Transitive Array: An Efficient GEMM Accelerator with Result Reuse

Transitive Array: An Efficient GEMM Accelerator with Result Reuse

Cong Guo*
Duke University
Durham, USA
cong.guo@duke.edu

Chiyue Wei
Duke University
Durham, USA
chiyue.wei@duke.edu

Jiaming Tang
Massachusetts Institute of Technology
Cambridge, USA
jmtang@mit.edu

Bowen Duan
Duke University
Durham, USA
bowen.duan@duke.edu

Song Han
Massachusetts Institute of Technology
Cambridge, USA
songhan@mit.edu

Hai Li
Duke University
Durham, USA
hai.li@duke.edu

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

Abstract

Deep Neural Networks (DNNs) and Large Language Models (LLMs) have revolutionized artificial intelligence, yet their deployment faces significant memory and computational challenges, especially in resource-constrained environments. Quantization techniques have mitigated some of these issues by reducing data precision, primarily focusing on General Matrix Multiplication (GEMM). This study introduces a novel sparsity paradigm, **transitive sparsity**, which leverages the reuse of previously computed results to substantially minimize computational overhead in GEMM operations. By representing transitive relations using a directed acyclic graph, we develop an efficient strategy for determining optimal execution orders, thereby overcoming inherent challenges related to execution dependencies and parallelism. Building on this foundation, we present the **Transitive Array**, a multiplication-free accelerator designed to exploit transitive sparsity in GEMM. Our architecture effectively balances computational workloads across multiple parallel lanes, ensuring high efficiency and optimal resource utilization. Comprehensive evaluations demonstrate that the Transitive Array achieves approximately **7.46**× and **3.97**× speedup and **2.31**× and **1.65**× energy reduction compared to state-of-the-art accelerators such as Olive and BitVert while maintaining comparable model accuracy on LLaMA models.

CCS Concepts

• **Computer systems organization** → **Single instruction, multiple data**; **Systolic arrays**.

*Cong Guo is the corresponding author of this paper.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731043

Keywords

Matrix Multiplication, Quantization, Sparsity, Transitive Array

ACM Reference Format:
Cong Guo, Chiyue Wei, Jiaming Tang, Bowen Duan, Song Han, Hai Li, and Yiran Chen. 2025. Transitive Array: An Efficient GEMM Accelerator with Result Reuse. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731043

1 Introduction

Deep neural networks (DNNs)[17, 35] have become foundational to modern artificial intelligence, demonstrating transformative potential across various applications. Among these, large language models (LLMs)[7, 14] stand out for their ability to generate human-like text, understand complex language, and perform advanced reasoning, making them crucial in natural language processing[6, 53]. However, as DNNs scale, they introduce substantial memory and computational overheads that challenge their deployment efficiency, particularly in resource-constrained environments [33, 50].

Quantization [30] has thus emerged as a critical research avenue, offering methods to reduce model size and computational load by representing parameters with lower precision while aiming to retain model accuracy. Most quantization approaches aim to reduce the data precision of general matrix multiplication (GEMM) operations, which are fundamental to DNNs [10, 32], as they represent the core computational workload in both training and inference. Recently, quantization has regained popularity in LLMs. AWQ [38] achieves a 4-bit compression of weights with negligible accuracy loss. Additional advanced algorithms (e.g., BitNet1.58 [57]) push the limits further by targeting 1-bit weight quantization. Through such innovations, quantization significantly enhances LLM efficiency and deployability.

Building on quantization, this study explores novel opportunities to accelerate GEMM operations. Bit-slicing has proven to be an effective technique in DNN accelerators [1, 9, 42, 49]. It decomposes each bit of a quantized matrix into multiple binary 1-bit matrices,

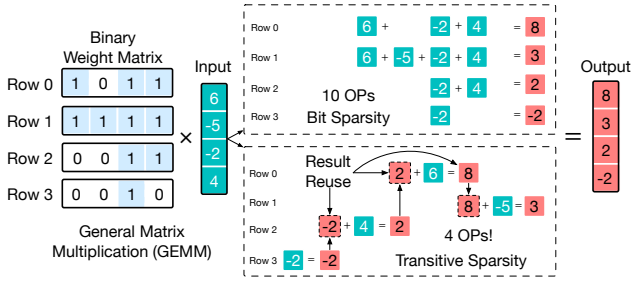


Figure 1: Comparison of bit sparsity and transitive sparsity (Ours) on the binary general matrix multiplication.

as illustrated in Fig. 2. By leveraging bit-slicing, we transform quantized integer-based GEMM into binary GEMM, as shown in Fig. 1. Bit-slice accelerators exploit bit-level sparsity, achieving computational reductions of up to 50–60% [9]. However, our analysis reveals even greater sparsity potential beyond bit sparsity, highlighting further opportunities for optimization.

This study introduces a novel sparsity paradigm termed **transitive sparsity**, which leverages previously computed results to minimize overall computations. As illustrated in Fig. 1, consider Row-0 of the binary matrix, represented by the bit pattern 1011, which requires the accumulation of values 6, −2, and 4, and Row-2, represented by 0011, which involves the accumulation of −2 and 4. Since both rows share the accumulation of −2 and 4, we can reuse the computed result of Row-2 for Row-0, significantly reducing the accumulation operations in GEMM. In the example shown in Fig. 1, transitive sparsity enables a 2.5-fold reduction in operations (from 10 to 4) compared to bit sparsity and a 4-fold decrease over dense GEMM. Furthermore, it is worth noting that transitive sparsity is algorithm-agnostic and lossless, preserving the computational accuracy of quantized GEMM operations. Our evaluation demonstrates that transitive sparsity theoretically reduces overall computations by 8× (i.e., 87.5% sparsity) for the LLaMA-7B [51] compared to dense GEMM without accuracy loss after the quantization.

To harness the benefits of transitive sparsity, we propose a novel computer architecture called the **transitive array**. However, we encountered several challenges in efficiently implementing transitive sparsity. First, transitive sparsity imposes strict requirements on execution order. For example, if Row-0 is computed before Row-2, it prevents reuse of Row-2's results. For weight tensors, we can precompute an optimal execution order offline. However, attention operations [54] in LLM present a complication, as they involve dynamically generated activations (e.g., the Query and Key tensors). This dynamic nature makes it challenging to determine execution order on the fly without incurring significant overhead. This dynamic limitation is also observed in most existing accelerators [9, 21, 42], which consequently lack adequate support for attention layers in their designs. Instead, these accelerators predominantly focus on fully connected (FC) layers with pre-processed weights. Second, even with an optimized execution order, the strict dependencies within transitive sparsity result in inherently serial execution, greatly restricting parallelism. Additionally, the irregularity of sparsity patterns makes it difficult to balance workloads across multiple parallel lanes, as varying sparsity levels lead to inconsistent computational loads and underutilization.

Fortunately, we identified certain properties within transitive sparsity that help address these challenges. Leveraging these insights, we can represent the transitive relations of the transitive sparsity using a directed acyclic graph, precisely a Hasse graph [12, 26]. This representation enables us to reduce the overhead of generating an optimal execution order, achieving linear complexity compared to the cubic complexity typically associated with GEMM. Following this, we assign rows of the bit-sliced matrix across multiple parallel lanes and have proven that this approach eliminates dependencies among parallel lanes. Additionally, to address workload imbalance, we introduce a simple yet effective method to ensure balanced distribution across multiple lanes before execution, making sparse execution more regular and highly efficient.

