ZeD: A Generalized Accelerator for Variably Sparse Matrix Computations in ML

Pranav Dangi
dangi@comp.nus.edu.sg
National University of Singapore

Zhenyu Bai*
zhenyu.bai@comp.nus.edu.sg
National University of Singapore

Rohan Juneja
rohan@comp.nus.edu.sg
National University of Singapore

Dhananjaya Wijerathne
dmd@comp.nus.edu.sg
National University of Singapore

Tulika Mitra
tulika@comp.nus.edu.sg
National University of Singapore

ABSTRACT

Modern Machine Learning (ML) models employ sparsity to mitigate storage and computation costs; but it gives rise to irregular and unstructured sparse matrix operations that dominate the execution time and require specialized accelerators to meet the performance and energy targets. Contemporary sparse matrix accelerators, optimized for extreme sparsity, frequently fall short in addressing the variable and moderate degrees of sparsity prevalent in most ML models. Variable sparsity leads to inefficiency in the storage and processing of matrices. In response to this challenge, we propose an adaptive and generalized architecture design, *ZeD*, capable of accommodating the variably sparse matrix computations in ML models. Our innovative design integrates a bit-tree compression format and zero-detection hardware, resulting in highly efficient packing, storage, retrieval, and processing of sparse matrices. Furthermore, we propose a matrix row reorganization strategy based on sparsity similarity to substantially enhance memory reuse. Synthesis results of *ZeD* demonstrate a 3.2× improvement in performance per area over state-of-the-art solutions across a spectrum of ML workloads characterized by wide-ranging sparsities.

CCS CONCEPTS

• **Computer systems organization → Special purpose systems**.

KEYWORDS

Sparse Tensor Computations, Machine Learning Hardware, Sparse Compression Formats, Hardware Acceleration

ACM Reference Format:
Pranav Dangi, Zhenyu Bai, Rohan Juneja, Dhananjaya Wijerathne, and Tulika Mitra. 2024. ZeD: A Generalized Accelerator for Variably Sparse Matrix Computations in ML. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3656019.3689905

*Corresponding Author

PACT '24, October 14–16, 2024, Long Beach, CA, USA

ACM ISBN 979-8-4007-0631-8/24/10.
https://doi.org/10.1145/3656019.3689905

1 INTRODUCTION

The escalating demands for storage and computing resources in machine learning (ML) have outpaced the development of specialized accelerator architectures tailored to address these challenges. A substantial portion of the computational workload in ML models is attributed to convolutions in convolutional neural networks (CNN) and attention mechanisms in models such as Transformers (e.g., Vision Transformers (ViT) and Natural Language Processing models like GPT). These operations fundamentally rely on matrix multiplications [17, 20, 24]. A widely adopted strategy to mitigate computation, communication, and storage needs for matrix multiplications is to employ sparsification and quantization techniques [3, 9].

Sparsification in model weights is achieved through pruning, which reduces the model size by trimming redundant weights and activations while trying to retain accuracy. Sparsity in activations often arises due to zero-valued results from non-linear functions like the rectified linear unit (ReLU). Inherently, this sparsification is unstructured, creating random sparsity patterns and hence introducing irregularity in memory access and processing, leading to inefficient inference performance. To make the model more hardware-friendly, prior works have attempted to resort to structured sparsity [3, 23, 26]. This requires sparsification under hardware-aware constraints where pre-defined structures that can arguably benefit from data locality are incorporated into the pruning mechanism. Unfortunately, these constraints reduce freedom on the software side, where ML research and new ML models are constrained within hardware-defined boundaries, which exposes accuracy-compression trade-offs. In fact, recent Transformer works show that, when pruning with a similar accuracy target, unstructured sparsity achieves 90% compression while structured sparsity reaches only 70% compression [20]. Moreover, enforcing such structure on dynamically generated sparse matrices like activations or attention masks is often not feasible. It is imperative to study unstructured sparsity despite its irregularity.

Contemporary sparse matrix accelerators are often optimized for hyper sparsity (> 99.9% sparsity); consequently, they frequently fall short of addressing the variable and relatively lower degrees of sparsity prevalent in most ML models. Figure 1 shows the current landscape of sparse matrix workloads, where ML workloads lie in the moderate sparsity regions, while scientific computing matrices belong to the extreme case of hyper-sparse matrices. We observe the variation in the degrees of sparsity across various matrices from ML and popular scientific workloads, This includes scientific matrices from *SuiteSparse*[7, 42]; sparse *BERT* and *DeiT* models[22,

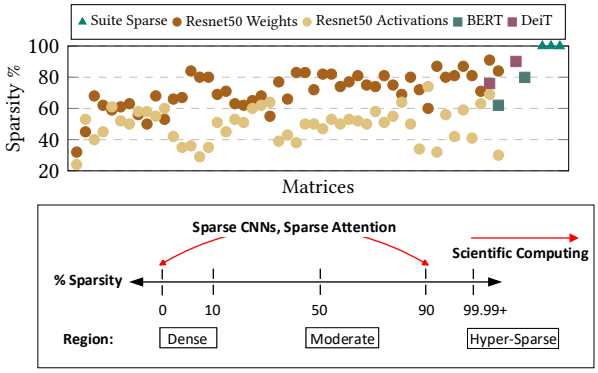

Figure 1: Variation in sparsity

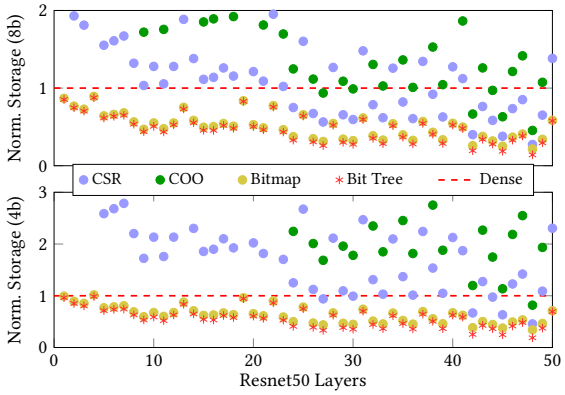

Figure 2: Comparison of storage costs of various sparse formats, normalized to the dense format (uncompressed) for 8-bit and 4-bit precisions.

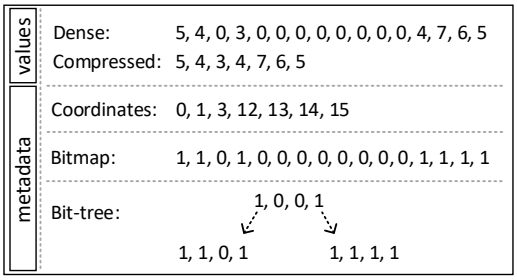

Figure 3: Types of sparse metadata: Coordinate-based (for CSR, COO), and Bit-sparse (for Bitmap, Bit-tree)

41] representing NLP and ViT applications; and different layers of *ResNet50* [12] for CNNs. Clearly, sparse matrices from scientific computing are hyper-sparse (> 99.9% sparsity), whereas matrices from ML workloads are only moderately sparse (≤ 90% sparsity), with a large variation in sparsity between different models and even within one model. This unstructured sparsity of variable degrees in ML is challenging from both storage and computation perspectives.

Storage Challenge. Traditional sparse matrix accelerators, tailored for hyper-sparse matrices, often utilize coordinate-based formats such as Compressed Sparse Row (CSR) and Coordinate Lists (COO) [15, 27, 37, 42]. These formats encode the positions of each non-zero element using elaborate metadata, allowing the accelerator to access the non-zero elements with minimal additional hardware. While these formats excel with hyper-sparse matrices containing only a few non-zeros, the associated metadata overhead becomes significant as the density of the matrices increases, incurring substantial storage and memory traffic overhead.

Further, in the inference of ML models, sparsification is often coupled with quantization, which compresses numerical values by reducing their bit-width without substantial loss in accuracy. In the context of quantized sparse models, where the actual data size undergoes significant reduction (e.g., 4-bit precision), the high overhead imposed by metadata from sparsification becomes dominant. This is because while quantization reduces the cost of storing the numerical values of the data elements, the metadata overhead remains constant, amplifying the relative inefficiency of these formats.

