GoSPA: An Energy-efficient High-performance Globally Optimized SParse Convolutional Neural Network Accelerator*

Chunhua Deng
Department of ECE
Rutgers University - New Brunswick
Piscataway, NJ, USA
chunhua.deng@rutgers.edu

Yang Sui
Department of ECE
Rutgers University - New Brunswick
Piscataway, NJ, USA
yang.sui@rutgers.edu

Siyu Liao
Amazon Research
Seattle, WA, USA
liasiyu@amazon.com

Bo Yuan
Department of ECE
Rutgers University - New Brunswick
Piscataway, NJ, USA
bo.yuan@soe.rutgers.edu

Xuehai Qian
Department of Computer Science
University of Southern California
Los Angeles, CA, USA
xuehai.qian@usc.edu

Abstract—The co-existence of activation sparsity and model sparsity in convolutional neural network (CNN) models makes sparsity-aware CNN hardware designs very attractive. The existing sparse CNN accelerators utilize intersection operation to search and identify the key positions of the matched entries between two sparse vectors, and hence avoid unnecessary computations. However, these state-of-the-art designs still suffer from three major architecture-level drawbacks, including 1) hardware cost for the intersection operation is high; 2) frequent stalls of computation phase due to strong data dependency between intersection and computation phases; and 3) unnecessary data transfer incurred by the explicit intersection operation.

By leveraging the knowledge of the complete sparse 2-D convolution, this paper proposes two key ideas that overcome all of the three drawbacks. First, an implicit on-the-fly intersection is proposed to realize the optimal solution for intersection between one static stream and one dynamic stream, which is the case for sparse neural network inference. Second, by leveraging the global computation structure of 2-D convolution, we propose a specialized computation reordering to ensure that the activation is only transferred if necessary and only once.

Based on these two key ideas, we develop GoSPA, an energy-efficient high-performance Globally Optimized SParse CNN Accelerator. GoSPA is implemented with CMOS 28nm technology. Compared with the state-of-the-art sparse CNN architecture, GoSPA achieves average 1.38×, 1.28×, 1.23×, 1.17×, 1.21× and 1.28× speedup on AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt workloads, respectively. Also, GoSPA achieves 5.38×, 4.96×, 4.79×, 5.02×, 4.86× and 2.06× energy efficiency improvement on AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt, respectively. In more comprehensive comparison including DRAM access, GoSPA also shows significant performance improvement over the existing designs.

Index Terms—CNN, Hardware Accelerator, ASIC, Sparse, Convolution

I. INTRODUCTION

Convolutional Neural Networks (CNNs) have achieved unprecedented success in many artificial intelligence (AI) tasks, such as image classification, object detection, video analysis etc. Due to the computation based on 2-D convolution over multiple large-size activation maps, CNNs are very computation and storage intensive, suffering large amount of data movement. To accelerate the execution of CNNs, especially for the inference phase, designing domain-specific CNN hardware accelerators has become a promising solution because of the significant improvement on speed, throughput and energy efficiency thanks to customized design methodology. To date, many CNN accelerator designs have been proposed and reported in academic papers [2]–[6], [9]–[18], [20]–[24], [26]–[31], [33], [34], [37]–[44], [46]–[54], [57]–[60]. Meanwhile, CNN hardware accelerators, especially for inference-only, are also actively investigated by many startup companies because of the huge market of low-power embedded vision.

Among several types of existing CNN accelerators, the sparsity-aware designs are particularly important and attractive because of the much higher energy efficiency and processing throughput compared to the non-sparsity-aware ones. Motivated by these benefits, several sparse CNN hardware architectures [1], [3], [7], [18], [32], [36], [60] have been developed and proposed recently. Among them, SCNN [36] is the first novel dataflow that considers both activation and weight sparsity, thereby achieving high hardware performance. However, SCNN is not optimal since it incurs architecturally-wasted multiplications and unnecessary data transfer. To realize convolution between a sparse kernel and sparse activation map, SCNN first performs multiplications among all the non-zero weights and non-zero
activations, and then selects the required products. Since the convolution operation only needs to pair each non-zero weight with a subset of non-zero activation values instead of all of them, the SCNN’s Cartesian product-based introduces many unnecessary multiplications by design, which correspondingly incurs unnecessary data transfer.

To achieve the optimal scenario with only necessary multiplications in the original algorithm, SparTen [18] and ExTensor [21] attempt to directly search and identify the positions of the matched entries between two sparse vectors. Such positions, known as “key” in SparTen and “intersection” in ExTensor, once correctly located, can be used to perform efficient sparse inner product without unnecessary multiplications.

In this paper, we refer this consistently as intersection-based method. Since the computation of convolutional layer can be interpreted as a stack of inner products, the intersection operation can be used to construct sparse CNN accelerator. In particular, SparTen has demonstrated the potential to achieve much higher performance than SCNN and thus can be viewed as the state-of-the-art sparse CNN accelerator.

While SparTen and ExTensor perform only the necessary computations, they suffer from three major drawbacks. First, the intersection operation, which locates the matching non-zero pairs—the novel mechanism that achieved most speedups—incurs high overhead. Specifically, to identify non-zero pairs, SparTen uses prefix sum, an expensive operation that is actually more powerful than functionality needed (identifying non-zero pairs); while ExTensor relies on an intersection unit which uses content addressable memory (CAM) to scan and search.

