HIRAC: A Hierarchical Accelerator with Sorting-based Packing for SpGEMMs in DNN Applications

Hesam Shabani*, Abhishek Singh*, Bishoy Youhana*, Xiaochen Guo*†
*Lehigh University †Qualcomm, Inc., USA
{hes318, abs218, bby223, xig515}@lehigh.edu

***Abstract*—The state-of-the-art deep neural network (DNN) models use pruning to avoid over-fitting and reduce the number of parameters. In order to improve storage and computational efficiency, only nonzero elements are stored, and their locations are encoded into a sparse format. Sparse General Matrix Multiplication (SpGEMM) is the kernel computation of DNN-based applications. One challenge of computing SpGEMM is to avoid multiplying zero elements while keeping hardware utilization high in hardware accelerators that consist of processing element (PE) arrays. Prior work tackling this challenge typically requires complex interconnection networks, which adds high area and energy costs.**

This work proposes a HW/SW co-design architecture to compute SpGEMM efficiently without requiring complex interconnection networks. A novel fast packing algorithm, SorPack, is proposed to convert a sparse matrix into a dense matrix that increases PE utilization. The key idea is to sort columns and rows inside each submatrix based on the number of nonzero elements. The goal is to keep the partial sums that need to be added together close to each other, hence can be added locally and avoid the use of complex interconnection networks. In addition, a new tile-based hierarchical architecture, HIRAC, is proposed to provide a scalable system that maximizes the parallelism of the PEs. The HIRAC architecture consists of a novel PE array design and interconnection network tailored for DNN applications. The SorPack algorithm complements the HIRAC to further improve hardware utilization and overall system performance. Based on the evaluation results, HIRAC achieves an average of 3.2× speedup on a single layer of DNN as compared to the state-of-the-art sparse DNN accelerator SIGMA. In addition, HIRAC has a 9.5% area reduction and a 32% power reduction as compared to SIGMA. An end-to-end evaluation on a DNN model shows an 8.2× runtime reduction over the TPU.

I. Introduction

DNN models are widely used in artificial intelligence (AI) applications such as speech and image recognition, robotics, self-driving, etc. DNN models have significantly improved accuracy on many AI tasks as compared to previous generations of neural network models. However, these models require many parameters to be trained and stored, which demands high computational complexity and considerable data movement between on-chip and off-chip.

One of the crucial steps for widely deploying DNNs models in AI systems is to design efficient hardware architectures. *DNN accelerators* [6], [13], [20] have been proposed as a promising solution to tailor the hardware to the specific need of DNN computation. These accelerators typically consist of specialized PEs and application-specific on-chip networks. For example, tile-based architectures are well studied and applied frequently in DNN accelerator designs [4], [5], [20], [22].

From the application side, one way to reduce the computation complexity in the state-of-the-art DNNs models is to decrease the number of parameters in the weight matrices by using different pruning methods to eliminate small weights, making the weight matrix sparse. In addition, applying non-linear layers (e.g., pooling, ReLU, and dropout) to the activation feature maps can make these activation matrices sparse as well. As a result, SpGEMM becomes an important computational kernel prevalent in DNN workloads, which is typically a performance bottleneck when these applications run on GPUs and accelerators.

The state-of-the-art accelerators such as TPU [13] leverage systolic arrays as the compute platform to speed up dense GEMM operations. However, TPU does not support sparsity properly and typically suffers from low utilization of the PEs for SpGEMM computations. For simple hardware design, systolic arrays restrict data movement to one-directional for gathering the partial sum results. The PEs have low utilization when zeros are mapped to the PEs.

SIGMA [20] proposed to handle the limitations of TPU for SpGEMMs in DNN applications by leveraging a hierarchical design to interconnect different PEs to improve the parallelism of the multipliers that are computing nonzero elements. However, SIGMA adds considerable area overhead and power consumption because of the high flexibility and complexity of the interconnection network in the non-blocking distribution and reduction networks.

Sparse-TPU [7] proposed packing algorithms and index matching mechanisms to efficiently process sparse matrices in a condensed format to process sparse matrix-vector multiplication in a systolic array architecture. However, Sparse-TPU is burdened by the greedy search algorithm's complexity when combining candidate columns.

This work proposes a simple and fast packing algorithm, SorPack, to efficiently convert sparse matrices to denser ones. The SorPack algorithm takes advantage of partitioning and sorting before packing the sparse matrices. The uniqueness of SorPack is the sorting of columns and rows inside each submatrix based on the number of nonzero elements. This helps to keep the partial sums that need to be merged close to each other, improving the possibility of reducing the partial sums before sending them to the network, which reduces data movement. In addition, the algorithm is simple, which enables

it to be applied to activation matrices online.

Along with SorPack, a novel SpGEMM accelerator, HIRAC, is proposed to take advantage of the optimization result of SorPack. Statistics gathered from matrices with targeted sparsity range are used to make design decisions. HIRAC is an innovative hierarchical design to tackle the challenges of SpGEMM computations. The innovations of HIRAC include a new PE subarray design with same cycle mergers and a simple interconnection network that moves the data toward one direction based on the desired dataflow. Combining SorPack with HIRAC improves overall hardware utilization and system performance.

Based on the evaluations, HIRAC has greater speedup, smaller area, and lower power consumption as compared to SIGMA. Area saving of HIRAC is due to the simple interconnection network and the hierarchical design of the PE subarray. The SorPack algorithm enhances the power saving and performance improvement of HIRAC.

The key contributions of this paper include the following:

- An HW/SW co-designed architecture to perform SpGEMM efficiently for state-of-the-art Sparse DNN applications.
- The SorPack algorithm is a novel fast packing algorithm to efficiently convert sparse matrices to dense ones to improve PE utilization and system performance.
- The HIRAC, a hierarchical accelerator for Sparse GEMM with low network complexity and high system performance.
- Evaluations on representative matrices in DNN workloads to cover different SpGEMMs irregularity and sparsity ranges.
- Design space explorations of the HIRAC and SorPack to reduce hardware complexity and improve system performance.

II. Background and Related Work

This work's key ideas center around preprocessing sparse matrices and designing tailored hardware accelerators for DNN and SpGEMM. Before introducing the proposed architecture, this section discusses the background and related work on sparse matrix preprocessing and accelerators for DNNs and SpGEMMs.

A. Preprocessing Methods

Recent work [7], [14] proposed offline preprocessing packing algorithms to condense sparse input matrices into denser ones and then mapped them onto the systolic array to perform SpGEMMs. Offline preprocessing can typically improve computation efficiency. However, the latency overhead can be high for large inputs.

A column combining method has been proposed by Kung et al. [14], which packed sparse matrices and used them in a systolic array-like architecture. Kung's work presented efficient bit-serial multipliers and accumulators in the systolic array. It improves system scalability by using a given number of parallel buses on each column of the 2D systolic array. However, the design only supports integer arithmetic. Therefore, the system cannot support high-precision applications. Sparse-TPU [7] can efficiently handle sparse matrices in a condensed representation using an index matching mechanism and value holding functionalities on a systolic array-based system. It was inspired by the column combining preprocessing method in [14] and proposed three packing algorithms based on a greedy algorithm. In the Sparse-TPU work, a collision-aware packing algorithm has been proposed to increase the matrix density and hardware utilization. This algorithm selects sparse columns to be combined into a denser column without changing the row positions of nonzero elements. A collision happens between the selected columns that have one or more nonzero elements with the same row indices. The collision-aware algorithm is used as a baseline for evaluations in Section VII. This algorithm first partitions the rows in the matrix and then performs collision-aware column merging. The final step is to partition the columns in the matrix. This column merging algorithm is an iterative algorithm, which can be time-consuming for the large matrices in the DNN applications. Also, the dataflow in Sparse-TPU is designed for sparse matrix-vector multiplication (SpMV), which is not optimized for SpGEMM in DNN applications [27]. The proposed SorPack algorithm is a non-iterative method and thus exhibits lower latency overhead as compared to the collision-aware packing algorithm [7].