Figure 2 compares the overall storage costs of different layers of a sparse ML model *ResNet50*, encoded with various sparse matrix formats normalized against the uncompressed dense matrix at 8-bit and 4-bit precisions. The variation in storage costs among these formats is influenced by the meta-data overhead, which depends on the degree of sparsity. The CSR and COO formats tend to exhibit considerably high overhead. Interestingly and counterintuitively, the compressed models in CSR/COO formats are sometimes even less efficient than just using the dense format with all the zeros included. CSR/COO metadata can bloat the storage requirement by up to 2-3X in such cases compared to the simple, dense format. As the trend towards quantized model inference gains momentum, where the ML models move towards extremely low precisions [2, 25], the impact of this overhead becomes more pronounced, exacerbating the inefficiencies of CSR and COO formats. On the other hand, these coordinate-based sparse formats relieve the hardware from needing any costly additional indexing circuitry, work well with state-of-the-art sparse dataflows, and are thus widely used. However, the inadequacies of these coordinate-based formats underscore the pressing need to explore sparse representation techniques that incur minimal overhead across a broader range of sparsity while effectively handling irregularities for efficient processing by the hardware accelerator.

The bitmap format uses one bit per element to indicate whether the element is zero, making its storage complexity dependent solely on the size of the input matrices rather than their sparsity, which allows for better compression in moderately sparse matrices. We adopt a hierarchical bit-tree storage format to extend this to a wider range of sparsity while maintaining low storage costs and enabling streamlined processing. Figure 3 illustrates the bit-tree format, which can be deemed as a lossless compression of the bitmap format, with its origins in Huffman trees [35]. The bit-tree hierarchically encodes the zeros in a bitmap, allowing the runtime skipping of sequences (packs) of zeros systematically. Bit-sparse formats like Bitmaps and Bit-Trees maintain a consistently low storage overhead, as shown in Figure 2, as their metadata size primarily depends on the matrix size and not the degree of sparsity.

Work	Type	Efficiency	Sparsity Level	Application
[3, 23, 26]	Structured	High	Low	CNN
[8, 24]	Structured	High	Low	Sparse Attention
[29, 33, 42]	Unstructured	High	Hyper-sparse	Scientific Computing
[15, 17, 27]	Unstructured	Low	High	CNN
[14, 22, 27]	Unstructured	Low	High	Sparse Attention
ZeD (Ours)	**Unstructured**	**High**	**Low to High**	**CNN, Sparse Attention**

Table 1: Categorization of Sparse Accelerators

Computational Challenge. Table 1 categorizes prior accelerators based on the application domain and the achieved or expected sparsity level to maintain optimal accuracy and efficiency. In this context, efficiency serves as a qualitative proxy for the achieved performance relative to the area, power, and memory bandwidth. Contemporary ML accelerators predominantly adopt structured sparsity [1, 3, 8, 23, 26] to achieve significant hardware efficiency, although this leads to limited and lower compression achieved for the same accuracy.

One of the basic building blocks of existing sparse matrix computations involves indexing into non-zero elements of operands by comparing their indices (i.e. metadata) [16, 21] and frequent random access of the sparse vectors (matrix rows). Sparse tensor algebra accelerators for scientific computing [4, 10, 29, 33, 42] using CSR/COO formats demonstrate high efficiency even for unstructured data due to novel dataflows, compression formats, and memory access techniques. However, they are tailored to matrices within the hyper-sparse domain. When targeting moderate to low unstructured sparsity like what is seen in ML workloads [14, 15, 17, 22, 27], they do not exhibit the same level of efficacy due to high overhead for metadata access. The growing number of non-zero elements per row increases random access and escalates metadata indexing and comparisons per essential compute operation, directly affecting performance and efficiency. This demonstrates that employing techniques tailored for extreme cases to address broader problems results in unforeseen inefficiencies. Due to the variability of machine learning workloads, it is crucial to develop efficient and general methods for retrieving and processing the non-zeros.

While bit-sparse formats like bitmaps and bit-trees offer efficient storage alternatives, they are challenging from an accelerator design perspective. Conventional bitmaps require dynamic decoding of arbitrarily long binary vectors to access non-zero elements, introducing complexity, reducing scalability, and hindering hardware acceleration. As mentioned earlier, prior works use the CSR or COO formats for this reason. We overcome this issue by adopting the bit-tree format that allows us to skip consecutive zeros systematically during processing. More importantly, we achieve this with minimal hardware overhead by carefully designing the packing strategy of bit-trees, circumventing the hardware complexities normally associated with bitmap access. Furthermore, we analyze the memory access patterns of the matrices to reorganize their execution and substantially enhance memory reuse, thereby alleviating the memory bottlenecks inherent to sparse dataflows.

Contribution: We present *ZeD*[1], a generalized architecture designed to accelerate variably sparse and unstructured machine learning workloads. Our contributions include the following:

- We mitigate storage and memory traffic overheads by adopting highly efficient bit-tree structures for packing the sparse metadata of the compressed matrix.
- We design a low overhead, multi-pass, parallel zero skipping mechanism to retrieve and process non-zeros from bit-trees.
- We study the parallelism of the dataflow and memory-access patterns in the sparse matrices to propose a pre-processing mechanism that reorganizes and groups execution of input rows to maximize memory reuse.

Overall, ZeD proposes a general and efficient architecture that harnesses wide-ranging sparsity within unstructured data to achieve 3.2× better performance/area than prior state-of-the-art accelerators and a 3.4× reduction in memory traffic.

2 STORAGE FORMAT & DATAFLOW

In this section, we first discuss various sparse storage formats and then introduce the hierarchical bit-tree storage format. Following this, we present our optimized algorithm, which utilizes a dataflow designed to exploit bit-tree sparsity and accommodate variable unstructured sparsity in matrices. Our approach includes an output-stationary, tiled, row-wise product matrix multiplication dataflow tailored to the bit-tree storage format. Finally, we propose an offline row access reorganization and grouping strategy to alleviate memory traffic bottlenecks associated with the dataflow.

2.1 Hierarchical Bit-Tree Storage Format

Figure 5 presents the trend of the storage costs of the dense and commonly used sparse formats [10, 15, 17, 27, 32, 38] for a $M \times N$ matrix with varying density. Coordinate-based formats (CSR and COO) use meta-data, typically an entire list or pointers to store the coordinates of each non-zero element. The coordinate is used to index non-zero elements during matrix multiplication to effectively compute only the non-zero products. The size of the largest matrices in the target application domain determines the number of bits allocated for this coordinate encoding. This can lead to significant, unexpected overhead for the variable sparsity levels commonly encountered in ML workloads, sometimes even exceeding the storage cost of the dense format (Figure 2). In contrast, bit-sparse formats like Bitmaps employ a simple binary encoding for non-zero elements, resulting in a consistent storage overhead regardless of sparsity level. This makes them highly suitable for a wide range of sparse matrices from a compressed storage perspective. However, conventional bitmaps require dynamic decoding of arbitrarily long binary vectors to access non-zero elements, introducing complexity, reducing scalability, and hindering hardware acceleration.

Efforts have been directed towards utilizing coarse-grained pointers and bitmaps [16, 19, 38], which involve compressing and skipping larger, entirely zero tiles within the matrix. This hierarchical organization aims to eliminate completely zero tiles during matrix operations on conventional dataflow architectures. However, this approach struggles to tackle sparsity effectively. It necessitates storing and computing all zeroes within a block that contains at least one non-zero element, resulting in performance and efficiency degradation. This drawback becomes particularly apparent in moderately sparse matrices where nearly every block may contain at least one non-zero element. Skipping at a coarser granularity needs

[1]for Efficient, General **Ze**ro **D**etection

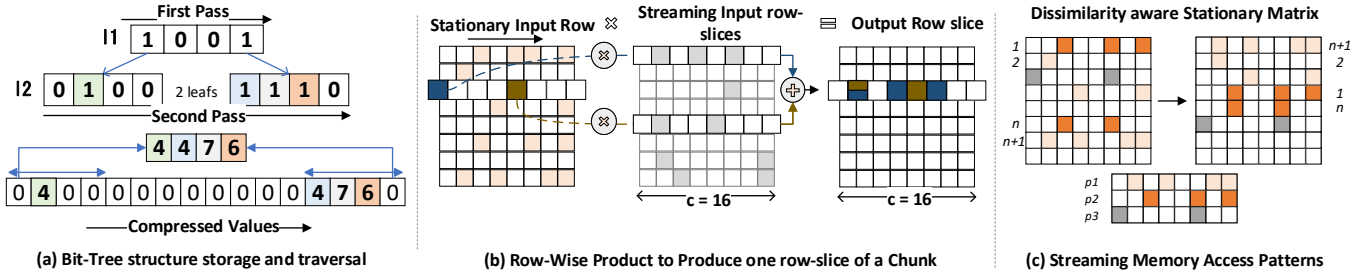

Figure 4: (a) Illustration of a 2-level bit-tree: 4 non-zeroes in a 16-element slice. (b) Working of row-wise product (c) Extracting Streaming Patterns from a Stationary Matrix

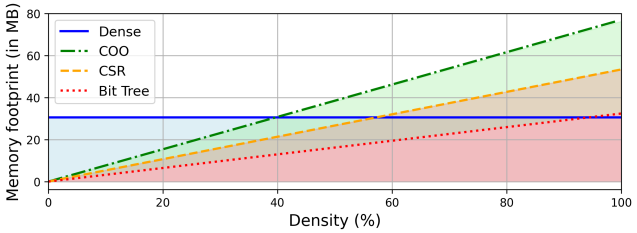

Figure 5: Memory footprint of sparse storage formats

to be combined with compression and skipping at a finer granularity to avoid any zero-valued storage and computations.