As revealed by SparTen’s breakdown report, in terms of both area and power consumption, the hardware cost of prefix sum and its associated priority encoder dominate the cost of the entire datapath of SparTen, e.g., 10 times and 4 times higher than MAC array, respectively. This means prefix sum and its affiliate hardware occupy 62.7% and 46.0% area and power consumption of the entire SparTen, respectively.

Second, in both SparTen and ExTensor, the latency of intersection operations is in the critical path of execution and causes frequent execution stall. To mitigate the effects, SparTen amortizes the cost of prefix-sum with more computations—performing the inner product in the input channel dimension, instead of kernel size. Consider a filter with size $3 \times 3 \times 100 \times 1$ corresponding to $\text{kernel height} \times \text{kernel width} \times \text{input channels} \times \text{output channels}$. It is convoluted with input feature map of size $7 \times 7 \times 100$ corresponding to $\text{input height} \times \text{input width} \times \text{input channels}$. Typically, we first compute the inner product between the $3 \times 3$ window of the kernel in one channel and the “covered” elements in the input feature map of the same channel. In this way, the prefix-sum of the two streams of length 9 is first performed before computing inner product of the same length. Instead, SparTen performs one inner product of length 100 for the same kernel weight position across all channels per prefix-sum. However, this idea increases the demand of SRAM access. It is because performing inner product in input channel dimension requires more weights (e.g., 100)—larger than the kernel size—but the registers may not be able to keep all of them. It leads to similar issues as register spills in compiler, and weights need to be repeatedly loaded from SRAM to the limited registers. Finally, SparTen and ExTensor only avoid unnecessary computation but not data transfer, because the two non-zero vectors need to be loaded before performing the intersection between them. Section II-B will analyze this drawback in detail.

The key motivation of the paper is that by leveraging the knowledge of the complete 2-D convolution, we can develop the efficient and specialized intersection mechanisms that overcome all of the three drawbacks. Following the terminology of Extensor [21], the operands of an intersection operation are known as streams. The stream can be either dynamic, i.e., known only at execution time, or static, i.e., known before the computation. For DNNs, weights constitute the static streams while activations form dynamic stream. While the mechanisms in Extensor is necessary to cover the most general case—intersection with two dynamic streams, we observe that when the intersection is performed between a dynamic and a static stream, more efficient implementation is possible. DNN inference acceleration is a typical and important case that falls exactly to this category. We propose two key ideas that lead to a better sparsity-aware DNN inference accelerator.

The first idea is on-the-fly intersection. When one of the streams is static, how the dynamic stream should be matched is known before computation and does not change. Thus, we can encode the static sparsity information in the data-flow that brings the dynamic stream and “filter” the zeros as well as non-zeros in the dynamic stream. In this way, all values delivered to the compute units are necessary to perform computation. For DNN inference, all the static sparsity information needed is a bit map of weights—they should be matched with the non-zero activations at known locations. The simply but effective idea avoids the three drawbacks. Clearly, it replaces the expensive intersection hardware cost, e.g., prefix-sum in SparTen and CAM in Extensor, with lightweight information embedded in dataflow. The unnecessary data transfers are filtered during activation data delivery. Overall, with the individual intersection operations removed from critical path, the design leads to a more streamlined dataflow.

Our second idea is specialized computation reordering for DNNs. In SparTen and Extensor, a given activation can be paired at different times with different weight elements, and will be fetched multiple times. Why not performing all the checking relevant to an activation together? It is exactly the insight of our idea. Based on on-the-fly intersection, we modify the dataflow such that a given activation is checked with all relevant positions in the bit map of weights, and a non-zero
of 2-D convolution, we name SPARSITY × DNN A × O × × exist in the original convolution algorithm. Not Cartesian Product architecturally-wasted multiplications ideal zero-value multiplications intersection techniques such as pruning results in models with large amount to the need of reducing model size, the model compression accelerators. Improvement than the existing DRAM-included sparse CNN off-chip DRAM access, GoSPA also shows great performance energy efficiency improvement on AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt, respectively. In the more energy efficiency improvement on AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt, respectively. More importantly, GoSPA achieves average 1.38×, 1.28×, 1.23×, 1.17×, 1.21× and 1.28× speedup on AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt, respectively. More importantly, GoSPA achieves 5.38×, 4.96×, 4.79×, 5.02×, 4.86× and 2.06× energy efficiency improvement on AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt, respectively. In the more rigid comparison including energy consumption incurred by off-chip DRAM access, GoSPA also shows great performance improvement than the existing DRAM-included sparse CNN accelerators.

II. SPARSITY-AWARE DNN ACCELERATORS

The sparsity of DNN models is an important problem. Due to the need of reducing model size, the model compression techniques such as pruning results in models with large amount of zeros. However, the smaller models do not necessarily lead to improved performance, the processing throughput is affected by the indirection and irregular accesses to the sparse weights. Thus, it is critical to design DNN accelerators with sparsity support. For convolution, the computation is composed of many products of weights and activations. When either of them is zero, the product becomes zero and the multiplication can be avoided. A sparsity-aware architecture should be able to avoid some or all unnecessary multiplications. Next we discuss three representative designs and their drawbacks.