In addition, recent work [11] proposes online preprocessing methods by implementing dedicated hardware units for skipping the zero values in the input feature map. It searches for nonzero elements in each lane within the search window during the runtime. However, the search process is limited by the search window size, which increases hardware complexity to address the overlap between search windows. Additionally, inter-lane search requires multiple multiplexers, which increases hardware complexity and area. The proposed SorPack algorithm will run on the CPU, hence does not add hardware overhead for packing matrices.

Besides the above mentioned preprocessing techniques, various compressed storage formats (i.e., CSR [15], TJDS [18], ELLPACK [31]) have been used to store nonzero values and their corresponding indices. These compression methods significantly reduce storage requirements, especially in highly sparse matrices. However, predicting the locations of nonzero elements in the input matrices with matching indices in SpGEMM computation is difficult without decoding the sparse format first [28]. Thus, index matching increases the latency for data fetching.

Sorting-based compression methods like TJDS, SELL-C-σ format [2] applies sorting and reordering in their compression method. The goal of using the sorting step is to improve cache utilization. The proposed SorPack algorithm also uses a sorting step, but the goal is to keep the partial sums that need to be added together close to each other to reduce the data movement on the global wires. This makes the partial sum merging more efficient in the proposed sparse DNN accelerators.

B. Accelerator Designs for DNNs and SpGEMMs

Accelerators should pay attention to providing system efficiency by leveraging adequate dataflow to perform SpGEMM computation. Many hardware accelerators have been proposed to speed up DNNs and SpGEMMs. There are three types of dataflows commonly adopted by these accelerators to optimize for data reuse: inner product [8], [20], outer product [9], [30], and row-wise product (column-wise is similar) [24], [29]. The inner product dataflow computes the dot product of a row and a column of the input matrices for one output matrix element before moving on to the next output matrix element, which achieves good output reuse but poor input reuse. The inner product works better for relatively denser input matrices (DNN sparsity ranges) [29], where a row and a column of the input matrices have nonzero elements with the same indices. The outer product computes the tensor product of a column and a row of the input matrices before moving on to another pair of a column and a row. The outer product can have better input reuse but worse output reuse as compared to the inner product, which typically favors sparser input matrices [9], [30]. The row-wise product (a.k.a. Gustavson's algorithm) takes one row of the first input matrix to compute a row of the output matrix before moving on to the next row. As compared to the outer product, row-wise typically generates less memory traffic and has simpler merging operations for partial sums in highly sparse matrices such as matrices in graph analytics applications [29]. The inner product is chosen for the proposed HIRAC due to the relatively higher nonzero density in the matrices of the DNN applications. In SpGEMM computation, a weight matrix (KN in Fig. 1) is multiplied with an activation feature map matrix (MK). The weight matrix is called *stationary matrix* because the weights are mapped to and stay in the PEs until the corresponding elements in the feature map matrix are multiplied by the weights. The activation feature map matrix is called *streaming matrix* because its elements are sent through different PEs columns.

The hierarchical interconnection network is a common feature in the accelerators, which can improve parallelism and reduce data movement. Eyeriss v2 [5] presents a hierarchical mesh network for a small mobile convolutional neural network that supports sparse activation feature maps and weights stored in a compressed format. The implementation cost of accumulation increases linearly in the architecture design. Google's TPU [13] is a collection of multiplier-accumulator (MAC) units arranged in a 2D grid to speed up dense GEMM computation. However, TPU cannot support sparsity properly due to its rigid structure, which faces the challenge of maintaining high PE utilization. Facebook (Meta) designed a specialized computing platform to accelerate recommendation systems and the natural language processing [1], which includes six power-efficient accelerator cards and a single-socket host CPU, enabling them to perform complex and large models for their targeted workloads that cannot efficiently run on CPUs. Simba [22] is proposed as the first chiplet-based DNN accelerator. Each of the Simba chiplets can be used as an inference accelerator, while multiple Simba chiplets can be packaged together to deliver data-center-scale throughput across a wide range of workloads. In a Simba-like architecture, communication latency is crucial in improving large-scale systems' performance and energy efficiency. This proposed HIRAC is inspired by the hierarchical design in Simba. SIGMA [20] was inspired by Simba, which proposed a hierarchical design to compute SpGEMM using a set of Flexible Dot Product Engine (Flex-DPE) units to construct a Flexible Dot Product Unit (Flex-DPU). SIGMA maps only nonzero elements of the stationary matrix to the Flex-DPEs and uses separate networks to improve the flexibility of data movement as well as hardware utilization and performance. SIGMA uses the Benes topology for non-blocking data distribution and a novel tree-based Forward Adder Network (FAN) as the reduction network to enable efficient irregular SpGEMM computations for DNNs. These design choices enabled SIGMA to handle the sparsity efficiently. The flexibility of SIGMA improved weight distribution and reuse, which decreased the implementation cost of the reduction network by designing the spatial accumulation with FAN as compared to the Eyeriss v2 [20]. However, the interconnection networks in SIGMA add significant area and energy overhead [21] due to its complexity. Additionally, the PE utilization of SIGMA is determined by the sparsity of the streaming matrix. As a result, the SIGMA design does not perform as efficiently when streaming a sparser matrix compared to a denser one [20]. Moreover, SIGMA requires preprocessing to find the nonzeros before mapping the stationary matrix, which also adds latency and energy overhead. SIGMA is the closest work to the proposed HIRAC for the same kind of targeted applications. HIRAC uses a simple interconnection network to reduce the area and energy overhead of SIGMA while achieving an average of $3.2\times$ performance improvement on a single layer of DNN computation.

In addition to the related work discussed above, other recent studies target alternative applications (*e.g.,* language processing) that also involve sparse matrices and SpGEMM computations. These accelerators typically have different dataflows or targets on sparse matrices after structured pruning. SPOTS [23] presents a hardware accelerator using flexible systolic array architecture for sparse convolutional neural networks (CNNs), which supports structured pruning. Structured pruning sets weights to zero at specific block sizes or defined locations, which might degrade model accuracy in some applications. Sanger [17] proposed and evaluated an HW/SW co-design for the BERT language model leveraging a reconfigurable systolic array architecture. In Sanger, the SW component applies dynamic structured pattern pruning in the attention matrix. S^2 Engine [27] proposed a new systolic architecture and a collective element (CE) array for maximizing the data reuse in sparse CNNs. Some other work proposed row-wise accelerator designs for highly sparse matrices in which the nonzero elements range from $10^{-6} - 1\%$ [24], [29]. However, the sparsity ranges go beyond the scope of typical DNN applications. Therefore, these designs are not suitable for DNN

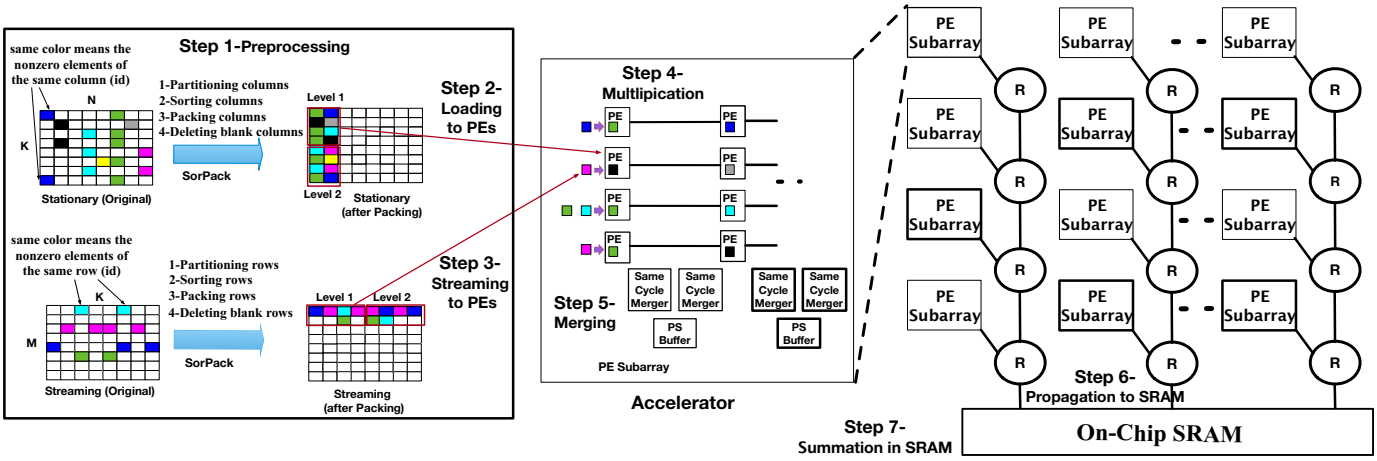

Fig. 1. An overview of proposed HW/SW co-design architecture composed of preprocessing and accelerator parts.

applications.

