Cascading Structured Pruning: Enabling High Data Reuse for Sparse DNN Accelerators

Edward Hanson
Duke University
Durham, North Carolina, USA
edward.t.hanson@duke.edu

Shiyu Li
Duke University
Durham, North Carolina, USA
shiyu.li@duke.edu

Hai 'Helen' Li
Duke University
Durham, North Carolina, USA
hai.li@duke.edu

Yiran Chen
Duke University
Durham, North Carolina, USA
yiran.chen@duke.edu

ABSTRACT

Performance and efficiency of running modern Deep Neural Networks (DNNs) are heavily bounded by data movement. To mitigate the data movement bottlenecks, recent DNN inference accelerator designs widely adopt aggressive compression techniques and sparse-skipping mechanisms. These mechanisms avoid transferring or computing with zero-valued weights or activations to save time and energy. However, such sparse-skipping logic involves large input buffers and irregular data access patterns, thus precluding many energy-efficient data reuse opportunities and dataflows. In this work, we propose Cascading Structured Pruning (CSP), a technique that preserves significantly more data reuse opportunities for higher energy efficiency while maintaining comparable performance relative to recent sparse architectures such as SparTen. CSP includes the following two components: At algorithm level, CSP-A induces a predictable sparsity pattern that allows for low-overhead compression of weight data and sequential access to both activation and weight data. At architecture level, CSP-H leverages CSP-A's induced sparsity pattern with a novel dataflow to access unique activation data only once, thus removing the demand for large input buffers. Each CSP-H processing element (PE) employs a novel accumulation buffer design and a counter-based sparse-skipping mechanism to support the dataflow with minimum controller overhead. We verify our approach on several representative models. Our simulated results show that CSP achieves on average 15× energy efficiency improvement over SparTen with comparable or superior speedup under most evaluations.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; *Systolic arrays*; *Real-time system architecture.*

KEYWORDS

ML Acceleration, Model Compression, Hardware/software co-design, Low Power Microarchitecture

ACM Reference Format:
Edward Hanson, Shiyu Li, Hai 'Helen' Li, and Yiran Chen. 2022. Cascading Structured Pruning: Enabling High Data Reuse for Sparse DNN Accelerators. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527419

1 INTRODUCTION

Deep Neural Networks (DNNs) are increasingly prevalent due to their use on a wide range of applications. DNNs consist primarily of two layer types—convolutional and fully-connected—that have vastly different data movement-related bottlenecks. Fully-connected layers are known for having a large number of parameters, causing it to be weight-data dominant. Meanwhile, convolutional layers reuse the weights across feature maps. This sliding-window nature of convolutional layers causes it to be activation-data dominant.

Many recent accelerators aim to solve the data movement bottleneck using various approaches. Works such as Cambricon-X [38] leverage weight sparsity by holding the model in a compressed format, thus skipping ineffectual weights. Other works [5, 37, 41] additionally induce weight sparsity in a specialized manner that benefits the accelerator. However, the induced weight sparsity of these approaches only benefit regularity of the data and do not target the data reuse opportunities across processing elements (PEs).

Another approach to reducing data movement is real-time compression of activation data. As shown in Fig. 1, repeated activation data accesses comprise the bulk of data movement energy. These works, including SparTen [10], SCNN [28], EIE [11], and Cnvlutin [2], target the inherent sparsity of feature maps induced by nonlinear layers such as ReLU. Unlike weights, the sparsity of activations are not known ahead of time, so accelerators that target activation sparsity rely on complex mechanisms such as content addressable memory (CAM) to identify ineffectual activations. This procedure increases complexity of the controller and limits options for efficient dataflows. Additionally, large input buffers are required to store and locate ineffectual activations, thus increasing the footprint of the accelerator. For example, SparTen allocates $0.97KB$ per multiply-and-accumulate (MAC) unit while Cambricon-S allocates $2.01KB$ per MAC. The additional savings produced by skipping activation data may not be worth these area and static power overheads. In fact, as shown by ExTensor [16], improved performance of sparse tensor processing does not necessarily come from the *magnitude* of sparsity; rather, it comes from the sparsity *pattern*.

Motivated by these observations, we present Cascading Structured Pruning (CSP). CSP is a collaborative approach at the algorithm (CSP-A) and hardware levels (CSP-H). To reduce the storage demands and data movement of weights, CSP-A structurally prunes the weights. The key distinction of CSP-A from previous approaches is that the induced weight sparsity preserves the sequential access of activation data by considering the temporal progression of the dataflow. Then, CSP-H employs a novel dataflow to exploit the compressed weight format and sequential activation data accesses to eliminate redundant activation reads. Rather than skipping ineffectual activations, CSP-H reduces activation data movement energy by maximally reusing activation data across its PEs and limiting the movement of partial sums to within each PE. The main contributions of our work include:

- We introduce a novel structured pruning algorithm called CSP-A. CSP-A removes subsequent weights that correspond to the same coordinate of a tensor product across timesteps of a dataflow. The resulting sparse weight structure enables sequential access of the activation data.
- We propose the CSP-H accelerator. CSP-H leverages the sparse weight structures induced by CSP-A with two novel dataflows for one-time access of activation data. The PEs of CSP-H employ a new accumulation buffer to support the dataflow with low controller complexity.
- We analyze opportunities to further improve the area and energy efficiency of the PEs via *periodic* partial sum truncation. Results show that periodically truncating the partial sums to lower bit-widths incurs negligible accuracy loss.

We verify our approach on various representative models, including AlexNet [19], VGGNet [32], ResNet [13], Inception [34], and Transformer [35]. We then compare our design to state-of-the-art accelerators, including Cambricon-S [41] and SparTen [10]. Results show that CSP achieves on average 15× energy efficiency over SparTen while maintaining comparable speedup under most evaluations.

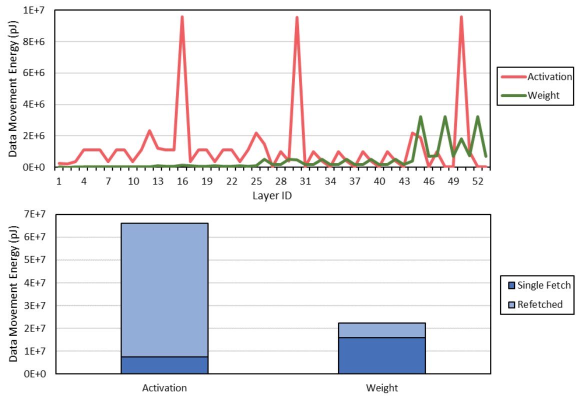


Figure 1: Data movement-related energy on ResNet-50. (Top) Layer-wise. (Bottom) Unique vs. re-fetched data.

2 BACKGROUND AND MOTIVATION

2.1 Sparse DNN Processing

Before discussing recent approaches to sparse tensor processing, we first review key terminologies. The computationally dominant layers of a DNN involve *input feature map* (IFM) and *filter* tensors, where individual elements of IFMs and filters are called *activations* and *weights*, respectively. Filters and feature maps are comprised of *kernels* and *channels*, respectively. The spatial position of each element in a kernel or channel is called a *pixel*. In this work, we define a *chunk* to be a collection of filters and a *pass* to be a single instance of tensor mapping to the hardware. For example, if a PE array pins the filters to the PEs (i.e. weight stationary), a single pass concludes when all computations related to this set of weights are complete. The primary operation of these layers are matrix multiplications of the IFMs and filters to produce *output feature maps* (OFM). Using a dense representation of IFMs and tensors, accessing individual elements is straightforward because they can be stored sequentially in memory. However, DNN filter tensors are known to be around 50−90% sparse [16]; popular activation functions such as ReLU introduces many zeroes into IFMs while filters often contain many redundancies that are removed using popular pruning methods [9, 12, 14, 15, 23]. To incur lower memory footprint and less compute, the tensors can be stored and executed in compressed formats such as Compressed Sparse Row (CSR) or using bit-masks. No matter the chosen compressed format, tensor compression introduces an additional structure that preserves the location (i.e., *coordinate*) of non-zero values.