A. SCNN: Cartesian Product Based Method

The first sparsity-aware CNN accelerator is SCNN [36]. To avoid the unnecessary multiplications due to the zero operand (we call them zero-value multiplications), SCNN directly performs the Cartesian Product among the non-zero activations and weights. While avoiding all the zero-value multiplications, the design artificially induces multiplications among non-zero values that do not exist in the original convolution algorithm. We call them as architecturally-wasted multiplications. The problem is demonstrated in Figure 1. With the same example, it is easy to see that the weight 3 and -3 should never be multiplied with activation -1 and -2, since the weight can never cover these activations no matter which position the kernel is at. Thus, the four multiplications that will be performed in SCNN are all artificially wasted. Similarly, the activation 3 and 1 should never be multiplied with weight -3, incurring another two architecturally-wasted multiplications. Moreover, SCNN also incurs unnecessary data transfer: activation -1 and -2 should not be read from SRAM at all because they only involve in architecturally-wasted multiplications.

In a nutshell, SCNN eliminates all zero-value multiplications but introduces architecturally-wasted multiplications that should not be performed at the first place, and unnecessary data transfers. For a sparsity-aware architecture, the ideal scenario is to only perform necessary computations in the original algorithm. Clearly, SCNN is not optimal.

B. SparTen and ExTensor: Intersection Based Method

The “ideal” sparsity-aware computation, which only performs necessary computations, is achieved by recent architectures SparTen [18] and ExTensor [21]. The major computation steps of them are shown in Figure 2. Both solutions identify the matching non-zero value pairs between kernel weights and the corresponding activations with a conceptual intersection operation, then perform the multiplications between each non-zero pair. In our example, they need to perform four inner product operations to generate the four elements in the result. The two schemes differ in how the intersection operation is implemented. Unlike SCNN, they achieve the desired goal of only performing the necessary multiplications.

However, the two designs suffer from three drawbacks. First, the hardware cost of the intersection operation is high. SparTen realizes that with prefix sum; while ExTensor relies on content addressable memory (CAM) to scan and search. Both methods incur high hardware and energy overhead.
III. EFFICIENT SPECIALIZED INTERSECTION

This section discusses the motivation and insights of the proposed ideas. We show that the intersection operation can be implemented much more efficiently when the knowledge of specific computation structure is considered.

A. Observation: Dynamic and Static Streams

We use the terminology in ExTensor [21] and call operands of an intersection operation as streams. A stream can be considered as a sequence of values in a vector. An intersection operation takes them and produces the common non-zero elements as bit mask in the static stream, given that the corresponding index is marked as non-zero in SSF. The input of computation is the static stream is known and does not change, based on the execution of computation. A good example of static stream is the weights of a sparse DNN model, which is trained and deployed before inference.

As explained in ExTensor, intersection operation is an essential primitive to express sparse computation. For a given algorithm, we first need to move sparse data of two stream to the intersection unit, perform the intersection, and then the computation based on the outcomes of intersection. This procedure is applicable to the most general scenarios where both streams are dynamic. Figure 3 (a) shows high level procedure. Here moving data of $DS_1$ and $DS_2$ before performing intersection $\cap$ is necessary because data is not known before.

Our key observation is that for 2-D convolution in DNNs, the stream for sparse weights is static. Both SparTen and ExTensor still use the same general procedure to handle the intersection between sparse activations, i.e., the dynamic stream $DS_1$, and sparse weights, i.e., the static stream $SS_1$ (see Figure 3 (b)). The key motivation is that, for the computation based on a dynamic stream and static stream, e.g., 2-D convolution, intersection operation can be performed in a much more efficient manner. Next, we demonstrate the insights and how it can avoid the three drawbacks of current architectures.

B. On-The-Fly Intersection

The intersection operation identifies the non-zero pairs of elements in the two input streams. Our key idea is that, for a pair of dynamic and static stream, it is possible and better to perform the logically equivalent intersection on-the-fly when the dynamic stream is brought to the compute unit. Since the static stream is known and does not change, based on its sparsity information, we can generate a conceptual static sparsity filter (SSF), which can be augmented in the data path that brings the elements in the dynamic stream. The idea is shown in Figure 3 (c). SSF simply indicates the indices of the non-zero elements as bit mask in the static stream, given a non-zero element in the dynamic stream, we can directly check whether the corresponding index is marked as non-zero in SSF. The input of computation is the static stream $SS_1$, and the elements in $DS_1$ that belong to the outcome of intersection between $SS_1$ and $DS_1$. Note that the second input is obtained with on-the-fly intersection without using an explicit intersection unit.

This simple but elegant idea can effectively solve the three drawbacks in SparTen and ExTensor. First, the high cost hardware structure to explicitly perform intersection is no longer needed. Second, the computation does not need to frequently wait for the outcomes of intersection. With common techniques such as pipelining, the execution can be made very efficient. Third, by embedding SSF in the data path of dynamic stream, we can avoid transferring the unnecessary element.
Fig. 3: (a) Intersection between two DS. (b) Intersection between DS and SS (SparTen/ExTensor’s method). (c) On-the-fly intersection between DS and SS (Our method).

as early as possible, instead of performing intersection after elements in both streams are staged in the intersection unit.

Readers may wonder why such simple idea was not used in SparTen and ExTensor. We think that it is due to two reasons. The goal of SparTen is to optimize the CONV layer, and it is compared to SCNN. This design indeed eliminated the wasted computations in SCNN using intersection. However, this operation was not presented explicitly as a primitive to facilitate sparse computation. It is not trivial to realize our observation without specifically considering intersection at the higher level. In contrast, ExTensor is an important milestone that explicitly defined intersection and developed its hardware implementation. Just because it is the first paper, it considers the most general design, which is indeed needed if both streams are dynamic. Thus, our idea is a natural further exploration of the intersection primitive considering the property of sparse computation. For dynamic/static stream intersection, we claim that an accelerator should always apply the principles of our idea and perform on-the-fly intersection. Although more specialized, we believe such scenarios are common, for example, sparse recurrent neural network (RNN) inference and finite element analysis also involve a static stream. In this paper, we demonstrate the design of a complete architecture for 2-D convolution.