III. System Design Overview

The goal of the proposed HW/SW co-design architecture is to take advantage of the interaction between the software (*i.e.*, the preprocessing algorithm) and the hardware (*i.e.*, the accelerator) to improve the system performance and reduce the hardware complexity. In general, some design decisions in the preprocessing algorithm have to be made considering the hardware organization and vice versa.

The software part of the proposed architecture focuses on packing the sparse matrices in the DNN workloads using the proposed SorPack algorithm. The hardware part dedicates to executing SpGEMM efficiently through mapping the nonzero elements to PEs and merging the corresponding partial sums through merging units in the HIRAC in a pipelined manner.

For each layer of a DNN architecture that requires a SpGEMM computation, a weight matrix is multiplied with an activation matrix. Fig. 1 shows an illustration of performing a SpGEMM operation on the proposed HW/SW co-designed architecture, which consists of software preprocessing and hardware acceleration. In the preprocessing step (step 1), the SorPack algorithm is applied to both streaming and stationary matrices to condense them. Either the weight or activation matrix can be stationary or streaming. For the weight matrices in DNN inference, the preprocessing step can be performed offline. In step 2, a packed stationary matrix is loaded to the PE subarray. In step 3, a packed streaming matrix is sent to the PE array. In step 4, streamed elements to the PEs are multiplied with the nonzero stationary elements. In step 5, the partial sums are merged within the PE subarrays. In step 6, partial sums are propagated from the PE subarrays to the SRAM. In step 7, partial sums are added to corresponding entries in the SRAM. The following sections will discuss each of these steps in detail.

IV. The SorPack Algorithm

One goal of a packing algorithm is to convert a sparse matrix to its denser representation that contains fewer zeros to increase the PE utilization in a PE array. However, a denser representation can challenge the merging of partial sums that share the same indices. These partial sums can be produced by different PEs or, at a different time, far away from each other.

In this work, the SorPack algorithm is proposed, which aims to improve packing density while keeping partial sums to be merged close to each other spatially and temporally. In the SpGEMM operation, a streaming matrix with the size of $M \times K$ is multiplied by a stationary matrix with the size of $K \times N$. The dot product of each row of the streaming matrix and column of the stationary matrix has to be computed. In each dot product, nonzero elements with matching indices in the streaming matrix row i and stationary matrix column j contribute to summing up the element C_{ij} in the result matrix C. In order to avoid the additional complexity of index matching for multiplication, the proposed SorPack algorithm does not change the column position of the nonzero elements in the streaming matrix or the row position of the nonzero elements in the stationary matrix. During the streaming process, each column of the streaming matrix is sent to the row of the PE array that stores the corresponding row of the stationary matrix. Therefore, every nonzero streaming element must be multiplied by every nonzero stationary element within the row of the PE array. Index matching is required only for summing up the partial sums.

The nonzero elements in the same stationary column or the same streaming row should not be split into different locations far away from each other to keep the partial sums of a group close to each other for merging. For example, assuming all the nonzero elements in the stationary matrix are directly packed toward the left within each row, the *column split distance* after packing of two nonzero elements in the same column is the difference in the number of nonzero elements in their corresponding rows on the left of this column. The SorPack algorithm keeps the row/column split distance small by sorting and rearranging the rows in the streaming matrix and the column in the stationary matrix based on the number of nonzero elements in that row/column before packing.

As shown in Algorithm 1, SorPack has four steps (lines

39-45), which include two steps before packing to limit and reduce the column/row split distance. This algorithm uses a stationary matrix as an example, while the following explanation covers both stationary and streaming matrices to make it general. The four steps include: 1) *Partitioning* the columns and rows of the matrix into $P \times P$ sub-matrices (lines 3-15). Further column and row position rearrangements are limited within the partition. The purpose of this partitioning step is to limit the column/row split distance to be smaller than P, which is a configurable parameter. 2) *Sorting* columns of the stationary matrix and rows of the streaming matrix within each partition based on the number of nonzero elements in the columns and rows respectively from large to small (lines 17-23) and rearrange their positions (lines 24-28). 3) *Packing* the stationary and streaming matrices by moving nonzero elements next to each other towards the left and the top within the rows and columns, respectively, within each partition (lines 29-34). 4) *Deleting* all-zero columns in the stationary matrix and all-zero rows of the streaming matrix after packing, which helps to increase matrix density and PE utilization (lines 35-38).

Fig. 2 shows an example of applying the proposed SorPack. In this example, the partition size P is 4. Partitioned tiles in the same row of the stationary matrix or the same column of the streaming matrix are called a *level*. When a PE subarray row cannot fit the entire stationary level, tiles in that level are loaded to the next row. The tiles in a streaming matrix level are sent to all PE subarrays that store the corresponding stationary matrix level. After sorting, dense rows in the streaming matrix are placed on the top of each partition, whereas dense columns in the stationary matrix are placed on the left of each partition. Because of the sorting, the nonzero elements in the dense rows and columns stay in the same row/column (e.g., I, E, J, K in the streaming matrix and b, s, t in the stationary matrix). Both matrices' size is reduced after deleting all blank rows and columns, which results in memory space saving and runtime reduction.

Impact of the SorPack Steps on the Hardware Design. A critical goal of the SorPack algorithm is to increase the merging of partial sums at the same cycle and column, which does not require hardware to buffer the partial sums and their corresponding labels for comparing and merging. Increasing the percentage of the same-cycle, same-column merging can improve performance, reduce data movement, and reduce hardware complexity.

*a) **Impact of Partitioning**:* The *Partitioning* step determines the maximum column/row split distance, which also influences packing density, performance, and energy consumption. Smaller P size limits the column/row split distance and increases the same-cycle/col merging percentages. As a result, the number of partial sums to be moved on the network might be reduced, and the merging latency and energy efficiency can be improved. However, a smaller P size reduces matrix density after packing, which leads to bigger matrices being loaded and streamed. This can increase the loading and streaming latency and energy. After applying the packing algorithm, the ratio of the number of nonzeros and the product of the number of columns and rows is defined as the *condensing factor*. Thus, the higher condensing factor is better since it demonstrates higher condensing after applying the packing algorithm. Fig. 3 shows the impact of partitioning sizes on merging and condensing factor.

*b) **Impact of Sorting**:* The uniqueness of the SorPack is the application of the *Sorting* step. This step helps to increase the same-cycle/col merging because the sorting is based on the number of nonzeros. After packing, the splitting distances are minimized. Table I presents the same-cycle/col merging percentage comparison with and without sorting for a variety

Algorithm 1: SorPack Algorithm

```
1  Input: Stationary matrix M, Partitioning size pSize
2  Output: Sorted and packed partitions
3  Function PARTITION(M, pSize):
4      nRows, nColumns = M.shape
5      for i ← 0 to nColumns − 1 do
6          for j ← 0 to nRows − 1 do
7              tmpSubPartition = {}
8              aRowIndex = min(pSize, nRows - j)
9              aColIndex = min(pSize, nColumns - i)
10             for counter ← 0 to aRowIndex do
11                 tmpSubPartition += M[j + counter][i: i +
                       aColIndex]
12             j += pSize
13             partitions += tmpSubPartition
14         i += pSize
15     return partitions
16 Function SORT(subPartition):
17     nRows, nColumns = subPartition.shape
18     numNonZeroElement[0:nColumns - 1] = 0
19     for each column ∈ subPartition do
20         for each element ∈ column do
21             if element ≠ 0 then
22                 numNonZeroElement[column.index]++
23     sIndices = quickSort(numNonZeroElement)
24     tmpArray = subPartition
25     j = 0
26     for each i ∈ sIndices do
27         subPartition[i] = tmpArray[j]
28         j += 1
29 Function PACK(subPartition):
30     nRows, nColumns = subPartition.shape
31     for i ← 0 to nRows − 1 do
32         for j ← 0 to nColumns − 1 do
33             if subPartition[i][j] = 0 then
34                 delete(subPartition[i][j])
35 Function DELETE-COLUMNS(subPartition):
36     for each column ∈ subPartition do
37         if column.isBlank() then
38             delete(column)
39 Function SorPack(M, pSize):
40     partitions = PARTITION(M, pSize)
41     for each subPartition ∈ partitions do
42         SORT(subPartition)
43         PACK(subPartition)
44         DELETE-COLUMNS(subPartition)
45     return partitions
```

