DenSparSA: A Balanced Systolic Array Approach for Dense and Sparse Matrix Multiplication

***Abstract*—Numerous studies have proposed hardware architectures to accelerate sparse matrix multiplication, but these approaches often incur substantial area and power overhead, significantly compromising their usage in dense scenarios. On the other hand, systolic arrays deliver high efficiency for dense matrix operations, but their application to sparse matrices remains challenging. An ideal design should process both dense and sparse matrices with high efficiency to satisfy performance and versatility requirements.**

In this paper, we introduce DenSparSA, a balanced systolic array centralized architecture that can execute sparse matrix computations with minimal overhead to original dense matrix computations. DenSparSA supports both single-side and dual-side unstructured sparse matrix multiplications with high efficiency. At the same time, the additional hardware required for managing sparsity is compact and decoupled from the conventional systolic array, allowing for minimal power overhead when switched back to dense matrix operations via circuit gating. The proposed design is implemented with Nangate 45nm. Implementation results show that DenSparSA achieves a speedup ranging from 1.9× to 22× compared to the classic systolic array for sparse workloads, while maintaining relatively low area and power overhead. For dense workloads, the power overhead can be reduced to 12% for BF16 and 5% for FP32. Compared with existing solutions for sparse acceleration, DenSparSA delivers competitive (0.82×-1.32×) efficiency in sparse scenarios and 1.17×-2.28× better efficiency for dense scenarios, indicating a better balance between both situations.

***Index Terms*—Sparsity, Systolic Arrays, Acceleration, Neural Network**

I. Introduction

Systolic arrays have emerged as a leading solution for computing dense matrix operations, exemplified by Google's Tensor Processing Unit (TPU) [1], a state-of-the-art systolic array architecture. The TPU's core matrix multiplication engine employs a 256 × 256 array of identical processing elements (PEs) to perform 8-bit Multiply-and-Accumulate (MAC) operations. In its third-generation design (TPU v3) [2], the architecture features a 128 × 128 array capable of supporting floating-point (FP) arithmetic.

As deep neural networks increasingly adopt sparsity to improve efficiency, there is a growing demand for systolic arrays to reap the benefits from unstructured sparse matrices while maintaining their efficiency in dense matrix multiplication. By providing robust support for both dense and sparse computations, systolic arrays can deliver substantial performance gains for AI accelerators, spanning a wide range of applications from training to inference [3].

To enable systolic arrays for sparse matrix multiplication, researchers have proposed various approaches. However, many of these methods either rely heavily on software-assisted algorithms [3] or introduce significant overheads, such as increased power and area costs, to support unstructured single-sided and dual-sided sparsity. Although these overheads are affordable by the computation savings achieved through sparsity, they remain problematic for dense matrix multiplication, where no such savings are realized. These overheads significantly constrain the use of systolic arrays to sparse matrix multiplication, undermining their original efficiency in processing dense matrix operations. In general-purpose scenarios, such as tensor processing units (TPUs), efficient support for dense matrix multiplication is equally critical.

This paper presents DenSparSA, a novel architecture and microarchitecture design for systolic arrays aimed at efficiently supporting general sparse matrix multiplication (i.e., including both single-sided and dual-sided unstructured sparse matrix multiplications) while maintaining the efficiency in dense matrix multiplication. The key contributions of this paper can be summarized as follows:

- Unified Support for Unstructured Sparsity: DenSparSA introduces an innovative and efficient systolic array architecture that effectively handles both single-sided and dual-sided unstructured sparsity, with no reliance on fixed sparsity pattern, software-preprocess or external sparsity filters.
- Decoupled Hardware for Efficient Sparsity Management: The design incorporates compact, decoupled hardware modules to efficiently manage sparsity. Hardware support for sparsity could be turned off by clock-gating without interfering with the computation of dense matrix multiplication. Therefore, power overhead for dense workloads are greatly reduced, bringing about better generality in both dense and sparsity scenarios.
- Comprehensive Evaluation and Benchmarking: We implement the proposed DenSparSA using ASIC designs, demonstrating speedups from 1.9× to 22× for sparsity, with relatively low overhead compared to the classic systolic array. When compared to existing solutions in multiple workloads, DenSparSA achieves competitive area efficiency (0.82×-1.32×) and power efficiency (1.17×-2.28×).

The following content is organized as follows: Sec. II presents the background and related work for the systolic array and related approaches for sparse matrix multiplication. Sec. III introduces the methodology incorporating dataflow and architecture designs of DenSparSA. Sec. IV describes the microarchitecture designs to support the data flow. Sec. V discusses the extensive experiments and compares the DenSparSA with the existing solutions. Sec. VI concludes this paper.