This work presents a comprehensive solution and novel architecture to enable transitive sparsity in GEMM operations. This work provides a new perspective on transitive sparsity and insights for future designs of more versatile GEMM accelerators. Compared to state-of-the-art accelerators like Olive [21], which leverages outlier-aware quantization, and BitVert [8], which employs bit-slice techniques, our solution achieves an approximate **7.46**× and **3.97**× speedup and **2.31**×, **1.65**× energy reduction, under similar model accuracy of LLaMA [51]. Our contributions are summarized as follows:

- We define a novel sparsity paradigm, **Transitive Sparsity**, which leverages previously computed results to minimize GEMM computations, thereby achieving significant reductions in operational overhead.
- We design a novel computer architecture, **Transitive Array (TA)**, tailored to exploit transitive sparsity and effectively address challenges related to execution order and parallelism. Our design also efficiently supports the quantization of Attention layers.
- Transitive Array is **multiplication-free**, achieving high efficiency by eliminating multiplication operations.
- Through comprehensive evaluations, we demonstrate the effectiveness of the **Transitive Array** architecture, showcasing substantial speedups and energy reductions compared to existing accelerators.

2 Background and Motivation

This section presents the motivation for transitive sparsity, grounded in our novel observations of sparsity patterns within GEMM.

2.1 Background: Quantization and Bit-slicing

As illustrated in Fig. 2, quantization and bit-slicing [1, 9, 42, 49] serve as foundational techniques for implementing transitive sparsity. After quantization, a 4-bit integer matrix of shape (4×4) is decomposed into four bit-level matrices. These matrices are then reorganized into a binary weight matrix with a shape of (16×4).

In general, for an S-bit quantized matrix of shape $(N \times K)$, we can reorganize it into a $(S \cdot N \times K)$ binary matrix. Building on previous work [1, 49], we demonstrate that maintaining sufficient precision in addition and accumulation operations—thereby preventing overflow or underflow—ensures that the bit-slicing method achieves the same results as the original quantized GEMM without any loss, even when adjusting the accumulation/reduction order for integers.

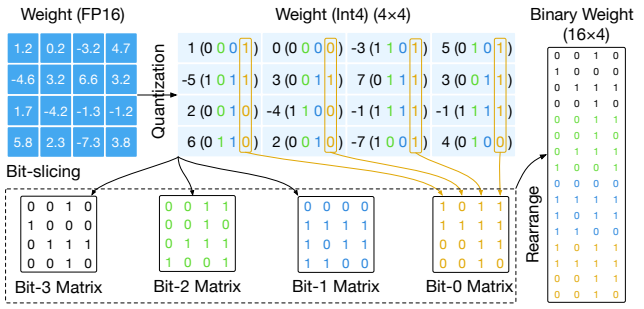


Figure 2: Quantization and bit-slicing. A weight tensor (FP16) is quantized into an Int4 weight matrix (4×4). Bit-slicing decomposes each bit level of the Int4 matrix and reorganizes them into a binary weight matrix of shape (16×4).

2.2 Motivation: Transitive Sparsity

Following these preliminaries, Fig. 3 illustrates our motivation for transitive sparsity. We first slice (❶) an 8-bit weight tensor with N rows and $T = 4$ columns into a matrix with $8 \times N$ rows and T columns, where each row represents a binary row with the T-bit width. We define each binary row as a **Transitive Row (TransRow, TR)**, which serves as the fundamental unit of our design. The width of each TransRow denoted as T, plays a crucial role in determining the efficiency of our approach.

In this study, we exemplify transitive sparsity using a 4-bit TransRow width. Transitive sparsity is also prevalent in higher bit widths (e.g., 8-bit). Each TransRow is identified by an index that indicates its corresponding bit level in the original integer. These TransRows are then accumulated with appropriate shifts. We employ 2's complement representation for integers. In practice, we represent all one-bits as positive 1, and all TransRows can be represented by **unsigned integers**, as illustrated in Fig. 3 ❷.

Subsequently, specific rows can be selected. For instance, consider four TransRows with values 11 (1011), 15 (1111), 3 (0011), and 2 (0010). Notably, some TransRows share identical 1s. For example, the TransRow representing 11 (1011) encompasses the same 1s as the TransRow representing 3 (0011). Exploiting this property allows us to eliminate redundant computations for TransRows. Originally, the TransRow 11 (1011) required three accumulations of 6, −2, and 4, skipping the 0 bit. As illustrated in Fig. 3 ❸, by reusing the result, we only need to perform an additional accumulation for 2 (the result of TransRow 0011) and 6, corresponding to the difference bits 1000 (i.e., 1011⊕ 0011 = 1000). We term this approach **Transitive Sparsity**, which reuses the results of preceding TransRows. Consequently, this characteristic of the bit-weight tensor significantly reduces the computational load for GEMM.

Challenges. However, as discussed in Sec. 1, adopting transitive sparsity in Deep Neural Networks (DNNs) introduces several challenges. For example, as shown in Fig. 3(b), the execution order of the four TransRows deviates from the original sequence. This strict execution order requirement renders parallel computing infeasible. Additionally, generating the execution order is highly complex. Specifically, we must determine which pairs of TransRows satisfy the transitive sparsity criteria among potentially hundreds of rows, leading to significant overhead in computing the execution order. A naive implementation would result in substantial inefficiencies

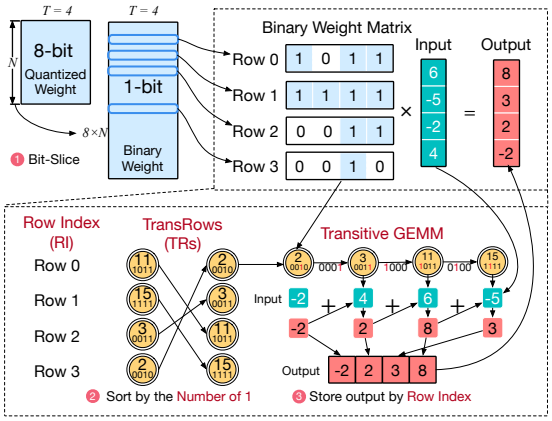


Figure 3: Motivation of Transitive GEMM.

in hardware utilization or increased time overhead. In this study, we aim to address these issues efficiently.

2.3 Efficient Representation

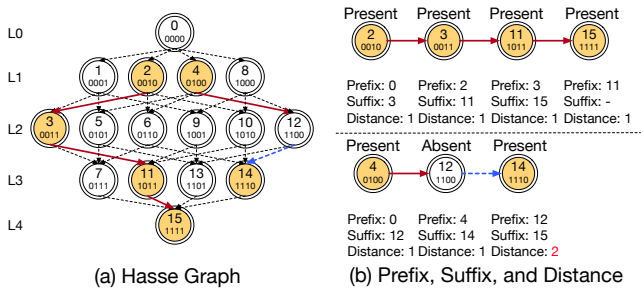
Fortunately, we have identified a beneficial and intriguing property within GEMM. As illustrated in Fig. 3, neighboring TransRows inherently possess a partial ordering relationship characterized by a single bit flip. This indicates that result reuse exhibits strong partial order relations. Inspired by this observation, we represent transitive sparsity using a **Hasse graph** [26], a specialized **Directed Acyclic Graph** that depicts a finite partially ordered set. Fig. 4(a) illustrates all partial order relations within a 4-bit TransRow. Normally, we ignore the Level 0, which can be skipped without computation. This representation effectively and significantly reduces the complexity of our design, which will be detailed in the following section.