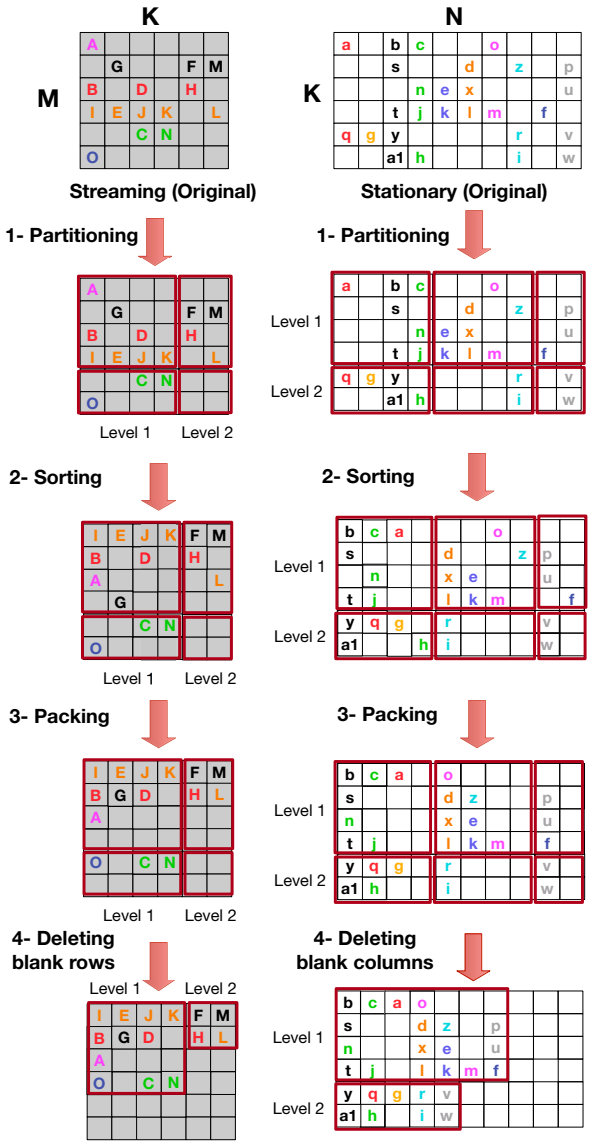

Fig. 2. An example of applying the SorPack in the streaming and stationary matrices.

TABLE I
THE SAME-CYCLE/COL MERGING PERCENTAGE COMPARISON OF SORPACK WITHOUT SORTING VS. WITH SORTING.

Percentage of Zero Elements	Without Sorting	With Sorting
Stationary=50%, Streaming=50%	8.7%	20.9%
Stationary=50%, Streaming=70%	7.6%	12.8%
Stationary=70%, Streaming=70%	5.8%	9.8%
Stationary=80%, Streaming=90%	1.3%	2.1%

Both stationary and streaming matrices are 100×100, and P is 4.

of matrix sparsity ranges. As shown in the table, the *Sorting* step increases the percentage of same-cycle/col merging of partial sums, especially when matrices are relatively denser.

V. THE HIRAC DESIGN

An important design consideration of HIRAC is to efficiently locate partial sums to be accumulated. HIRAC requires an index labeling to find partial sums that need to be added

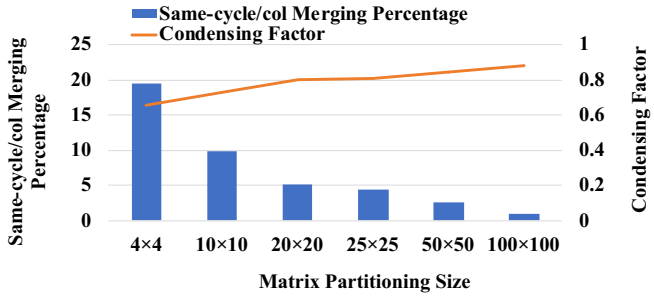

Fig. 3. Condensing factor and the percentage of partial sums with the same-cycle/col merging for different matrix partitioning sizes. The result is from the 100×100 matrix size, and the sparsity of the stationary and streaming matrices are 50% and 70%, respectively.

together. The label consists of the column id of the original stationary matrix and the row id of the original streaming matrix. Accumulation occurs in the merging units, and the partial sums can only accumulate together when labels match.

Based on a study of the target matrices, 42% of the partial sums do not need to be merged when the sparsity of stationary and streaming matrices are 50% and 70%[1], respectively. This number increases for sparser matrices.

Fig. 4 shows an example of a preprocessed streaming and stationary matrix after applying SorPack. The elements in the streaming matrix are the row ids of the original streaming matrix, and the elements in the stationary matrix are the column ids of the original stationary matrix. This figure shows when and where the PEs produce each partial sum. The merging can take place under three situations 1) at the same cycle, same column (e.g., (18,3)), 2) at different cycles but the same column (e.g., (1,1)), and 3) across different columns (e.g., (19,2)). The first situation is preferred for a simple hardware implementation of accumulations. The proposed HIRAC design includes a novel PE array, interconnection network, and On-Chip memory (SRAM) to cover all of the above mentioned situations efficiently.

A. PE Array

The PE array is hierarchically organized into multiple PE subarrays. Each PE subarray has columns of PEs, *Same Cycle Merger*, and Partial Sum Buffer (*PS Buffer*) as shown in Fig. 5. The Same Cycle Merger is a dedicated unit to handle same-cycle/col merging efficiently without requiring any buffering, which takes advantage of the SorPack that maximizes the same-cycle/col merging.

1) PE Columns: In a column of PEs, each PE computes multiplications when labels match. The results are then sent to the same cycle merger dedicated to this column or the corresponding router. If a stationary column id does not match with any other column id in the same PE column, no partial sum produced by this PE would be able to merge with other partial sums in this column. These partial sums are sent to the router, and the corresponding PEs can be identified when the stationary matrix is loaded.

[1]These numbers are the middle of targeted sparsity ranges for both matrices.

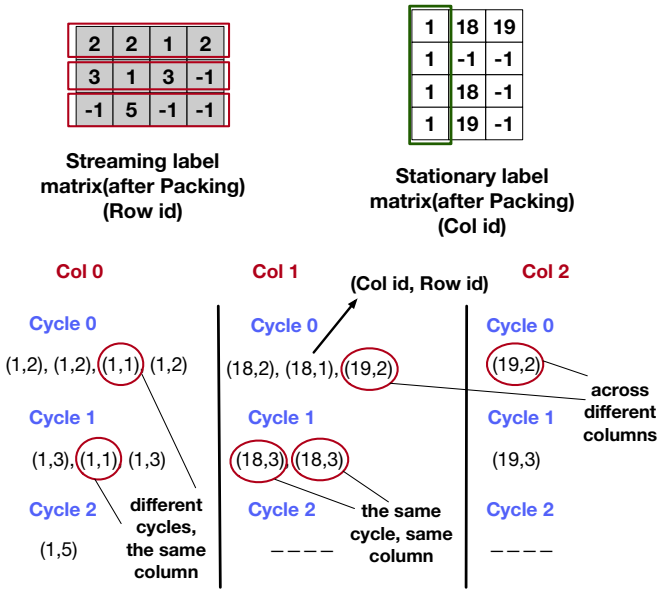

Fig. 4. Examples of partial sums to be merged produced under different situations. A -1 col or row id means a zero element that needs to be skipped.

