Griffin: Rethinking Sparse Optimization for Deep Learning Architectures

Jong Hoon Shin, Ali Shafiee, Ardavan Pedram, Hamzah Abdel-Aziz, Ling Li, and Joseph Hassoun

Samsung Semiconductor Inc., San Jose, CA

{jhshin.1, ali.shafiee, ardanavan.p, hamzah.a, ling.li, j.hassoun}@samsung.com

Abstract—This paper examines the design space trade-offs of DNNs accelerators aiming to achieve competitive performance and efficiency metrics for all four combinations of dense or sparse activation/weight tensors. To do so, we systematically examine the overheads of supporting sparsity on top of an optimized dense core. These overheads are modeled based on parameters that indicate how a multiplier can borrow a nonzero operation from the neighboring multipliers or future cycles. As a result of this exploration, we identify a few promising designs that perform better than prior work. Our findings suggest that even the best design targeting dual sparsity yields a 20%-30% drop in power efficiency when running on single sparse models, i.e., those with only sparse weight or sparse activation tensors. We found that one can reuse resources of the same core to maintain high performance and efficiency when running single sparsity or dense models. We call this hybrid architecture Griffin. Griffin is 1.2, 3.0, 3.1, and 1.4× more power-efficient than state-of-the-art sparse architectures, for dense, weight-only sparse, activation-only sparse, and dual sparse models, respectively.

I. INTRODUCTION

In deep neural networks (DNNs) [37], rectified linear unit (ReLU) [33] and weight pruning [20]–[22], [35] enable accelerators to mitigate ineffectual computations (i.e., those with at least one zero operand) [46]. The former approach induces sparsity in activation tensors by zeroing out negative elements. While the latter approach induces sparsity in weight tensors by pruning insignificant weights. Although both approaches have shown promising results in several applications [41], [47], [66], they might not be always enabled. For instance, to improve DNNs accuracy, DNN developers might prefer dense non-linear activation functions, which don’t result in having as many zeros, such as swish [56], GeLU [28] or leaky ReLU [42]. Similarly, they might avoid weight pruning as it significantly increases training time, drops network accuracy, or because the network is already pre-trained dense. Therefore, both activation and weight tensors can be dense or sparse, categorizing DNN models and execution modes in four categories based on (activation, weight) tensor types: (dense,dense), (dense, sparse), (sparse,dense), and (sparse,sparse).

An accelerator might be optimized specifically for each of the above categories. However, DNN model categories are usually unknown at the design time for inference accelerators and might switch between different modes during training. Unfortunately, the optimal design point to run a category of DNNs is only optimal for the same type of DNN models. For example, architectures that are optimized for weight-only sparsity are not as efficient for activation-only sparsity models (i.e., (sparse, dense)) and cannot fully take advantage of dual sparse models (i.e., (sparse, sparse)). Even architectures that support dual sparsity do not get the best area and power efficiency for (dense, sparse) or (sparse, dense) models (see Section III).

At the edge, where area and power have strict budgets, it is challenging to support all categories of workloads efficiently, especially when both compute unit and SRAM are optimized for a specific category so that the share of sparse overheads on top of the dense design becomes significant [52], [69]. Examples of such dense accelerators include ARM Ethos-N77 (1MB per 2048 MACs) [1], NVDLA (0.5MB per 1024 MACs) [2]. For instance, in the case of DNN models with (sparse, sparse) tensors, an accelerator with significant sparsity overheads to gain substantial speedup is justifiable [18], [32]. However, it does not perform efficiently for other categories.

Unlike some of the previous work [18], [50], we consider the possibility of efficiently running all categories of DNN execution modes, early, at the design time. We start with an efficient dense baseline that exploits high degrees of parallelism and data locality. Then we identify sources of borrowing effectual operations from future computations across several dimensions of blocking and time, and mainly exploited resources (e.g., multiplexers and buffers) that can be re-purposed from one category of execution mode to another. We devise a novel hybrid architecture to reuse the hardware overheads in dual sparse architectures for DNN models that are only sparse in either activation or weight tensors. We call this hybrid architecture Griffin which enhances dual sparse architectures for both weight-only and activation-only sparse DNN models thus maintaining power and area efficiency. We corroborate the effectiveness of our approach by comparing our design with state-of-the-art architectures for all model categories.

To reach an optimal design for each category as well as an efficient hybrid architecture, we create an analytical model, verified by a simulator, to investigate the sources of overheads based on parameters that indicate how a multiplier can borrow a nonzero operation from the adjacent multipliers or future cycles. Our model offers a framework to quantify several sparse architectures including some of the prior work such as Convolutional [7], Cambricon-X [70], and Bit-Tactical [13]. We further identify new design points that are more power and area-
1: for m in [0, M] // step=M
2: for n in [0, N] // step=N
3: for k in [0, K] // step=K
4: for all m in [0, M] // step=1, dimensions da1, db1
5: for all n in [0, N] // step=1, dimensions da3
6: for all k in [0, K] // step=1, dimension db3
7: i = Mm1 + m2; j = Nn1 + n2; k = Kk1 + k2
8: c(i, j) = a[i, k] × b[k, j],

Fig. 1. The pseudo-code depicting the blocking of \( C+ = A \times B \) with \( A_{M \times K}, B_{K \times N}, \) and \( C_{M \times N} \) matrices. The first two loops (Line 1-2) tile the codes so that it fits on the GEMM accelerator core. The accelerator realizes the inner loops (lines 3-9) and unrolls the operation in three dimension \( (M_0, N_0, K_0) \). The core requires that matrices \( A \) and \( B \) to be reshaped as 3D tensors. The maximum distance from which a non-zero element can be borrowed across each of the dimensions of these input tensors are represented as \( da_1, da_2, da_3 \) for \( A \) and \( db_1, db_2, db_3 \) for \( B \).