To conveniently present our design, we introduce three concepts: **prefix**, **suffix**, and **distance** within the Hasse graph, as shown in Fig. 4(b). We define the preceding node (e.g., 3) that provides the reused result as the **prefix**, correspondingly, the latter as the **suffix**, e.g., Node 11. Based on transitivity, a further subsequent Node 15 can continue to reuse the result of its prefix, i.e., 11.

The **distance** between two nodes with a partial ordering is defined as the difference in their levels, corresponding to the difference in the number of 1s they contain. As illustrated in Fig. 4(b)-bottom, the **distance** is transitive and depends on the presence of specific TransRows and nodes. The distance is adjusted if an intermediate node is absent from the TransRows. For example, the distance between Node 4 and Node 14 is 2 because Node 12 is absent. However, Node 14 still considers Node 12 as its prefix, requiring Node 12 to pass the result from Node 4 to Node 14. Therefore, a larger distance implies that more add operations are required from the corresponding prefix TransRow. When the distance is 1, we achieve the most efficient result reuse between two different TransRows. Additionally, TransRows with identical values have a distance of 0 and can directly reuse each other's results.

2.4 Parallelism and Load Balance

After effectively representing transitive relations using a Hasse graph, we observe that nodes within the same level possess no partial ordering relationships, as they cannot share results. For instance, Node 2 cannot share its result with Node 4 because they differ in their bit representations. Consequently, transitive sparsity



(a) Hasse Graph (b) Prefix, Suffix, and Distance

Figure 4: Hasse graph and definition of prefix, suffix, and distance.

exhibits parallelism horizontally across each level of the Hasse graph.

Parallelism. Specifically, for an S-bit width transitive sparsity, the maximum parallelism is achieved at Level-$\frac{S}{2}$, corresponding to the binomial coefficient $\binom{S}{\frac{S}{2}} = \frac{S!}{\left(\frac{S}{2}\right)!\cdot\left(\frac{S}{2}\right)!}$. For example, Level 2 in a 4-bit graph exhibits a parallelism of 6, while Level 4 in an 8-bit graph exhibits a parallelism of 70. However, leveraging the maximum parallelism would result in the underutilization of other levels. Therefore, we typically utilize the granularity corresponding to Level 1, which is 4 for a 4-bit width and 8 for an 8-bit width TranSparsity. This strategic choice of granularity ensures balanced utilization across all levels of the Hasse graph, optimizing both computational efficiency and hardware resource allocation.

Data Independence. Additionally, each node in the Hasse graph can be transitioned by only one prefix node. For example, Node 11 can be transitioned from only one of its prefix nodes, such as Node 3, Node 9, or Node 10. Therefore, we can assign only one prefix to every node in the graph. Consequently, we can divide the graph into a forest comprising T independent trees starting from Level 1 for T-bit transitive sparsity. Therefore, we can safely divide the Hasse graph into multiple independent trees, each node having one in-degree edge, as shown in Fig. 4(a). This division ensures parallelism and that each node has a prefix node with correct partial ordering.

Load Balance. After dividing the Hasse graph to enable parallelism, it is essential to balance the resulting trees to prevent extreme workload imbalances across parallel lanes, which can lead to inefficient resource utilization and performance bottlenecks. We design an effective round-robin-like traversal method by leveraging the rule that each node has only one prefix. This method allows us to select an available prefix node for each node, thereby evenly distributing workloads among the trees only using a simple node number counter for supervision.

Finally, this section presents a streamlined representation of transitive sparsity and theoretically addresses the challenges of parallelism and load balancing. This framework paves the way for more efficient architectural designs, which are detailed in the subsequent sections.

3 Scoreboard Design

In this section, we introduce the **Scoreboard** mechanism, designed to efficiently compute the execution order using a Hasse graph representation. We propose two types of Scoreboard: **Static** and **Dynamic**. The static Scoreboard operates offline, generating execution information at the tensor level. In contrast, the dynamic Scoreboard processes smaller sub-GEMM operations online, ensuring compatibility with all GEMM kernels and enabling seamless deployment for large language models (LLMs). To accommodate the dynamic nature of attention mechanisms in Transformer architectures, the Scoreboard efficiently determines the execution order through dedicated hardware design, while simultaneously optimizing parallelism and workload balance at runtime. Both static and dynamic mechanisms leverage the same Scoreboard algorithm, as illustrated in Fig. 5, Alg. 1, and Alg. 2

3.1 Hamming-order Execution

Unlike general applications, which handle irregular and dynamic instructions, the instructions and data accesses in GEMM operations within DNN models exhibit strict regularity. This regularity enables us to predefine execution orders for all instructions before they are issued. Based on our analysis using the Hasse graph, the level of a node corresponds to the number of 1s (i.e., PopCount) in its binary representation. For instance, Node 11 (1011) belongs to Level 3. Leveraging this property, we sort the incoming *TransRows* based on their Hamming weight (i.e., PopCount) [34] instead of their integer values, defining this execution order as **Hamming-order**.

As shown in Fig. 5 ❶, TransRows are sorted exclusively by their **number of 1s**. Moreover, since there is no inherent partial ordering within nodes at the same level, the sorting mechanism does not enforce any order among nodes with identical PopCount. This approach significantly simplifies the sorter design, reducing its complexity and hardware overhead.

3.2 Scoreboarding

To achieve a balanced forest with independent trees, we design an efficient Scoreboarding algorithm optimized for hardware implementation, as depicted in Fig. 5 ❷-❺.

First, we record (Step ❷) all present TransRow and their counts in the Hasse graph. Subsequently, we perform **forward** and **backward** passes to build and prune the Hasse graph for transitive sparsity, maintaining prefix information with the smallest distance for each node.

Forward Pass for Prefix. As shown in Alg. 1 and Step ❸, we hierarchically update each node's prefix information without interdependencies. Therefore, we need to traverse all nodes by the Hamming order (Line 3) in Step ❸. Initially, we initialize the nodes (Lines 1-2) and traverse all nodes in a forward partial ordering (Lines 3-4). For transitive sparsity, we select node prefixes with a distance of less than 4 (Line 7). In practice, even in an 8-bit TransRow, nodes with a distance of 3 are rarely observed (< 0.1% within 128 TransRows). In Line 8, we set a temporary distance of 0 for nodes with a valid distance and $Count > 0$. This indicates that the node will be executed and has a valid prefix, as at least one TransRow has been processed. Consequently, the node can serve as a prefix, resetting the distance for subsequent suffixes. Following this, we generate locally the suffix based on the Hasse graph and send the prefix information to all its suffixes with an incremented (+1) distance (Lines 9-10), embodying the transitivity of TranSparsity. Lines 11-13 set the distance and prefix bitmaps.