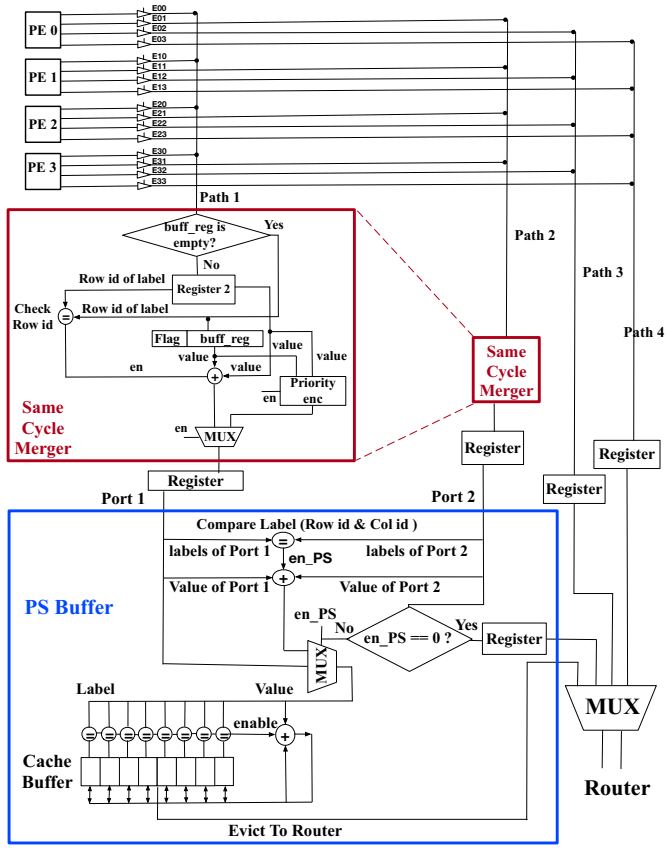

Fig. 5. An illustration of the PE subarray of the HIRAC.

2) Same Cycle Merger: The same cycle merger adds the partial sums produced at the same cycle, same column with matching labels. The partial sums that have matching column ids are sent to the same cycle mergers from of PEs in the same column via *tri-state buffers* as highlighted in Path 1 and 2 in Fig. 5. The tri-state buffer is responsible for serially transferring the partial sums from PEs. Same Cycle Merger compares partial sums row ids to identify the correct merging. About 14% of the total partial sums should be merged at the same cycle and column in the PE subarray when the sparsity of stationary and streaming matrices are 50% and 70%, respectively. Therefore, the same cycle merger plays an important role.

The same cycle merger, as shown in Fig. 5, consists of a pair of registers, an adder, and a comparator. When the partial sum arrives at the same cycle merger, it checks whether the register is empty or not. If it is empty, it means this is the only partial sum available at that cycle in the same cycle merger. Otherwise, it is stored in the second register. When both registers are full, the comparator checks the row ids. If the row ids match, the two partial sums are added together. A priority encoder is used to transfer partial sums from both registers to the PS buffer when the row ids are not matched.

3) PS Buffer: A PS buffer is associated with each PE column, which is used to compute the additions of partial sums from different cycles in the same column or when more than two partial sums from the same cycle need to be merged. About 23% of the merging is on partial sums from different cycles, same column within the PE subarray when the sparsity of stationary and streaming matrices are 50% and 70%, respectively. PS Buffer requires comparators and sequential logics to buffer partial sums and their corresponding labels to search for potential merging candidates. The PS buffer highlighted in blue in Fig. 5 consists of two adders, a multiplexer, a comparator, a cache, a register, and a set of lookup comparators. Streaming is stalled in the column when the PS buffer is full, and the entries are evicted based on a FIFO order before streaming is resumed. When a partial sum arrives at the PS buffer, the labels from the two same cycle mergers are compared. This is to cover the situation where more than two labels from the same cycle merger are the same. If they match, merging occurs, and the result is stored in the cache. Otherwise, the partial sum is stored in the cache directly or sent out to the router when the cache is full.

B. Interconnection Network

A single direction forwarding is supported to keep the interconnection network simple. Each PE subarray is connected to a router, which includes a set of the input-output buffers. The main goal of the router is to transfer the partial sums from PE subarrays to the SRAM for the final merging of matching partial sums from different columns. Fig. 1 illustrates the topology of the proposed interconnection network. The partial sums can be sent to the router directly from the PE subarray or from another router.

C. On-Chip SRAM

All of the partial sums are sent to the on-chip SRAM to check if they can be further merged. This covers situations where partial sums are produced in different columns or different PE subarrays. Therefore, comparators and adders

are required at the on-chip SRAM to handle these remaining partial sums that need to be merged.

As discussed in Section IV, the stationary matrix is loaded level by level, and the corresponding level of the streaming matrix are streamed accordingly. This requires the on-chip SRAM to hold the entire output matrix to capture all potential partial sums to be merged before sending the results to the off-chip. One optimization is to keep a tile of the output matrix and change the loading and streaming order such that all of the partial sums for this tile are computed before the next tile is computed. Fig. 6 shows an illustration of the optimization to compute the tiled output matrix.

Multi-bank direct mapped cache is considered for the SRAM design since the output result matrix in the DNN application is dense [28]. After each tile computation, the SRAM is cleared before computing the next output result matrix tile. It saves the on-chip area for last level merging.

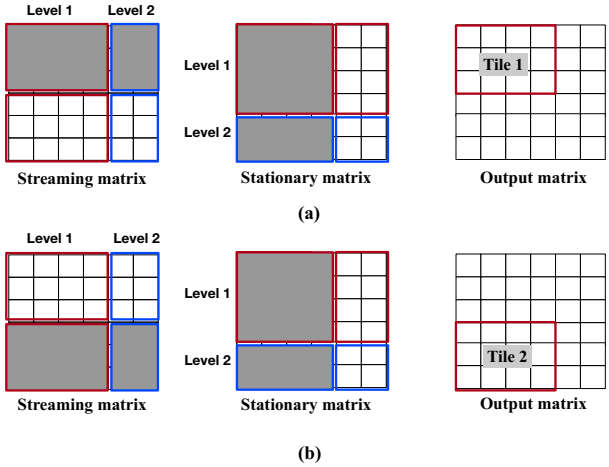

Fig. 6. An illustration of the optimization to compute the tiled output matrix. The red block represents submatrices in level 1, and the blue block represents submatrices in level 2. Shaded submatrices in (a) and (b) are the ones to load and stream to compute tiles 1 and 2.

VI. Experimental Setup

An in-house cycle-accurate simulator is used to model the timing and functional behavior of the HIRAC architecture to evaluate the performance. Architectural parameters of the HIRAC with different configurations are listed in Table II.

To compare performance with SIGMA and TPU, an analytical model is adopted directly from the SIGMA work. The proposed SorPack algorithm is implemented in Python as described in Section IV to generate packed streaming and stationary matrices, which are then fed as inputs to the HIRAC. The area and power of HIRAC are estimated by synthesizing with the 45nm Synopsys standard cell library [25]. The results are scaled to 28nm. BookSim 2.0 [12] is used to estimate the power and area of the global wires in the interconnection network. CACTI [3] is used to estimate the power and area of the SRAM. The sparsity in the activation feature maps ranges between 50-98% [26], and the sparsity range for weights from pruning varies during training iterations from 10% to 90% [20]. Table III shows the matrix dimensions of real-world DNN workloads [19], [20] that are used in the evaluation.

TABLE II
Architectural parameters of HIRAC.

Common Parameters	
Technology	28nm
Clock Frequency	500 MHz
Total Num. of PEs	16384
Data Type	BFP16 Multiplier, FP32 Adder
Channel Width	46 bits
Num. of Channels Per Router	32
Router's In/Output Buffer Size	32 bits
Cache Buffer in PS	128 bytes
On-Chip SRAM	64 KB (32 banks, Direct-Mapped)
Sensitivity Study Parameters	
2×2 **PE subarray:**	
Num of Same Cycle Merger:	2
Num of PS buffer:	2
4×4 **PE subarray:**	
Num of Same Cycle Merger:	8
Num of PS buffer:	4
8×8 **PE subarray:**	
Num of Same Cycle Merger:	32
Num of PS buffer:	8

TABLE III
Common DNN workloads dimensions.