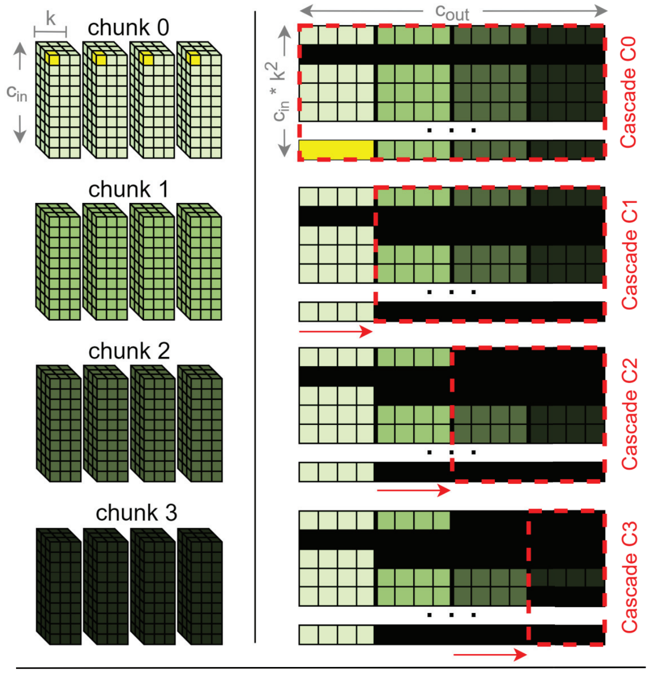
Sparse tensors can be exploited for speedup by removing ineffectual computations. In the context of computing DNN matrix multiplications, ineffectual computations come from multiplications involving a zero-value activation, zero-value weight, or both. As discussed in ExTensor [16], matrix multiplications are computed by multiplying matching coordinates between the unrolled IFMs and unrolled filters matrices and summing the products at each output pixel. Only products involving coordinates of nonzero elements impact the final result – these tuples of effectual coordinates are called *intersections*. We leverage the temporal progression of dataflows by *pushing intersections towards the beginning* of the chunk-wise computation, which we thoroughly discuss in Section 3. Conversely all pruned coordinates are pushed towards the latter portion of the computation, which allows us to use an early stop mechanism, rather than a costly sparse-skipping mechanism (Section 5).

2.2 Motivation

Although convolutional filters have a relatively lower number of weights, they incur a large amount of activation data reuse, causing activation data movement to dominate energy cost. As shown in MAESTRO [21], the bulk of on-chip and off-chip energy consumption comes from data movement and large on-chip buffers (i.e. > 90%). In fact, most of this energy cost comes from PEs re-fetching activation data from on-chip and off-chip buffers, shown in Fig. 1. The large on-chip buffer characteristic is especially true with 2-way, unstructured sparse accelerators, such as Cambricon-S [41] and SparTen [10], that rely on complex sparse-skipping logic to search and squeeze all ineffectual activations stored in the buffers. The unstructured sparse activation data targeted by these accelerators leads to *irregular access patterns* of input feature maps, which limits reuse opportunities across the parallel PEs. The underlying reason for these inefficient access patterns is simple – although a given *intersect* of the sparse computation is pruned, subsequent computations may still involve the intersect, generating irregularity between subsequent passes. This motivates us to propose a new pruning method called CSP-A. Under CSP-A, pruned intersections pertaining to a specific activation are maintained for subsequent filters. Section 3 details the proposed algorithm.

3 CASCADING STRUCTURED PRUNING

In this section, we give the formulation as well as the intuition of CSP-A. When processing a convolutional layer, activation data is reused across spatial positions or across filters. Because the structure of a kernel does not change during inference, the coordinate of ineffectual weights remains constant within a kernel. However, the same cannot be said *across* filters; subsequent computation passes may need to consider the aforementioned coordinate for the next filters, which causes irregularity in activation data access. Therefore, we design a novel pruning method that pushes all pruned weights towards the later filters, thus preventing



```
M = c_in * k^2                          # Define num. filter rows
arr_w = 4                               # Define HW mapping param.
for(c=0; c<c_out; C+=arr_w)             # Each cascade
  for(r=0; r<M; r++)                    # Each filter row
    if(l1_norm(W[r][c:-1]) < d[n])      # Pruning threshold
      W[r][c:-1] = 0                    # Prune cascade of sub-rows
```

Figure 2: Cascading Structured Pruning. Sub-row example in yellow.

subsequent computation passes from needing to consider coordinates that were pruned in the earlier filters.

3.1 Terminology

Now, we recall and define key notation used in this work. Suppose a layer has c_{out} filters. Each filter has c_{in} channels and kernel size of k^2. Also, let M represent the number of elements in each filter, where $M = c_{in} \times k^2$. As shown in Fig. 2, the weight tensor is flattened into a 2D matrix with the shape of $c_{in}k^2 \times c_{out} = M \times c_{out}$. Each row spans c_{out} filters and represents the same location in each filter. The filters are separated into N groups (called *chunks*) and the portion of the filter row that corresponds to this group of filters is called a *sub-row*. Lastly, we define cascade $C\langle n \rangle$ as the collection of filters spanning chunk n to the last chunk. Notice that the weight tensor can be flattened across any of its dimensions to produce a 2D matrix; we choose these specific dimensions in this case study because they align with popular dataflows, such as output stationary (OS) and weight stationary (WS). This scheme also applies to fully-connected (FC) layers by representing the tensor as $M = f_{in}$ columns and f_{out} rows. All blackened squares in Fig. 2 represent pruned weights.

3.2 Model Training

To achieve the desired sparsity structure, we utilize group LASSO from structured sparsity learning [36] (SSL) at the granularity of a *cascade* rather than individual chunks. Although previous works apply group LASSO at various granularity, our work is the first to apply group LASSO at overlapping regions of the filter by considering the temporal aspect of dataflow. The regularization term is defined as

$$R(W_l) = \lambda \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} || w_{j,[i:N]} ||_2, \tag{1}$$

where $w_{j,[i:N]}$ is the j-th row of the i-th cascade and λ is the regularization strength.

Despite Equation 1 achieving the desired regularization pattern in practice, it also heavily skews the regularization term to penalize later filters. Take Fig. 3 for example. A filter tensor containing four chunks will have four cascades. The largest cascade, $C0$, will apply the regularization term to all four chunks. The next largest cascade, $C1$, does the same to the latter three chunks and so on. Therefore, the total number of times that the regularization term is applied to a filter tensor with N chunks is equal to

$$RT = \sum_{i=0}^{N-1} (N - i) = \frac{N(N+1)}{2}. \tag{2}$$

Here, the last chunk is affected by the regularization terms for all cascades, thus over-penalizing the last chunk. With this skewed regularization penalty, later filters are unfairly penalized, thus risking accuracy degradation. Therefore, we augment Equation 1 to scale the regularization strength of each cascade. For a given cascade, we prefer smaller cascades to penalize its chunks *less* (i.e., regularization term is scaled to its length). Therefore, we set the numerator of the scaling factor for the n-th chunk to be

$$RC_n = N - n. \tag{3}$$

Incorporating the scaling factor to Equation 1, the updated regularization function becomes

$$R(W_l) = \lambda \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (|| w_{j,[i:N]} ||_2 \times \frac{RC_i}{RT}). \tag{4}$$

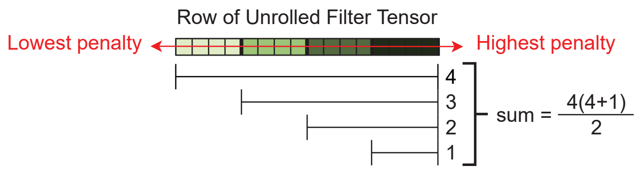


Figure 3: Over-regularization penalty on later chunks.

Pruning and Fine-tuning. Once trained with the regularization term, the model is ready to be pruned to exhibit CSP sparsity characteristic. Motivated by SSL, we prune based on the L_1 norm of the L_2 norm using standard deviation. Specifically, we set

$$\delta_n = STD(w_{[0:M-1],[n:(n+1)]}) \times q, \tag{5}$$

where q is a hyper-parameter for tuning the pruning threshold. Pseudo-code for CSP-A pruning is shown in Fig. 2. Once pruned, the model is then fine-tuned to regain accuracy loss. The overhead of fine-tuning process is 20-50% of a standard training process, e.g., 100 epochs for CIFAR10 and 50 epochs for ImageNet. This overhead is reasonable for the target workload because models only need to be trained once before being employed for inference tasks indefinitely.

3.3 Compressed Representation

Once the filter tensors exhibit CSP sparsity behavior, they can be compressed in a way that enables sequential access of both activation and weight data in a fashion similar to CSR, but while removing indirect addressing resulting from CSR's row and column pointers. Instead of using an array of pointers, we can simply record the number of chunks within each filter row within a *chunk counts* array. The weight matrix can then be stacked such that consecutive unpruned chunks of a filter row are grouped. Because the resulting representation interleaves chunks within each filter row, we call this compressed format *weaved compression*. This format allows us to design a simple early stop mechanism rather than a complex sparsity skipping mechanism in hardware. This format can then be extended to logically group *multiple non-zero filter rows* (T of them) to support different feeding patterns in the dataflow (detailed in Sections 5.3 and 5.4). Fig. 4 depicts the resulting compressed format of the working example in Fig. 2 with two example filter row grouping, T=1 and T=2.