II. Background and Related Work

A. Systolic Array

Systolic arrays, also known as Matrix Processing Units (MXUs) or General Matrix Multiplication Accelerators, are specialized hardware architectures designed for parallel dense matrix multiplication, a fundamental operation in deep neural networks [4]. As shown in later Fig. 2, the systolic array consists of a grid of processing elements (PEs) and a three-level memory hierarchy (buffers) to execute computations in parallel, enabling highly efficient parallel processing [5]. Each PE performs multiply-accumulate operations and communicates with adjacent PEs or buffers, making systolic arrays adept at managing input data and weight sharing. They can be scaled to meet computation demands while reducing memory bandwidth requirements. This efficiency in handling dense matrix multiplication makes systolic arrays integral to accelerating deep neural networks (DNN) tasks [4]. However, while systolic arrays are highly effective for linear operations like dense matrix multiplication (e.g., im2col-based convolution), they cannot efficiently compute sparse matrix

multiplications, which are common in compressed DNNs and other heavy computation tasks.

B. Sparse Matrix Multiplication

Matrix multiplication constitutes a major workload in many domains, including the training and inference of DNNs and scientific computation such as graph computation [6, 7] and fluid dynamics [8]. Among these workloads, sparsity is a common scenario, leading to the possibility of extra speedup and improved efficiency by skipping the zero-valued elements that do not contribute to the computation result. A special case of sparsity, structured sparsity, refers to the situation of having predictable patterns of zeroes in matrices, allowing for optimization in memory and computation in deep learning models as predictable sparsity patterns reduce the complexity of hardware support and are approachable by special pruning methods [9]. Different DNN models lead to different scenarios of sparsity. CNN-based models have sparsity in both the weight matrix and activation matrix and sparsity of transformer-based models exists mainly in the weight matrix. The hardware overhead induced by supporting sparsity is often referred to as sparse tax [10, 11], and different strategies in handling sparsity lead to different trade-offs between overall performance and sparse tax.

C. Existing Solutions for Sparse Matrix Multiplication

To reduce hardware costs on random sparsity patterns, many works apply software-hardware co-design to gain speedup from sparsity. Structured sparsity in DNNs is welcomed by both research outcomes like Highlight [11] and commercial products like NVIDIA Ampere [12]. Also, designs like Sense [13] apply a special pruning method on CNN to produce balanced sparsity and reduce the hardware complexity but lack generality for other workloads. However, structured pruning tends to result in larger precision loss and lower sparsity, limiting the effect of speedup [14], and pruning specific to accelerator architecture compromises with the generality of the hardware platform.

Given the limit of structured or specially pruned sparsity, solutions targeting unstructured sparsity emerge. Existing ASIC solutions for unstructured sparse matrix could be divided into two categories: some are based on slightly modified TPU-like systolic array [3, 15, 16] while others are not [17, 18, 19].

STPU [3] and Mentha [15] are two TPU-like designs that adopt data packing to tackle sparse workloads. STPU applies a co-design of packing algorithm and hardware support to leverage sparsity, requiring packing to be done offline. Mentha is also oriented on data packing, but the packing algorithm could be conducted either online or offline. While achieving moderate speedup, STPU and Mentha incur relatively low cost overhead since the modification to a TPU-like baseline is limited. S2 Engine [16], another TPU-like design, does not require data packaging but is limited to dual-side sparsity in CNN. Though having a small cost overhead, designs based on systolic arrays are limited either in performance or in working scenarios.

Aside from systolic arrays, other architectures, on the other hand, achieve higher speedup but induce rather large overhead. SIGMA [17] targets acceleration of irregularly shaped and sparse matrices, leveraging single-side sparsity solely. DSSA [19] leverages dual-side sparsity with a systolic architecture, providing better speedup for CNN workloads than SIGMA while still incurring large overhead. Trapezoid [18] is a solution with high generality, utilizing both single-side and dual-side sparsity for high speedup over a wide range of scenarios. Although these designs provide higher performance for sparse workloads, they involve rather large areas and power overhead, leading to reduced efficiency when handling dense workloads. While numerous previous designs have been proposed to accelerate sparse calculation, they couldn't meet the need to provide a general hardware solution to achieve high performance and efficiency with both sparse and dense workloads.

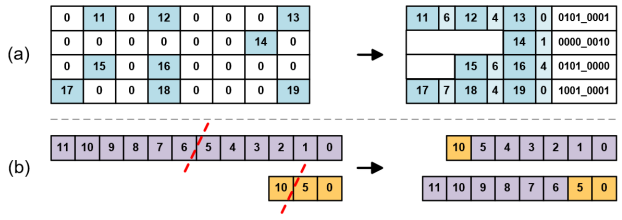


Fig. 1. Examples of dataflow compression in DenSparSA. (a) shows Vector Compression for dual-side sparsity and (b) shows Vector Permutation for single-side sparsity.

III. Dataflow and Architecture Design

In this section, we introduce the dataflow and architecture designs of the proposed DenSparSA.