Workloads	M size	N size	K size
NCF (Set 0)	2048	1	128
Transformer (Set 1)	84	1024	4096
GNMT (Set 2)	1632	36548	1024
Nvidia GPU (Set 3) [19]	6912	2048	4096
Nvidia GPU (Set 4) [19]	27648	384	4096
Transformer (Set 5)	31999	1024	84

VII. Evaluation Results

A. The SorPack Algorithm

The proposed SorPack algorithm is faster than the collision-aware packing algorithm in the Sparse-TPU work. Fig. 7 shows the runtime comparison of the SorPack and the collision-aware algorithms for three different matrix sizes and sparsities for both stationary and streaming matrices. The result shows about $32\times$ average reduction in runtime for the smallest matrix. As the matrix size increases, a greater runtime reduction is expected. The collision-aware algorithm partitions a matrix vertically based on the number of PE rows in the hardware design. Then greedy column merging is performed inside each partition. A bigger matrix has more columns for the iterative algorithm to find compatible candidates to merge; hence, the algorithm takes longer for each iteration. In contrast, the SorPack algorithm partitions both vertically and horizontally. The packing step in the SorPack algorithm occurs inside each partition with a constant partition size independent of the matrix size, and the packing step is a faster non-iterative single-pass process. Hence, SorPack is a scalable algorithm as the matrix size gets bigger.

The impact of the sorting step in the SorPack on the overall system performance is also evaluated. Fig. 8 shows the results for different sparsities. The performance improvement is due to the increased percentage of the same-cycle/col merging after applying the sorting step in the SorPack. In the DNN application, the nonzero elements have a uniform distribution [16].

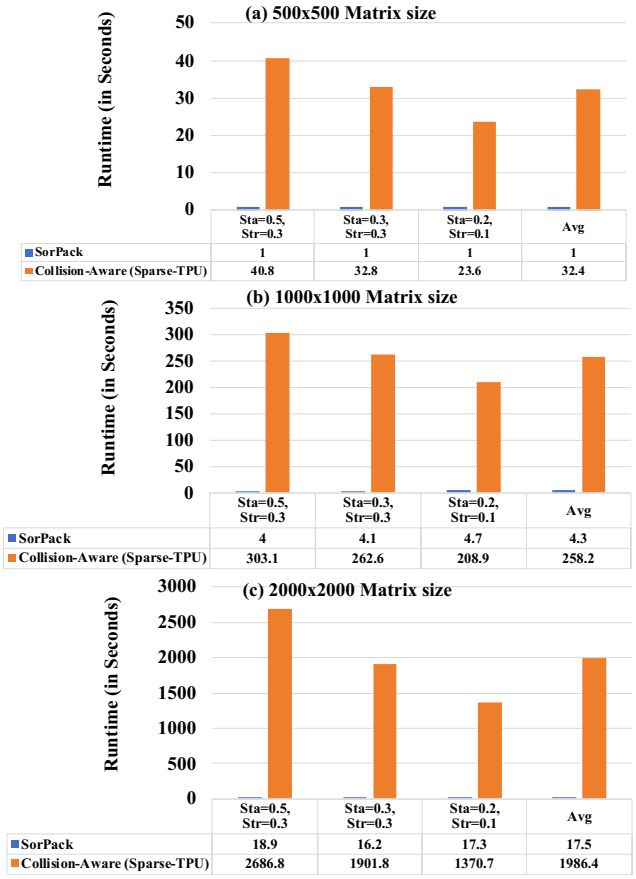

Fig. 7. The runtime comparison of the SorPack and the collision-aware algorithm for different matrix sizes (a-c) and sparsities.

The percentage of the same-cycle/col merging is expected to have greater improvement after applying the sorting step in matrices with non-uniform distribution of nonzeros (*e.g.*, adjacency graphs of real-world networks).

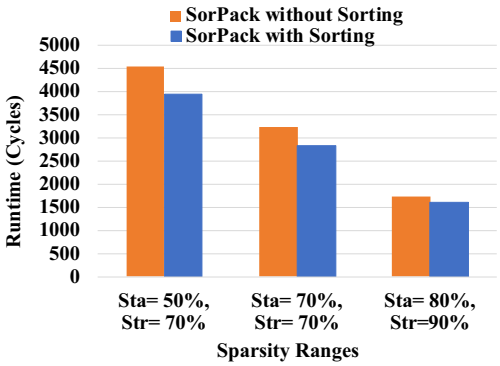

Fig. 8. Effects of the Sorting step on system performance. The matrix size is 100×100.

Sorting also helps to reduce data movement energy because it allows more merging to happen within the PE subarrays. After applying sorting, the network energy is reduced by 10%, and the SRAM energy is reduced by 8.5%.

B. The HIRAC Architecture

The HIRAC architecture is evaluated and compared against the SIGMA and TPU architectures, which are faithfully modeled for the same data type and architecture configurations. TPU and SIGMA have 128×128 MAC units and 128×128 PEs, respectively. SIGMA has the same number of adders and multipliers as the TPU does.

1) Speedup: Fig. 9 shows the overall speedup of the HIRAC as compared to the SIGMA when computing matrices with different sparsities for matrix sizes in Table III. The HIRAC works most efficiently between the sparsity ranges of 40% to 90%, which covers many important applications [20], [26]. One reason for using the SorPack is to increase the same-cycle/col merging percentage. There are fewer partial sums that would need to be merged for graph analytic-like applications with hyper-sparse graphs (sparsity $>$ 90%). For these applications, the benefit of using SorPack would be less. The results in Fig. 9 are normalized to the performance of TPU. Generally, when the sparsity range is high in both matrices, about $3\times$ speedup can be achieved. One main reason for this speedup is the use of the SorPack algorithm, which makes the streaming matrices dense. Therefore, in the cases where the streaming matrices are sparser, more improvement is expected. When the streaming matrix is denser, the speedup of the HIRAC over SIGMA is smaller. This is because SIGMA does not remove zeros from streaming matrices.

2) Area and Power: The area of HIRAC is estimated based on the synthesis of standard cells without place and route. To make a fair comparison with TPU and SIGMA, the layout overhead of HIRAC is conservatively overestimated. The layout area overhead of HIRAC is considered the same percentage for the HIRAC as it is for SIGMA, even though the interconnection network of the HIRAC is simpler than it is in SIGMA. The wire area of the interconnection network is estimated separately using a similar methodology used in the BookSim, and then the layout area overhead is added on top of the estimated area of HIRAC. Since the TPU and SIGMA did not consider the area of SRAM in the evaluation, the SRAM area is not included in the comparison but will be reported separately. Fig. 10 shows the area breakdown of all three designs. The area of the HIRAC is 58.7 mm^2, which is 9.5% smaller than the area of SIGMA. The reason for this improvement is the simpler interconnection network of the HIRAC as compared to SIGMA. However, the HIRAC adds a 24.6% area overhead as compared to the TPU because of the interconnection network and its conservative layout overhead. The local buffer and logic area in the HIRAC is 1.4 mm^2, which includes the buffer and logic used in the same cycle merger and the PS buffer. This area is higher than it is in SIGMA and TPU because of the use of more buffers in the HIRAC. The on-chip SRAM in HIRAC adds a 2.4 mm^2 area overhead, which is 4.1% of the total area.