C. Specialized Computation Reordering

In this section, we consider another optimization when applying on-the-fly intersection in 2-D convolution. Figure 4 shows the computation composed of four 1-D inner products in the same setting, i.e., kernel size $2 \times 2$ with weights $A, B, C, D$, and activation matrix size $3 \times 3$ with activation $1, 2, ..., 9$. We can clearly see that a given activation element can be paired at different times with different weight elements. For example, activation 5 is paired with $D$ in inner product #1, with $C$ in inner product #2, ... With on-the-fly intersection, while fetching activation $(1, 2, 4, 5)$, we can check whether 5 should be sent to compute unit. Here, the sparsity information of $(A, B, C, D)$ is encoded in SSF, details will be shown shortly in our concrete architecture. Then, for inner product #2, activation 5 will be fetched again and checked with $C$.

Based on this observation, it is natural to come up with our second idea. A given element in the dynamic activation stream should be checked with all possible pairing weights together, and if both non-zero, should be sent to the corresponding compute units for multiplication. This is shown on the right of Figure 4. This idea can avoid the repeated data transfer of the same activation, and ensures that each activation is only transferred if necessary, and only once. It can be simply achieved by reordering computation during on-the-fly intersection. For example, when activation 5 is transferred it is checked with $(A, B, C, D)$ using SSF; while activation 4 is checked with $(A, C)$, etc. The only problem left is that, when a compute unit receives an activation, it should have the information of its position in the matrix and multiply that with proper weights. Thus, in our design, we piggyback convolution ID (CID) and position ID (PID) for each activation for this purpose. The details will be discussed shortly.

Readers who are familiar with DNN accelerator data flow can realize that we essentially reorder the computation to adopt weight stationary [8], instead of inner product used in output stationary-based SparTen and ExTensor. Weight stationary means the computation operations related to a given weight are all performed before moving on to the next weight. The output stationary can be understood similarly. As analyzed in many prior dense DNN works (e.g. Timeloop, Maestro and Interstellar [25], [35], [56]), different stationary data flows determine different orders of computation. However, unlike in the dense DNN design that “different dataflows are able to achieve similar and close-to-optimal energy efficiency” [56], an important insight of this work is we argue that weight stationary is uniquely suitable for sparsity-aware DNN accelerators. If we produce one output at a time, we unavoidably only look at a subset of activations, e.g., $(1, 2, 4, 5)$ in Figure 4, losing the opportunity of leveraging global computation structure and also incurring additional data transfer (one of the drawbacks of SparTen and ExTensor). Based on this property, we name our architecture as GoSPA, an energy-efficient high-performance Globally Optimized SParse CNN accelerator.

Before moving to the details of GoSPA, We would like to note that besides SparTen and ExTensor, some other SpMV accelerators, such as ESE [19], also studied the efficient intersection between a dynamic stream and a static stream. However, all of these existing SpMV designs (SparTen, ExTensor and ESE), are designed and optimized for sparse inner products as the primitive operation. As analyzed in Figure 2, such design philosophy is not optimal for sparse 2-D convolution; while our proposed computation reordering is the optimal solution for performing on-the-fly intersection in 2-D convolution. We believe this simple yet optimal approach is particularly important for sparse DNN research – designing
sparse CNN accelerator, which needs to consider two-sided weight and activation sparsity, is much more challenging and non-trivial than designing sparse RNN accelerator (e.g. ESE), which only needs to consider one-sided weight sparsity.

IV. GoSPA: DATAFLOW

Based on the observations and ideas elaborated in Section III, we develop the detailed dataflow of GoSPA. In this section, we first discuss the mechanisms to identify the matching pairs of non-zero weight and activation in the 2-D convolution. Without loss of generality, we assume both the filter kernel and input activation matrices are square, and the numbers of both input and output channels are 1. Figure 5 shows the two examples with different strides. Each weight and activation is indexed using the coordinates of the weight and activation matrices, respectively. From this figure it is seen that, given the shapes of filter kernel \((F \times F)\) and activation matrices \((H \times H)\) as well as the stride value \(S\), the set of matching activations for each weight and the set of matching weights for each activation are completely determined. The mathematical relationship between the coordinates of matching pairs is shown in the dashed boxes of Figure 5.

To conveniently perform the multiplication between matched weights and activations in processing units, we use convolution ID (CID) and position ID (PID) to jointly represent each activation, and use PID to solely represent each weight. The property of this notation is that, given the PID of a weight, we immediately know all activations that should be multiplied with it. We show the specific assignment of CID and PID on the right of Figure 5. For the 2-D array of the matching activations, the activations of the same column shares the same CID, as they are involved with the same inner product; and meanwhile the activations of the same row share the same PID, as they correspond to the same matching weight that is also assigned with the same PID value. With this CID/PID-based indexing, realizing the reordered computation in Figure 4 becomes very convenient: each activation with PID value \(p\) is first multiplied with the weights with the same PID value \(p\), and then all the products are accumulated using a CID-wise way. Notice during window sliding procedure, the same activation can be assigned with different (PID, CID) pairs; while the weight is always assigned with the fixed PID.
two examples in Figure 5. Given the pre-known convolution parameters $F$, $H$, and $S$, the CID/PID of each activation $a_{xy}$ and the PID of its matching weights can be directly calculated as described in the dashed box of Figure 6. Based on the calculated CID and PID, the sparse 2-D convolution that fully utilizes the two optimizations proposed in Section III can now be easily realized. Figure 7 shows an example of the optimized processing procedure. Each non-zero activation, together with its coordinates $3$, are first serially sent to the CID/PID generator. Another input of this generator is the weight bit mask (the conceptual SSF discussed in Section III-B), which helps to filter the non-zero activation lacking the matching non-zero weights. The output of CID/PID generator is a stream of non-zero activations having the matching non-zero weights as well as the corresponding CID/PID information. Since one activation can have multiple matching weights, multiple CID/PID may be generated for the same activation. The hardware implementation detail of the CID/PID generator will be discussed next in Section V. In the final computation phase, each non-zero activation is multiplied with the non-zero activation with the same PID, and all the products are then accumulated according to the CID information.