A. Dataflow Support for Sparsity Computation

For systolic arrays, the total computation latency depends on the time required for data to fully pass through the array. Therefore, the quest for speedup is essentially a quest for shorter data passage time, and acceleration could be made if the hardware support calculation upon compressed dataflow with shorter length and shorter passage time. Based on such principle, a system of compressed dataflow is designed to enable general speedup in different situations of sparsity. The dataflow proposed here would serve as the target for our microarchitecture design.

1) Vector Compression for Dual-side Sparsity:

For dual-side sparsity, both input matrices could be compressed by excluding zero elements, as illustrated in Fig. 1(a). The input matrices that enter the array vertically and horizontally are respectively compressed according to column and row vectors, leading to a shortened length of input on both directions. After compression, each remaining element carries an index that marks its position in the original vector, and a series of elements holds a bit mask representing its sparsity pattern.

However, such a compression method will result in index mismatch when the compressed vectors arrive at the processing element (PE) arrays and lead to incorrect computation results. We will discuss in detail how to design microarchitecture to address this issue in section IV(A).

2) Vector Permutation for Single-Side Sparsity:

In the case of single-side sparsity, only one of the input matrices could be compressed while the other is left complete. This leads to a problem: even though computation between mismatched elements is made possible by certain hardware modification, there is no effect of speedup since the data passage time of the dense input matrix remains unchanged. This leads to restricted usage, since transformer-based DNN models holds only single-side sparsity.

To address this problem, we propose a method called Vector Permutation (VP) to balance the length of compressed sparse vectors and uncompressed dense vectors. As is shown in Fig. 1(b), each input vector is cut into two halves by the median index, after which the later halves are exchanged between each pair of vectors symmetric about the diagonal. The vector length after permutation is roughly the average of the original length of the two paired vectors. Please note

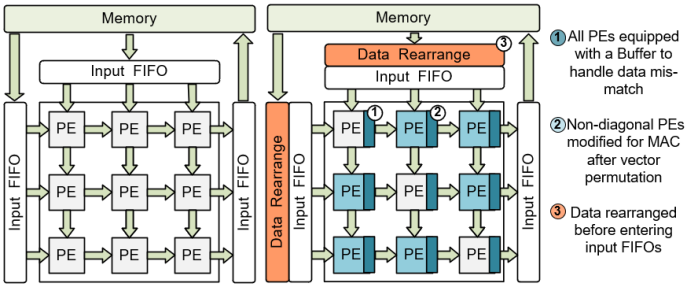


Fig. 2. DenSparSA architecture compared with the conventional systolic array.

that the ascending sequence of the element indices are maintained after permutation, which plays a key role in DenSparSA's approach to accelerating with single-side sparsity..

For single-side sparsity, VP is a must in our design to achieve speedup. For dual-side sparsity, VP could also provide the additional effect of speedup by balancing the length of inputs on two sides. The hardware implementation of vector permutation and computation details of permuted vectors will be discussed in section IV(B).

3) Vector Filter for Extra Speedup:

In matrix multiplication scenarios such as DNN, a weight matrix is pre-known. If the pre-known weight matrix has considerable sparsity, such sparsity could be utilized ahead of computation to achieve extra speedup. In such a case, DenSparSA acquires a merged sparse vector out of a number of vectors in the pre-known sparse weight, after which the other input matrix is filtered by this merged sparse vector to shorten its length before computation.

Such an approach brings about better speedup but results in array partitioning and corresponding cost overhead, the details of which will be discussed in section IV(C).

B. Architecture Support for Proposed Dataflow

To support the proposed dataflow, the new hardware architecture should be able to 1) perform Multiply-and-Accumulate (MAC) operations between compressed vectors with mismatched data, 2) perform MAC between permuted vectors and 3) rearrange the data inputs to realize vector permutation and filtering. In order to fulfill the requirements as listed above, we make a series of modifications to a classic systolic array which has the same architecture as the matrix multiplication unit in the Tensor Processing Unit (TPU), and propose a new architecture based on it, as illustrated in Fig. 2.

The fundamental concept behind this design comprises mainly three essential microarchitecture improvements: ① All PEs are equipped with a buffer to handle data mismatch. ② Non-diagonal PEs are slightly modified to support VP, while buffered PEs on the diagonal are naturally adaptable to VP. ③ Data are rearranged before entering the input FIFOs in order to implement VP and filtering. The above modifications, especially ① and ② which contribute to the major part of hardware overhead, are carefully designed to reduce their impact on the inherent structure of a classic (TPU-like) systolic array. The operations of the PEs and buffers are pipelined to minimize changes to the critical path. The buffers are compact and can be decoupled from the array, allowing them to be clock-gated when switching back to dense matrix multiplication. As a result, support for sparsity is achieved with reduced impact on the area and power efficiency of the original systolic array during dense workload execution.