In this paper, we focus on architectures that support both sparse and dense DNN models, efficiently. We first review an optimized dense architecture in Section [II-A] Then we review different types of sparse architectures.

### A. Dense Architecture

Most dense DNN accelerators rely on a customized unit for general matrix-matrix multiplication (GEMM). GEMM, defined as \( C++ = A \times B \), is the main building block of popular DNNs such as CNNs [36], [45] and Transformers [15]. For layers such as convolution layer (CL) and fully connected layer (FL), input tensor, layer’s parameters (weights), and output tensor are represented as \( A_{M \times K}, B_{K \times N}, \) and \( C_{M \times N} \), respectively.

In FL the kernel is represented as 2D matrix \( B_{K \times N} \), and the input activations as vectors of length \( K \). A batch of input activations can therefore be represented as a 2D matrix \( A_{M \times K} \) with \( M = \text{Batch size} \) that after multiplication by kernels results into a batch of outputs \( C_{M \times N} \). In a convolution layer, the kernel is represented as a 2D matrix \( B_{K \times N} \) with \( K = C_{in} \times R \times S \) and \( N = C_{out} \), where \( C_{in}, R, S, \) and \( C_{out} \) are the number of input channels, filter height, filter width and the number of output channels, respectively. As a result, the input feature map is reshaped as a 2D matrix \( A_{M \times K} \) with \( M = H_{in} \times W_{in} \) and \( K = C_{in} \times R \times S \), where \( H_{in} \) and \( W_{in} \) are the height and width of each input channel. [19] For transformer-based models, GEMM operations appear in the self-attention and feed-forward layers. The self-attention layer leverages GEMM operations to transform token vectors to key, query, and value vectors. Moreover, checking the similarity between all the generated query and key vectors is performed by GEMM.

GEMM can be implemented in hardware with two main optimizations: (1) Memory hierarchy optimization using blocking to minimize the size and the data movement between different levels of the memory hierarchy [52], [53], [61], [69], and (2) unrolling the nested loops in space to exploit parallelism and minimize energy per access [34], [44], [52], [53], [68], [69]. Figure [1] illustrates the high-level structure of the dense GEMM accelerator and how the operation is mapped onto it. As shown in the figure, the 3D realization of GEMM requires both matrix A and B to be rearrangedblocked in three dimensions as well. More specifically, each rows/columns of Matrix A/B, respectively, is stored in a 2D fashion in SRAM bank. Therefore, each element in A/B is adjacent to other elements in three dimensions as shown in Figure [1] for \( a_{i,j} \) and \( b_{i,j} \). The adjacency of elements in the three dimensions is key to our approach of modeling sparse architecture described in Section [III] and Section [V]. More specifically an element can be borrowed from maximum distance of \( da_1, da_2, da_3 \) \( (db_1, db_2, db_3) \) across all neighboring dimensions.

### B. Sparse Models and Architectures

There are four categories of DNN models as described in Section [I] Sparsity of A/B in DNN.dense, DNN.A, DNN.B, and DNN.AB categories are dense/dense, sparse/dense, dense/sparse, sparse/sparse, respectively. Analogously, there

### II. PROBLEM DEFINITION

The rest of this paper is organized as follows. Section [II] defines the problem and base dense architecture. Section [III] describes architectures that only support weight or activation sparsity. Section [IV] explores supporting both activation and weight sparsity. Section [V] presents our methodology and Section [VI] presents the results. Section [VII] goes over the prior work. Finally, we conclude this paper in Section [VIII].
Throughout the next two sections, we quantify the overheads da (gray: borrowing from skipped on-the-fly. To avoid under-utilization, zero operations are replaced from neighboring entities (c) in metadata from preprocessing is later used to select appropriate values from Matrix A. The zero entities on Matrix A is detected and provide additional logic to find zero operands, either by approach minimizes the control overhead, dense datapath is concurrently [5], [8], [9], [11], [26], [30], [40], [49]. While this approach minimizes the control overhead, dense datapath is unable to skip ineffectual operations [45]. Sparse architectures provide additional logic to find zero operands, either by preprocessing or on the fly detection and skipping of ineffectual operations. The skipped operations would be replaced with nonzero operations from future cycles of the same or adjacent multipliers. The adjacent multipliers are those that their operands are within proximity elements. We consider two elements as close in Matrix A/B, if they are close in da1, da2, and da3 or db1, db2, and db3, respectively (See Figure 1).