We adopt a hierarchical bit-tree format that encodes sequences of consecutive zeros as a single zero at a higher level, similar to hierarchical Run-Length Encoding used in Huffman trees [35], to address the challenges associated with variable sparsity. Bit-trees evolve from bitmaps by encoding packs of consecutive zeros in the bitmap as a single zero at a higher level. Figure 5 illustrates that the bit-tree offers the most compact memory footprint across varying densities, except in the cases of hypersparse and dense matrices.

Bit-trees can be designed with an arbitrary number of levels, recursively condensing sequences of multiple zeros into a single zero. Based on experimental evaluation, we encode/pack a sequence of four zeros as a single zero. The greater the number of levels, the more efficient the encoding for matrices with higher sparsity. For the purpose of this discussion, focusing on moderate sparsity, we use a two-level bit-tree with the levels denoted as l1 and l2, illustrated in Figure 4(a). In this representation, a group of four consecutive elements at l2 is referred to as a "leaf", and a non-zero leaf exhibits one-to-one mapping with its corresponding actual values in the uncompressed matrix. The compressed values stored in memory do not consist of any zeroes with the exception of a small amount of zeroes required for padding at the end of a tile's storage to maintain memory alignment. Decoding operates inversely; each non-zero element in l1 corresponds to a leaf in l2 with one or more non-zero values, while every zero in l1 corresponds to four consecutive zeros in l2 that do not need to be stored or computed. We present a sensitivity analysis of the storage format parameters in the experimental evaluations.

Given the binary nature, the storage cost of bit-tree is similar to a naïve bitmap for moderate sparsities. In fact, it is better at higher sparsities and is significantly more amenable to hardware decoding. To the best of our knowledge, ours is the first work to introduce bit-trees for accelerating sparse matrix multiplication. Capstan [32] employs bit-trees for vectorizing addition on non-zeroes clustered along the diagonal in hyper-sparse matrices. However, their approach calculates addresses explicitly by nesting iterations over multiple levels for these extremely sparse matrices, which becomes inefficient at medium sparsity levels. In contrast, we use the knowledge of the relative positioning of non-zeroes in the tree and exploit data locality within bit-trees through our dataflow. We introduce low-cost architectural support to efficiently retrieve the required data at different levels during matrix multiplication. More specifically, we use simple, myopic zero-detection hardware that employs multiple passes on four-bit sequences to find the non-zeroes' relative positions at each level of the tree. These relative positions help us dynamically map the non-zero values to their desired compute positions. This localized zero-detection facilitates parallel and efficient sparse vector operations in our dataflow, ensuring performance efficiency and scalability.

2.2 Dataflow and Memory Access

We customize the dataflow, the corresponding tiling strategy, and the memory access patterns to optimize performance for retrieving and processing on non-zeroes in the bit-tree. Our observation indicates that the packing of non-zero elements in the bit-tree aligns well with the state-of-the-art row-wise sparse matrix multiplication dataflow [4, 13, 33, 42]. Additionally, while traversing bit-trees using our tiling strategy, we also reorganize the accessing of the rows based on sparsity patterns, alleviating known irregular memory access issues associated with the dataflow.

2.2.1 **Row-wise Product Dataflow:** In the literature, various accelerators utilize different dataflows for sparse matrix multiplication. These dataflows are typically categorized based on the operand that remains stationary and the sequence in which matrix dimensions are traversed [17, 33]. Selecting an appropriate dataflow that matches the storage format and tailoring an architecture accordingly are critical factors in addressing inefficiencies and bottlenecks in sparse matrix multiplication. Prior accelerators utilizing inner-product and outer-product dataflows face inefficiencies with sparse inner-join, irregular merging of large partial sum matrices, significant on-chip memory requirements, and challenges related to load

imbalance [11, 29, 33]. These dataflows have limited parallelism and are also not suitable for handling bit-trees.

Gustavson's algorithm [13] enables row-wise product, an efficient and highly parallel algorithm for sparse matrix multiplication used by some state-of-the-art works [4, 33, 42]. Here, all the non-zero elements from a single row of the stationary matrix (input 1) are multiplied by the non-zero entries from the corresponding rows of the streaming matrix (input 2). The row indices of the streaming matrix are determined by the column positions of the non-zero values from the stationary matrix, as shown in Figure 4(b). The sparse vectors (partial sums) thus produced are accumulated in the corresponding row of the output matrix.

While the adoption of row-wise product undeniably mitigates many of the inefficiencies with prior dataflows, it still has a few drawbacks. Notably, it has the issue of random and frequent irregular accesses to the streaming rows. Further, it exhibits inefficiencies in the sparse vector addition process required to merge partial sum rows. This necessitates the accelerators using Gustavson's dataflow to use high-radix mergers, reconfigurable interconnects, and high-bandwidth memory resources [27, 33, 42].

The challenges posed by partial sum mergers and irregular streaming become more pronounced as we transition from hyper-sparse matrices to relatively denser matrices characterized by a higher number of non-zero elements (nnz) per row. A greater nnz per row implies increased irregular off-chip traffic associated with the streaming matrix and a higher number of index comparison operations for each partial sum row merge operation required to generate the final output row. Recognizing these issues, we present an algorithm that identifies sparsity patterns to reorganize and group the execution order of stationary matrix rows, mitigating the irregularity in accessing streaming matrix rows during inference. Moreover, we introduce a multi-pass zero-detection mechanism on bit-trees in hardware to effectively address partial sum merging bottlenecks by circumventing explicit addressing through low-overhead control logic.

2.2.2 **Row Access Reorganization:** The conventional method for computing row-wise product faces a memory bottleneck when fetching streaming rows [13, 33, 42]. This occurs because while row-wise product reduces output partial sum traffic, it comes at the cost of heightened traffic of the streaming matrix. During sparse matrix multiplication, these streaming matrix rows are accessed frequently and randomly depending on the positions of non-zero elements in the stationary matrix, as depicted in Figure 4(b). For instance, computing the highlighted output row in the figure necessitates accessing both the first and fifth rows in the streaming matrix. Although these rows are retrieved on-chip, it's uncertain whether they will be utilized for the subsequent output row's calculation, which is totally contingent upon the non-zero coordinates of the next stationary row. Our objective here is to maximize the reuse of these on-chip fetched rows, thereby decreasing off-chip traffic and minimizing the likelihood of main memory access stalls when a row is not present on-chip. To achieve this, we propose a metric to evaluate streaming row traffic based on stationary rows' characteristics. The execution order of stationary input rows is reorganized according to their dissimilarity. Similar rows are processed sequentially, improving data reuse and reducing off-chip traffic for streaming rows. Importantly, this reorganization is informed by a thorough analysis of the dataflow and does not require any information about the streaming matrix.

The row-wise product dataflow enables parallelism at multiple levels. Processing one stationary row independently produces one output row, and we face no sequential interdependency between the execution of different rows in the stationary matrix. This parallelism inherent in row-wise product yields the ability to process rows out of order. We define a dissimilarity index between every ordered pair of stationary matrix rows. Revisiting the dataflow, dissimilarity, in this context, quantifies the additional streaming rows that must be retrieved when consecutively processing stationary rows. Specifically, it is the number of positions in the second row that lack a corresponding non-zero value in the first row. For instance, in Figure 4(c), the dissimilarity between row *1* and row *2* is 1. This is because the streaming row required by row *2* isn't already fetched during the execution of row *1*. The dissimilarity index is defined for every pair of consecutively executing rows in a matrix. Our approach employs this dissimilarity awareness at the row level to efficiently execute stationary input matrix rows based on memory-access patterns. During the processing of the $n+1$'th row, streaming rows corresponding to these additional non-zeros over the n'th row necessitate fetching from the main memory for multiplication. The dissimilarity index for an entire matrix is thus the sum of dissimilarities between each pair of consecutive rows. We leverage this dissimilarity metric to establish an optimization criterion for offline reorganizing and grouping of stationary rows.