IV. Microarchitecture Design and Optimization

This section will dive into the details of the microarchitecture, explaining the design and operation of hardware modules required to support previously proposed dataflow.

A. Buffered PE for Index Matching

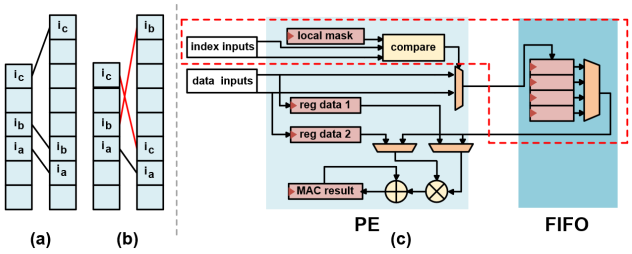


Fig. 3. Dataflow (a, b) and hardware structure (c) of buffered PE. All components in (c) marked by a reg triangle are clock-triggered.

Data in compressed vectors are mismatched when they arrive at a processing element (PE), of which an archetypal situation is shown in Fig. 3(a) with i_x denoting the ascending indices of corresponding elements. One element of a matching pair arrives at the PE earlier than another, for which extra storage is needed to compute multiplication correctly. Also, the intersection between matching pairs as illustrated in Fig. 3(b) is impossible as it conflicts with the ascending order followed by indices in the same vector. Therefore, extra storage needed to handle data mismatch could be implemented in the form of a FIFO buffer. The interact logic between a PE and its buffer is shown in Algorithm 1. The PE holds a local bit mask denoting the indices of matching pairs, by which it could tell whether an element should be preserved for future calculation. Values in the leading one of the matching pairs are pushed, and a value is popped from the buffer when the later one arrives.

Algorithm 1: Access Logic of Buffer

Input: $index_A, data_A, index_B, data_B$
begin
  if $index_A > index_B$ **then**
    if $index_A$ *found in mask* **then**
      push $data_A$
    if $index_B$ *found in mask* **then**
      pop
  if $index_B > index_A$ **then**
    if $index_B$ *found in mask* **then**
      push $data_A$
    if $index_A$ *found in mask* **then**
      pop

Apparently, conducting buffer access and Multiply-and-Accumulate (MAC) sequentially in the same cycle would result in a significantly longer critical path, thus buffer access and MAC are pipelined in our design. Conventionally, a PE in a systolic array does calculations upon data received at the input ports. In DenSparSA, the calculation is done upon data transferred to neighbor PEs, i.e., the data at the output ports. Thus, the inputs to the multiplier is the data that arrived a cycle ago. If the two input indices match naturally, MAC upon the two input values are done in the next cycle. If the latter one of a matching pair arrives, a pop signal is sent to the buffer, and MAC is done between popped data and one

of the data passed to a neighbor PE. The pipeline logic is described in Algorithm 2.

Algorithm 2: Pipeline Logic of Buffered PE

Input: $index_A, data-in_A, index_B, data-in_B$
Output: $data-out_A, data-out_B$
begin
  if $index_A = index_B$ **then**
    do MAC on data outputs in the next cycle
  if $index_A > index_B$ & $index_B$ *found in mask* **then**
    do MAC on popped data and $data-out_B$
  if $index_B > index_A$ & $index_A$ *found in mask* **then**
    do MAC on popped data and $data-out_A$

The hardware structure of a buffered PE is shown in Fig. 3(c), where trivial details such as connections with other PEs are eliminated for simplicity. The combinational and sequential logic in the buffer contributes to the major part of area and power overhead. A merit of our design is that PEs are decoupled with their buffers when performing dense matrix multiplication, during which turning the buffer off by clock gating would not affect the function of the PE. When the compute array is switched to dense workloads, the clock connected to components framed in a red dotted line in Fig. 3(c) is gated, and the dynamic power due to index propagation, local bit mask refreshment, index comparing and FIFO will not contribute to power overhead.

FIFO depth is a crucial parameter in our design. When a FIFO is full, the operation on the corresponding PE has to stall, leading to a slowdown in computation. Thus, large FIFO depth reduces the overall stall rate but results in larger hardware overhead. On the other hand, FIFOs constitutes the main part of overhead in our design. Though they do not account for significant chip area, they lead to larger power overhead. Therefore, a compromise should be made between hardware overhead and lower stall rate. Since the stall rate is hard to model mathematically, we apply the Monte Carlo method to simulate the overall stall ratio when FIFOs are configured with different depths. This is done by feeding random sparsity patterns with varying level and combination of sparsity to a self-built cycle-accurate simulator, and an expectation of stall rate is calculated from the simulation outcomes. The result is shown in Table. I. In our implemented design, the FIFO depth is chosen to be 6 and the exact stall rate is 8.61×10^{-4}.

TABLE I
RELATION BETWEEN STALL RATE AND FIFO DEPTH.