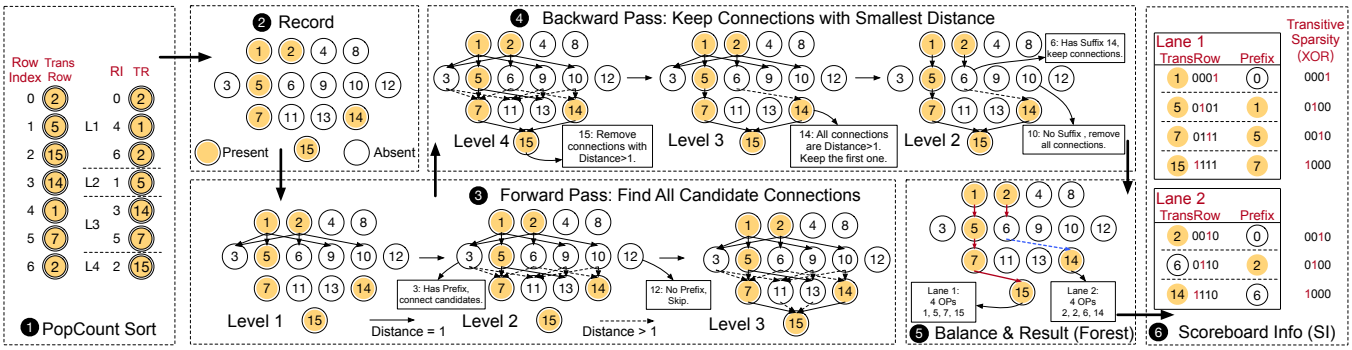


Figure 5: Scoreboarding process. Step ❶: Sort TransRows by Hamming order. Step ❷: Record present TransRows in the Hasse graph. Step ❸ (Forward pass): Assign candidate prefixes to all nodes. Step ❹ (Backward pass): Propagate suffix requests for nodes without a Distance = 1 prefix, searching for the prefix with the shortest distance and retaining the closest prefix for all nodes. Step ❺: Generate a balanced forest. Step ❻: Output the final Scoreboard information (SI).

Algorithm 1: 4-bit Scoreboarding Forward Pass.

Input: Node list, $N[16]$ { int Count; int Distance; Bitmap Prefix[4]; Bitmap Suffix; }

```
1  for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] do
2      N[i].Distance = +∞
3  for i in [0, 1, 2, 4, 8, 3, 5, 6, 9, 10, 12, 7, 11, 13, 14] do
4      Forward(i)
5  def Forward(int idx):
6      int dis = N[idx].Distance
7      if (dis >= 4 and idx != 0) then return
8      if ((N[idx].Count > 0) or idx == 0) then dis = 0
9      while suffix ∈ GetSuffixFromHasse(idx) do
10         SetPrefix(suffix, dis + 1, idx)
11 def SetPrefix(int idx, int Distance, int prefix):
12     N[idx].Prefix[Distance - 1].insert(prefix)
13     N[idx].Distance = min(N[idx].Distance, Distance)
```

Algorithm 2: 4-bit Scoreboarding Backward Pass.

Input: Node list, $N[16]$ { int Count; int Distance; Bitmap Prefix[4]; Bitmap Suffix; }

```
1  for i in [15, 7, 11, 13, 14, 3, 5, 6, 9, 10, 12, 1, 2, 4, 8] do
2      Backward(i)
3  def Backward(int idx):
4      int dis = N[idx].Distance
5      if 1 < dis < 4 and N[idx].Count > 0 then
6          prefix = GetPrefixFromHasse(N[idx].Prefix[dis - 1])
7          SetSuffix(prefix[0], idx) // Only the first prefix.
8  def SetSuffix(int idx, int suffix):
9      N[idx].Suffix.insert(suffix)
10     N[idx].Count = 1
11 Keep PB with the smallest distance; remove others.
```

For example, as depicted in Fig. 5 ❸, Nodes 1 and 2 will have their temporary distances set to 0 due to their *Count* > 0. Subsequently, Node 5 receives prefix information from Node 1 with a distance of 1, recording them into prefix bitmap. Similarly, all nodes update their prefix information with corresponding distances. A special case is Node 14, which has a distance of 2 from Node 2. Due to its *Count* > 0, Node 14 still sends a distance of 1 to Node 15. This mechanism retains more available prefix information, forming a more effective graph.

Backward Pass for Suffix. As shown in Alg. 2 and Step ❹, the backward pass is a reverse process of the forward pass to retrieve suffix information. The algorithm primarily constructs paths for nodes with *Count* > 0 and distance > 1 to their valid prefix nodes (Lines 5-7). Notably, we select only one prefix (defaulting to the first available) from the first valid Prefix Bitmap (PB) field to build the path (Line 7). Selecting multiple prefixes could result in redundant paths for a node. For instance, in Fig. 5 ❹, Node 14 has two prefix nodes, 6 and 10, each with a distance of 2. If both prefixes are selected, two paths would be formed: $2 \rightarrow 6 \rightarrow 14$ and $2 \rightarrow 10 \rightarrow 14$. This redundancy leads to unnecessary computations. Therefore, we retain only one path by keeping the first prefix.

Subsequently, we set the suffix information and update the prefix node's *Count* to 1 (Lines 8-10). For example, Node 6's *Count* is set to 1, designating Node 14 as its suffix. If Node 6 has a distance > 1 and < 4, the backward pass would continue tracing the path to the first node with distance 1, forming a complete path to Node 14 within a distance of 4. After that, we will only keep the prefix bitmap with the smallest distance (Line 11). This mechanism ensures each node maintains a single prefix, facilitating efficient parallelism and preserving correct partial ordering within the Hasse graph.

Balanced Forest. As shown in Fig. 5 ❺, after the forward and backward passes, we maintain a concise Hasse graph with valid prefixes having the smallest distances, thereby constructing the forest. We implement a workload counter and priority supervision to assign each node a Lane ID, ensuring balanced distribution across the trees. For example, Node 15 has two prefixes, Node 7 and Node 14. Since Node 2 corresponds to two TransRows, Node 15 is assigned to Lane 1 to achieve load balancing.

Scoreboard Information (SI). Using this approach, we successfully construct a balanced forest through the Scoreboard mechanism. In practice, the Hasse graph is transformed into a simplified table, referred to as Scoreboard Information (SI), which contains two key elements: each TransRow and its corresponding Prefix, as illustrated in Fig. 5 ❻. The overhead of SI is minimal. The total

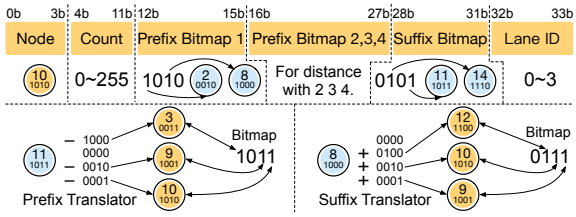


Figure 6: Bit field diagram of 4-bit Scoreboard and prefix and suffix translators.

memory requirement for SI is: $2 \times T \times 2^T$ bits, where T is the TransRow width. When $T = 8$, the SI needs only 512 Bytes of memory, which is insignificant and stored in the on-chip buffer.

3.3 Static Scoreboard

The execution order for all tensors in a DNN model, including weight and activation tensors, can be precomputed. For activation tensors, a small calibration dataset is used to generate the activation tensors. We first transform the quantized tensor into a binary matrix, which is then divided into T-bit TransRows. All TransRows within the tensor are recorded, and the final static Scoreboard Information (SI) is computed offline using our proposed algorithm in Fig. 5.

However, static SI may introduce efficiency challenges. GEMM relies on tiling (Sec. 4.1), and for a given tile, the prefix of a TransRow in the static SI may not exist in that tile, leading to an **SI Miss**, similar to a cache miss. An SI miss disrupts the prefix-suffix path, significantly impacting performance. We conduct experiments to evaluate this impact in Sec. 5.8. To ensure that the Scoreboard remains transparent to users and minimizes SI misses, we propose an efficient hardware-supported **dynamic Scoreboard** for enhanced performance.

3.4 Dynamic Scoreboard

Fig. 6 illustrates the bit fields of a single entry in the 4-bit dynamic Scoreboard design.