Analogous to the principle of batching, where on-chip-fetched weights are reused, reorganizing the execution of stationary matrix optimizes the reuse of streaming rows already fetched. For each row in the stationary input matrix, if there exists a hypothetical parent row with no dissimilarities, the row is absorbed as a sub-pattern of the parent pattern. This implies that processing a row after its parent row wouldn't necessitate extra off-chip access. For example, in Figure 2(c), no additional access would be needed if row *2* is executed after row *n+1*. Analyzing the rows results in a set of distinct patterns with dissimilarity indexes greater than zero, labeled as *p1, p2, p3... pn*, as depicted in Figure 4. By leveraging insights into global patterns in the matrix and considering the maximum available streaming rows storage capacity on-chip, we can reorder stationary row access to maximize data reuse. Parent patterns typically contain a higher number of non-zero elements, where most of the latency in fetching streaming rows can be efficiently masked. Subsequently, processing all sub-patterns of a parent pattern sequentially reduces redundant off-chip traffic significantly.

We then analyze the dissimilarity among the parent patterns within the matrix. We group rows with lower dissimilarities into balanced clusters for processing. For instance, if executing pattern *p2* after *p1* requires two additional streaming row accesses, while executing pattern *p3* after *p2* requires three extra accesses, it's logical to execute them in the sequence *p1*, *p2, p3*. Moreover, if patterns *pn* and *p1* exhibit high dissimilarity (indicating minimal reuse), we segregate such patterns into separate groups. These groups do not benefit from sequential processing and can be processed in parallel. Hence, patterns with lower dissimilarities are organized into balanced clusters, taking advantage of the row-wise independence of the stationary side. This reorganized and grouped

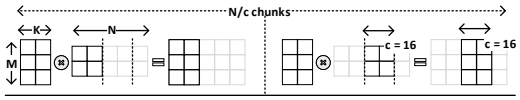

Figure 6: Tiled Row-wise product

execution collectively diminishes random access to the streaming matrix, consequently reducing unnecessary memory traffic.

2.3 Memory Access Optimizations

2.3.1 **Output Workspace:** Kjolstad et al. [21] introduced the notion of a workspace as a temporary tensor, typically dense, allowing for rapid insertion and random access in sparse matrix operations. When integrated at an architectural level, this workspace in the local memory is mapped to the main memory either as sparse or dense tensors, with explicitly decoupled data orchestration [30]. A dense output workspace enables efficient and only essential processing over non-zeroes in the input matrix while requiring minimal additional hardware for indexing into the workspace. In our execution model, this indexing translates to output partial sum redirection, which is already performed by the zero detection hardware in conjunction with crossbars, helping us exploit data locality. Further, upon analyzing the ML workloads, we find that, given the moderate sparsities on the inputs, the average sparsity of output matrices is merely 8.1%. Given our prior analysis of storage formats for variably sparse matrices, it becomes evident that maintaining a dense workspace for these matrices is preferable not only for quick insertion and access but also from storage and on-chip traffic standpoints. We further elaborate output workspaces through our tiling strategy ahead.

Algorithm 1 Output-Stationary, Tiled Row-wise Product

1: **Input:** Matrix MK, Matrix KN
2: **Output:** Matrix $C \in \mathbb{R}^{M \times N}$
3: **for all** chunks in matrix C **do** ▷ Figure 6
4: $C_{chunk} \in \mathbb{R}^{M \times c}$
5: **for all** rows in matrix MK **do**
6: $C_{row\text{-}slice} \leftarrow$ corresponding row in C_{chunk}
7: **for all** nz_i in MK row **do** ▷ Figure 4(b)
8: $strRow \leftarrow$ nnz in corresponding KN row slice
9: $psum_i \leftarrow nz_i \cdot strRow$
10: **end for**
11: $C_{row\text{-}slice} \leftarrow \sum_i psum_i$
12: **end for**
13: **end for**

2.3.2 **Tiled Row-Wise Product on Bit-Trees:** Through Figure 6 and Algorithm 1, we detail the data movement and tiling strategy employed for the multiplication of a compressed $M \times K$ (stationary) matrix with a compressed $K \times N$ (streaming) matrix. The stationary matrix is processed one row at a time, involving sequential access to non-zero elements and parsing the associated bit-tree to decode the relative position of the next non-zero. These positions are then utilized to retrieve streaming rows and their respective bit-trees for multiplication. The streaming matrix is partitioned into chunks with c=16 columns, and rows within a chunk are called row-slices. A chunk comprises M row-slices (line 4), and a slice encompasses four l1 nodes or up to four l2 leaves of the bit-tree, corresponding to 16 values in the original matrix. While processing all the row slices of a streaming matrix chunk, the corresponding partial sum (psum) slice of the output row remains stationary for accumulation, forming the output workspace (line 11). Each processing element that processes one output row has one output workspace register file. Streaming slices (line 8) within a chunk are managed using a scratchpad between successive executions of the stationary rows. The row-access reorganization strategy explained earlier is applied to the stationary matrix to maximize the reuse of streaming slices within the scratchpad.

This tiling of the sparse matrix according to its actual dimensions is data-agnostic and is known as coordinate space tiling [16]. When adopting coordinate-space tiling, we face a challenge when fetching the bit-tree l1 of a completely zero slice, necessitating a flush before resuming computation for the next row-slice. This could result in stalls when handling large hyper-sparse matrices, as they are unsuitable for efficient processing with two-level bit-trees and require higher-order bit-trees. ZeD's algorithm is generalizable and scalable to such hyper-sparse matrices, only requiring multiplexing of the same microarchitecture to support the higher-order bit-trees.

3 ZED ARCHITECTURE

We design ZeD based on the tiled row-wise product dataflow on the bit-tree storage format introduced previously. ZeD is structured in an output stationary format, focusing on stationary partial sum accumulation, enabling parallelism at multiple levels of granularity. The dataflow enables the computation of output rows independently and concurrently. A Processing Element (PE) can individually compute a single output row to exploit this parallelism. Given the nature of the dataflow, each PE works independently and in parallel and can start the execution of new rows asynchronously. Figure 7(a) illustrates a high-level overview of our Architecture. Employing N PEs in parallel enables the simultaneous computation of N rows of the output matrix. Furthermore, within each PE, we exploit the opportunity for parallelism during the scalar-vector multiplication of non-zero elements from the stationary matrix with the non-zero vectors formed by the streaming matrix rows. Given the ordered nature (sorted by position) of streaming data within a row, this operation can be carried out concurrently alongside zero-skipping on bit-trees, with metadata management reserved solely for the final merge operation that accumulates stationary partial sums.

Besides reducing storage overhead, the bit-tree enables an efficient multi-pass skipping mechanism. This multi-pass skipping structure facilitates myopic (near-sighted) lookahead zero-detection within each level, enabling efficient indexing into the subsequent level at minimal hardware cost.

3.1 Processing Element

3.1.1 **Zero Detection Unit:** The Zero Detection Unit (ZDU) is the key component for decoding a bit-tree. Given the ordered nature of streaming data, the ZDU serves as a simple hardware extension designed for on-the-fly decoding of each level of the bit-tree, which synchronizes non-zero values to their actual position in the streams.