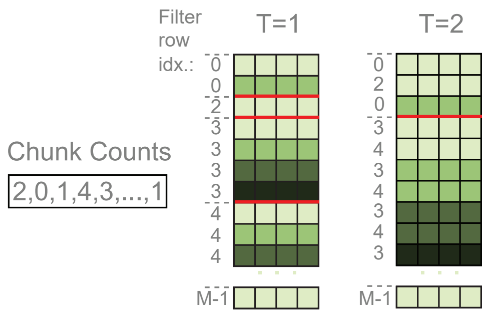


Figure 4: Weaved Compressed Format.

4 DESIRED BEHAVIOR

CSP-A conceptually shrinks the unfolded filter tensor across subsequent passes. Fig. 5(a) displays this behavior using a weight-stationary (WS) PE array. All activations that are fed into the PE array's rows are propagated until it encounters a pruned weight. Once this happens, the activation data is no longer needed by subsequent PEs, illustrated with the red 'X'. Conversely, all previous PEs are guaranteed to require the activation. This scheme constrains the pruning behavior to enable a sequential activation data feeding pattern.

From the above example, it is clear that CSP significantly reduces control complexity of a sparse accelerator by providing a useful constraint to the sparsity structure. In other words, we can now exploit weight sparsity using an *early stop* mechanism rather than a sparse-skipping one. However, it is equally important to identify a suitable parallelized compute scheme to maximally leverage CSP's reduced complexity to improve accelerator performance and efficiency. As an exercise, we present the **Leader-Follower** pipeline, illustrated in Fig. 5(a). Here, the leader PE array processes a pass of activation data on the earliest chunk of the filter tensor. Each PE row corresponds to a filter row, which is distinguished by the varying shades of red in the figure for the first batch of filter rows. The leader array then propagates its activations to the follower array, thus improving the activation data reuse by up to the width of the array (arr_w). Upon encountering a pruned sub-row, the follower array must fetch activations for the next filter row. Although this scheme enables pipelining of PE array structures, it suffers from two major issues. (1) Bandwidth demand of the global activation buffer scales linearly with the number of pipelined PE arrays. (2) Without adequate reorganization of the filter rows, PE stalls (depicted by PEs marked with 'ST') are almost guaranteed due to the load difference across the arrays.

Next, we present the **Serial Cascading** PE array, illustrated in Fig. 5(b). Here, the PE array interleaves computation for all chunks using an output stationary (OS) dataflow. For visual simplicity, we align all weights and activations in the figure, although a real implementation may stagger the inputs. Each PE holds all partial sums corresponding to its filter column within each chunk while individual input activations are held within each PE and continually reused until the first pruned chunk is reached. Although this format cannot benefit from inter-PE-array pipelining like the Follower-Leader configuration, it is *stall-free* and enables **one-time activation read** without moving the partial sums between the PEs. In the next section, we detail the microarchitecture to implement the Serial Cascading PE array.

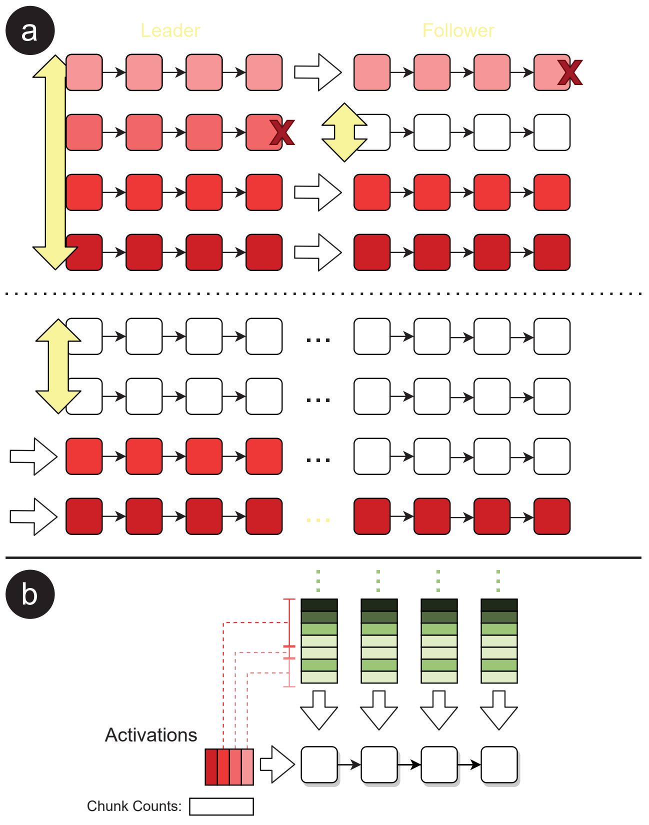


Figure 5: Example CSP-enabled parallel architectures. (a) Leader-Follower. (b) Serial Cascading.

5 CASCADING MICROARCHITECTURE

In this section, we detail the implementation of CSP-H. CSP-H employs the Serial Cascading behavior discussed in Section 4. First, we discuss a novel accumulation buffer composed of circular register bins; this structure holds partial sums for all chunks of a filter row in an output-stationary manner, which is crucial to supporting Serial Cascading. Next, we explore several register bin optimizations and detail the implementation of a PE. Lastly, we tie the target behavior and PE design together with two new dataflows called Input pseudo-Output/-Weight Stationary (IpOS and IpWS, respectively).

5.1 Circular Register Bins

As defined in Section 4, we aim to maximally reuse input activations across *all* chunks. Using conventional dataflows, only one chunk of output channels (i.e., filters) can be mapped onto the PE array at a time. To support multi-chunk mapping, we expand the partial sum accumulation register within each PE. The most straightforward implementation would be to simply add more accumulation registers with a mechanism

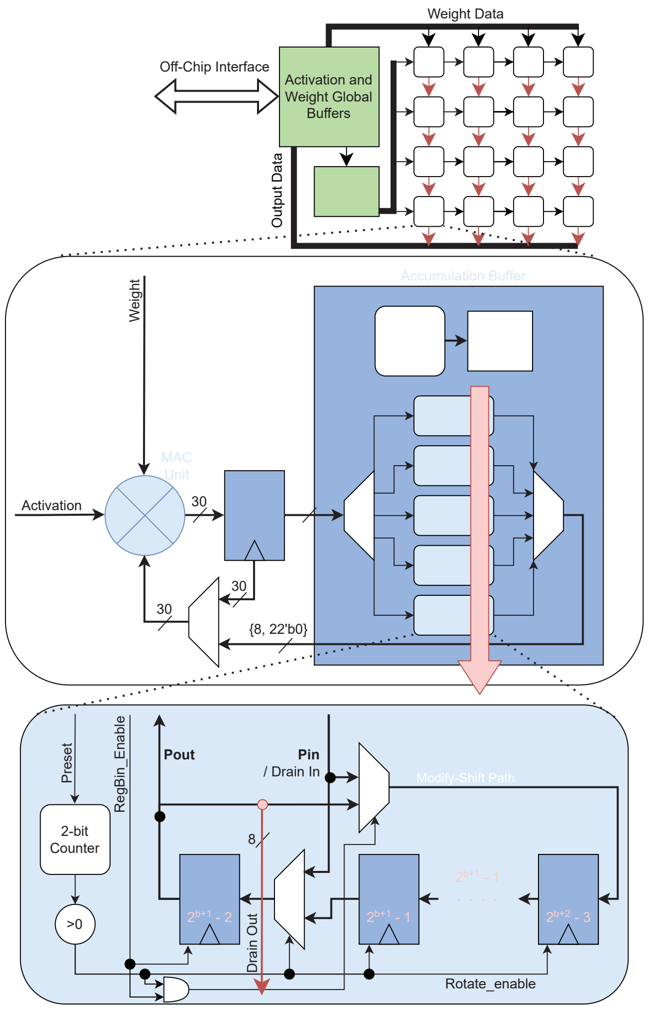


Figure 6: CSP-H, PE, and RegBin microarchitecture.

to arbitrarily select among them (i.e., MUX). However, complexity of MUXes does not scale well. Coupled with the fact that read-modify-write (RMW) of partial sums is on the critical path of single-cycle MAC operations, depending on a high-latency arbitrary switching mechanism would degrade performance of a PE. Instead, we require a scalable solution that would support as many concurrent chunks as possible without impacting PE performance.