The total power consumption of the HIRAC is 15.2 W, including the dynamic and leakage power, which is a 32% reduction as compared to SIGMA. However, the HIRAC con-

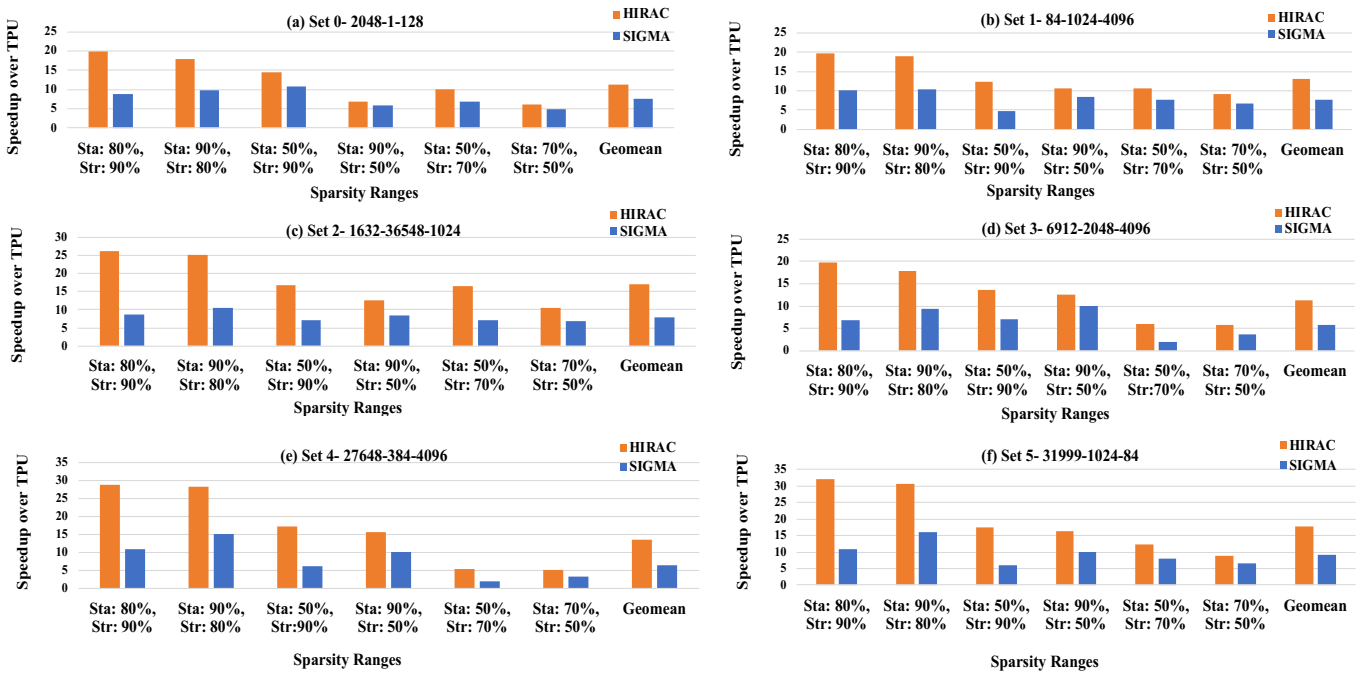

Fig. 9. Speedup comparison of the HIRAC and SIGMA over the Google TPU for representative matrices in DNN workloads (a-f). Sta:80% Str:90% means the stationary matrix has 80% zeros, and the streaming matrix has 90% zeros.

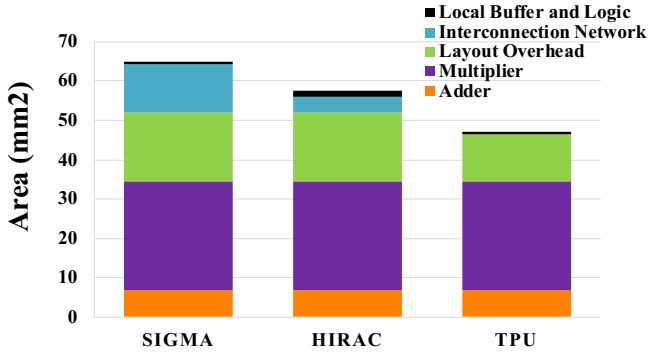

Fig. 10. Area breakdown of the HIRAC.

sumes 23.8% more power than TPU because of the improved PE utilization as well as the use of more buffers, logic, and the interconnection network in the design. Although consuming more power is a side effect, better energy efficiency is achieved due to the overall speedup. For example, the HIRAC reduces 14.2× of energy as compared to TPU for the geomean of Set 5 workload in Fig. 9(f).

C. Sensitivity Study

The partitioning sizes of the SorPack algorithm, the PE subarray sizes in the HIRAC architecture, and the number of banks of the on-chip SRAM can significantly influence the performance of the proposed architecture. This section reports the sensitivity study results of these configuration parameters.

1) Partitioning Sizes of SorPack: Fig. 11 demonstrates the speedup over the SIGMA for different matrix partitioning sizes P of SorPack. When the matrix partitioning size is small, the percentage of the same-cycle/col merging partial sums increases, as shown in Fig. 3. Therefore, more merging would happen inside the PE subarray, and fewer unmerged partial sums need to be sent through the interconnection network to the on-chip SRAM, which improves the speedup significantly, as shown in Fig. 11(a). However, when the matrix size is larger, a larger partitioning size P can reduce the matrix size more, which reduces the rounds of loading and streaming. For the sensitivity result (Fig. 11(b)) with larger matrices like the set 3 workload in Table III, the 2048×2048 partition size has an advantage of 20% and 39% reduction in the rounds of loading and streaming, respectively, as compared to the 4×4 partition size. However, the 4×4 partition size still outperforms the larger ones.

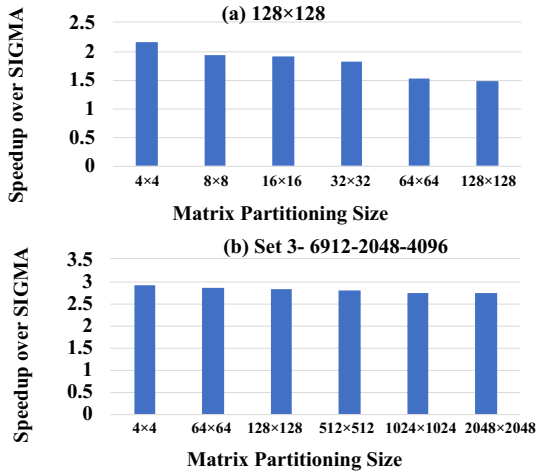

Fig. 11. Speedup over the SIGMA for different matrix Partitioning size P of the SorPack in two different matrix sizes of (a) 128 × 128 and (b) the Set 3 workload. The sparsity of the stationary and streaming matrices are 50% and 70%, respectively.

2) PE Subarray Sizes in the HIRAC: PE subarray size in the HIRAC can be chosen independently from the matrix partitioning size P in the SorPack. Table IV presents area, power, and cycle runtime evaluation results for the different PE subarray sizes. Based on the results, as the size of the PE subarray increases, area and power are reduced. The main reason is a reduction in the area overhead of the interconnection network due to the reduced number of routers in the design. However, the overall cycle runtime reduces as the PE subarray size decreases due to an increase in the number of PE subarrays working parallelly. So, there is a trade-off for choosing the PE subarray size in the HIRAC, which should be based on whether the system has a stringent power constraint or prioritize a fast runtime.

TABLE IV
AREA, POWER, AND CYCLE RUNTIME FOR DIFFERENT PE SUBARRAY SIZES.

PE subarray size	Area	Power	Cycle runtime
2×2	64.7 mm^2	16.1 W	4559
4×4	58.7 mm^2	15.1 W	5106
8×8	56.7 mm^2	14.7 W	5830

100×100 matrices are used with 80% and 90% sparsity for stationary and streaming matrices, respectively, to evaluate the sparser cases, and P is 4.

3) Number of Banks of the On-Chip SRAM: Choosing the number of banks of the on-chip SRAM can influence the area, power, and access parallelism of the SRAM. The number of bank conflicts reflects whether the number of banks is enough to support the required parallelism. As the number of banks increases in the SRAM, the number of bank conflicts decreases. Therefore, the number of banks improves the system performance. However, increasing the number of banks would increase the area and power. Fig. 12 shows the cycle runtime, the number of bank conflicts and the area as the number of banks increases. Beyond 32 banks does not improve the number of SRAM conflicts and cycle runtime; however, the area will continue to increase. Thus, 32 banks are used for the on-chip SRAM.