In general, detecting zero operations and replacing them with nonzero ones imposes additional overheads on top of the dense core. Figure 2 illustrates the overheads based on the dimension of the two adjacent operands when only one of Matrix A or B are sparse. Matrix B is known before execution, hence it is preprocessed before being written into SRAM banks. Preprocessing replaces zero entities with nonzero entities from the neighboring elements, which generates the metadata as well as a more compressed form of Matrix B. Figure 2(a) shows the case where b0,0 and b2,0 are zero and replaced by b3,0 and b4,0, respectively. More specifically, the solid black arrow along db1 dimension represents that non-zero b3,0 is sent to the zero b0,0. The two solid golden arrows send b4,0 to the diagonally neighboring element b2,0 along db2 and db1 dimensions. Therefore, for the pair (b0,0,b3,0) and pair (b2,0,b1,0), the borrowing distances are (db1,db2,db3) = (1,0,0) and (db1,db2,db3) = (1,1,0), respectively. In either of these cases, extra MUXs, called AMUX, before operands in A are needed. These MUXs select appropriate A-elements via the dashed black/golden arrows, based on the metadata derived from matrix B.

In Figure 2(b), b0,1 is replaced with b3,0 which is sent to diagonal direction through two consecutive green arrows along db1 and db2 dimensions. Thus, the borrowing distance is (db1,db2,db3) = (1,0,1), which leads to performing the computation in a neighboring PE (i.e., PE0,1). Therefore, AMUX and extra adder tree are required to navigate the

### III. Sparsity Overhead Analysis

In a typical dense GEMM accelerator, multipliers share operand fetch logic and all of them execute operations concurrently [5], [8], [9], [11], [26], [30], [40], [49]. While this approach minimizes the control overhead, dense datapath is unable to skip ineffectual operations [45]. Sparse architectures provide additional logic to find zero operands, either by preprocessing or on the fly detection and skipping of ineffectual operations. The skipped operations would be replaced with nonzero operations from future cycles of the same or adjacent multipliers. The adjacent multipliers are those that their
ABUF also requires MUXs, but they can be shared between BBUF. ABUF are also fetched to a buffer called BBUF. This architecture allows replacing a zero value by two consecutive solid orange arrows along \( da \) dimensions (Therefore, distance is \((da_1, da_2, da_3) = (1, 0, 1)\)). The associated B-element, \( b_{1,0} \), in BBUF is selected by BMUX. (dashed orange arrow) In this case, an extra adder tree is required to send partial-sum values to the correct accumulator (accumulator in PE\(_{0,0}\)), as the multiplication is performed in the adjacent PE (i.e. PE\(_{0,0}\)) with a different accumulator (dashed blue arrows). When only one of the input matrices is sparse, we can define single sparse architectures based on the maximum distance of borrowing nonzero operands in each dimension to replace zero operands with nonzero ones.

**Definition III.1.** Sparse.\( A(da_1, da_2, da_3) \) is an architecture that only supports sparsity in matrix \( A \) of the GEMM operation. This architecture allows replacing a zero value \((x_1, x_2, x_3)\) with a nonzero value \((x_1 + \Delta_1, x_2 + \Delta_2, x_3 + \Delta_3)\) where \( \Delta_i \leq da_i \).

**Definition III.2.** Sparse.\( B(db_1, db_2, db_3) \) is an architecture that only supports sparsity in matrix \( B \) using preprocessing. This architecture allows replacing a zero elements \((x_1, x_2, x_3)\) with a nonzero element \((x_1 + \Delta_1, x_2 + \Delta_2, x_3 + \Delta_3)\) where \( \Delta_i \leq db_i \).

Therefore, the single sparse architectures described in Figure 2(a), (b), (c), and (d) can be specified as Sparse.\( B(1,0,0) \), Sparse.\( B(1,0,1) \), Sparse.\( A(1,1,0) \), and Sparse.\( A(1,0,1) \), respectively. In the architectures mentioned above, there are \( da_1 \times da_2 \times da_3 \) and \( db_1 \times db_2 \times db_3 \) potential candidates for replacing a zero operand in \( A \) and \( B \), respectively. In this work, we use a similar priority mechanism as [13] whenever multiple nonzero candidates exist. For both Sparse.\( A \) and Sparse.\( B \) families of architectures, we recognize the following sources of overhead: ABUF, AMUX, BBUF, BMUX, and adder tree (ADT). The depth of ABUF and BBUF, the fan-in of AMUX and BMUX, and the number of required adder trees depend on the limits of distance for replacement elements in different dimensions. Table II expresses the dependency for both Sparse.\( A \) and Sparse.\( B \). In this table, we also provide other restricted instances from each family of architectures to reveal the impact of replacing elements in each direction. Using equations in Table II we can directly estimate and compare the cost of different single sparse architectures. For example, Sparse.\( A(1,1,0) \) shown in Figure 2(d) can be upgraded to Sparse.\( A(1,1,1) \) by enabling \( da_3 = 1 \), which requires twice larger AMUX fan-in, and one extra adder-tree per PE.

**Load Balancing:** For unstructured sparse input matrix \( A \) and \( B \) the zero operands are not necessarily evenly distributed. This issue still exists after preprocessing \( B \) or on-the-fly zero skipping on \( A \). Coarse-grain load balancing is an effective approach to distribute nonzero values and improve performance utilization [18], [32]. In this approach a GEMM operation is decomposed into smaller blocks and each block is assigned to available idle PEs. In this work, however, we consider a recently proposed fine-grain approach by shuffling input matrix \( A \) and \( B \) along their second dimension (i.e., \( da_2 \) and \( db_2 \)) in the core [29]. The shuffling happens, over the dense matrices \( A \) and \( B \), before applying preprocessing or entering the buffer for on-the-fly zero skipping. While there are many ways to perform shuffling, we observe that simple permutation is sufficient. Thus, if an element is located in \((i_1, i_2, i_3)\) in an input matrix, it will be relocated to \((j_1, i_2 \bmod K_0, i_3)\), where \( K_0 \) is the size of the second dot product unit (see Figure 1).