Node and Count: Each entry contains a Node identifier (e.g., Node 10) and an 8-bit *Count* field, representing the number of TransRows sharing the same value as the node. The Count field facilitates load balancing across the trees.

Prefix Bitmaps (PBs): We allocate four Prefix Bitmap fields, each corresponding to prefix indices at distances of 1, 2, 3, and 4.

Suffix Bitmap (SB): The Suffix Bitmap records suffix nodes that are absent from the current set of TransRows.

Lane ID: Each entry includes a Lane ID field that records the lane identifier associated with the node. This facilitates the distribution of tasks across multiple parallel lanes.

Furthermore, the Scoreboard can be extended to accommodate an 8-bit Hasse graph by maintaining the same fields but increasing the bit-width accordingly to handle the larger number of nodes.

Thanks to the Hasse representation, each node is shown to have a limited set of prefixes and suffixes. Leveraging this property, we propose an efficient encoding and decoding mechanism to map bitmap representations into prefix and suffix indices. As shown in Fig. 6-bottom, the proposed method introduces a **Prefix Translator** and a **Suffix Translator** to compress and recover transitive sparsity information effectively. The Prefix Translator derives node indices by applying a 1-to-0 bit flip to the prefix bitmap, while the Suffix Translator reconstructs suffix indices through a 0-to-1 bit flip. This design eliminates the need to explicitly store prefix and suffix data, significantly reducing significant (T times for T-bit) memory overhead. Moreover, the method achieves fast and accurate decoding with minimal hardware complexity, making it highly suitable for high-performance systems.

Finally, we get only one prefix for each node. It is important to note that the **distance** in the algorithm is a conceptual construct. In the hardware implementation, a straightforward logic circuit utilizing bitmaps can accurately determine the distance without introducing conflicts or dependencies, thanks to the efficient data structure design. Consequently, we are able to fully exploit the inherent parallelism of the Hasse graph by processing each level concurrently. This is facilitated by a multi-way Scoreboard design, which achieves high efficiency during on-the-fly operations.

4 Transitive Array

Fig. 7 and Fig. 8 shows Transitive Array design and an example of processing TransArray computation.

4.1 Tiling

Tiling is a fundamental technique for partitioning GEMM operations into smaller tiles, optimizing execution for DNN accelerators. In Fig. 8 ❶, we illustrate an example featuring an S-bit weight matrix of shape $(N \times K)$, an 8-bit input matrix of shape $(K \times M)$, and a 32-bit partial sum output matrix of shape $(N \times M)$. Initially, tensors are stored in off-chip DRAM. For each TransArray, only a small tile is processed in the on-chip buffer, consisting of a weight tile $(n \times k)$, an input tile $(k \times m)$, and an output tile $(n \times m)$. Using the bit-slice method, the weight tensor is decomposed into a binary weight tile of shape $(S \cdot n \times k)$. The pre-quantization and bit-slicing processes are performed offline, while all subsequent steps execute at runtime. During computation, the TransArray unit processes a sub-tile consisting of a weight tile $(S \cdot n \times T)$ and an input tile $(T \times m)$, where T represents the TransRow width.

4.2 Scoreboarding

In Fig. 8 ❶ bottom, TransArray can select only one type of Scoreboard Information (SI) between static and dynamic. The **static SI** is stored in DRAM and pre-fetched by TransArray before GEMM computation, incurring no runtime overhead. Thus, **all sub-tiles in the tensor share the same static SI.** In contrast, the **dynamic SI** is generated at runtime using the hardware Scoreboard when the weight sub-tile $(S \cdot n \times T)$ is sent to the on-chip network. Unlike the static SI, **each sub-tile has its own private dynamic SI**, enabling optimal prefix for each TransRow. As illustrated in In Fig. 7 (a), the sub-tile and the Scoreboard unit can be shared by multiple TransArray units.

4.3 Dispatch

Next, the TransArray dispatches TransRows along with their corresponding Scoreboard Information (SI) in Fig. 7 (b) Dispatcher and Fig. 8 ❷ and ❸. During this dispatch process, TranSparsity is computed using a simple XOR gate. For example, when the TransArray receives **TransRow 7** (`0111`), it first retrieves its **Prefix 5** (`0101`)

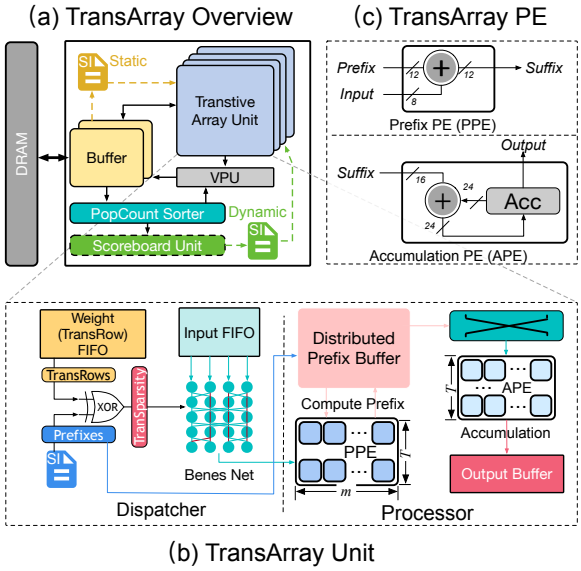


Figure 7: (a) Overview of Transitive Array architecture. (b) One Transitive Array unit. (c) Prefix PE (PPE) and Accumulation PE (APE) design.

from the SI. Then TransArray uses XOR ($\oplus$) to prune TransRow into **TranSparsity**: $7 \oplus 5 = 2$ (0010). The TranSparsity value (0010) corresponds to the element in the input matrix, i.e., -2 in Fig. 8 ❹. Additionally, the Prefix (0101) is sent to the Prefix Buffer, allowing retrieval of the precomputed prefix result, -1 for Prefix 5 (0101). Thus, instead of each TransRow accumulating all 4 input elements as in the original dense GEMM, it now requires only a single accumulation, effectively saving $T\times$ operations for T-bit TranSparsity.

This reduction is the primary source of sparsity in our design. As a result, our architecture achieves up to **75%** sparsity for 4-bit TranSparsity and **87.5%** sparsity for 8-bit configurations. Furthermore, after the first dispatch, each node in the Scoreboard replaces its Prefix with its own index. Subsequent identical TransRows can reuse the result from the first computed TransRow. This significantly improves efficiency.

4.4 Distribution Network

In Fig. 7 (b) Dispatcher and Fig. 8 Step ❸, the TransArray needs to fetch the input data and prefix data. We adopt the Benes network [4] to support input data access, which is an established non-blocking network widely adopted in studies [46, 55]. The Benes network has only $2\log(N)+1$ levels for an N-input and N-output configuration, resulting in very high hardware efficiency. For the prefix buffer, we employ a distributed buffer design where each prefix buffer operates independently. This approach significantly reduces the overhead associated with prefix buffers by eliminating the need to store all prefix nodes and avoiding complex network designs.