Expanded Accumulation Buffer. Notice from Fig. 5(b) that the the defined chunk-wise mapping is sequential, which reflects the access pattern for each accumulation register. We leverage this sequential RMW access pattern by implementing a *circular* accumulation buffer. As the MACs for each chunk are processed, partial sums are propagated through the circular buffer and only the head of the buffer needs to be accessed for accumulation. In order to support arbitrary length pruned filter rows, the circular buffer needs to be separated into segments called *register bins* (RegBin).

Fig. 6 depicts the overall accelerator, accumulation buffer, and RegBin microarchitecture. Individual RegBins are implemented as circular buffers that only propagate partial sums upon receiving an enable signal and encountering a requested chunk size greater than a specified threshold. FSM-based control logic iterates the current chunk index to enable specific RegBins for RMW. Each RegBin must be sized accordingly to support stall-free computation. Consider Fig. 7. Here, the highlighted registers are the ones accessed for one MAC operation while red arrows signify a rotate of the RegBin. If the chunk size for a given cascade only reaches the head of a RegBin (i.e., cycle 1), the partial sum data present at the head can be directly accessed without triggering the rotate logic of the RegBin; this saves energy related to switching power by avoiding needlessly propagating data through the RegBin. However, in the worst case, the chunk size of the cascade reaches only the second register in the RegBin (i.e., cycle 4). In this case, the RegBin must complete a full rotation on-time before computations of the next filter row require partial sum data from that RegBin (i.e., cycle 7). As a result, we size consecutive RegBins exponentially. Specifically, given a RegBin RB_b, $b \in \mathbb{N}_0$, we determine its length as

$$len(RB_b) = 2^{b+1}. \qquad (6)$$

In total, the proposed accumulation buffer contains five RegBins, allowing it to span $2 + 2^2 + 2^3 + 2^4 + 2^5 = 62$ chunks. Assuming an array width of $arr_w = 32$, this is equivalent to supporting up to 1,984 concurrent filters. Most modern DNN layers contain at most 1,024 filters, so the proposed design comfortably supports the common case.

Internal to each RegBin is counter-based FSM that leverages the sequential access patterns to minimize complexity. A decrementing counter is preset once the FSM receives *chunk count* $>$ *rot_tresh*, where the rotate threshold for a given RegBin b is specified by

$$rot\_thresh_b = \begin{cases} 0 & \text{if } b = 0 \\ 2^{b+1} & \text{if } b \geq 1 \end{cases}. \qquad (7)$$

While the value of the counter is nonzero, the RegBin continues to circulate, even when the RegBin is not selected by the accumulation buffer for RMW; this allows the RegBin to reset on-time while waiting for the next filter row. If the RegBin is enabled *and* rotating, the RMW changes to a read-modify-shift (RMS), where the last register is directly written to during the circular shift, thus maintaining correctness without adding another stage to the RMW operation.

Flushing Accumulation Buffer. Now that we have covered the loading and compute mechanisms of the accumulation buffer, it is equally important to discuss the *offloading* mechanism. Efficiently offloading the accumulated activations is important to minimize the stall between passes. In

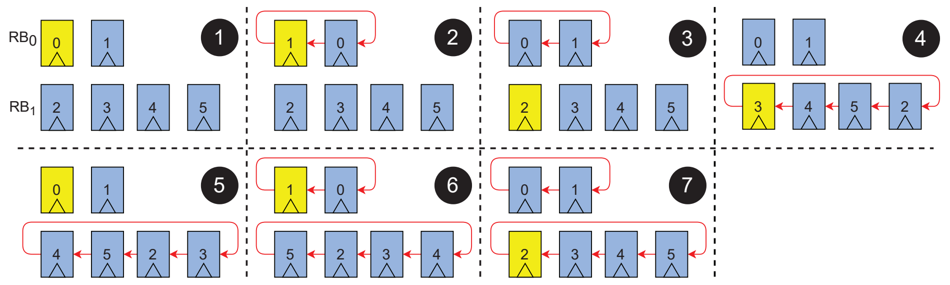


Figure 7: Running example of circular RegBin stall-free access.

a conventional PE array architecture, the accumulated activations are propagated between PEs towards the global buffer designated to hold the outputs; this is easy because at most one output is held within each PE, thus limiting the bandwidth requirement of the global buffer to the dimension of the PE array. In contrast, the proposed expanded accumulation buffer may store up to 62 output activations per PE. One direct approach to flushing these buffers is to place all entries onto a wide bus, thus maintaining the single cycle flush of the conventional architecture. However, the bandwidth demand of the output global buffer would increase by 62×, which would severely limit the scalability of the PE array. On the other extreme, we could take a true serial approach and flush the accumulation buffers one entry at a time; but, this could stall the next pass by a number of cycles equal to the size of the largest dirty RegBin. Instead, we avoid both of these penalties through again leveraging the sequential RMW access pattern by serially flushing all B RegBins at the same time. In other words, each RegBin flushes its outputs serially, so the flush output of a RegBin is only 8-bit. All RegBins flush at the same time with a total drain bus bandwidth of $(8 \times B)$-bit. RegBins with the same ID (e.g., $b = 0, 1, \ldots$) in subsequent PEs buffer the outputs of the previous PE while simultaneously draining its own. Here, the flushing procedure would only incur a two cycle penalty (i.e., equal to the size of the first RegBin) while allowing the next pass' computation to overlap with the flushing.

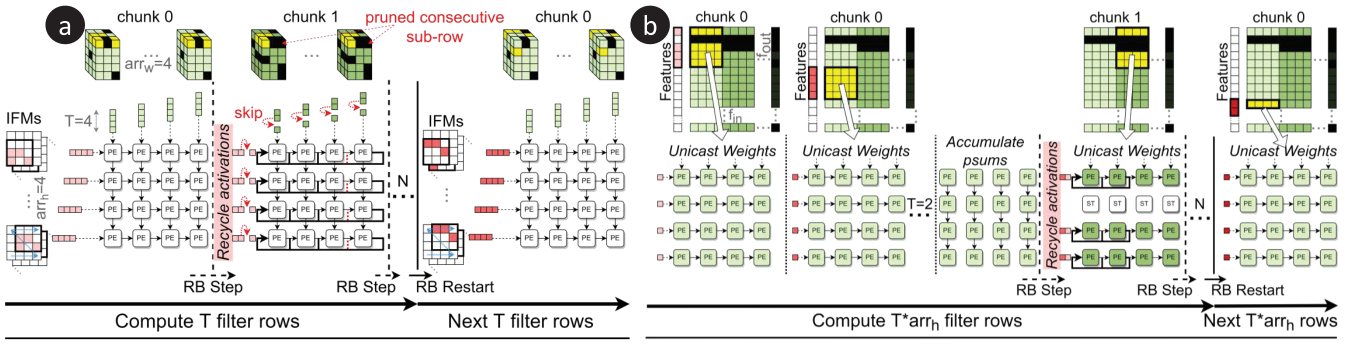
5.2 Register Bin Optimizations

The proposed circular RegBins enable efficient activation and partial sum reuse, but are prone to significantly increasing area and power consumption within each PE. Although state-of-the-art edge DNN accelerators are able to leverage 8-bit input activation and weight quantization, partial sums still require significantly higher precision to maintain high accuracy. As discussed Sze et al. [33], the required precision for partial sums is typically $26 - 32$ bits for popular DNNs. This precision requirement significantly increases the area and power consumption of the register-based accumulation buffer and necessitates some hardware-algorithm optimizations.

Periodic Partial Sum Truncation. Truncating the partial sums will significantly reduce the size of the accumulation buffer, thus reducing area and power cost. However, naïvely reducing partial sum precision may incur significant accumulation error and degrade model accuracy. To address this we introduce an *Intermediate Partial Sum Register* (IR) between the MAC unit and RegBin, shown in Fig. 6. In conjunction, we design two new dataflows (discussed in Section 5.3 and 5.4) so that each chunk can accumulate up to T MACs at full precision prior to truncating the partial sum. Here, T is the number of filter rows concurrently processed by the accelerator and grouped by weaved compression, and is typically set to $T = arr_w$ to align with the PE array dimensions. *This approach recovers almost all accuracy loss from partial sum truncation while significantly reducing area and power cost of each PE.* Section 7.2 details this accuracy-power trade-off.