FIFO depth	0	1	2	3	4
Stall rate	0.998	0.724	0.323	0.109	0.028
FIFO depth	5	6	7	8	9
Stall rate	0.0056	0.000861	3×10^{-6}	2×10^{-8}	0

B. Calculation upon Permuted Vectors

As is discussed in III, Vector Permutation (VP) promotes speedup by balancing the length of inputs. The buffered PEs described in V-C could be easily adapted to MAC between permuted vectors. We'll begin with the situation of a single PE, and then extend the design to a whole PE array.

In VP, original vectors are cut according to the middle index of the longer vector. Therefore, only the part with larger indices than that middle index are exchanged during permutation of the later halves of the vectors. Such a process maintains the ascending order of the indices in each vector, as could be shown in Fig. 1(b). On the other hand, the logic of FIFO-buffered PE is applicable for inner products of matched elements as long as the indices are ascending in each vector. Consequently, an isolated buffered PE naturally supports VP.

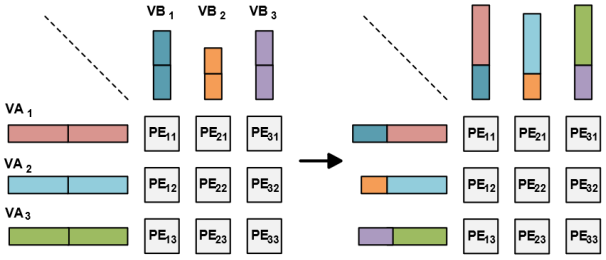


Fig. 4. Showcase of Vector Permutation for a PE array.

The situation is a bit different for an array of PEs. In this case, each side of the array receives a batch of vectors, and vector permutation is performed between each pair of vectors symmetric about the diagonal, as illustrated in Fig. 4. For the PEs on the diagonal, the situation is the same as compared to an isolated PE, since the former and later halves contribute to inner products between the same pair of original vectors. For non-diagonal PEs, however, the former and later halves contribute to inner products between different vector pairs.

Assume the longer vectors have length $2n$ or $2n+1$. For PE_{ij}, the MAC result in the first n cycles accounts for vector_{Ai} and vector_{Bj}, while MAC result in later cycles accounts for vector_{Aj} and vector_{Bi}. An exactly opposite situation holds true for PE_{ji}, which is symmetric to PE_{ij} about the diagonal. Thus, by exchanging the MAC result between PE_{ij} and PE_{ji} after the nth cycle for once, each of the pair of symmetric PE gets the correct MAC result between a certain pair of permuted vectors at the end. The hardware cost for this is just a MUX for each non-diagonal PE to decide whether the MAC result should be refreshed based on the result from the same PE or another PE.

C. Data Rearrange

Modules for data rearrangement are introduced to conduct VP and filtering. In a classic design of a systolic array, input data are held by double-buffered FIFOs before entering the compute array. In our design, a filter and a router are inserted at the interface of each input FIFO and array memory. The detailed microarchitecture is illustrated in Fig. 5. The bit masks of a group of sparse vectors are merged by OR operation, producing a merged mask to be used as a filter for the denser input matrix. The index of each element from the dense matrix is compared with the merged bit mask, and only elements with a matching index can enter the input FIFO. Filtering is only conducted for the denser side of the inputs since the benefit of filtering an already compressed sparse input is limited as compared to its overhead.

The router is a combination of two 2-1 MUXs, determining which source should the input FIFOs receive data from. Dense input vectors filtered by the same mask share the same series of indices and consequently the same middle index, therefore the flag signal of MUXs could be shared between them, reducing the complexity of control. Routing of this form is applied between each pair of input vectors symmetric about the diagonal of the array.

With filtering comes the need to partition the compute array, since dense inputs filtered by a certain group of sparse vectors are no longer applicable to another group of sparse vectors. The size m of a sparse

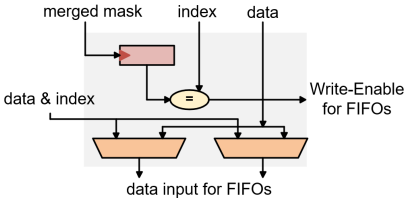


Fig. 5. Data rearrange at array-memory interface.

vector group to be merged into a mask is positively correlated with overheads of partitioning as m is exactly the partition granularity, and negatively correlated with the speedup effect as smaller m brings about higher sparsity in the merged mask. As for our design where medium sparsity constitutes the major workload, m is chosen to be 4 to make a balanced compromise between overhead and speedup.