Additionally, we incorporate a crossbar between Step ❸ and Step ❹ for data arrangement in the prefix buffer to avoid bank conflicts originating from the row index. For each cycle, only T vectors are sent to the double buffer in T-bit TranSparsity. If the T vectors correspond to the same memory bank based on their row indices, conflicts may arise when they are stored in the same buffer bank. To mitigate this, we introduce a queue within the

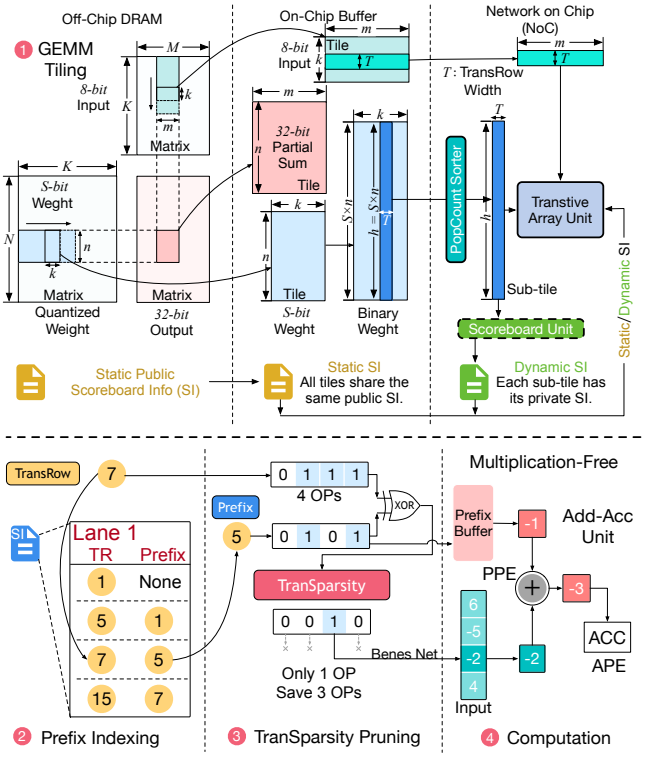


Figure 8: Processing flow of the Transitive Array unit. ❶: GEMM tiling layout and Scoreboard Information (SI). ❷: Retrieving the prefix for each TransRow using SI. ❸: Pruning for TranSparsity. ❹: Computation.

crossbar to buffer the partial sums and arrange them appropriately. We implement a double buffer mechanism so that the partial sum buffer overlaps and conceals the overhead associated with buffering, thereby enhancing overall system efficiency. We also put a shifter in place to shift their value to accommodate their bit-level according to their indices.

4.5 Processing Element

In Steps ❹, we utilize adders to complete all computations for the TransArray. For the TransArray, we design two types of Processing Elements (PEs).

As shown in Fig. 7 (c), the first PE is the **Prefix Processing Element (PPE)**, which is responsible for transitively computing the prefix results. According to our analysis in Sec. 2.1, maintaining sufficiently high precision in the adder allows us to perform lossless computations based on integers. Therefore, we employ a 12-bit adder to satisfy the precision requirements for PPE. The PPE design is highly concise, featuring only one 12-bit adder that adds the input and prefix sum, stores the result in the prefix buffer. The TPE array is formed by T lanes, each with $m = 32$ adders for T-bit width TranSparsity.

The second PE is the **Accumulation Processing Element (APE)**, depicted in Fig. 7 ❾, which is responsible for accumulating the final results. The APE only comprises a 24-bit accumulator. It accesses the partial sums from the prefix buffer and efficiently

combines them to produce the final output. The APE array has the same shape as the PPE, i.e., $T \times 32$ for T-bit TranSparsity.

As shown in Fig. 7 (a), we incorporate vector units (VPU) to support operations beyond GEMM (e.g., de-quantization, softmax, etc.), similar to previous studies [21, 23, 36]. According to the latest study [56], we utilize group-wise quantization to further improve model accuracy. When the group size is 128, the vector unit applies an integer scale factor to re-scale the partial results for each $128/T$ tile with T-bit TranSparsity. Therefore, we can efficiently overlap the overhead.

TransArray inherently supports the mixed-precision design. First, for the weight tensor, the bit-slicing method decomposes arbitrary bit-width weight tensors into binary matrices, making mixed-precision design inherently compatible with the TransArray design. For the activation tensor, since our processing elements (PEs) only use adders, it is straightforward to configure them for different bit widths. To support 8-bit activation, we employ a 12-bit Partial Product Engine (PPE) and a 24-bit Accumulation Processing Engine (APE). Additionally, they can be easily split into two 6-bit PPEs and two 12-bit APEs to support 4-bit activation.

4.6 Scheduling

Due to the reordering of execution, we implement multiple double-buffering mechanisms to divide the computation into three stages: (1) Dynamic Scoreboarding, (2) TPE array for prefix, and (3) APE array for output.

First, Stages 2 and 3 utilize arrays of identical shapes to compute the prefix and the final output, respectively. Specifically, TransRows with a prefix of distance 1 can complete the PPE and APE computations within a single cycle. However, TransRows with distances > 1 require additional cycles for the PPE to process. Consequently, the PPE array will consistently require more cycles than the APE. For the PPE, if we have n TransRows, the number of cycles required will exceed n cycles. However, APE will have constantly n cycles.

For Stage 1, due to the Hamming-weight sort, the dynamic Scoreboard can process a maximum number of nodes given by: $\min(n, 2^T)$ where n is the number of TransRows, and T is the bit-width of TranSparsity. To perform the sort, we implement a bitonic sorter [3] with $O(\log^2 n)$ time complexity, resulting in significantly fewer cycles. Futhermore, we configure an T-way Scoreboard to provide parallel Scoreboarding for T-bit TranSparsity, ensuring that: $\frac{\min(n,2^T)}{T} < \frac{n}{T}$, , which means Scoreboarding time is always less than that of PPE and APE.

Finally, this three-stage pipeline ensures that the critical path of TransArray is still on the PPE array. Fortunately, according to our analysis, only approximately 1.67% of TransRows in our design have distances greater than 1. Thus, both the PPE and APE arrays achieve nearly full utilization.

5 Evaluation

5.1 Methodology

Accelerator Baselines. We compare our Transitive Array with five baselines, including Bitfusion [48], ANT [23], Olive [21], Tender [36] and BitVert [8]. BitFusion [48] introduces bit-level dynamic composability for DNN acceleration. ANT [23] proposes a hardware-friendly framework for a fixed-length adaptive numerical data type. Olive [21] quantizes models by utilizing the space of normal values to accommodate outliers. Tender [36] decomposes activation tensors along feature dimensions into sub-tensors, with scale factors set to powers of two. BitVert [8] introduces bi-directional bit-level sparsity, ensuring at least 50% sparsity, and provides a bit-level binary pruning technique.

Accelerator Implementation. We build a cycle-level simulator to analyze the performance of the Transitive Array. For other architectures, we use their open-sourced simulator, ANT [23], which all build on. The hardware architecture of both the Transitive Array and the baselines is implemented using System Verilog. We synthesize the RTL logic with Synopsys Design Compiler, utilizing ARM's standard cell library in a commercial 28nm process to determine the corresponding area and static/dynamic power consumption. We use Cacti 7.0 to assess buffer area and power at 28nm. We maintained all baselines and the Transitive Array at the same 28nm process and a frequency of 500MHz. We rewrite all baseline PE implementations according to their design to ensure a fair comparison. We evaluate most experiments with dynamic Scoreboard, except the Sec. 5.8, in which we compare and explore the static and dynamic Scoreboard using random and real data.

Benchmark. We run LLaMA versions 1, 2, and 3 [15, 51, 52] on the Wikitext dataset [43], using the perplexity (PPL) [5] of the generated sentences as the performance metric. Since TransArray's results depend on the data distribution of weights and activations, we extract all the model's weights and activations and evaluate them with our simulator. However, because TransArray requires both activations and weights, resulting in an unacceptable memory footprint trace, we only extract the first Transformer block with a prefill sequence length of 2048. This approach is feasible because all Transformer blocks are identical and exhibit similar computational behavior.