V. GoSPA: Hardware Architecture

A. Overall Organization

Figure 8 shows the overall architecture of GoSPA. All the weights and the activation results are stored in the off-chip DRAM. When executing the computation for a CONV layer, GoSPA first reads the weights, input activations as well as their corresponding metadata (index) from DRAM, and then sends them to the activation processing unit (APU). The sparse weights are represented with metadata bitmap as shown in Figure 13, which is named weight sparsity pattern (WSP), while the activations are represented with well-known compressed sparse row (CSR) format. APU is responsible for calculating PID and CID for each activation. After that, by checking the pre-known PIDs of the current weights in the PE, the non-zero activations that have the non-zero matching weights are sent to different processing elements (PEs). Each PE contains local weights in SRAM. The entire PE array performs parallel computation to perform the 2-D convolution between multiple filter kernels and activation maps. After PEs’ processing, the results, which are partially sparsified by the ReLU units, are sent back to DRAM in the CSR format.

B. Activation Processing Unit (APU)

APU processes the sparse data and generate ID information as well as preparing correct non-zero activations to the corresponding PEs. Figure 9 shows the inner architecture of APU, which contains two main modules (Stage-1 and Stage-2) plus an activation SRAM. This equipped SRAM here stores rows of input activations of current channel, and it operates as a circulator buffer: when a new row of activation map comes in, it replaces the oldest row in the buffer. Such design strategy significantly reduces DRAM access incurred by the overlap of
convolution operations. It also reduces the overhead of CID storage. By limiting the processing in several rows of the input activations, the CID can be represented with 6-bit to satisfy common CNN models. Please note that the PID information is embedded in which FIFO-A the activation is stored – the activation with PID as $i-1$ is always stored in the $i$-th FIFO-A. Next, we describe the details of Stage-1 and Stage-2s.

**Stage-1.** The Stage-1 module of APU generates CID and PID information for each non-zero activation data read from SRAM, and then it groups different activations according to their different PID information. Figure 10 shows the inner architecture of APU ID generation process, which consists of two blocks, pre-processing, ID generation. Pre-processing block converts the metadata (index) of each CSR-format activation to its coordinates $x$ and $y$, and then calculates $P_x$, $P_y$, $C_x$, $C_y$ as indicated in Figure 5. Then, ID generation block, which consists of $G^2$ copies of IDGen unit, generates all the possible CID/PID information for the current processed activation $a_{xy}$. Each IDGen unit contains 4 comparators and 2 multiply-add units to perform CID/PID calculation as described in Figure 6. Meanwhile, for boundary check, a valid signal is generated to indicate whether the pair of $m$, $n$ in the current IDGen unit correspond to the valid CID/PID information or not. After that, a group of FIFO-As build backup logs for those activations and their corresponding CID/PID, and prepare to send those information to the Stage-2 for further processing. Notice that the processing of Stage-1 of APU is not the system bottleneck because: 1) The read and write access to FIFO-A array are inherently parallel via using multiple IDGen units in Figure 10 for the same activation value as well as the delicate arrangement of PID-wise FIFO-A components; and 2) each FIFO-A component has sufficient FIFO depth, e.g. 64-depth in our design, which can efficiently mitigate the potential problem if sparsity is too high.

**Stage-2.** Because the non-zero activation data collection and grouping scheme in Stage-1 does not distinguish the activations paired with the non-zero weights from the activations paired with the zero-valued weights, the Stage-2 module of APU is in charge of only sending the non-zero activations to the PEs that store the corresponding matched non-zero weights. To that end, Stage-2 consists of a routing module and an array of FIFO-B components. The routing module is used to send the desired content from $N_{PID}$ FIFO-A components to $N_{PE}$ different FIFO-B components, and each FIFO-B component has one-to-one connection to one PE.

The inner architecture of routing module is shown in Figure 9. The routing module receives $N_{PID}$ pairs of non-zero activation and its CID from $N_{PID}$ FIFO-A components. In each clock cycle, a multiplexer is used to select one pair from these $N_{PID}$ input pairs. Specifically, for all the $N_{PID}$ input pairs, they are selected in a sequential way. Notice that because the FIFO-A groups activations according to PID, when the $i$-th pair of activation and CID is selected, the corresponding PID of that activation is also already known as $i-1$. After that, the selected activation and its corresponding CID/PID are broadcast to $N_{PE}$ FIFO-B components for being paired with weights in PEs. Consider we should only pair the non-zero activations with the non-zero weights, $N_{PE}$ switches are needed to determine whether to send the current selected activation to $N_{PE}$ FIFO-B components or not. For the same activation, the switching signals for different switches can be different – the switching signal for the $k$-th switch is based on the weight sparsity pattern (WSP) stored in the $k$-th PE, namely WSP$_k$. As illustrated in Figure 11 and Figure 13, WSP, in the format of bit mask, is a type of metadata representing the sparsity information of weights. Since for a pair of matched weight and activation, they must have the same PID (see Figure 5 and Figure 6), so for one activation with PID$=i-1$ to be sent to the $k$-th FIFO-B component (and the $k$-th PE), it should only be paired with the non-zero weight with PID$=i-1$. In other words, for the $k$-th switch controlled by WSP$_k$ from $k$-th PE, only when its associated WSP$_k[i-1]$ is 1, it will be ON to allow the current selected activation and CID/PID to
be sent to the \( k \)-th FIFO-B component.