Note that shuffling happens on both Matrices \( A \) and \( B \). To navigate the elements of \( A \) to its corresponding \( B \)'s elements, rotation-based shuffling requires a \( K_0 \times K_0 \) crossbar between SRAM and ABUFs. Therefore we limit the shuffling to local rotations only between four consecutive elements (in \( da_2 \) and \( db_2 \)) to reduce the \( K_0 \times K_0 \) crossbar to multiple \((K_0/4) \times 4 \times 4\) crossbars. In our experiments, this localization does not impact the load balancing. We use the notation \textit{shuffle = on} and
This is achieved by enabling zero operand replacement from \( \Delta \). Similarly, in Matrix B, it allows replacing a zero operand with a nonzero value. A hybrid architecture replaces a zero operand in architecture that supports sparsity in both A and B. We define dual sparse architectures as follows:

**Definition IV.1.** Sparse.AB\((da_1, da_2, da_3, db_1, db_2, db_3)\) is an architecture that support sparsity in both A and B. This architecture replaces a zero operand in \((x_1, x_2, x_3)\) with nonzero value \((x_1 + \Delta_1, x_2 + \Delta_2, x_3 + \Delta_3)\) where \(\Delta \leq da_1\). Similarly, in Matrix B, it allows replacing a zero operand with nonzero value \((y_1 + \Delta'_1, y_2 + \Delta'_2, y_3 + \Delta'_3)\) where \(\Delta'_i \leq db_i\).

In this section, we explain our approach using a walk-through example, shown in Figure 3. The following seven steps are required to support dual sparsity.

1) **Preprocessing B:** Since B is known before the execution, it is preprocessed to a compressed format with metadata. The preprocessed elements of B in SRAM are fetched in BBUF, which holds a window of current elements every cycle. In our example, \(b_{1,0}, b_{3,0}, b_{5,0}, b_{6,0}, b_{7,0}, b_{10,0}\) are kept in BBUF.

2) **Generating zero masks:** ABUF keeps the elements of A corresponding to the element of B currently residing in BBUF. For each of these elements in ABUF, a single mask would be generated indicating whether it is zero or not. \(a_{0,0}, a_{0,1}, a_{0,4}, a_{0,10}, a_{0,11}\) are the elements of A in ABUF with mask bit equals 1.

3) **Filtering zero masks:** using the metadata from B, the zero mask get updated by zeroing those mask bits when their corresponding weight is zero. In our example, \(a_{0,0}, a_{0,4}, a_{0,10}\) become zero.

4) **Arbitration:** The remaining ones in the zero mask associate with operations with both nonzero operands in A and B. In this step, these ones are detected and selected.

5) **Index generation:** For the selected ones in the zero mask, their indices in A and B are extracted using priority encoders which detect the first unused non-zero value in each bit-index pair list and generate the corresponding indices. In our example, \(a_{0,0}, a_{0,4}, a_{0,10}\) and \(a_{0,11}\) become zero.

6) **Operand selection:** Using the indices generated in Step 6, two vectors of operands are selected from ABUF and BBUF.

7) **Execution:** The selected vectors are fed into the PE for execution.

Fig. 3. Dual sparse architecture steps. (1) Preprocessing Matrix B, (2) Generating zero mask for Matrix A, (3) Filtering out the zero mask of A, (4) Selecting nonzero elements, (5) Generating index, (6) Selecting elements of A and B using indices, and (7) Execute the dot product.

shuffle = off to indicate whether an architecture support rotation-based shuffling or not.
We discuss how we derive these parameters for the optimal performances in Section VI. This configuration requires 9-entry ABUF, 3-entry BBUF, 9-input AMUX, and 3-input BMUXs, and one extra adder tree. Without hybrid architecture, this design point downgrades to Sparse.A(2,0,0) and Sparse.B(2,0,1) for DNN.A and DNN.B models, respectively. In this case, the main overhead, such as the large ABUF would not be underutilized (the above-downgraded models only require 3 entries of the ABUF). On the other hand, Griffin morphs into a more aggressive configuration for DNN.A, it morphs into Sparse.A(2,1,1). In each subfigure, we highlight the part of ABUF and BBUF used in the configuration for PE0,0.

**Figure 4.** Griffin is a hybrid architecture that morphs into different configuration for different categories of benchmarks. (a) When running dual sparse benchmark, Griffin allows (2,0,0,2,0,1) borrowing from (da1, da2, da3, db1, db2, db3). (b) When running benchmark with only sparse Matrix B, it morphs into Sparse.B(2,0,1). (c) When running benchmark with only sparse Matrix A, it morphs into Sparse.A(2,1,1).