5.2 Design Space Exploration

The TranSparsity is a generalized design for arbitrary bit-widths. Thus, we first explore the design space to determine our final hardware implementation. We employ the tiling approach to divide the large matrix into small tiles. Specifically, the performance of the TransArray is significantly impacted by the tile size, i.e., the row number (N) and the bit width (T) of TransRows for a bit matrix. All experiments on this sub-section are based on dynamic Scoreboard and random data.

Bit Width (T). As illustrated in Fig. 9 (a), we explore the influence of tiling size on N and T. We find that T determines the upper bound and lower bound of sparsity and density for TranSparsity. Even if we fully utilize the sparsity of TranSparsity, we can only achieve a lower-bound density around $\frac{1}{T}$ because we must perform at least one accumulation operation for every T-bit element. For example, 8-bit TranSparsity requires one accumulation every 8 bits. For 8-bit TranSparsity, we achieve **Pareto Optimality** with reasonable overhead. Beyond 8-bit, the density increases before reaching a row size of 256. Notably, 10-bit TranSparsity requires 4× the hardware overhead of 8-bit TranSparsity but offers comparable sparsity at a tile row size of 256. Therefore, we choose **8-bit TranSparsity** to implement in this study.

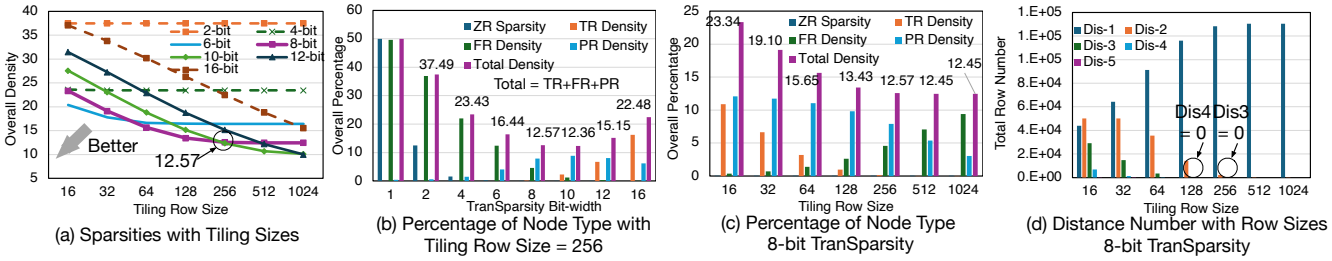


Figure 9: Design space exploration among various bit-width and tiling row sizes: (a) and (b), and 8-bit TranSparsity with tiling row sizes: (c) and (d) on a 1024×1024 random 0-1 matrix.

As mentioned earlier, not all TransRows will utilize both PPE and APE. The utilization of PPE and APE significantly impacts the overall design efficiency. We summarize the features of TransRows based on the utilization of PPE and APE for partial sums (PSum) and final sums (FSum). This classification helps us understand the efficiency of TranSparsity in depth. We identify four computation patterns based on the combination of PSum and FSum. **Zero Row (ZR):** Represents a node with no partial or final sums, allowing us to skip computations entirely. **Transitive Reuse (TR):** Represents a node responsible for transitioning partial results to its suffix node, utilizing only the PPE without APE. **Full Result Reuse (FR):** Represents nodes that have been computed previously, allowing subsequent TransRows to reuse the results without requiring prefix computations or PPE. **Prefix Result Reuse (PR):** Represents nodes that require both PPE and APE to add the prefix result and perform accumulation.

As illustrated in Fig. 9 (b) with a tiling row size of 256, before 8-bit TranSparsity, the FR (Full Result Reuse) constitutes the majority of TransRows, leading to underutilization of the TransArray and higher density. After 8-bit, higher bit widths such as 10, 12, and 16 introduce more TR (Transitive) nodes. This results in an extremely sparse Hasse graph consisting of only 256 rows. The 8-bit TranSparsity strikes a balanced trade-off between sparsity and hardware overhead.

Row Number (N). As shown in Fig. 9 (c), we conducted experiments varying the number of rows from 16 to 1024 using 8-bit TranSparsity. We observed that when the number of rows exceeds 256, the density of 8-bit TranSparsity stabilizes. This stability occurs because most TransRows are captured by the Hasse graph, resulting in only a few Transitive Result (TR) nodes. Consequently, increasing the row number beyond 256 does not yield additional sparsity benefits. Even if all nodes can reuse results, every 8 bits still requires one addition operation. Therefore, we set the **maximum row number to 256** for 8-bit TranSparsity. With this configuration, each TransArray unit can support 32×8 tile for 8-bit weights and 64×8 tile for 4-bit weights.

Additionally, we provide detailed statistics on the distances of the nodes. As depicted in Fig. 9 (c), when the row number is 128, there are no nodes with a Distance of 4, and similarly, for a row number of 256, there are no nodes with a Distance of 3. In practice, this means we only need to provide three prefix bitmap fields for nodes with a Distance of 3. TransRows with a Distance ≥ 4 are treated as outliers and dispatched at the end of other operations.

Table 1: Specifications of One TransArray Unit.

Bit-width	T = 8-bit TranSpasrity.
TransRow Number	Max 256 1-bit TransRow.
Weight Tiling	N = 32 for 8-bit Wgt; N = 64 for 4-bit Wgt.
Input Tiling	M = 32 for 8-bit Input.
PPE Array	8×32 12-bit Adder.
APE Array	8×32 24-bit Adder.
NoC	An 8-way Benes net and X bar.
Scoreboard	Tow 8-way 256-entry tables; A sorter.
Buffer Size: 80KB	8KB Weight; 8KB Input; 22KB Output; 18KB Prefix; 24KB Double Buffer.

5.3 Area Comparison

Table 1 and Table 2 present the design configurations and area analysis of our TransArray architecture compared to the baseline designs. We implement **six TransArray units** in the final GEMM accelerator design. TransArray yields a lower computational core area overhead compared to all five baselines. This low area overhead mainly comes from our simplified PPE and APE design, which only requires accumulation operations, while other baselines employ multi-bit multiplication units, incurring quadratic hardware complexity. Therefore, we are able to place a NoC within our array to flexibly support buffer access while maintaining a lower area overhead. Note that we also consider the Scoreboard to ensure a dynamic Scoreboard. To ensure a fair comparison, we configure a smaller buffer size (480KB) for our design compared to other baselines (512 KB and 608 KB).

Table 2: The area of core components and buffers for TransArray and other baseline models using a 28nm process.