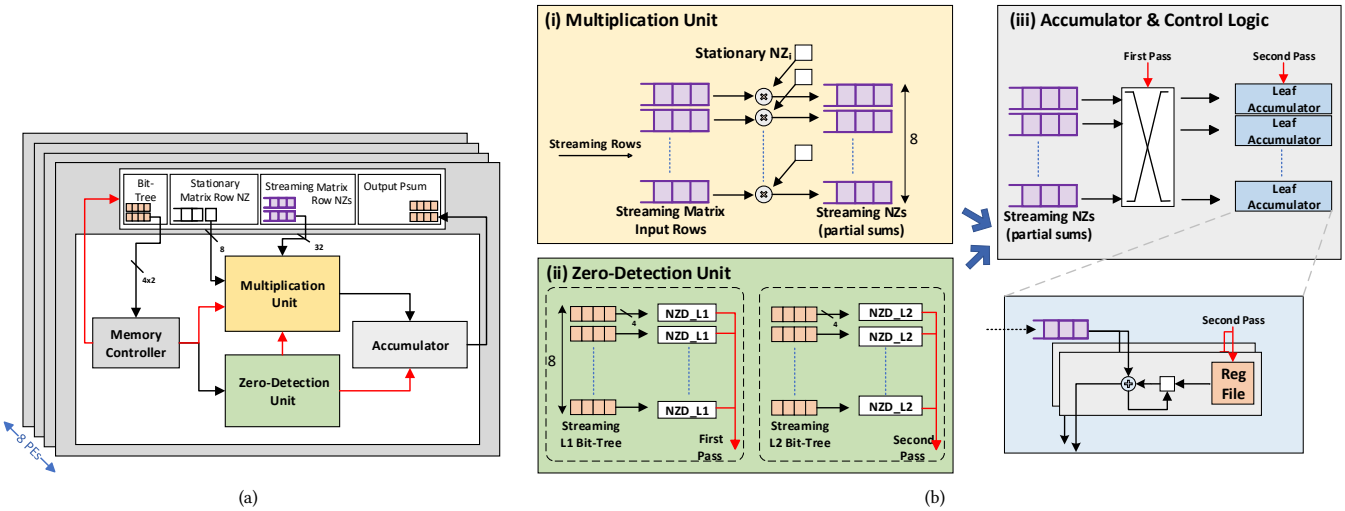

Figure 7: (a) Architectural Overview: datapath (black) & control flow (red) (b) Working of various components of a PE

We use "zero-detection" and "non-zero-detection" interchangeably due to their analogous nature over a binary vector. The ZDU detects the first 'set' bit in a leaf, returns its position, and masks, i.e. 'unsets' the bit (denoting 'completed') before moving to the next cycle. This enables decoding the position of non-zero elements within a leaf while skipping zero positions every cycle till the leaf has no non-zero left. Given that the tiling strategy at l1 and the sizes of leafs at l2 form packed bits of four, processing only four bits at a time is sufficient, regardless of the level. Consequently, these 'myopic' units only necessitate bitwise operations on four bits every cycle to identify non-zero positions. For example, if a leaf has three non-zeros, the ZDU returns their positions in order over three cycles.

The ZDU is designed to meet the requirements of our multi-pass skipping mechanism. Depending on the depth of the bit-tree vector being parsed, the output of a zero-detection unit (corresponding to the n'th pass, where n is the level) is employed as a control signal to manipulate the merge operation in the accumulator. We later see that this myopic, bitwise non-zero detection hardware collectively occupies less than 3% of the total on-chip area.

3.1.2 **Multiplication Unit:** Each PE has a multiplication unit consisting of eight SIMD-style multipliers. The multipliers are dedicated to executing scalar multiplication of non-zero elements from the stationary row with their respective streaming matrix rows. The memory controller coordinates the arrangement of these streaming matrix rows, corresponding to each stationary non-zero in the row-wise dataflow elucidated earlier. This scheduling is synchronized with the input bit-trees streamed to the Zero-Detection unit, which produces accumulator control signals for each multiplier output. The Multiplication unit produces partial sum row-slices streamed to the accumulator for the partial sum accumulation.

3.1.3 **Accumulators:** Using bit-trees and zero-detection enables a priori lookahead into the merging operations, which is one of the known inefficiencies in the dataflow. As discussed earlier, we maintain a dense workspace for accumulation in the output row. The dense workspace is realized as a register file. Utilizing 4 × 2 leaf accumulators (Figure 7(b)) enhances efficiency in mergers within a row-slice. Each leaf accumulator retains a stationary leaf (four elements) in its local register file. The first pass configures the crossbar, directing the multiplier output to the appropriate leaf accumulator. Subsequently, the second pass determines the precise position within the designated leaf for accumulation. Given that a leaf accumulator keeps four values stationary, the second pass serves as a two-bit control signal to index into the specific register among the four in the register file for accumulation.

3.1.4 **Control Path:** We first parse the top (l1) level of our streaming bit-tree wherein the first pass output from the non-zero detection unit enumerates non-zero second-level leafs. Each zero-skipped at l1 corresponds to four zeroes in l2. For a completely zero l1, the corresponding datapath has to flush and restart computation for the next row-slice. The first pass output is also used to determine the configuration of the crossbar. This essentially determines the leaf to which the value belongs and thus enables redirecting, i.e. mapping each value to its designated leaf's accumulator. Chunk values are padded to the nearest four for memory alignment. Using the first pass results we index into the second level of our bit-tree. Non-zero-detection on the second level yields the second pass output that indexes into local positions (between 0-3) of non-zero values inside a leaf. For an adder within each stationary leaf accumulator in Figure 7(b), the second pass acts as a control signal for register selection between the leaf-width-sized local register file, i.e. indexing into the workspace. The second pass is a two-bit signal for four-element leafs.

The leaf accumulators are decoupled from the multiplier output using buffers. Each of these last-level accumulators functions with no leaf inter-dependence, i.e., one leaf accumulator only accumulates values in four locations in the dense space. Buffers enable congestion mitigation by reassigning work to leafs that are free after finishing the accumulation of their assigned row-slices of an entire chunk. Leaf accumulators require synchronization only after processing one chunk. Analysis of our architecture over various

Workload	Name	Dimensions	Sparsity %	
		M, K, N	MK	KN
CNN	R9	64, 576, 3136	50.93	55.98
Resnet50	R19	512, 128, 784	63.61	46.98
layers	R29	256, 1024, 196	82.80	39.41
	R39	256, 2304, 196	80.56	67.43
	R49	2048, 512, 49	84.69	69.75
ViT	DeiT-B	196, 196, 768	90.07	-
NLP	BERT-B	384, 384, 768	79.52	-
Synthetic	Syn1	1000, 300, 800	30.00	25.00
Synthetic	Syn2	1000, 300, 800	40.00	85.00

Table 2: Workload Sparsities and Dimensions

sparse ML matrices shows that the buffers are small and incur low overheads as the producer (multiplication unit) and consumer (accumulators) have identical parallelism, and buffers primarily function to mitigate leaf-level load imbalances.

This architecture is easily scalable to accommodate higher-level bit-trees and/or varied sizes of leafs, requiring adjustments in the tiling scheme, multiplexing the zero-detection hardware and/or changing its width. Furthermore, the hardware cost remains minimal as the zero-detection unit processes only small (myopic) and deterministic lengths of bit-vectors at any level of the bit-tree, distinguishing bit-trees from naive bitmaps.

4 EXPERIMENTAL EVALUATION

4.1 Workloads

We analyze our architectural decisions, corresponding performance, utilization, and ZeD memory traffic across a wide range of real-world ML workloads. In order to showcase the effectiveness of our approach and underscore the importance of segregating pruning algorithms from the hardware they operate on, we examine various machine learning models. We choose models that are pruned using state-of-the-art unstructured pruning techniques, resulting in less than 1% accuracy loss. These techniques are oblivious of the hardware, rendering absolute freedom in pruning. We select the ResNet50 model on the ImageNet-1K dataset pruned to an average of 80% sparsity [12]. Given the variations in sparsity within a model, we evaluate Resnet50 at two granularities: we look at its individual layers (R9, R19, up to R49) as well as the average performance on the entire model (Resnet50). Like prior works, the convolution operations in the CNN model are mapped as matrix multiplication operations using a Toeplitz transform [6]. We consider the sparse-dense score-value matrix multiplication for evaluating sparse attention [24, 36]. To obtain the sparse scores, we use the pruned BERT-Base model (language) from [12], evaluated on the SQuAD dataset with a context length of 384 and the DeiT-Base model (vision) from [41] that is aggressively pruned to up to 90% sparsity. Additionally, we introduce two synthetic workloads designed to exhibit characteristics absent in these limited real-world models. *Syn1* simulates a relatively denser matrix multiplication scenario, whereas *Syn2* is meant to demonstrate the efficacy of our approach in efficiently processing dense stationary and sparse streaming matrices. The actual sparsity levels and the dimensions of the matrices are shown in Table 2.