**Figure 4(b)** illustrates Griffin morphing into Sparse.A(2,1,1) for DNN.A models. In this mode, the entire three elements of BBUF are needed. In addition, the extra adder tree of each PE is also reused because these configurations allow borrowing from da3. However, there are three main changes to the dual sparse mode. (1) Sparse.A(2,1,1) requires three entries of ABUF from the current row and two from the neighboring one. Since ABUF has nine spaces from nine entries, the element from the neighboring ABUF is also copied into the current ABUF. (2) The process of zero skipping and arbitration becomes more complicated as borrowing from da2 direction is permitted. However, just one arbiter is needed per row of PE since only A is sparse. Moreover, the control logic in each PE is bypassed. (3) Due to enabling borrowing from da2 direction, the fan-in of BMUXs should increase from three to five. Table III summarizes the difference between Griffin and dual sparse architecture for DNN.A and DNN.B models.

**V. Experimental Setup**

The performance evaluation is based on a cycle-accurate simulation model developed in Python and PyTorch for several pruned DNN benchmarks. The python-based simulator receives weight and activation tensor blocks from pytorch and pre-process weight tensors, if necessary, and then computes
the number of cycles per block of tensors according to the borrowing strategy. Our simulation pipelines consider stalls due to output synchronization, SRAM bank conflicts, and ABUF/BBUF fullness. The cycle-accurate simulation estimate the inference end-to-end latency for the given benchmarks. The weight and activation (B,A) sparsity ratios, accuracy, and latency (i.e., number of cycles) with dense matrices for these benchmarks are listed in Table IV.

We consider the dense architecture depicted in Figure 1 as our baseline. The datapath configuration for this architecture is \( (K_0, N_0, M_0) = (16, 16, 4) \) with 1024 MAC operations per cycle. Our default multiplier precision for the MAC units is INT8.

The baseline on-chip memory is mainly A SRAM (ASRAM) and B SRAM (BSRAM). We optimized the baseline memory for better area efficiency by allocating only 512KB for ASRAM and 32KB for BSRAM. These SRAM sizes are within the range of efficient memory hierarchy design and consistent with commercial DNN accelerators such as NVDLA (512KB per 1024 MACs) [2] and ARM Ethos-N77 (1MB per 2048 MACs) [1]. Note that, the area overheads to support sparsity are mostly in the computation cores. Thus, a large ASRAM can be misleading as its area/energy consumption overshadows the impact of sparse overheads on the cores [52]. For baseline architecture, ASRAM and WSRAM bandwidth are 51.2GB/s and 204.8GB/s, respectively. To exploit the full sparsity speedup, SRAM BW should be equal or more than the multiplication of the normalized speedup and the baseline bandwidth. We used 50GB/s DRAM bandwidth which is enough to avoid any performance drop.

On top of the dense baselines, we also evaluate our architecture against three state-of-the-art sparse (SOTA) architectures: BitTactical [13], TensorDash [43], and SparTen [18]. We denote the dual sparse architectures TensorDash and SparTen as TDash.AB and SparTen.AB, respectively. We refer to the weight-only sparse architectures BitTactical to TCL.B, which can be considered as single sparsity version of TDash.AB with weight preprocessing. We also use SparTen.B and SparTen.A to refer to one-sided SparTen optimized for DNN.B and DNN.A, respectively. Table V summarizes the SOTA architectures that we compare against. For a fair comparison, we implemented these SOTA architectures with the same SRAM capacity in SystemVerilog and consistent with prior work [7], [30], [70].

To estimate power and area, we implemented the baseline, our proposed architecture, and the three SOTA sparse architectures with the same SRAM capacity in SystemVerilog and synthesized them using Synopsys DesignCompiler [4] and Synopsys memory compiler with 7nm technology libraries. We consider 800 MHz clock frequency and 0.71V voltage for our synthesis processes. The experimental setup and configurations are summarized in Table IV.

We use the geometric mean to estimate the evaluation metrics which include normalized speedup and power/area efficiency. For power and area efficiency metrics, we consider effective \( TOPS/W \) and \( TOPS/mm^2 \) defined as follows.

\[
\text{Effective TOPS}/W = \text{sparsity speedup} \times \text{dense TOPS}/W
\]

\[
\text{Effective TOPS}/mm^2 = \text{sparsity speedup} \times \text{dense TOPS}/mm^2
\]

### VI. Results

In this section, the design space exploration results of sparse architectures such as weight-only sparse, activation-only sparse, and dual sparse architectures are reported. We identify the optimal design points for each sparse architecture class, and also propose a hybrid architecture Griffin which performs well in three different DNN categories, DNN.A, DNN.B, and DNN.AB.