Figure 11 shows the example processing schemes of Stage-1 and Stage-2. For stage-1, FIFO-A\( s \) store different non-zero activations with its corresponding PID information. Specifically, for the FIFO-A \( \#i \), it stores the non-zero activation value with PID=\( i \). For FIFO-B \( \#i \), its corresponding PE (PE \( \#1 \)) has WSP as 1010, which means the weights with PID as 1 and 3 are zeros. In such case, the content of FIFO-A \( \#2 \) and \( \#4 \) are not sent to FIFO-B \#1. The same rule applies to FIFO-B \#2, only data in FIFO-A \#2 and \#3 are sent into this FIFO. Notice that when the activation data in one FIFO-A component is allowed to be sent to certain FIFO-B components, within this FIFO-A the activations are read out in a sequential way – in each cycle one activation is read out. For instance, as shown in Figure 11, it takes 4 cycles to send the 4 activations in FIFO-A \#1 to FIFO-B \#1.

**C. Processing Elements (PEs)**

After receiving ACT, CID and PID from APUs, PEs identify the correct non-zero weights for the current input non-zero activations and perform multiplication and accumulation. Figure 12 shows the inner architecture of PE. Here \( N_{REG} \) (typically 2) registers are used to store the non-zero weights. In each clock cycle, when one non-zero activation value in the ACT sequence arrives, the corresponding non-zero weight will be selected and paired with the current activation value. Figure 13 illustrates the details of this pairing. For instance, when two registers are equipped in the PE, initially the first two non-zero weights, namely \( \text{Curr}_Wgt \) and \( \text{Next}_Wgt \) in the current filter, along with their PIDs, are stored in the registers. Then, in each clock cycle the current coming PID is compared with \( \text{Curr}_Wgt_{PID} \) to determine whether there is a PID match or not. If so, the corresponding weight (\( \text{Curr}_Wgt \)) is multiplied with the activation value arrives in the current cycle; otherwise another weight (\( \text{Next}_Wgt \)) will be multiplied with the activation, and meanwhile the PE will update the registers via replacing current weight and PID by next weight and PID as well as reading new weight and PID information from SRAM (see Figure 13). Notice that the temporal pattern of PID ensures additional stationary opportunity – the significant weight reuse at the register level. As shown in Figure 13, because the same PID values always occur in the adjacent cycles, the corresponding PID-indexed weight value always stay in the registers at its maximum possible period, therefore significantly improving register-level weight reuse and reducing SRAM access.

Another direct benefit brought by the processing scheme of GoSPA is the reduction in the unnecessary timing cost. As shown in Figure 14, the dataflow of SparTen causes inherent strong data dependency between prefix sum (and its associate priority encoder) and PEs. Each time the PEs must stall until prefix sum circuits identify the matched pairs. Please note that such waiting time is caused by inefficient identifying and matching non-zero pair in each PE, and it is different from the under-utilized PEs due to imbalanced sparsity across multiple PEs. On the other hand, the dataflow of GoSPA makes a pipeline-like computing procedure become possible. More specifically, with the use of PID and CID, identifying the matched pair is not a random search, but can be properly calculated using 2-D convolution’s inherent spatial pattern. Therefore, APU is always able to provide the desired non-zero activation value matching the current non-zero weight in the PEs, and thereby eliminating the unnecessary waiting time.
D. Architectural Overhead Analysis

Similar to all the other sparse CNN accelerators, GoSPA needs additional hardware modules for processing sparse 2-D convolutions. For GoSPA, those architectural overhead includes extra FIFO access, PID/CID generation as well as storing/reading metadata (WSP) for PEs and switches. Fortunately, thanks to the proper use of PID/CID information, GoSPA is able to process sparse 2-D convolution in a much more efficient way than the prior works. More specifically, once the CNN shape parameters are given, the relationship between the matched weight and activation is already pre-determined offline (see Figure 5), and hence the PID/CID generation only requires simple logic (see Figure 6). Meanwhile, the routing module in APU only requires simple multiplexer and switches. Compared with the complicated high-cost prefix sum used in SparTen and content addressable memory (CAM) used in ExTensor, the simple sparsity-handling hardware in GoSPA brings much lower overhead. As shown in Figure 18(a) in Section VI, the area and energy consumption of architectural overhead of GoSPA is only 10.6% and 14.5%, respectively; while SparTen suffers 62.7% and 46.0% extra area and power consumption for handling sparsity.

VI. EVALUATION

A. Experimental Methodology

We develop a simulator to build a behavior model and use Verilog to build a bit-accurate cycle-accurate verified RTL model, which is synthesized with CMOS 28nm library using Synopsys DC Compiler. After that, we use Synopsys IC Compiler to place and route on the synthesized netlist. Finally, PrimeTime PX is used to estimate power consumption via using gate-level netlist and switching activity interchange format (SAIF) file.