V. Implementation and Evaluation

A. Implementation

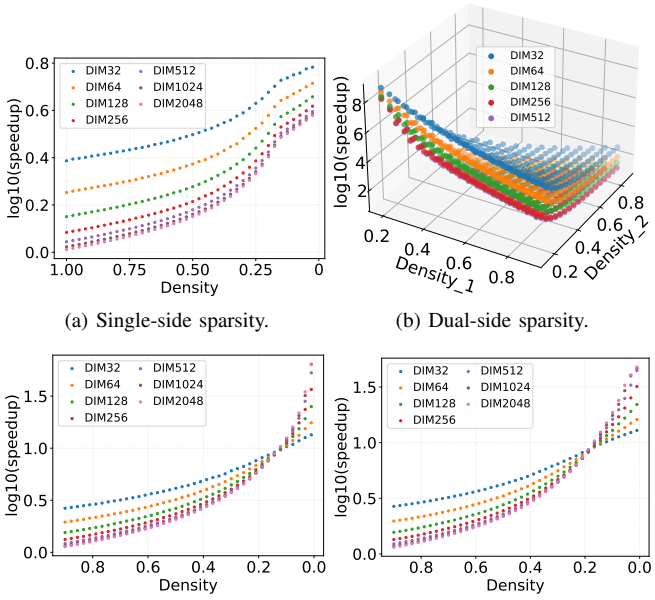
We implemented the proposed DenSparSA design and the baseline systolic array design, originally used by Tensor Processing Units (TPUs), in RTL. The designs were synthesized at 400MHz using OpenROAD and Innovus with the NanGate 45-nm open-cell library. We assume 128×128 PE configuration across all designs. In DenSparSA, each processing element (PE) includes a FIFO with a depth of 6, and the array works by 4×4 partition for Vector Filtering. The multiply-accumulate (MAC) units are designed to support BF16 or FP32 multiplication and accumulation. In the following, we compare the proposed DenSparSA with the original systolic array (SA) and related state-of-the-art approaches.

B. Performance and Cost Breakdown

1) Speedup Performance:

We evaluate the performance of DenSparSA in terms of speedup over a conventional TPU-like systolic array. Matrix with dimensions ranging from 32 to 2048 are taken into account. The relation between speedup and dual-side sparsity is shown in Fig. 6, with speedup ratios expressed in log forms. The co-effects of sparsity on both sides are shown in Fig. 6(b). For the quantitative showcase, Fig. 6(c) and Fig. 6(d) show the cases of having equal sparsity on two sides and having sparsity x on one side with sparsity range $[x - 0.2, x + 0.2]$ on another side. For sparsity ranging from 50% to 95%, DenSparSA achieves 1.9× - 5.5× gmean speedup with single-side sparsity and 2.6× - 17.5× gmean speedup with dual-side sparsity, as compared with baseline design of the systolic array.

As can be seen in both figures, there's a speedup for matrices with smaller dimensions even when the sparsity is very low. This is due to the effect of solely partitioning since a smaller array size brings about lower loading latency, a side effect of our partitioning that is intended for filtering vectors. With the decrease in matrix density, the speedup effect converges to the case of matrices with larger dimensions, as the advantage brought about by just partitioning becomes trivial. For single-side sparsity displayed in Fig. 6(a), the growing trend of speedup rate is retarded as sparsity grows high. This is because the denser one of the two matrices has to be compressed by filters to achieve higher speedup, and when the sparsity is high, the time cost for matrix compression exceeds the time cost for computing the compressed matrix, since the former time cost is fixed given the length of the dense input matrix and the latter one goes down when the sparsity of another input matrix is very high. Therefore, there is a saturation on the overall speedup effect. Allocating more hardware resources for data compression could help with the saturation, but would not be cost-effective for DNN-level sparsity. Such saturation is not observed for dual-side sparsity.



(a) Single-side sparsity. (b) Dual-side sparsity.

(c) Dual-side with equal sparsity. (d) Dual-side with varying sparsity.

Fig. 6. Speedup under different situations of sparsity.

2) Cost and Overhead Breakdown:

Table II shows the area breakdown of DenSparSA. The analysis of area usage is focused on the computing array since no modification is made to the structure of cache and external memory. The total overhead is 15% for BF16 multiplier and 12% for FP32 multiplier. The hardware overhead mainly comes from three sources: additional buffers, additional control logic in PEs, and data rearrangement, as reflected in the increased area in PEs and input modules. Although each PE gets additional logic for buffering, the overall area overhead is limited, since sequential cells account for a rather small proportion of the whole area, and the additional control logic is not complex. The overhead of buffered PE is further mitigated as larger multipliers are applied, since the growth of hardware cost for buffers, i.e., MUXs, comparators, and registers are in a roughly linear relation with data width, making it hard to catch up with the growing hardware cost of multipliers.

TABLE II
Area Breakdown Compared with Baseline Design.