#### A. Weight-Only Sparsity Support:

For weight-only sparse architectures, we limit our analysis to those with AMUX fan-in smaller than or equal to eight, as larger MUXs would severely impact power efficiency. Figure 5(a) shows normalized speed up, with respect to the dense baseline, for all possible configurations with the above constraints. We remove the case with \( db1 = 1 \), as it is far from the optimal points. Below are the key observations from Figure 5:

1. Larger \( db1, db2, db3 \) leads to higher speed-up, with \( db1 \) has more impact than the other two parameters because it decides the ideal maximum speed-up to \( 1 + db1 \).
2. \( db3 > 0 \) can boost the performance by up to 48% (\( db3 = 1, B(4,0,0,off):1.7 \times \rightarrow B(4,0,1,off):2.5 \times \) and 72\% (\( db3 = 2, B(4,0,0,off):1.7 \times \rightarrow B(4,0,2,off):2.9 \times \)) with increasing power overhead of 10\% and 20\%, respectively.
3. Shuffling is effective, mostly for \( db1 > 2 \) and leads to up to 43\% improvement in speedup (\( B(6,0,0,off):1.9 \times \rightarrow B(6,0,0,on):2.7 \times \) (See Figure 5(a)).
4. Shuffling plays similar role as \( db2 \) because both shuffling and \( db2 \) mitigate the load imbalance between different \( k \) indices in GEMM operation. Therefore, when the shuffling is used in Sparse.B designs, the impact of \( db2 \) get diminished as shown in the speedup results from B(2,db2,0, on/off).
5. Balancing \( db2 \) and \( db3 \) is more effective than using only one large parameter between them because the speedup gain from \( db2 \) and \( db3 \) gets saturated as they increases. For example, B(2,1,1,2):2.6× outperform B(2,2,0,0):2.4× and B(2,0,2,0):2.4×.
6. Shuffling can boost the performance of nonzero \( db2 \) with a lower cost as shown in Figure 5(b).
Fig. 5. Impact of B-matrix routing configurations. We use shorter notation B(\(\text{db}_1, \text{db}_2, \text{db}_3\)) to refer to Sparse.B(\(\text{db}_1, \text{db}_2, \text{db}_3\)). When we indicate shuffling-on/off of the sparse design, ‘on’ or ‘off’ index is added as Sparse.B(\(\text{db}_1, \text{db}_2, \text{db}_3, \text{on/off}\)). (a) Normalized speedup with respect to the dense baseline for different configurations. The shaded bars are the results with shuffling. (b) Effective power efficiency for different design points for DNN.B benchmark (y-axis) and DNN.dense benchmark (x-axis), (c) Effective area efficiency for different design points for DNN.B benchmark (y-axis) and DNN.dense benchmark (x-axis).

**TABLE VI**

| Architecture | A-matrix Routing | B-matrix Routing | Shuffle |
|--------------|------------------|------------------|---------|
| Sparse.B*    | \(da_1\) | \(da_2\) | \(da_3\) | \(db_1\) | \(db_2\) | \(db_3\) |
| Sparse.A*    | 2       | 1       | 0       | 4       | 0       | 1       |
| Sparse.AB    | 2       | 0       | 0       | 2       | 0       | 1       |
| Griffin      | conf.B   | -       | -       | 8       | 0       | 1       |
|              | conf.A   | 2       | 1       | 1       | 8       | 0       |
|              | conf.AB  | 2       | 0       | 0       | 2       | 0       |

Figure 5(b),(c) show the power and area efficiency of weight-only sparse architecture families, for both pruned models (shown in y-axes) and non-pruned models (shown in x-axes). Looking into the Pareto optimal design points, Sparse.B(4,0,0,off), Sparse.B(4,0,1,off), and Sparse.B(4,0,2,off) show 95%, 127%, and 130% increase in power efficiency for pruned network, respectively. These three designs only impose 10%, 16%, and 22% power overhead respectively, compared to dense baseline models. (Listed in Table VI) Among the three designs, we chose Sparse.B(4,0,1,off) as an optimal design point for weight-sparse architecture, Sparse.B*, which shows high TOPS/W on DNN.B with minimal efficiency loss in DNN.dense. Finally, note that while SparTen.B shows 3.9× speedup, it hurts the power efficiency by 26% and increases only 1% area efficiency with DNN.B, compared to the baseline, due to large buffers, MUXs, and control path logic. On the other hand, TCL.B improves both area and power efficiency. However, we found that adding shuffling and \(db_3 > 0\) can significantly increase power efficiency for TCL.B up to 47%.

**B. Activation-Only Sparsity Support:**

We narrow down the design space exploration of architectures that support activation-only sparsity to those with AMUX/BMUX fan-in smaller than or equal to eight. This is based on our observation that designs with larger AMUX/ BMUX also require deeper BBUF, which is expensive and reduces power and area efficiency. Figure 6(a) shows the power and area efficiency of these design points for a design with (shown in y-axes) and without ReLU (shown in x-axes). We observe that,

1. \(da_1\) is not as important as \(db_1\) in Sparse.B architectures because the average sparsity level due to ReLU is close to 50% that gives ideal speedup of \(\sim 2x\). \(A(2,1,0,\text{on}):1.83\times \sim A(3,1,0,\text{on}):1.89\times\)

2. \(da_3 > 0\) leads to power and area efficiency drop with insignificant speedup \(A(2,1,0,\text{on}):1.83\times \rightarrow A(2,1,1,\text{on}):1.93\times \rightarrow A(2,1,2,\text{on}):1.97\times\), while power/area overhead for \(A(2,1,1,\text{on})\) and \(A(2,1,2,\text{on})\) are 9%/4% and 17%/6%, respectively.

3. Shuffling boosts performance by up to 40% \(A(4,0,1,\text{on}):1.28\times \rightarrow A(4,0,1,\text{on}):1.79\times\).

4. Performance of designs with \(da_1 \geq 4\) is limited since they cannot use \(da_2 > 0\) due to the AMUX fan-in size limit \((\text{cf. AMUX}=1 + da_1 \times (1 + da_2) \times (1 + da_3))\).