4.2 State-of-the-art Baselines:

We use three state-of-the-art accelerators for comparison, Dual Side Sparse Tensor Core (DSTC) [38], Eyeriss [17], & Flexagon [27]. DSTC introduces zero skipping to the tensor-core architecture using a tiled bitmap format. It implements an outer-product matrix multiplication on tensor cores of a GPU where the adapted bitmap allows it to skip warps corresponding to entirely sparse tiles of the output matrix. The sparse version (v2) of Eyeriss implements an inner-product matrix multiplication. It stores weights and activations using a CSC format and implements a comparison-based skipping to eliminate ineffectual accesses and computations on weights corresponding to zeroes in activations. Flexagon introduces a reconfigurable Network-on-Chip (NoC) that can support multiple kinds of dataflow while using the CSR format for its architecture. We choose the three architectures to cover the fundamental techniques utilized in contemporary sparse accelerators in the literature. DSTC encompasses all the functionalities of sparsity-supporting tensor-core variant architectures while employing a low-cost bitmap format [38]. Eyeriss shows skipping through CSR storage [17]. Meanwhile, Flexagon, through reconfigurability, incorporates characteristics from various hyper-sparse matrix accelerators and tailors them for deep learning tasks [27].

4.3 Evaluation Framework

We design ZeD in RTL, and synthesize it with a target frequency of 500MHz using the Synopsys Design Compiler on a commercial 22nm technology node for area and power estimations. We use a subset of evaluation inputs to determine an average activity level for ZeD's components and finally use a 15% higher activity level for conservative power estimates. For further exploration and evaluation of design-tradeoffs, we develop a cycle-accurate simulator for ZeD. To better understand ZeD's results, we also model ZeD-naive without row-reorganization. We estimate the performance, memory traffic, and utilization of various components of the architecture using the cycle-accurate simulator. We use an eight-PE model of ZeD with a total of 50KB on-chip SRAM to store the stationary non-zeroes, streaming matrix rows, output partial sums, and corresponding bit-trees. The SRAM is paired with a 16GB/s off-chip DRAM memory. We use CACTI 7.0 [5] to model these memories at the same tech node.

We model the baseline architectures with the same theoretical peak performance (corresponding to 64 INT8 MAC units) and then compare the achieved performance and efficiency. We use Sparseloop [40] to model DSTC-like, Eyeriss-like, and representative dense tensor core-like baselines. Sparseloop also reports the area and energy consumption of these architectures. We use the STONNE [28] simulator and the numbers reported in the paper for Flexagon's performance, power, and area results. We model the available configurations of Flexagon with a 64-unit wide multiplier network and scale up the performance for the other dataflow configurations according to the original paper for a fair comparison.

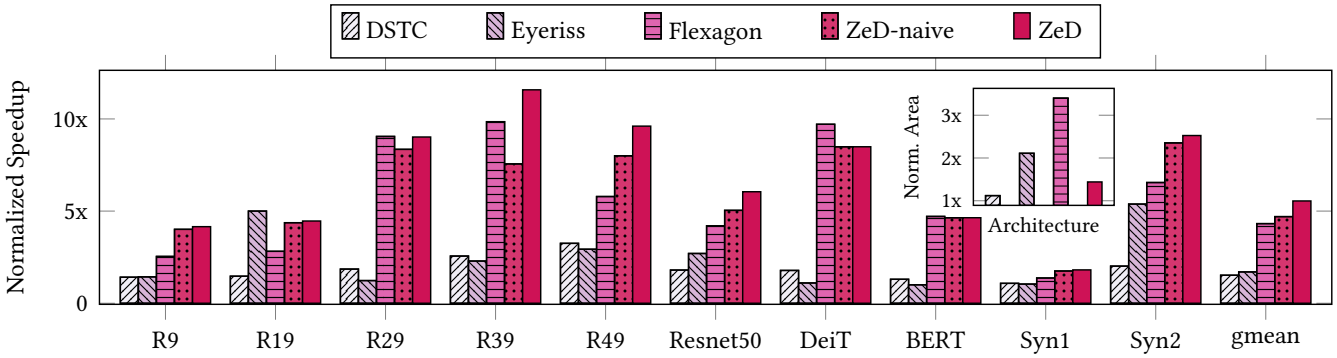

Figure 8: Speedup with respect to the dense baseline. Inset: Area of the architectures normalized to the dense baseline

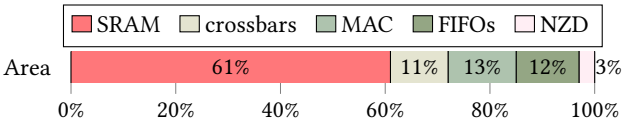

Figure 9: Breakdown of ZeD area

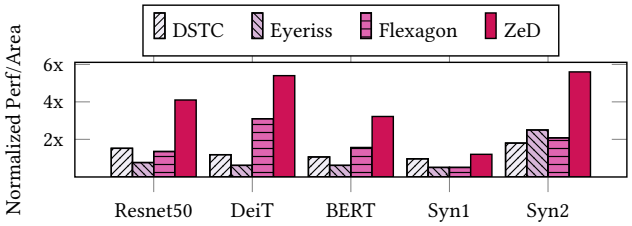

Figure 10: Application-wise Performance per Area of the architectures, normalized to the dense baseline

4.4 Experimental Results

We first evaluate the on-chip area and power consumption of ZeD. As seen earlier, bit-trees require negligible storage space compared to their pointer-based CSR counterparts. Figure 9 demonstrates a breakdown of the total 0.235mm^2 on-chip area of ZeD. The on-chip SRAM, which stores the non-zeroes in the stationary row, output workspace, bit-trees, and the streaming matrix rows, takes up 62% of the on-chip area. The collective footprint of the non-zero detection units in each PE (NZD) contributes to merely 3% of the total area. Moreover, only 4.5% of the total 59mW of on-chip power is consumed by the non-zero detection units, which shows its outstanding performance with minimal overheads. This analysis further validates that our approach of using a non-elaborate, highly compressed format and introducing additional hardware on-chip to detect and process the compressed values dynamically is more efficient than storing the matrices in a comparatively elaborate elaborate CSR format.

4.4.1 **Performance Analysis:** We next evaluate the performance of ZeD with respect to the state-of-the-art accelerators on the workloads from Table 2. Compared to the dense baseline, ZeD achieves a maximum speedup of 13.4x and a gmean speedup of 5.9x. Figure 8 compares the speedup various accelerators can achieve over the dense baseline. DSTC uses multiple bitmaps and relies on a costly global buffer for extensive partial sum aggregations associated with outer-product, which reduces scalability and can cause significant slowdowns when dealing with moderately sparse inputs that accumulate to very dense outputs. Owing to this inefficient dataflow and inefficient bitmap access, ZeD constantly offers more than 2x speedup and gmean 3.5x speedup over DSTC. Eyeriss employs costly comparisons to skip accessing and processing weights corresponding to zero-valued activations. Further, the CSC format that enables this skipping optimization proves to be quite costly, leading to a higher on-chip resource requirement but achieving only modest speedups for most of the workloads. Consequently, ZeD sees a gmean 2.9x speedup on Eyeriss.

Flexagon's reconfigurable NoCs enable it to statically reconfigure its architecture's data flow to align with specific sparsity patterns' requirements, thereby demonstrating efficient handling of variable sparsities. Sparse attention matrices like BERT and DeiT require decoding and streaming in non-zeroes only from the first input matrix. Despite high sparsity, these computations maintain regularity as the second input matrix is dense, enabling strided access and streamlined merging. In contrast to DSTC, Flexagon and ZeD employ a similar execution model in this context. Still, overall, ZeD attains gmean 1.2x speedup on Flexagon across such workloads.

To put Flexagon's performance into perspective, we conduct an iso-area performance comparison of the different accelerators for our workloads. Flexagon consumes more than 3x the area of DSTC and more than 2.2x the area of ZeD, as shown in Figure 8. The reconfigurable interconnects help mitigate irregularity but come at the cost of significantly higher area and power consumption, as also noted in the original work [27]. Further, Flexagon's use of compressed-sparse rather than bit-sparse formats requires higher on-chip memory and area requirements and unnecessary memory traffic in an already memory-intensive task. The memory requirements are further accentuated given the redundancy required to support all kinds of dataflow on the architecture. ZeD achieves gmean 2.7x better performance per area across the evaluated workloads on Flexagon, attributed to our low-cost algorithm-architecture co-design that can handle matrices with variable sparsities efficiently.