Breakdown (mm^2)		**BF16**				**FP32**			
		TPU-SA		DenSparSA		TPU-SA		DenSparSA	
Array	Reg	102.1	14.3	116.4	25.2	186.7	27.4	207.5	36.6
	MAC		78.6		78.8		136.2		139.5
	Other		9.2		12.5		23.1		31.4
Input		1.5		2.8		2.8		4.8	
Output		1.7		1.7		3.2		3.2	
Other		6.2		7.5		10.8		12.96	
Total		111.5		128.4(+15%)		203.5		228.5(+12%)	

Table III shows the power breakdown of DenSparSA. Still, buffered PE contributes to the majority of the overhead, and the power overhead is more significant than the area overhead since buffers account for a larger proportion of power consumption. The overhead caused by buffers is partly diluted by the stage registers in multipliers pipelined to achieve higher clock frequency but is still a drawback

if no benefit could be made from accelerating the workload, in other words, if the workload is dense.

Unlike area statistics, the total power is reported respectively in sparse mode and dense modes. In sparse mode, routers and buffers of the PEs are turned on to facilitate computation on sparse matrices. In dense mode, unnecessary components as discussed in Section V-C are turned off by cutting off the dynamic power consumed for handling sparsity. Decoupling is quite effective in reducing power overhead for dense workloads. By clock-gating, power overhead in dense workloads is reduced to 12% and 5% for BF16 and FP32 multipliers. The feature of decoupled sparsity support put our design at an advantage when switched back to a dense workload, which will be further discussed in Section V-C through comparison.

TABLE III
POWER BREAKDOWN COMPARED WITH BASELINE DESIGN.

Breakdown (W)	BF16			FP32		
	TPU-SA	DenSparSA		TPU-SA	DenSparSA	
		Sparse	Dense		Sparse	Dense
Array	13.7	22.9(+52%)	14.4(+6%)	50.5	61.4(+21%)	52.0(+3%)
Input	2.4	3.7		4.7	7.0	
Output	0.8	0.8		1.3	1.3	
Other	0.6	0.7		0.9	1.0	
Total	17.5	26.1(+49%)	19.6(+12%)	57.6	71.2(+24%)	61.2(+5%)

C. Comparison among Existing Solutions

We compare DenSparSA comprehensively with existing solutions for accelerating sparse matrix calculation. The comparison is based on two principles. First, comparison is done one-by-one with each design according to the datatype and evaluation metrics proposed by each respective paper. This is because evaluation metrics vary from accelerating DNN workloads to solely accelerating different matrix benchmarks, and we lack information to derive each design's performance for a generalized theme. Second, statistics of performance, power, and area are normalized to a baseline design of a TPU-like systolic array. This is because even basic statistics such as the cost of a TPU-like baseline PE mismatch among different works, and a consistent comparison of absolute values between each work would be hard to conduct given such mismatch. Therefore, the speedup and overhead ratios over a TPU-like baseline design, as reported in all compared works, are chosen as a common measure for reference.

Fig. 7 shows the comparison of computing efficiency in sparse scenarios. Area efficiency, as displayed in Fig. 7(a), is calculated as speedup ratio over baseline TPU divided by area overhead, and the same is done for power efficiency. Workloads include dual-side dense (DD), single-side sparse (SD), and dual-side sparse matrix (SS) multiplication with sparsity on the order of 10^{-1}, the common sparsity for DNN-related computation, and a geometric mean is calculated for designs capable of both SS and SD workloads. As for the displayed bar chart, the efficiency is normalized to the better design between each comparison pair, with the original values of $(Perf/Area)$ and $(Perf/Power)$ marked on top of each bar. DenSparSA provides better area efficiency than SIGMA, DSSA and Mentha, and is slightly lower than Trapezoid. As for power efficiency, DenSparSA is 24% and 32% ahead of SIGMA and Mentha, and 18% and 7% less efficient than DSSA and Trapezoid. From the comparison above, we can conclude that DenSparSA delivers a competitive (0.82×-1.32×) computing efficiency among existing sparse matrix accelerators.

Figure 8 shows the comparison of computing efficiency in dense scenarios. For dense workloads, DenSparSA delivers the best (1.17×-2.28×) computing efficiency in terms of both area utilization and power consumption. The advantage in area utilization is due to the originally small area overhead, as the extra logic is quite compact and buffers do not account for a large proportion of chip area. On the other hand, the advantage of DenSparSA in power efficiency is largely due to the "pluggable" feature of decoupled modification upon baseline design. Since a major contributor of power overhead could be turned off, the power efficiency is rather close to the baseline model. In our evaluation, only clock gating for buffered PEs is modeled and simulated, which already provides a meaningful drop in power overhead. DSSA and Trapezoid provide slightly higher computing efficiency over DenSparSA in sparse workloads, but are outperformed by DenSparSA by a larger gap in dense workloads, indicating that DenSparSA may hold better versatility in scenarios where dense and sparse workloads are mixed together, and that DenSparSA maintains a better balance between leveraging sparsity acceleration and keeping efficient in conventional situations.