Sparse.A(2,1,0,\text{on}), which we selected as an optimal design point (Sparse.A*) among activation sparse architectures, leads to 26% increase in power efficiency for DNN.A models while reducing power efficiency by 18% for DNN.dense models. (Listed in Table VI) This configuration leads to 58% increase in area efficiency, for DNN.A with only 14% decrease for DNN.dense models.

We also observe that SparTen.A can achieve 2× speedup at the cost of 62% and 49% power and area overheads, compared to the optimized dense baseline. This is because SparTen.A does
We observe the following trends in Figure 7.

C. Dual Sparsity Support:

The family of dual sparse architectures has seven parameters, namely $da_1$, $da_2$, $da_3$, $db_1$, $db_2$, $db_3$, and shuffling. To explore the design space we allow configurations that lead to AMUX fan-in that is not larger than 16. Compared to weight-only and activation-only sparsity, we consider larger fan-in, as dual sparse design can achieve higher performance, hence, can tolerate higher area and power overhead. Several design points satisfy this restriction and we show the best-performing ones in Figure 7(a). We observe that $4.9 \times$ speedup is achieved by Sparse.AB(2,0,0,4,0,2,on) compared to the dense baseline.

D. Hybrid Architecture (Griffin):

To evaluate our hybrid architecture, Griffin, we consider the impact of hybrid enhancement on Sparse.AB*. When running DNN.B models, it is reconfigured to $conf.B(db_1, db_2, db_3, shuffle) = (8, 0, 1, on)$ with $3.5 \times$ speedup. This configuration of Griffin allows DNN.B models to achieve 25% and 42% better power efficiency and area efficiency, respectively, compared to Sparse.AB*, as shown in Figure 8(b). The architecture uses $conf.A(da_1, da_2, da_3, shuffle) = (2, 1, 1, on)$, for running DNN.A models, with $1.94 \times$ speedup, which translates to 23% and 20% power efficiency and area efficiency improvement, respectively (Figure 8(c)). The efficiency improvement in DNN.B and DNN.A was achieved by only $\sim$1% drop of power efficiency and area efficiency in both DNN.dense and DNN.AB.
We use shorter notation $AB(da_1, da_2, da_3, db_1, db_2, db_3)$ to refer to Sparse.AB$(da_1, da_2, da_3, db_1, db_2, db_3)$. When we indicate shuffling-on/off of the sparse design, 'on' or 'off' index is added as Sparse.AB$(da_1, da_2, da_3, db_1, db_2, db_3, on/off)$. (a) Normalized speedup with respect to the dense baseline for different configurations. The shaded bars are the results with shuffling. (b) Effective power efficiency for different design points for DNN.AB benchmark (y-axis) and DNN.A benchmark (x-axis), (c) Effective Area efficiency for different design points for DNN.AB benchmark (y-axis) and DNN.A benchmark (x-axis).

All three configurations of Griffin are listed in Table VI. Griffin is also 1.2, 3.0, 3.1, and 1.4× more power-efficient and 3.8, 3.1, 3.7, and 1.8× more area-efficient compared to Sparten, the state-of-the-art dual sparse architecture to the best of our knowledge, for DNN.dense, DNN.B, DNN.A, and DNN.AB, respectively. Note that the conf.A of Griffin for DNN.A is not as effective as conf.B for DNN.B, since activation tensors are denser than weight tensors. Furthermore, activation sparsity is handled in real-time which is less effective than the preprocessing used for weight-only sparsity cases.

E. Hardware Overhead and Breakdown:

Table VII shows the power and area breakdown of the dense baseline as well as Sparse.B*, TCL.B, Sparse.A*, Sparse.AB*, Griffin, TDash.AB, and Sparten.AB in the order of increasing power efficiency. In the dense baseline, we observe that multipliers are dominant both in power and area. Allocating most of the resources to the compute unit makes the design more efficient and sparsity overhead more amplified. In both Sparse.B* and Sparse.A* sparsity cases, the control overhead is insignificant. Moreover, power and area overhead of shuffler is less than 1.0% and 0.7% compared to the dense, respectively. In all types of sparse designs, the main power overheads come from registers added to the data path pipeline and those needed for ABUFs and BBUFs. The selected design for Sparse.A*, Sparse.B*, Sparse.AB*, and Griffin increase the data path power, compared to the dense baseline, by 46%, 34%, 72%, and 73%, respectively. Moreover, these designs increase the area overhead with respect to the dense baseline, by 16%, 19%, 30%, and 32%, respectively. We also observe that the control path imposes 12% and 4.3% power and area overhead, in both Sparse.AB* and Griffin, as they require one control unit per PE.

Since TCL.B and TDash.AB added sparse logic designed on top of efficient 3D dense core like Griffin, the power and area cost of those architectures are similar to Sparse.B* and Griffin, respectively. Therefore, the higher speedup from Sparse.B* and Griffin caused better power and area efficiency compared to TCL.B and TDash.AB. In the case of Sparsten.AB, both the PE and BUFs cause inefficiency because the Sparten.AB with MAC-based architecture does not share accumulators (which consume 110mW) and uses BUFs with a depth of 128. This buffer makes the power and area increase by 416mW and $6.4 \times 10^5 \mu m^2$, respectively (larger than the power and area of baseline architecture).