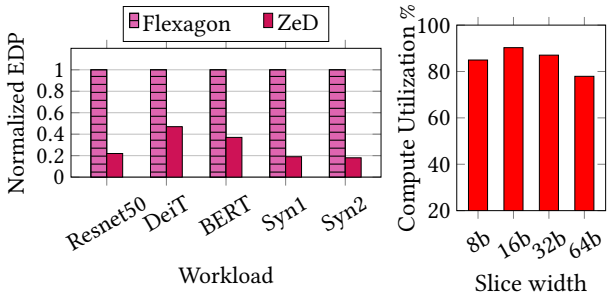

Figure 11: (a) EDP of ZeD normalized to Flexagon (lower is better); (b) Sensitivity to storage format parameters

Overall, ZeD achieves gmean 3.2x better performance per area over the three baselines for all the workloads in Figure 10. For a holistic analysis, we also compare the Energy Delay Product (EDP) of ZeD to Flexagon. Figure 11 highlights the significant EDP benefits of ZeD over Flexagon across the wide range of input matrices.

Ablation Studies: To better understand the enhancements from ZeD's architecture and row-reorganization, we conduct ablation studies to demonstrate these improvements separately. A naive ZeD (ZeD-naive) model without reorganization shows a gmean 2.62× improvement over the baselines, which can be solely attributed to ZeD architecture. Further, introducing row-reorganization reduces the proportion of memory access stalls from 34% to about 19% of the overall execution time (40% reduction). This improvement translates to an additional 1.25x performance increase (up to 1.7×) over the ZeD-naive model. Combined, these result in the gmean 3.2× improvement in iso-area performance over the baselines.

Since row-reorganization can currently only be applied offline, transformer matrices with dynamic sparsity, such as DeiT and BERT, do not incorporate row-reorganization for fair final results. However, recent research [41] indicates the potential for static sparsity in transformers, which, if incorporated, would enable ZeD to achieve an additional 1.4× speedup in these evaluations. The benefits of row-reorganization are particularly noticeable in matrices like R29, R39, and R49, where the weights have higher sparsity, causing significant dissimilarity and stalls in ZeD-naive.

4.4.2 **Sensitivity to storage format parameters:** ZeD's architecture design is primarily influenced by the packing size of the last-level leaf node and the width of the row slice. Figure 11 illustrates the impact of varying the row-slice width from 8 bits to 64 bits for four-element leafs. The utilization of ZeD compute elements reaches its peak at a slice width of 16 bits, with a notable decrease as we increase the slice width to 64 bits. Working with very small row slices risks encountering a larger number of entirely zero slices in the first pass on the bitree L1. This can lead to unnecessary stalls when transitioning to the next row slice within a chunk of the matrix without any computing. Conversely, larger row slices, implying larger tiles, are more susceptible to load imbalance due to uneven distribution of non-zero elements. This can result in congestion at leaf accumulators, requiring large decoupling buffers or reduced compute utilization. Furthermore, larger row slices diminish the effectiveness of the row reorganization strategy due to the increased probability of dissimilarity between two rows.

By exploring the different packing strategies in a leaf, we find that the average utilization of compute elements across these configurations stays within 3% of the optimum. Nevertheless, packing 2-bits in a leaf entails more complex control and crossbar logic, whereas 8-bit leaves demand more complex zero-detection logic and increased padding requirements. Our analysis suggests that a 16-bit slice of 4 four-bit leaves achieves the highest efficiency. *Caveat:* On processing a sparse matrix multiplication with hyper-sparse scientific matrices using the same leaf sizes and tiling strategies, ZeD suffers from low utilization. We see only 28% utilization for one such test application, pointing to the need for scaling up to higher-level bit-trees and/or larger tiles. As discussed earlier, the scope of this work is variably sparse ML workloads with < 99% sparsity. Pruning aims to trim redundant weights and activations in ML models while preserving essential information, leading to moderately sparse matrices compared to the inherently sparse scientific matrices. However, it is worth highlighting that our algorithm and architecture can easily adapt and generalize to handle hyper-sparse matrices. This adaptability requires straightforward modifications to packing levels and tiling in the software, along with adding more parallel hardware units for zero detection.

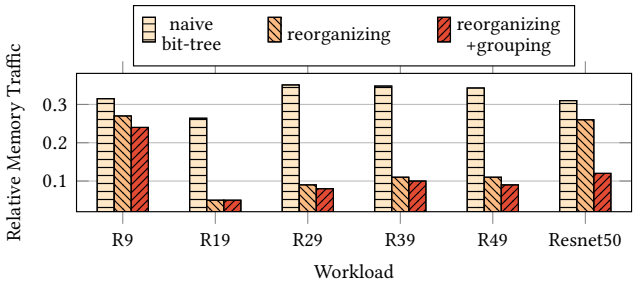

Figure 12: Reduction in memory traffic compared to CSR SpMM (lower is better)

4.4.3 **Effect of row reorganization:** The proposed matrix row reorganization and regrouping in ZeD significantly reduces off-chip memory traffic. Identifying sparsity patterns facilitates effective data reuse for each byte of streaming matrix data retrieved on-chip, leading directly to decreased energy consumption associated with main memory access. The average pre-processing time for row access configurations on a matrix in our workloads is approximately 0.7 seconds, a cost easily amortized across multiple inferences utilizing the same weight matrices. Figure 12 shows the relative memory traffic of ZeD in comparison to a naive CSR implementation of row-wise product matrix multiplication on Resnet50 layers with varying sparsities. Naive bit-tree denotes a version of ZeD with no memory access optimizations. When dealing with layers that do not entirely fit within the PE scratchpads, exploiting the absence of sequential interdependencies in output row computations and reorganizing stationary row executions result in a substantial 2.8x reduction in memory traffic over a naive implementation. By grouping (and separating) patterns based on dissimilarity scores and executing them

on different PEs while ensuring local data reuse, an additional 1.2x reduction in memory traffic is attained. Consequently, we achieve a total 3.4x reduction in memory traffic overheads compared to a naive model of ZeD.

5 RELATED WORKS

Storage Formats: We categorize prior research according to their use of various sparse storage formats. There has been lots of industry adoption [26] and research [23, 39] on utilizing structured sparse formats, particularly in ML. As discussed earlier, these formats enable regular processing but expose the accuracy-compression tradeoff due to hardware-enforced limitations.

In recent literature, the CSR format and its modified versions have emerged as the most prevalent sparse storage formats [4, 10, 18, 31–33, 42] for unstructured sparse matrices. These formats and associated accelerators are typically tailored for hyper-sparse computations in high-performance and scientific computing domains, often implemented on distributed scale-out systems. While these coordinate-pointer-based formats efficiently store and index matrices within scientific computing, our prior analysis reveals their inefficiency when evaluated with moderately sparse workloads. Some works like Flexagon [27], EIE [15], and Eyeriss [17] have adopted these formats for handling sparse ML workloads. Our comparison and analysis of these works in the previous section reveal that while the utilization of CSR-like formats, coupled with architectural enhancements, can yield satisfactory performance, it comes at the expense of increased on-chip resource utilization due to higher storage requirements and associated traffic. Additionally, these formats exacerbate the memory bottleneck, which is particularly bad for tasks like Sparse Matrix Multiplication, which is already bandwidth-bound.

While less prevalent, some bitmap-based storage formats have also been adopted previously, particularly in response to the inefficiencies observed in the CSR format [11, 31, 38]. Although these approaches are efficient in terms of storage, they often lack a mature indexing strategy and encounter limitations of the bitmap when indexing longer run-length of zeroes in sparse operands, necessitating expensive comparisons to eliminate ineffective compute.

ExTensor [16], DSTC [38], and SMASH [19] employ hierarchical storage techniques to facilitate skipping at coarser granularities. ExTensor uses a tiled CSR format to hierarchically intersect and eliminate all zero dimensions of computations across tensor tiles. DSTC and SMASH utilize hierarchical bitmaps for this elimination. SMASH incorporates hardware-accelerated explicit indexing into non-zero blocks using a bitmap management unit that exposes itself to the sparse application in software as an ISA extension. DSTC facilitates hierarchical bitmap-based operations while utilizing an outer-product matrix multiplication approach to skip entirely zero output tiles. While skipping at a coarser granularity proves effective for hyper-sparse matrices, it introduces inefficiencies for denser matrices, where they are forced to store and compute on a lot of zeroes with a higher likelihood of encountering non-zeros in each tile. Moreover, these approaches often depend on using explicit comparisons for intersections or costly output accumulations, failing to fully leverage their storage capabilities with an efficient dataflow and architecture. Fibertree [34] introduces a hierarchical storage abstraction for sparse matrices, allowing partitioning and processing of matrices at multiple granularities. This abstraction offers flexibility and expressibility for working with sparse tensors. Fibertrees can be realized in hardware using various sparse storage techniques, including CSR and even bit-trees.