Lower Switching Activity. By introducing the intermediate partial sum buffer, the circular RegBins are now given ample time to propagate partial sums in between chunks. Rather than using additional time to further scale the RegBins beyond the limit set by Equation 6, we maintain the current configuration and allow the controller FSMs to update once every T cycles. This significantly reduces switching activity, thus reducing dynamic power.

Clock Gating. Lastly, we apply a simple yet effective technique to further reduce power of the accumulation buffer. Most filter rows of each layer do not require accessing the costly RB_4 due to the high chunk pruning rate. In fact, later RegBins are accessed at a lower rate due because CSP-A penalizes the later non-zero chunks more. Due to the exponential scaling of the RegBins, RB_4 consumes almost as much power as the previous RegBins combined. Thus, we can further improve power efficiency by applying *clock gating* to unused RegBins at a **per-pass** granularity. The power efficiency is further improved using this technique under higher CSP sparsity rates. We discuss results on RegBin access frequencies in Section 7.6.



```
for(a=0; a<(OFM_h*OFM_w/arr_h); a++)        # Each group of OFM coords.
  for(Tr=0; Tr<(M/T); Tr++)                 # Each grouped T filter rows
    chunks = max(chunk_cnts[Tr*T:(Tr+1)*T]) # Total non-zero chunks in Tr
      for(n=0; n<chunks; n++)               # Each non-zero chunk
        # ---- Parallelized by PE array ---- #
          for_all(w=0; w<arr_w; w++)        # Each PE array col
            for_all(h=0; h<arr_h; h++)      # Each PE array row
              for(sr=0; sr<T; sr++)         # Each sub-row
                if(w[n*N+w, Tr*T+sr]==0)    # Skip pruned sub-rows
                  continue
                # OFM has shape [c_out, (OFM_h*OFM_w)]
                # w has shape [c_out, M]
                # IFM unrolled to shape [OFM_h*OFM_w, c_l*k*k]
                OFM[n*N+w, a*arr_h+h] += w[n*N+w, Tr*T+sr] \
                                       * IFM[a*arr_h+h, Tr*T+sr]
```

```
for(a=0; a<(f_in/(arr_h*T)); a++)          # Each group of in_features
  for(Tr=0; Tr<T; Tr++)                    # Each grouped T filter rows
    po = a*arr_h*T+Tr*T
    chunks = max(chunk_cnts[po:(po+1)*T])  # Total non-zero chunks in Tr
    for(n=0; n<chunks; n++)                # Each non-zero chunk
      for(t=0; t<T; t++)                   # Each psum set before truncation
        po2 = a*arr_h*T+t*T
        # ---- Parallelized by PE array ----#
        for_all(w=0; w<arr_w; w++)         # Each PE array col
          for_all(sr=0; sr<arr_h; sr++)    # Each sub-row
            # OFM has shape [f_out]
            # w has shape [f_out, f_in]
            # IFM has to shape [f_in]
            OFM[n*N+w] += w[n*N+w, po2+sr] * IFM[po2+sr]
        accumulate_psums()                 # Accumulate psums within cols
```

Figure 8: (a) Input pseudo-Output Stationary Dataflow. (b) Input pseudo-Weight Stationary Dataflow.

5.3 Dataflow: Input pseudo-Output Stationary

The combination of the *Serial Cascading behavior* and *Intermediate Partial Sum Truncation* informs the design of a new dataflow called *Input pseudo-Output Stationary* (IpOS). One computation loop of IpOS is illustrated in Fig. 8(a). Fig. 8 assumes the same sparsity pattern as Fig. 2 and each time step shows an explicit mapping between the yellow-highlighted filter elements and the input features. For the first T cycles of a group of filter rows (i.e., Tr), activation and weight data are fed into the PE array under the standard OS dataflow. After T cycles, partial sums are truncated and stored in the RegBins. The PE FSMs are then updated to prepare for the next chunk (i.e., 'RB Step'). Meanwhile, $T-|P_{Tr,n}|$ activations are recycled within their respective rows. Here, we consider $|P_{Tr,n}|$ to be the number of subrows skipped during the n-th chunk of the Tr-th group of filter rows. Because of the sequential access of activation data, the global controller can decrement the *chunk count* for each filter row to determine which PE to recycle activation data from. After $T-|P_{Tr,n}|$ cycles, partial sums of the next chunk are computed, and the process repeats until this set of partial sums of all chunks are computed. Specifically, this process of recycling activation data continues until the number of non-zero chunks, signified by *chunk count*, for all T filters rows is reached. Upon reaching *chunk count*, RegBin counters restart to operate on RB_0 in preparation for the next set of T filter rows (i.e., 'RB Restart'). Once all filter rows are processed, this concludes one iteration of IpOS. Each iteration targets distinct pixels of the OFM, similar to the conventional OS dataflow, and all iterations are sequentially computed for a layer.

5.4 Dataflow: Input pseudo-Weight Stationary

For any DNN layer that is efficiently supported, IpOS is the preferred dataflow because utilization of all PEs is not affected by the differences in sparsity across the sub-rows. However, not all layer types can be efficiently mapped onto an OS-like dataflow. An example are fully-connected (FC) layers, which make up the majority of computations in popular DNNs like the Transformer [35]. FC layers have no concept of spatial dimension, which would reduce the utilization of the PE array using IpOS dataflow to only a single row. This issue can be avoided by unrolling the filters directly onto the PEs under a WS-like dataflow. Enter the *Input pseudo-Weight Stationary* (IpWS) dataflow, illustrated in Fig. 8(b). Unlike IpOS, IpWS requires the weights to be unicasted to the PEs. Although this procedure has a larger bandwidth requirement for the filter GLB, we can coalesce the memory accesses to avoid introducing additional ports to the GLB; this can be done by reordering the weights in memory ahead of time. The rest of the dataflow is similar to IpOS with some key differences. Each iteration is now equivalent to the number of input channels divided by the product of arr_h and T. Another key difference is that we must include another step, *accumulate_psums*(), to ensure that the minimum number of accumulations between each partial sum truncation is met. This can be done by accumulating alternating rows of partial sums within each column, introducing a single cycle delay between each set of T sub-rows. The last difference is that IpWS has the potential to suffer from PE under-utilization similar to the Leader-Follower pipeline shown in Section 4. To minimize this underutilization, we greedily reorder the filter rows from least to most sparse; this increases the likelihood that sub-rows for each chunk share the same sparsity.

6 EVALUATION METHODOLOGY

To evaluate CSP, we analyze the impact of its components, CSP-A and CSP-H, both separately and combined. Firstly, we validate CSP-A's impact on model performance and sparsity. Then, we analyze the optimal truncation period for negligible accuracy degradation; this informs the appropriate values for both arr_w and T. Then, we compare CSP-H to recent relevant works and investigate our key sources of energy savings.

6.1 Model Performance Validation

To validate CSP-A's impact on model performance and sparsity, several representative models and datasets are selected to span a wide range of DNN layer types and complexity. Specifically, the selected models are AlexNet [19], VGGNet [32], ResNet [13], Inception [34], and Transformer [35]. AlexNet is a classic Convolutional Neural Network (CNN) that incorporates a large kernel size, 11×11, for the first layer. VGGNet is another classic CNN with repeating, small 3×3 convolutional kernels. ResNet is most known for its bottleneck layers and residual connections, which are layers consisting of 1×1 kernels that connect nonconsecutive layers. Inception is the most complex model of the chosen CNNs; it employs branches, aggressive residual connections, and various kernel shapes. The Transformer employs a collection of multi-headed attention and feed-forward layers, which can be conceptualized as FC layers. In terms of datasets, we choose CIFAR-10 [18], ImageNet [7], and WMT'16 German to English Translation. While image classification is scored using the percentage of correct classifications, translation quality is scored using BLEU [26], which measures the similarity between machine-translated text to a set of verified translations.

The selected CNN models are trained using Stochastic Gradient Descent (SGD), nesterov momentum of 0.9, initial learning rate of 0.1, and with cosine annealing. CIFAR-10 models are trained from scratch for 200 epochs with regularization strength of 0.01 and 0.0005 weight decay. The models are then pruned with a pruning threshold multiplier of $q = 0.75$ and fine-tuned for 100 epochs. On ImageNet, we use a pretrained model and retrain with regularization for 60 epochs with regularization strength of 0.0001, and using the ADAM optimizer. Fine-tuning phase lasts for 50 epochs. All other hyper-parameters are consistent with CIFAR-10. On WMT, the Transformer is trained from scratch with the same hyper-parameters from the original paper [35], regularization strength of 1, and for 400 epochs; it is then fine-tuned for 100 epochs. The chunk size used for CSP-A regularization and pruning is set to 32 for all models.