For the comparison between GoSPA and SparTen, we evaluate the performance on AlexNet, VGG-16, GoogLeNet, MobileNetV2, ResNet-18 and ResNeXt-50. The evaluated sparse models are trained and pruned using Pytorch with maintaining the same accuracy of dense models. The overall weight sparsity ratio after pruning is 63%, 62%, 68%, 30%, 60% and 60% for AlexNet, VGG-16, GoogLeNet, MobileNetV2, ResNet-18 and ResNeXt-50, respectively. For the comparison between GoSPA and other sparse CNN ASIC designs with DRAM-included and post-layout synthesis results, since Eyriess series and NullHop only report the performance on AlexNet, VGG-16, and MobileNet, our comparison with those works are based these three workloads.

B. Comparison with SparTen

Comparison on Speedup. To be consistent with SparTen, we utilize simulator to evaluate the speedup performance of GoSPA. For fair comparison, the simulator-based GoSPA has the same number of multipliers used in SparTen. Figure 15 shows the speedup performance of GoSPA and SparTen over the baseline dense architecture on different sparse CNN models. From the figures it can be seen that, compared with SparTen, because GoSPA avoids the frequent execution stall incurred by the data dependency between intersection and computation, GoSPA achieves $1.38 \times$, $1.28 \times$, $1.23 \times$, $1.17 \times$, $1.21 \times$ and $1.28 \times$ overall speedup across all the layers of AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt, respectively. Notice that GoSPA is slightly inferior to SparTen on three layers of GoogLeNet. This is because the very unbalanced weight distributions in these layers make SparTen, which uses extra hardware-based balancing module, can achieve more balanced weight distribution than GoSPA. We expect GoSPA would achieve higher performance than SparTen on these three layers if extra balance-specific hardware module is also used in GoSPA as SparTen does.

Comparison on Energy. To evaluate the energy efficiency of GoSPA architecture and have fair comparison with SparTen, we develop a design example with 8 PEs, where each PE contains four 16-bit multipliers, 256 24-bit accumulators, one 1024 $\times$ 16 weight SRAMs. Therefore, the total number of multipliers of this 8-PE GoSPA design example is 32, which is the same with the configuration reported in SparTen. The APU contains one $2720 \times 64$ SRAM bank to store the input activations to reduce DRAM access. The total area and power of this 8-PE GoSPA is 0.25 $mm^2$ and 29.64 mW, respectively.

Figure 17 compares the energy efficiency of GoSPA and SparTen on different workloads. It is seen that GoSPA achieves $5.38 \times$, $4.96 \times$, $4.79 \times$, $5.02 \times$, $4.86 \times$ and $2.06 \times$ higher energy efficiency over SparTen on AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt workloads, respectively. In-depth numerical analysis verifies such significant improvement on energy efficiency mainly come from the two architecture-level advantages of GoSPA over SparTen:

1) GoSPA has much less SRAM access than SparTen. The unnecessary data transfer, which SparTen/ExTensor suffers, is efficiently removed in GoSPA. As shown in Figure 16, GoSPA has an average of $5.5 \times$, $6.0 \times$, $3.9 \times$, $6.57 \times$, $5.73 \times$ and $2.76 \times$ less SRAM access than SparTen on AlexNet, VGG, GoogLeNet, MobileNet, ResNet and ResNeXt, respectively. Such huge saving significantly reduces energy consumption.

2) GoSPA needs much less hardware overhead for handling sparsity than SparTen. Figure 18(a) summarizes the hardware cost for the modules that are involved with handling sparsity in GoSPA and SparTen. It is seen that for SparTen the prefix sum and priority encoder consume 62.7% and 46% of the total area and power; while only 10.6% and 14.5% of the area and power consumption of GoSPA come from modules involved with handling sparsity. Such huge saving on hardware cost further brings better energy efficiency of GoSPA over SparTen.

C. Sensitivity of Load Imbalance to Sparsity

Load imbalance, which causes under-utilization of multipliers, is a common problem that all sparse CNN accelerators suffer. Fortunately, thanks to helping build backlog of input activations, the 2-stage FIFOs in the APU of GoSPA can efficiently alleviate this problem. Figure 19 illustrates the impact of FIFO depth on the average multiplier utilization for GoogleNet with different weight and activation density configurations. It is seen that with the FIFO depth increases,
Fig. 15: Speedup comparison between GoSPA and SparTen. For MobileNet, "Btlnck" denotes the bottleneck block of stacked convolutional layers [45]. For ResNeXt, "Group" means the stage of stacked convolutional layers with cardinality=32 [55].

Fig. 16: Normalized SRAM access comparison. For MobileNet, "Btlnck" denotes the bottleneck block of stacked convolutional layers [45]. For ResNeXt, "Group" means the stage of stacked convolutional layers with cardinality=32 [55].
TABLE I: Performance comparisons among different sparse CNN ASIC designs (scaled to 28nm).