F. Overall Comparison:

Figure 8 depicts the normalized area and energy efficiency of the several selected architectural configurations as well as the related work for four different types of DNN models. The goal for optimal design is to remain a top performer for all four categories of DNN models. This is only achieved by Griffin design. For DNN.dense models, most of the designs, except for the Sparten, perform in the same ballpark of efficiency and all less efficient of the dense baseline. That is expected as DNN.dense models do not reflect the gains in speedup associated with supporting sparsity. The reduced efficiency in Figure 8(a) can be interpreted as sparsity tax spent for the sake of the sparsity gain. Griffin shows more economic sparsity tax of 29%/24% for power/area efficiency than 42%/80% of Sparten.AB. However, these various designs show large...
contrast, especially in DNN.B and DNN.AB models and beat the baseline by factors of integer in area/energy efficiency. The exceptions are Spartan.AB and Sparse.A* still perform overall worse than the baseline if the task is not dual sparse. TCL.B is close to the baseline for DNN.dense and DNN.A models and performs reasonably well for DNN.B and DNN.AB cases. While TDash.AB shows similar efficiency to Sparse.AW* in DNN.dense, DNN.A, and DNN.B, it does not perform well in DNN.AB models, compared to Sparse.AB* and Griffin.

VII. RELATED WORK

Recently, several optimization techniques are proposed to improve compute and memory efficiency for DNNs [14], [59], [63]. Various deep learning accelerators improve DNNs performance by exploiting weight and/or activation sparsity [12]. Cambricon-X [70] and Bit-Tactical (TCL) [13] are two designs based on inner product compute units that focus only on exploiting weight sparsity. Cambricon-X leverages a queue based on inner product compute units that perform an outer product between two nonzero weight matrices in a compressed format using a systolic array. Cambricon-X exploits only activation sparsity, without shuffling, by compressing them in time (i.e., \( db1 \))

Other work proposed CNN architectures to support dual sparsity [13], [32], [50]. SCNN uses a 2D array of compute units that perform an outer product between two nonzero weight and activation vectors [50]. The outputs of the compute units are routed to their corresponding accumulator using a heavy-weight crossbar which introduces a substantial overhead.

ZeNA [32] is a dual sparsity architecture similar to Spartan. Both are based on sparse-aware MAC units which route computations in time (i.e., \( db1 \) and \( da1 \)) only for each MAC unit independently. However, they are limited in efficiency and scalability due to their limited data movement and reuse between MACs. In contrast, Griffin routes irregular sparsity among all core dimensions.

Eyeriss v2 [10] introduces a hierarchical mesh network for flexible processing of sparse weights and input activations directly in a compressed domain. In contrast to our work, it introduces large overheads degrading the performances of dense models. Sparse-TPU [25] incorporates index matching and value holding functionalities to efficiently process sparse matrices in a compressed format using a systolic array.

In addition, there are other sparse architectures which exploit bit-level sparsity in weights and/or activation such as Stripe [31], Bit-Pragmatic [6], Loom [59], Laxonic [58], Bit-Fusion [60], UNUP [38] and activation-side in TCL [13]. Bit-level sparsity requires extra overhead due to bit-serial operation which

![Fig. 8. Power efficiency vs Area efficiency in different DNN categories such as (a) DNN.dense, (b) DNN.B, (c) DNN.A, (d) DNN.AB.](image)
degrades the power/area efficiency compared to bit-parallel designs.

Outerspace [48] proposes a design based on cores to perform sparse outer products. SpArch [71] optimizes sparse GEMM by reducing memory footprint (DRAM access). Tensaurus [62] introduces a new sparse storage format, compressed interleaved sparse slice (CISS) for sparse-dense matrix operations. ExTensor [27] targets general sparse tensor algebra, optimizing the memory hierarchy at multiple levels through a general abstraction based on intersections between non-zero data coordinates. Spaghetti [3] designs a pattern-aware software scheduler to leverage sparsity patterns for optimal DRAM utilization. Sparse CNN FPGA accelerator [72] proposes a sparse dataflow to skip zero weights and minimize off-chip memory accesses. Procrustes [67] produces pruned models from the sparse DNNs training with an optimized Dropback algorithm. SIGMA [55] introduces a highly Flexible Dot Product Engine and Forward Adder Network to enable efficient GEMM computations in DNNs training.

**VIII. Conclusions**

This work describes a systematic approach to model the family of architectures that support various flavors of DNNs sparsity. We explore the design space and offer multiple insights regarding the best dimensions to borrow from for varieties of sparse architectures. We propose a hybrid architecture, Griffin, to enhance dual sparse architectures to perform close to optimal power efficiency even when both weight and activation tensors are not sparse. We evaluate the proposed design in contrast with previous work for different model categories (i.e., DNN.dense, DNN.B, DNN.A, and DNN.AB). The result shows that Griffin architecture improves power and area efficiency by up to 3.1 and 3.8×, respectively, compared to the state-of-the-art dual sparse architectures.
[72] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, “An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 1953–1965, 2020.

[73] M. Zhu and S. Gupta, “To prune, or not to prune: exploring the efficacy of pruning for model compression,” *arXiv preprint arXiv:1710.01878*, 2017.