6.2 Architecture Modelling and Comparison

Baseline Accelerators. DNN inference acceleration is a well-studied area and encompasses a rich diversity of approaches. For this reason, we select several accelerators that target efficient DNN inference from different angles. DianNao [3] is one of the earliest inference accelerators that leverages 3-level memory and targets the dominant MAC operation for high temporal data reuse and low complexity. It targets dense networks and is therefore a good baseline to observe the impact of sparsity-aware approaches. Cambricon-X [38] holds sparse CNN weights in a compressed format and applies efficient indexing to reduce activation data movement cost and allowing it to operate as a 1-way sparse accelerator. SparTen [10] implements 2-way compressed format using bit-masks in addition to employing offline and online load balancing schemes for improved utilization and lower data movement with significantly lower compute cycles. Cambricon-S [41] combines structured pruning, local quantization, and 1-way sparse computation skipping, making it a strong candidate to compare against our structured sparse approach. Each of these works have a common goal of accelerating DNN inference, which places end-to-end latency and energy efficiency as first-class priorities in our main analysis.

Architecture Modelling. All accelerators are modeled using cycle-accurate simulation to obtain cycle count and data movement traces on CSP-A-trained models. To simulate CSP-H, we modify SCALE-Sim [30] and implement the IpOS and IpWS dataflows, PE architecture, and weaved compressed format. Specifically, after training the models using CSP-A, we export the filters and feature maps of each layer, and apply weaved compression to the weights. The layer-wise data are then fed into the modified simulator to obtain cycle-accurate measurements. For DianNao, we use the Timeloop [27] simulator. Because this work does not target unstructured sparsity, we structurally prune entire ineffectual filters from the models to enhance their baseline results without modifying their implementation. Cambricon-X, Cambricon-S and SparTen do not have open-source simulation tool that we can leverage. Instead, we closely follow the designs discussed in these works to implement in-house

Table 1: Hardware Parameters.

Accelerator	Off-Chip Mem.	Global Mem. (pJ)	Number PEs	Mem./PE	MACs/PE	B/MAC
DianNao		NB: 36 KB (1.51 rd) SB: 36 KB (2.98 wt)	64	2 KB	16	0.195 KB
Cambricon-X		NBin: 36 KB (1.51 rd) NBout: 36 KB (2.98 wt)	64	2 KB	16	0.195 KB
SparTen	DDR3	N/A	1024	0.76 KB	1	0.778 KB
Cambricon-S	1 GB 64-bit bus 800 MHz	NBin: 32 KB (1.44 rd) NBout: 32 KB (2.64 wt) SIB: 8 KB (1.01 rd)	64	32 KB	16	2.070 KB
Ours	Per-Byte Energy: 766 pJ rd 780 pJ wt	InAct: 2 KB (0.84 rd) Wgt: 50 KB (1.76 rd) OutAct: 20 KB (2.83 wt)	1024	A&W: 2 B IR: 4 B Accum.: 62 B	1	**0.137 KB**

simulators. Lastly, because Cambricon-S is a cooperative algorithm-architecture approach, all models are re-trained using Cambricon-S' method. Performance and power consumption are scaled to match the PE count and global memory constraints presented in TABLE 1.

TABLE 1 presents the configurations for all accelerators used in this work. Two important parameters that directly influence computation throughput and data movement are **MAC unit count** and **global buffer size**. Therefore, for fair comparison, we constrain all accelerators to use 1024 parallel MAC units and 72 *KB* of global buffer space. Note that local buffers (i.e., exclusive to each PE) are tied to the specific architecture, so modifying this value may unfairly disadvantage the baselines. Instead, we compare the ratio of **total buffer-per-MAC** and consider it in our overall analysis. Less buffering per MAC is more area- and memory-efficient.

All accelerators are scaled to a clock frequency of 300 *MHz*, 65 *nm* technology node, and 8-bit computation for comparability. CSP-H is synthesized for area and power estimation using Synopsys Design Compiler and TSMC 65 *nm* library. Clock gating is implemented via latches and is included in the synthesized results. For high fidelity dynamic power estimation of *our work*, we simulate a sample workload in ModelSim and extract the switching activity (.saif) file. Because replicating this method for all baseline accelerators would require significant engineering effort, we instead conservatively estimate dynamic power of *baseline accelerators* with a per-MAC consumption of 0.081 pJ; this is extracted by synthesizing a MAC unit and averaging its dynamic power. Global buffers and off-chip memory are modelled using CACTI as SRAM and DRAM, respectively. Global and off-chip memory energy is computed by multiplying the number of accesses by the unit-energy cost of reads or writes. TABLE 1 reports the per-Byte access energy of each memory module. On-chip leakage power of the memory is also considered, but off-chip leakage power is not considered because it may be shared by other system components. Lastly, energy efficiency is computed as the number of inferences per unit energy.

7 RESULTS

7.1 CSP-A Evaluation

Accuracy and parameter sparsity of all trained models are shown in TABLE 2. Here, 'Base Acc.' refers to the accuracy of the pretrained model; meanwhile, 'Final Acc.' is the final model accuracy after pruning and fine-tuning. Also, 'Param. Spar.' refers to the percentage of zero-valued weights of the targeted layers. To separate the efficacy of our algorithm and architecture on convolutional and FC layers, we target only the convolutional layers for the CNN models and the FC layers for the Transformer. On CIFAR-10, we achieved over 70% parameter sparsity on all models with accuracy loss under 0.05%. InceptionV3 is able to be pruned particularly well (over 95% sparsity). Cambricon-S' pruning method achieves higher accuracy with similar sparsity on CIFAR-10 models because it applies pruning at a finer granularity than CSP-A. On ImageNet, we achieved a lower sparsity of at least 49% while keeping accuracy degradation to below 0.5%. The lower sparsity here is attributed to the increased complexity of the ImageNet dataset. Lastly, our method is able to produce impressive results on the Transformer, with a sparsity of 84% and *improvement* to the BLEU score by 9.6, an improvement of about 38%. In comparison, transformers.zip [4], a method that relies on iterative magnitude pruning, is only able to prune 30% with negligible accuracy loss because it does not utilize parameter regularization during training. To better understand the source of this performance improvement, we include additional results using SSL [36] across the output-channel dimension, unstructured L_2 regularization, and CSP-A with varying chunk size, all with similar sparsity strength and pruning threshold. As shown in TABLE 2, SSL across the output-channel dimension (which is equivalent to CSP-A with chunk size set to the number of output channels) significantly degrades the BLEU score with comparable sparsity to Ours. Simply applying L_2 regularization for unstructured sparsity gives significantly less accuracy degradation. Meanwhile, BLEU score gains from varying the chunk size parameter between 8 to 128 (i.e., Ours-8 to Ours-128 in the table) suggests that the 'sweet spot' for pruning the models attention heads across the attention heads dimension is related to the key dimension, $d_K = 64$. When setting the chunk size to d_K, entire sub-rows of heads are pruned. These results suggest that there is an interaction between

Table 2: Model accuracy and sparsity of CSP-A.

Dataset	Model	Method	Base Acc. (%/BLEU)	Final Acc. (%/BLEU)	Δ*Acc.* (%/BLEU)	Param. Spar. (%)
CIFAR-10	VGG-16	[41]	92.08	93.41	**+1.33**	87.5
		Ours	92.08	92.39	+0.31	**87.58**
	ResNet-50	[41]	93.57	94.34	**+0.77**	73.66
		Ours	93.57	93.54	-0.03	**73.91**
	InceptionV3	[41]	93.79	93.93	**+0.14**	93.76
		Ours	93.79	93.75	-0.04	**95.56**
ImageNet	AlexNet	[36]	57.37	57.47	+0.10	33.38
		[41]	56.55	56.84	-0.29	**74.79**
		Ours	56.55	56.72	**+0.17**	49.02
	VGG-16	[41]	71.59	71.27	**-0.32**	64.45
		Ours	71.59	71.24	-0.35	**73.72**
WMT'16 (DE-EN)	Transformer	[4]	28.09	27.65	-0.44	30.00
		[36]	25.61	21.38	-4.23	72.67
		[41]	25.61	23.64	-1.97	54.68
		l2-reg-flat	25.61	23.26	-0.35	72.46
		Ours-8	25.61	29.37	+3.76	79.48
		Ours-16	25.61	35.91	**+10.30**	83.00
		Ours-32	25.61	35.28	+9.67	**84.39**
		Ours-64	25.61	25.55	-0.06	82.35
		Ours-128	25.61	30.12	+4.51	55.87

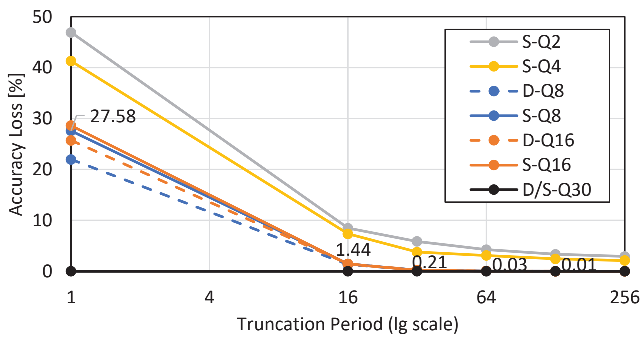


Figure 9: Periodic Partial Sum Truncation. 'D':dense, 'S':sparse, followed by precision of the RegBins.

the pruning granularity and the key dimension of the multi-headed attention layers, where enforcing CSP-A sparsity behavior can influence the performance of the attention.