<table>
<thead>
<tr>
<th>Design</th>
<th>Eyeriss</th>
<th>NullHop</th>
<th>Eyeriss V2</th>
<th>GoSPA</th>
<th>GoSPA-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Width</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Number of Multipliers</td>
<td>168</td>
<td>128</td>
<td>384</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Clock Frequency (MHz)</td>
<td>464</td>
<td>500</td>
<td>464</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Core Area (mm²)</td>
<td>2.27</td>
<td>6.3</td>
<td>4.08</td>
<td>2.67</td>
<td>1.76</td>
</tr>
<tr>
<td>Throughput (frames/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AlexNet</td>
<td>89.6</td>
<td>N/A</td>
<td>795</td>
<td>460.3</td>
<td>460.3</td>
</tr>
<tr>
<td>VGG</td>
<td>1.86</td>
<td>13.71</td>
<td>N/A</td>
<td>29.7</td>
<td>29.7</td>
</tr>
<tr>
<td>MobileNet</td>
<td>N/A</td>
<td>N/A</td>
<td>3412</td>
<td>1868</td>
<td>1868</td>
</tr>
<tr>
<td>DRAM Access (MB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AlexNet</td>
<td>15.4</td>
<td>N/A</td>
<td>4.6</td>
<td>4.1</td>
<td>2.59</td>
</tr>
<tr>
<td>VGG</td>
<td>321.1</td>
<td>42</td>
<td>N/A</td>
<td>58.8</td>
<td>36.38</td>
</tr>
<tr>
<td>MobileNet</td>
<td>N/A</td>
<td>N/A</td>
<td>3.9</td>
<td>1.81</td>
<td>1.01</td>
</tr>
<tr>
<td>On-Chip/System Power (mW)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AlexNet</td>
<td>278/336</td>
<td>N/A/N/A</td>
<td>461/1075</td>
<td>128.2/445.3</td>
<td>89.7/290</td>
</tr>
<tr>
<td>VGG</td>
<td>236/269</td>
<td>155/257</td>
<td>N/A/N/A</td>
<td>136.0/429.4</td>
<td>95.2/277</td>
</tr>
<tr>
<td>MobileNet</td>
<td>N/A/N/A</td>
<td>N/A/N/A</td>
<td>572/2808</td>
<td>145.0/713</td>
<td>100.7/418</td>
</tr>
<tr>
<td>On-Chip/System Energy Efficiency (frames/J)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AlexNet</td>
<td>322.3/266.7</td>
<td>N/A/N/A</td>
<td>461/1075</td>
<td>128.2/445.3</td>
<td>89.7/290</td>
</tr>
<tr>
<td>VGG</td>
<td>7.88/6.9</td>
<td>88.45/53.3</td>
<td>N/A/N/A</td>
<td>218.4/69.2</td>
<td>312/107.3</td>
</tr>
<tr>
<td>MobileNet</td>
<td>N/A/N/A</td>
<td>N/A/N/A</td>
<td>5966/1215</td>
<td>18550/4473</td>
<td>18550/4473</td>
</tr>
<tr>
<td>Area Efficiency (frames/mm²)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AlexNet</td>
<td>39.5</td>
<td>N/A</td>
<td>194.9</td>
<td>172.4</td>
<td>261.5</td>
</tr>
<tr>
<td>VGG</td>
<td>0.82</td>
<td>2.17</td>
<td>N/A</td>
<td>11.1</td>
<td>16.9</td>
</tr>
<tr>
<td>MobileNet</td>
<td>N/A</td>
<td>N/A</td>
<td>836</td>
<td>699.6</td>
<td>1061</td>
</tr>
</tbody>
</table>

*Eyeriss reports DRAM access with batch mode, where batch size is 4 for AlexNet, and 3 for VGG. Other accelerators use batch size=1.

![Fig. 17: Normalized energy efficiency comparison.](image1)

![Fig. 18: (a) Area and power overhead for handling sparsity. (b) APU area breakdown.](image2)

the multiplier utilization increases steadily. When using 64-depth or 128-depth FIFO, GoSPA can maintain very high multiplier utilization (> 90%) for most density configurations. Considering the density range of activation in common sparse CNN workloads is typically 30%-70% (summarized from the evaluated six workloads), to make good balance between high multiplier utilization and area/power overhead, we set the depths of FIFO-A and FIFO-B as 64 in our GoSPA design.

D. Comparison with Other DRAM-included Sparse CNNs

We also compare GoSPA with other DRAM-included sparse CNN designs (Eyeriss series and NullHop). For fair comparison, we scale the 8-PE GoSPA to 32-PE GoSPA to have the same number (128) of multipliers used in NullHop. In this scaled design, APU contains two 2720 × 64 SRAM banks. To be consistent with clock rate of other designs after considering the difference among different technology nodes, 32-PE GoSPA is synthesis under 500MHz operating frequency.

As shown in Table I, even the 16-bit GoSPA outperforms 8-bit Eyeriss V2 with respect to on-chip (excluding DRAM) and system (including DRAM) energy efficiency. In overall, GoSPA-R achieves at least 2.97×, 2.15× and 1.34× higher on-chip energy efficiency, system energy efficiency and area efficiency, respectively, on AlexNet. On VGG, GoSPA achieves at least 2.47×, 1.30× and 5.12× higher on-chip energy efficiency.
energy efficiency, system energy efficiency and area efficiency, respectively. On MobileNet, GoSPAR achieves 3.11×, 3.68×, and 1.27× higher on-chip energy efficiency, system energy efficiency, and area efficiency, respectively.

VII. CONCLUSION

This paper proposes GoSPA, a sparse CNN accelerator architecture. By using a novel on-the-fly intersection and reordering computation, GoSPA globally optimizes sparse 2-D convolution and hence avoids the limitations of the state-of-the-art sparse CNN designs. Evaluations show that GoSPA achieves significant hardware performance improvement than the state-of-the-art sparse CNN accelerators.

REFERENCES