Our proposed algorithm and architecture can seamlessly integrate at the lowest level within the hierarchy of these abstractions. This demonstrates ZeD's adaptability and generalized capability, showing its potential to enhance existing coarse-grained skipping techniques for various sparse matrix acceleration architectures.

Sparse Dataflows: Sparse accelerators can be broadly classified based on the dataflows they employ: Eyeriss [17] and SIGMA [31] utilize a naive inner product approach to compare and eliminate ineffective computations. SpArch [43] and OuterSPACE [29] accelerate SpMM using an outer-product dataflow similar to DSTC. They face challenges due to the high output partial sum merging cost. MatRaptor [33], InnerSP [4], and GAMMA [42] employ different storage, preprocessing techniques and merging hardware for their row-wise product dataflow implementations. Flexagon resorts to reconfigurability to efficiently support the three major dataflows mentioned here. The evaluation against Flexagon, DSTC, and Eyeriss thus encapsulates the major contributions of the other works and provides a comprehensive understanding of ZeD.

6 CONCLUSION

We present ZeD, with generalized architecture design considerations to tackle variable, unstructured, and random sparsity in ML models. We highlight the inefficiencies of contemporary sparse accelerators in handling matrices with variable degrees of sparsity. We exploit sparsity by efficient packing, storage, retrieval, and consequent traversal of highly compressed bit-tree structures and sparsity-pattern-based memory accesses. Our techniques combining a row-wise product dataflow with a bit-tree compression format and zero detection hardware enable parallelism at multiple granularities. The algorithmic and architectural enhancements enable efficient processing of matrices across a wide spectrum of sparsities commonly seen in ML workloads.

ACKNOWLEDGMENTS

We thank the Shepherd & the anonymous reviewers for their invaluable feedback, which helped improve the work. This research is supported by the National Research Foundation, Singapore, under its Competitive Research Programme Award NRF-CRP23-2019-0003.

REFERENCES

[1] Shivam Aggarwal, Kuluhan Binici, and Tulika Mitra. 2024. CRISP: Hybrid Structured Sparsity for Class-Aware Model Pruning. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. https://doi.org/10.23919/DATE58400.2024.10546782

[2] Shivam Aggarwal, Hans Jakob Damsgaard, Alessandro Pappalardo, Giuseppe Franco, Thomas B. Preußer, Michaela Blott, and Tulika Mitra. 2024. Shedding the Bits: Pushing the Boundaries of Quantization with Minifloats on FPGAs. arXiv:2311.12359 [cs.CV] https://arxiv.org/abs/2311.12359

[3] Bahar Asgari, Ramyad Hadidi, Hyesoon Kim, and Sudhakar Yalamanchili. 2019. ERIDANUS: Efficiently Running Inference of DNNs Using Systolic Arrays. *IEEE Micro* 39, 5 (2019), 46–54. https://doi.org/10.1109/MM.2019.2930057

[4] D. Baek, S. Hwang, T. Heo, D. Kim, and J. Huh. 2021. InnerSP: A Memory Efficient Sparse Matrix Multiplication Accelerator with Locality-Aware Inner Product Processing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, Los Alamitos, CA, USA, 116–128. https://doi.org/10.1109/PACT52795.2021.00016

[5] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (jun 2017), 25 pages. https://doi.org/10.1145/3085572

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan M. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *ArXiv* abs/1410.0759 (2014). https://api.semanticscholar.org/CorpusID:12330432

[7] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (dec 2019), 25 pages. https://doi.org/10.1145/3322125

[8] Hongxiang Fan, Thomas Chau, Stylianos I. Venieris, Royson Lee, Alexandros Kouris, Wayne Luk, Nicholas D. Lane, and Mohamed S. Abdelfattah. 2023. Adaptable Butterfly Accelerator for Attention-Based NNs via Hardware and Algorithm Co-Design. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois, USA) *(MICRO '22)*. IEEE Press, 599–615. https://doi.org/10.1109/MICRO56248.2022.00050

[9] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. arXiv:2103.13630 [cs.CV]

[10] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 21 (feb 2022), 49 pages. https://doi.org/10.1145/3508041

[11] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

[12] Manas Gupta, Efe Camci, Vishandi Rudy Keneta, Abhishek Vaidyanathan, Ritwik Kanodia, Chuan-Sheng Foo, Wu Min, and Lin Jie. 2024. Is Complexity Required for Neural Network Pruning? A Case Study on Global Magnitude Pruning. arXiv:2209.14624 [cs.LG]

[13] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (sep 1978), 250–269. https://doi.org/10.1145/355791.355796

[14] Tae Jun Ham, Sungjun Jung, Seonghak Kim, Young H. Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W. Lee, and Deog-Kyoon Jeong. 2020. Accelerating Attention Mechanisms in Neural Networks with Approximation. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), 328–341. https://api.semanticscholar.org/CorpusID:211296403

[15] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 243–254. https://doi.org/10.1109/ISCA.2016.30

[16] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[17] Yu hsin Chen, Tien-Ju Yang, Joel S. Emer, and Vivienne Sze. 2018. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9 (2018), 292–308. https://api.semanticscholar.org/CorpusID:131771552

[18] Eun-Jin Im and Katherine A. Yelick. 1999. Optimizing Sparse Matrix Vector Multiplication on SMP. In *SIAM Conference on Parallel Processing for Scientific Computing*. https://api.semanticscholar.org/CorpusID:42432358

[19] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 600–614. https://doi.org/10.1145/3352460.3358286

[20] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. 2023. Full Stack Optimization of Transformer Inference: a Survey. arXiv:2302.14017 [cs.CL]

[21] Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor algebra compilation with workspaces. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 180–192.

[22] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. 2022. The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 4163–4181. https://doi.org/10.18653/v1/2022.emnlp-main.279

[23] Z. Liu, P. N. Whatmough, Y. Zhu, and M. Mattina. 2022. S2TA: Exploiting Structured Sparsity for Energy-Efficient Mobile CNN Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 573–586. https://doi.org/10.1109/HPCA53966.2022.00049

[24] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 977–991. https://doi.org/10.1145/3466752.3480125

[25] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits. arXiv:2402.17764 [cs.CL]

[26] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. arXiv:2104.08378 [cs.LG]

[27] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3582016.3582069

[28] Francisco Muñoz-Martínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2021. STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 201–213. https://doi.org/10.1109/IISWC53511.2021.00028

[29] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. https://doi.org/10.1109/HPCA.2018.00067

[30] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 137–151. https://doi.org/10.1145/3297858.3304025

[31] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[32] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1022–1035. https://doi.org/10.1145/3466752.3480047

[33] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 766–780. https://doi.org/10.1109/MICRO50266.2020.00068

[34] V. Sze, Y.H. Chen, T.J. Yang, and J.S. Emer. 2020. *Efficient Processing of Deep Neural Networks*. Springer International Publishing. https://books.google.com.sg/books?id=iJ05zwEACAAJ

[35] Jan van Leeuwen. 1976. On the Construction of Huffman Trees. In *International Colloquium on Automata, Languages and Programming*. https://api.semanticscholar.org/CorpusID:37417891

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[37] H. Wang, Z. Zhang, and S. Han. 2021. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 97–110. https://doi.org/10.1109/HPCA51647.2021.00018

[38] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 1083–1095. https://doi.org/10.1109/ISCA52012.2021.00088

[39] Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel Emer. 2023. HighLight: Efficient and Flexible DNN Acceleration with Hierarchical Structured Sparsity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada,) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1106–1120. https://doi.org/10.1145/3613424.3623786

[40] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2021. Sparseloop: An Analytical, Energy-Focused Design Space Exploration Methodology for Sparse Tensor Accelerators. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 232–234. https://doi.org/10.1109/ISPASS51385.2021.00043

[41] H. You, Z. Sun, H. Shi, Z. Yu, Y. Zhao, Y. Zhang, C. Li, B. Li, and Y. Lin. 2023. ViT-CoD: Vision Transformer Acceleration via Dedicated Algorithm and Accelerator Co-Design. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 273–286. https://doi.org/10.1109/HPCA56546.2023.10071027

[42] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

[43] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 261–274. https://doi.org/10.1109/HPCA47549.2020.00030