7.2 Periodic Truncation

Whenever a microarchitectural change induces error in computation, it is important to carefully examine and tune the magnitude of accuracy loss. By truncating the partial sum registers, we introduce a energy-accuracy trade-off. Fig. 9 illustrates the accuracy curve while Fig. 12 illustrates the energy consumed by the PE array for the different partial sum register configurations. Accuracy loss is normalized to the full-precision scenario and is computed on ResNet-50. As expected, directly reducing RegBin precision from 30- to 8-bit (i.e., $T = 1$) greatly reduces energy consumption of the PE (i.e., roughly 2.8×), but induces a significant accuracy loss of 28%. By incorporating an IR to increase the period in which partial sums are truncated, we can recover most of the accuracy loss while simultaneously reducing energy consumption of the RegBins. Assuming a truncation period of 32 (which corresponds to $T = arr_w$), accuracy loss is reduced to 0.21%. Also, by reducing the frequency that RegBins are accessed, overall PE power is reduced by 5.3×.

Now, we discuss the minimum truncation period for negligible accuracy loss. At a truncation period of $T = 1$, it is assumed that the IR is completely bypassed and RegBins are directly accessed each cycle. Accuracy loss is normalized to the 30-bit precision scenario. As shown in this figure, accuracy loss quickly drops as truncation period increases. At the nominal truncation period of 32, accuracy loss is reduced to 0.21%. We can further decrease the accuracy loss by increasing truncation period. However, raising the truncation period to larger than arr_w will require buffering of the additional activation data to maintain IpOS or IpWS dataflows. Instead, by introducing one additional activation buffer for each PE, accuracy loss can be further decreased to 0.03%

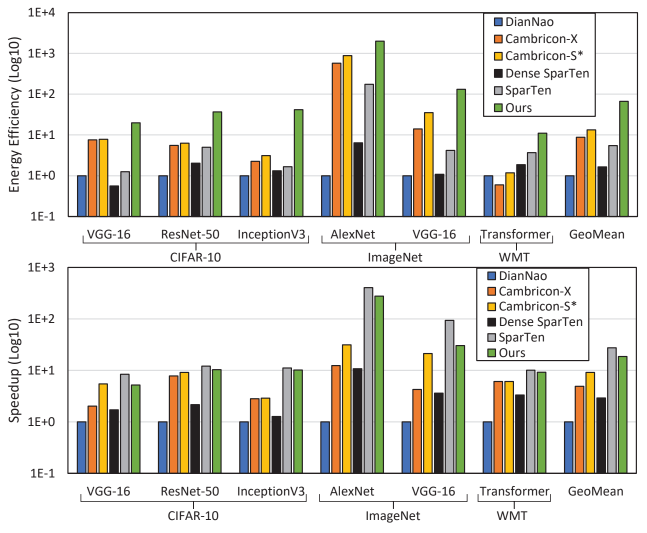


Figure 10: Overall Results. *Uses separately trained model.

at the cost of slightly increased area and complexity. Accuracy loss can also be mitigated by incorporating partial sum truncation inside the model training loop. Because the current accuracy loss is negligible (i.e., 0.03%), we leave this algorithmic approach for future work.

7.3 Accelerator Performance

Using the observations from Section 7.2, we choose $arr_w = 32$ with two activation input registers for a truncation period of $T = 64$. Overall results of the experiment are shown in Fig. 10. This figure displays the energy efficiency and speedup of all accelerators on the evaluated models, normalized to DianNao. We also include results of SparTen assuming dense execution as an additional baseline. On a majority of the models, CSP-H significantly improves overall energy efficiency over all baselines. Overall improvement on Transformer is lower because its FC layers tend to be *weight data* dominant rather than activation data dominant. Regardless of the number of activation data re-fetches avoided, all designs must incur the cost of reading the Transformer's large number of weights. By minimizing partial sum and activation data movement with the proposed IpOS and IpWS dataflows, CSP-H achieves an average 7.7× and 15× improvement in energy efficiency compared to Cambricon-X and SparTen, respectively, and on average 5× improvement compared to Cambricon-S. CSP-H guarantees one-time activation data access and limits partial sum movement to within its PEs; CSP-H also reuses activation data across all PEs and does not rely on large input buffers, thus improving overall energy efficiency. In contrast, SparTen's clusters are specifically designed to work independently and operate on unique slices of the output map; this results in each of the (32) clusters

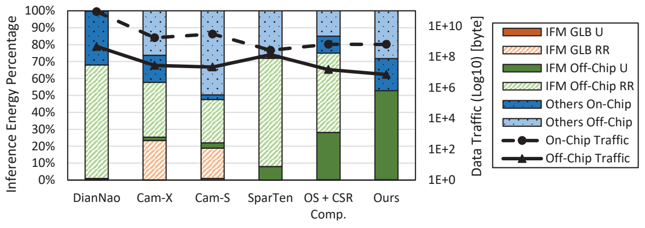
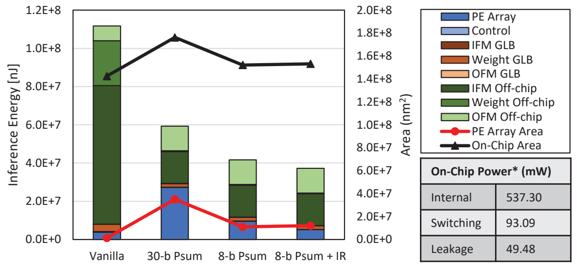


Figure 11: Energy Breakdown and Data Traffic.



On-Chip Power* (mW)	
Internal	537.30
Switching	93.09
Leakage	49.48

Figure 12: Energy, Power, and Area. *Power excludes SRAM.

redundantly accessing overlapped data from the input maps, which eclipses SparTen's nominal 50% activation movement savings achieved by compression.

In terms of speedup, CSP-H delivers comparable or superior performance to most of the state-of-the-art baselines, but has slowdowns of 1.4× compared to SparTen. SparTen achieves this speedup by emphasizing load balance within its independent clusters while skipping *all* ineffectual computations from zero-valued inputs and weights. CSP-H, on the other hand, only skips computations with zero-valued weights and relies on a simple greedy load-balancing scheme for its IpWS dataflow. Despite being 1.4× slower than SparTen, CSP-H experiences a 15× energy efficiency benefit. Because CSP's primary goal is to tackle energy inefficiency by removing redundant data accesses, we find the trade-off to be significantly in our favor. Additionally, the huge gap in $Buffer/MAC$ (shown in TABLE 1) between CSP-H and SparTen shows that we have extra budget to potentially hold more activation data by increasing the size of the activation GLB. Future works can exploit this extra capacity budget by pre-fetching more activation data and employing a sparse activation skipping mechanism on top of CSP-A to bridge the performance gap.

7.4 Activation Re-Fetch Energy Comparison

To further highlight the key sources of energy consumption for each of the accelerators, we isolate the energy consumed by refetching activation data (i.e., IFM RR) during a single VGG-16 inference in Fig. 11. Because DianNao does not compress the filter tensors, all activation data must be re-fetched for consecutive intersections in the convolution operation, leading to over 65% off-chip re-fetch energy. SparTen's inability to reuse activation data across its clusters due to its irregular IFM access pattern and independent clusters causes it to spend nearly 60% of its inference energy re-fetching off-chip activation data. Cambricon-X and Cambricon-S perform significantly better in terms of off-chip activation data re-fetch energy, but their buffer controller, BCFU/NSM, consumes a significant amount of energy locating and re-transporting non-zero activation data for unique intersections pertaining to individual PEs. To expose the benefits of the proposed IpOS dataflow, we include another data point (i.e., OS + CSR Compression) that applies standard CSR compression on top of an OS dataflow given the same hardware configurations; here, the amount of off-chip activation data is still significant (> 40%). As shown in Fig. 11, CSP-H completely removes all activation re-fetches, causing *unique* off-chip activation data fetches (i.e., IFM U) to dominate the overall energy consumption, which is an unavoidable cost regardless of accelerator design.