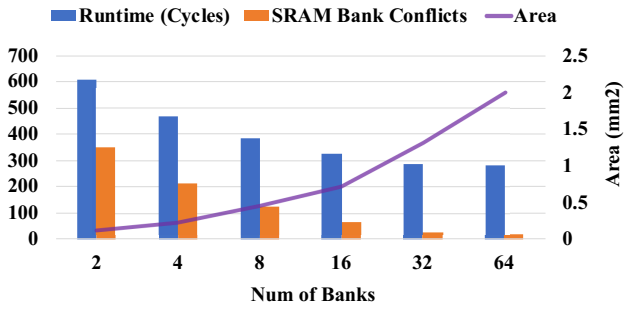

Fig. 12. Results of cycle runtime, SRAM bank conflicts and area for the different number of the banks.

D. End-to-End Evaluations of a DNN Workload

A pruned GNMT v2 model with the WMT16 EN-DE dataset is used to further evaluate the overhead of running SorPack for online preprocessing of sparse activation matrices in an end-to-end DNN application [10]. This model uses a 4-layer LSTM encoder and a 4-layer LSTM decoder, which require SpGEMM computations. The SorPack algorithm is applied offline to the weight matrices since they will not change during inference. The runtime overhead of SorPack is counted for preprocessing of the sparse activation matrices only, which is evaluated on an Intel Xeon Silver 4114 CPU at 2.2 GHz. The serial- and parallel- SorPack computation overhead is measured in isolation on a CPU and added to the accelerator time. The SorPack runtime includes all the steps of the Algorithm 1 wherein *SORT* takes 75% of the total runtime of serial-SorPack. The total runtime of this DNN application running on the HIRAC is compared with TPU, which does not require preprocessing. Running a parallel version of SorPack on multiple cores can reduce the preprocessing overhead, wherein each partition performs its sorting, packing, and deleting blank rows (Fig. 2) in parallel. This makes the execution time of the parallel-SorPack bound by the partition that takes the longest to sort, pack, and delete blank rows. As shown in Fig. 13, the HIRAC perform 1.5× and 8.2× better than TPU when running serial- and parallel- SorPack on the CPU. The performance of the SorPack can be further improved by optimizing the software implementation, which is left as future work.

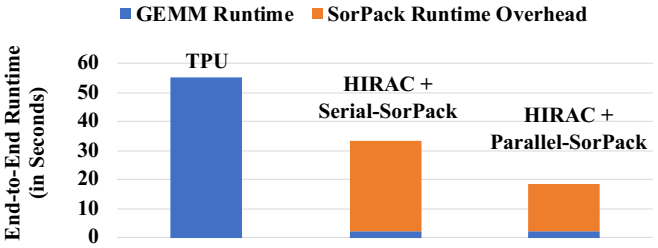

Fig. 13. An end-to-end runtime evaluation using GNMT v2 [10].

VIII. CONCLUSION

This paper proposes a novel sorting-based packing algorithm, SorPack, and a hierarchical SpGEMM accelerator, HIRAC, for DNN applications. The key idea is to use the sorting step in the SorPack to keep the partial sums that need to be merged close to each other to reduce data movement. The proposed SorPack algorithm is simple and faster as compared to the state-of-the-art packing algorithm used in the Sparse-TPU [7]. Representative matrices in the DNN workloads are used for the evaluation of the HIRAC to compare with the state-of-the-art sparse DNN accelerator, SIGMA [20]. The results show that HIRAC can achieve a significant speedup with lower power and area as compared to SIGMA.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation at Lehigh University under Grant CCF-1750826. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] M. J. Anderson, B. Chen, S. Chen, S. Deng, J. Fix, M. K. Gschwind, A. Kalaiah, C. Kim, J. Lee, J. Liang, H. Liu, Y. Lu, J. Montgomery, A. S. Moorthy, N. Satish, S. Naghshineh, A. Nayak, J. Park, C. Petersen, M. D. Schatz, N. Sundaram, B. Tang, P. Tang, A. Yang, J. Yu, H. Yuen, Y. Zhang, A. Anbudurai, V. Balan, H. Bojja, J. Boyd, M. Breitbach, C. Caldato, A. Calvo, G. Catron, S. Chandwani, P. Christeas, B. Cottel, B. Coutinho, A. Dalli, A. Dhanotia, O. Duncan, R. Dzhabarov, S. Elmir, C. Fu, W. Fu, M. Fulthorp, A. Gangidi, N. Gibson, S. Gordon, B. P. Hernandez, D. Ho, Y.-C. Huang, O. Johansson, S. Juluri, and et al., "Facebook," in *First-Generation Inference Accelerator Deployment at Facebook*, 2021.

[2] H. Anzt, S. Tomov, and J. Dongarra, "NVIDIA," in *Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs*, 2014.

[3] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, 2017.

[4] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[5] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," in *accepted for publication in IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[6] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 243–254.

[7] X. He, P. Subhankar, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge, "Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020.

[8] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18, 2018, p. 674–687.

[9] R. Hojabr, A. Sedaghati, A. Sharifian, A. Khonsari, and A. Shriraman, "SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[10] Intel, "NLP Architect by Intel AI Lab," Nov. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1477518

[11] J.-W. Jang, S. Lee, D. Kim, H. Park, A. S. Ardestani, Y. Choi, C. Kim, Y. Kim, H. Yu, H. Abdel-Aziz, J.-S. Park, H. Lee, D. Lee, M. W. Kim, H. Jung, H. Nam, D. Lim, S. Lee, J.-H. Song, S. Kwon, J. Hassoun, S. Lim, and C. Choi, "Sparsity-Aware and Re-configurable NPU Architecture for Samsung Flagship Mobile SoC," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[12] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 86–96.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *To appear at the 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada,*, 2017.

[14] H. Kung, B. McDanel, and S. Q. Zhang, "Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[15] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

[16] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the Value of Network Pruning," in *International Conference on Learning Representations*, 2019.

[17] L. Lu, Y. Jin, H. Bi, Z. Luo, P. W. Li, T. Wang, and Y. Liang, "Sanger: A Co-Design Framework for Enabling Sparse Attention Using Reconfigurable Architecture," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[18] E. Montagne and A. Ekambaram, "An optimal storage format for sparse matrices," in *Inf. Process. Lett.*, vol. 90, 2004.

[19] NVIDIA, "Matrix Multiplication Background User's Guide :: NVIDIA Deep Learning Performance Documentation," https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, 2022, [May 17, 2022].

[20] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[21] A. Samajdar, E. Qin, M. Pellauer, and T. Krishna, "Self Adaptive Reconfigurable Arrays (SARA): Learning Flexible GEMM Accelerator Configuration and Mapping-Space Using ML," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22, 2022, p. 583–588.

[22] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[23] M. Soltaniyeh, R. P. Martin, and S. Nagarakatte, "An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-Matrix Multiplication," in *ACM Trans. Archit. Code Optim.*, 2022.

[24] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[25] Synopsys, "Synopsys Standard Cell Libraries," https://www.synopsys.com/dw/ipdir.php?ds=dwc_standard_cell, 2019, [Version P-2019.03, March 2019].

[26] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng, "Dual-Side Sparse Tensor Core," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021, p. 1083–1095.

[27] J. Yang, W. Fu, X. Cheng, X. Ye, P. Dai, and W. Zhao, "S2Engine: A Novel Systolic Architecture for Sparse Convolutional Neural Networks," in *IEEE Transactions on Computers*, vol. 71, no. 06, 2022.

[28] C. Zhang, M. Bremer, C. Chan, J. Shalf, and X. Guo, "ASA: Accelerating Sparse Accumulation in Column-Wise SpGEMM," in *ACM Trans. Archit. Code Optim*, 2022.

[29] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[30] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient Architecture for Sparse Matrix Multiplication," in *2020 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[31] C. Zheng, S. Gu, T.-X. Gu, B. Yang, and X.-P. Liu, "BiELL: A bisection ELLPACK-based storage format for optimizing SpMV on GPUs," in *Journal of Parallel and Distributed Computing*, 2014.