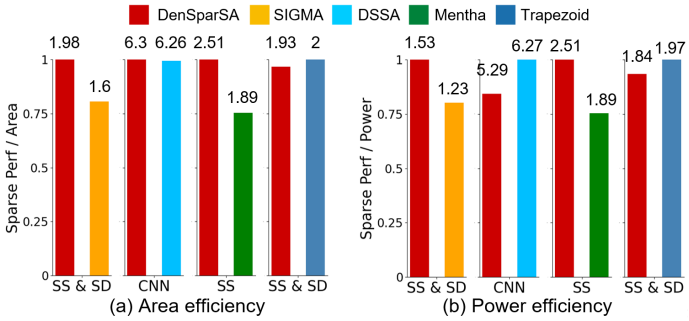


Fig. 7. Comparison with Existing Solutions in Sparse Scenarios

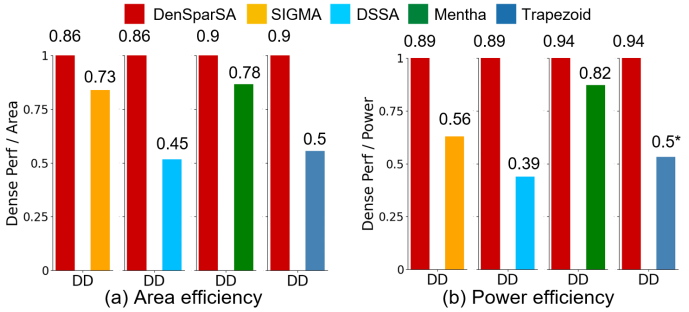


Fig. 8. Comparison with Existing Solutions in Dense Scenarios

VI. CONCLUSION

This paper presents DenSparSA, a systolic array centralized architecture that leverages general sparsity for speedup in matrix multiplication. The dataflow and architecture are proposed to shorten the data passage time and bring about speedup benefits, for which a series of hardware modifications are made upon the classic design of a 2D systolic array. The microarchitecture and circuit support for sparsity handling is compact and decoupled from the original design of the systolic array, therefore clock-gating could be applied to minimize power overhead when switched back to dense workloads. Compared with existing solutions, our design achieves competitive (0.82× - 1.32×) performance in general sparse matrix multiplication while holding a better balance between dense and sparse workloads with better (1.17× - 2.28×) computing efficiency for dense scenarios, maintaining the benefit from high power efficiency of original systolic arrays in dense scenarios.

REFERENCES

[1] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[2] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 63(7):67–78, 2020.

[3] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. Sparse-tpu: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[4] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Annual Design Automation Conference*, 2017.

[5] Jie Wang, Licheng Guo, and Jason Cong. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *FPGA*, 2021.

[6] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. *Procedia Computer Science*, 51:2453–2462, 2015.

[7] David Emms, Edwin R Hancock, Simone Severini, and Richard C Wilson. A matrix representation of graphs and its spectrum as a graph invariant. *arXiv preprint quant-ph/0505026*, 2005.

[8] EI Yakubovich and DA Zenkovich. Matrix approach to lagrangian fluid dynamics. *Journal of Fluid Mechanics*, 443:167–196, 2001.

[9] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

[10] Hyeonuk Sim, Jooyeon Choi, and Jongeun Lee. Spartann: Sparse training accelerator for neural networks with threshold-based sparsification. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 211–216, 2020.

[11] Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel Emer. Highlight: Efficient and flexible dnn acceleration with hierarchical structured sparsity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1106—-1120, 2023.

[12] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 3.2 the a100 datacenter gpu and ampere architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 48–50, 2021.

[13] Wenhao Sun, Deng Liu, Zhiwei Zou, Wendi Sun, Song Chen, and Yi Kang. Sense: Model-hardware codesign for accelerating sparse cnns on systolic arrays. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 470–483, 2023.

[14] Ziheng Wang. Sparsert: Accelerating unstructured sparsity on gpus for deep learning inference. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 31–42, 2020.

[15] Minjin Tang, Mei Wen, Yasong Cao, Junzhong Shen, Jianchao Yang, Jiawei Fei, Yang Guo, and Sheng Liu. Mentha: Enabling sparse-packing computation on systolic arrays. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[16] Jianlei Yang, Wenzhi Fu, Xingzhou Cheng, Xucheng Ye, Pengcheng Dai, and Weisheng Zhao. S 2 engine: A novel systolic architecture for sparse convolutional neural networks. *IEEE Transactions on Computers*, 71(6):1440–1452, 2021.

[17] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella1, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 58–70, 2020.

[18] Yifan Yang, Joel S. Emer, and Daniel Sanchez. Trapezoid: A versatile accelerator for dense and sparse matrix multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 931–945, 2024.

[19] Zhengbo Chen, Qi Yu, Fang Zheng, Feng Guo, and Zuoning Chen. Dssa: Dual-side sparse systolic array architecture for accelerating convolutional neural network training. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP)*, pages 1–10, 2022.