7.5 Component Breakdown

Fig. 12 shows the PE array area and end-to-end energy breakdown of one inference run on ResNet-50. 'Vanilla' refers to a conventional OS dataflow accelerator; here, it is clear to see two main characteristics: (1) off-chip data movement dominates the energy cost and (2) movement of activation data (specifically from the IFMs) composes a majority of off-chip energy cost. The other PE configurations explored in this figure pertain the various iterations of CSP-H discussed in this work. By leveraging CSP-A and proposed dataflows, all iterations of CSP-H significantly reduce off-chip data movement, and thus address the largest dissipator of energy. '30-bit Psum' configuration trades lower off-chip energy for high PE array energy due to the large power cost of the large partial sum buffers. Meanwhile the '8-bit Psum' configurations decrease the PE array's area and power by roughly 3×. These results are consistent with Section 7.2.

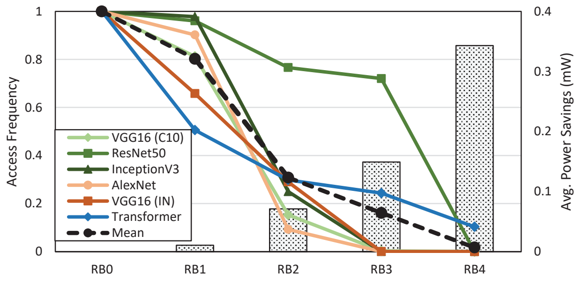


Figure 13: Access Frequency and Savings of RegBins.

7.6 RegBin access frequency

Fig. 13 illustrates the access frequency of each RegBin during an inference of all representative models, as well as the average power savings resulting from clock gating. As expected RB_0 is used 100% of the time because it is the first RegBin that must be accessed to accumulate the intersections from each filter row. Meanwhile RB_4 is activated less than 11% of the time for all runs. For models that achieve high prune rate, the frequency of RB_4 access drops to zero, signifying that RB_4 is unused. We further lower power of each PE by clock gating unused RegBins at a per-pass granularity and include it in the synthesized results. This produces an average power savings of $0.574mW$ for each PE, reducing the power consumption of each PE by 46%.

8 RELATED WORKS

Sparse DNN inference and structured pruning are well-studied areas with a rich collection of approaches. Recent works can be categorized by the sparsity **granularity** and **type** that they target. In terms of *granularity*, accelerators can exploit *bit-wise* sparsity via bit-serial computation [1, 31], *unstructured element-wise* sparsity of either activations or weights [2, 5, 6, 8, 11, 20, 29, 38], or *structured sparsity* via a co-designed pruning algorithm [17, 37, 41]. BitPruner [39] applies structured bit-wise pruning to benefit bit-serial architectures. Our approach also falls under the structured pruning category, but with one key distinction: the pruning framework is closely designed with the dataflow. In fact, CSP-A is orthogonal to BitPruner in that CSP-A can enhance BitPruner to achieve a cascading sparsity pattern, which can further improve the data reuse capabilities of the accelerator. To *train* the structured-sparse DNNs, algorithm-only methods like group LASSO [36], group-wise brain damage [22], and Mao et al. [25] optimize the trade-off between sparsity and accuracy while generating sparsity patterns that map well to a DNN accelerator. Unlike CSP, they do not consider the temporal aspect of the architecture's dataflow.

In terms of sparsity *type*, sparse DNN accelerators can target sparsity of the activations or weights (i.e., 1-way sparsity), or both (i.e., 2-way sparsity). Recent works that target 2-way sparsity are able to extract higher performance and energy efficiency [10, 16, 28, 40], but they rely solely on sparsity skipping to achieve higher performance proportional to tensor sparsity. CSP avoids sparsity skipping logic and instead incorporates an early stop mechanism based on the induced sparsity pattern. Sanger [24] is another 2-way sparse approach that targets the *dynamic* structures of attention-based models (i.e., Logit and Attend operators); it dynamically applies fine-grained structure pruning with a dataflow that is well suited for Logit and Attend operators. CSP-A is not a dynamic pruning method and instead targets the *static* elements of the attention layers, thus treating the Logit and Attend operators as dense.

9 CONCLUSION

This paper presents Cascading Structured Pruning, which flexibly prunes filter tensors in a structured manner while enabling contiguous activation data access pattern. Contiguous activation data creates the opportunity for compression schemes and dataflows with sequential activation access for improved data reuse. From the algorithm side, CSP-A induces this *cascading* pruned behavior by enforcing pruned weights that would be mapped to the same coordinates across subsequent timesteps of the inference dataflow. Once pruned, model sparsity ranges from $49-96\%$ with an accuracy degradation of less than 0.5% after pruning and fine-tuning. The hardware, CSP-H, then exploits the cascading pruned behavior with two novel dataflows to enforce one-time activation data access and limit movement of partial sums to within each PE. We then introduce a novel PE design that periodically truncates partial sums for minimal accuracy loss with lower power and area footprint than the straightforward approach. Our results show that CSP-H applied to CSP-A-trained models improves energy efficiency compared to state-of-the-art accelerators, all with comparable or superior speedup under most evaluations.

10 ACKNOWLEDGEMENTS

This work was supported in part by NSF 2112562, 1955246, and 1937435. This work was also supported by ARO W911NF-19-2-0107.

REFERENCES

[1] J. Albericio, A. Delmas, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*. ACM, 2017, pp. 382–394.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 1–13.

[3] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–284.

[4] R. Cheong and R. Daniel, "transformers. zip: Compressing transformers with pruning and quantization," *Technical report, Stanford University*, 2019.

[5] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 189–202.

[6] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "TIE: Energy-efficient tensor train-based inference engine for deep neural network," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 264–277.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[8] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," ser.

MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 395–408.

[9] X. Ding, G. Ding, Y. Guo, J. Han, and C. Yan, "Approximated oracle filter pruning for destructive CNN width optimization," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 97. PMLR, 09–15 Jun 2019, pp. 1607–1616.

[10] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 151–165.

[11] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. IEEE Computer Society, 2016, pp. 243–254.

[12] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[14] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI'18. AAAI Press, 2018, p. 2234–2240.

[15] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4340–4349.

[16] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 319–333.

[17] H. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 30, no. 7, pp. 2093–2103, 2020.

[18] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[20] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 821–834. [Online]. Available: https://doi.org/10.1145/3297858.3304028

[21] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 754–768.

[22] V. Lebedev and V. Lempitsky, "Fast ConvNets using group-wise brain damage," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2016, pp. 2554–2564.

[23] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[24] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[25] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the granularity of sparsity in convolutional neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017, pp. 1927–1934.

[26] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002, p. 311–318.

[27] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. S. Emer, "Timeloop: A systematic approach to DNN accelerator evaluation," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019*. IEEE, 2019, pp. 304–315.

[28] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," vol. 45, no. 2. New York, NY, USA: Association for Computing Machinery, Jun. 2017, p. 27–40.

[29] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, "ADMM-NN: an algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 2019, pp. 925–938.

[30] A. Samajdar, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic CNN accelerator," *CoRR*, vol. abs/1811.02883, 2018.

[31] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, "Laconic deep learning inference acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 304–317.

[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[33] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[34] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015.

[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, pp. 5998–6008.

[36] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 2016, pp. 2074–2082.

[37] J. Yu, A. Lukefahr, D. J. Palframan, G. S. Dasika, R. Das, and S. A. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017, pp. 548–560.

[38] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 2016, pp. 20:1–20:12.

[39] X. Zhao, Y. Wang, C. Liu, C. Shi, K. Tu, and L. Zhang, "BitPruner: Network pruning for bit-serial accelerators," in *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 2020, pp. 1–6.

[40] Y. Zhao, X. Chen, Y. Wang, C. Li, H. You, Y. Fu, Y. Xie, Z. Wang, and Y. Lin, "Smartexchange: Trading higher-cost memory storage/access for lower-cost computation," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 954–967.

[41] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through A cooperative software/hardware approach," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 15–28.