Arch.	Computation Core			Buffer
	Component	Array	Area (mm^2)	
TransArray (6 Units)	PPE (50.3μm^2)	$6 \times (8 \times 32)$	**0.443**	480KB
	APE (101.7μm^2)	$6 \times (8 \times 32)$		
	NoC (19520μm^2)	$6 \times (1)$		
	Socreboard (92507μm^2)	1		
BitFusion	8-bit PE (548μm^2)	28×32	0.491	512KB
ANT	4-bit PE (210μm^2)	36×64	0.484	
Olive	4-bit PE (319μm^2)	32×48	0.489	
BitVert	8-bit PE (985μm^2)	16×30	0.473	
Tender	4-bit PE (329μm^2)	30×48	0.474	608KB

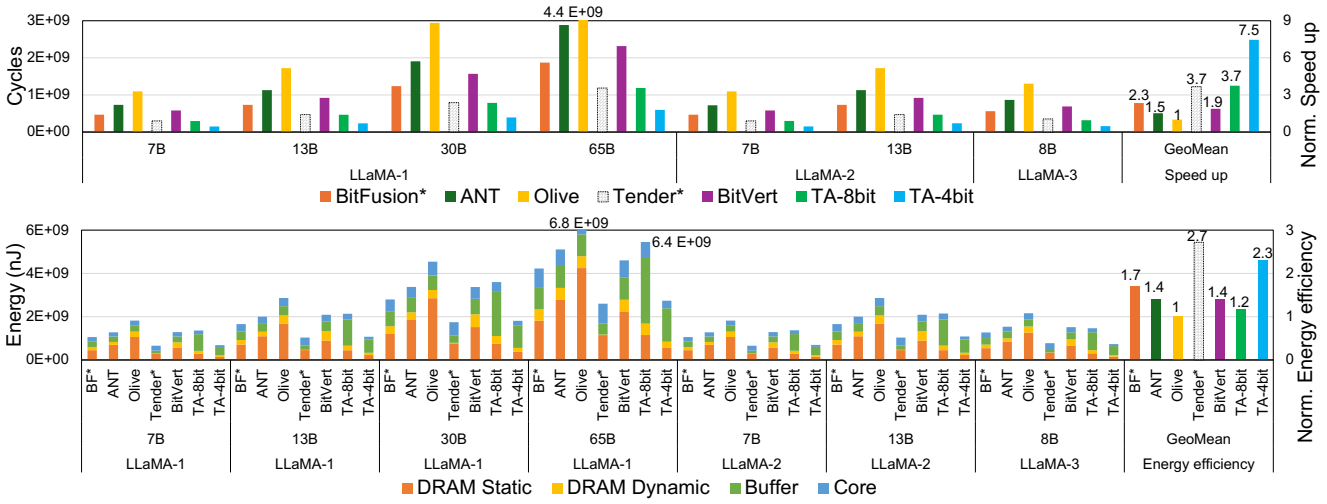


Figure 10: Runtime and energy consumption on FC layers of LLaMA models. *BitFusion (8-bit) and Tender (4-bit) exhibit unacceptable perplexity (PPL) scores. ANT, Olive, BitVert, and TransArray (4-bit) achieve similar PPL levels on LLaMA.

Table 3: The Perplexity on Wikitext data with Tender (TD), BitFusion (BF), Olive (OL), BitVer (BV), ANT with group quantization, and TransArray (TA) with group quantization.

Arch	TD-4	BF	OL	TD-8	BV	ANT	TA	TA	FP16
FC Act.	4bit	8bit	8bit	8bit	8bit	8bit	**Int8**	Int8	FP16
FC Wgt.	4bit	8bit	8bit	8bit	8bit	8bit	**Int4**	Int8	FP16
L-1 7B	23.85	9.50	5.86	5.87	—	**5.82**	**5.82**	5.75	5.68
L-1 13B	13.68	8.46	5.28	5.28	—	**5.20**	**5.20**	5.14	5.09
L-1 30B	12.07	6.70	4.37	4.27	—	4.32	**4.24**	4.17	4.10
L-1 65B	8.85	5.34	3.80	3.74	—	3.76	**3.66**	3.57	3.53
L-2 7B	36.47	10.68	5.73	5.77	—	**5.58**	5.62	5.56	5.47
L-2 13B	55.08	16.11	5.06	5.09	—	5.20	**5.01**	4.95	4.88
L-3 8B	28.60	22.56	6.70	7.17	**6.24**	6.27	6.59	6.39	6.14

5.4 Model Accuracy

We benchmarked several post-training quantization (PTQ) methods on the Wikitext dataset using LLaMa models, as shown in Tbl. 3. Perplexity (PPL) served as the performance metric—lower values indicate better performance. We reproduced their results using open-sourced code, except for BitVert [9], for which we reported only the available results from its paper. Due to the lack of optimization for quantization, BitFusion exhibits a larger gap compared to the FP16 results. Other accelerators achieve near-lossless performance at the 8-bit level due to their quantization-aware optimized architecture designs. We modified ANT to support group-wise quantization for a fair comparison. Please note that Tender [36] only supports 4-bit PEs without mixed precision. However, its 4-bit PPL is unacceptable. Therefore, in the following evaluation, only the results of BitFusion and Tender are provided for reference. Additionally, please note that these accelerators (except BitFusion) cannot support the Attention layer due to their complicated weight pre-processing. We maintain the Attention layer with FP16 precision.

Due to the generalized design of *TA*, it can broadly support state-of-the-art (SOTA) quantization frameworks without specific requirements. We implement TransArray in Qserve [39], the SOTA quantization framework from prior work. In contrast, other frameworks present a high barrier to algorithmic optimization due to their specialized designs for specific data types. For instance, SOTA quantization methods, such as SmoothQuant [61], suppress outliers in weight tensors for integer optimization. This contrasts with Olive, which benefits from large outliers.

Our proposed *TA* architecture consistently achieves competitive PPL scores across all model sizes, utilizing Int4 weights and Int8 inputs, and without Attention quantization for a fair comparison. These significant improvements stem from the generalized integer-based design without special requirements.

5.5 Performance and Energy on FC Layers

Settings. Fig. 10 shows the overall performance and energy consumption across various accelerators **on only FC layers of the LLaMA models**. The results for BitFusion with 8-bit PE and Tender with 4-bit PE are provided only for reference due to their large PPL loss. The remaining accelerators maintain similar accuracy levels for LLaMA. ANT and Olive utilize mixed-precision designs with 8-bit evaluations. BitVert employs bit-slice techniques with 50% sparsity. We provide two types of precision for TransArray: 4-bit weights and 8-bit weights on real data.

Iso-Precision Comparison. We first compare TransArray with 8-bit weights against other baselines. Due to the greater difficulties in quantizing LLM, the mixed-precision advantages of ANT and Olive disappear. They are even slower than BitFusion because of the larger overhead from their complex PEs, which support outliers and adaptive data types. Although BitVert has a more complicated PE design, it utilizes bit sparsity with a fixed 50% sparsity based on bit-slice techniques and achieves a 1.9× speedup over Olive on LLMs, consistent with the results reported in its paper. Despite TransArray needing to slice 8-bit into 8 accumulation operations, it can fully exploit 87.5% sparsity with TranSparsity. This translates to 8× and 4× speedups over dense GEMM and bit sparsity-based architectures, respectively. Additionally, with an extremely streamlined

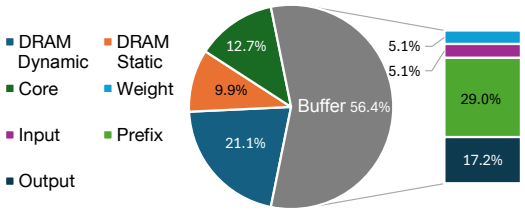


Figure 11: TransArray energy breakdown on LLaMA-1-7B.

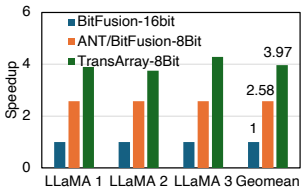


Figure 12: Speedups on Attention layers of LLaMA models over BitFusion.

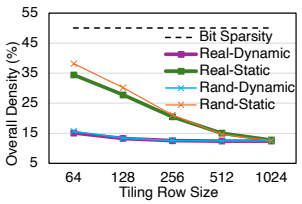


Figure 13: Static and dynamic Scoreboard comparison on real and random (Rand) data.

PE design, TransArray with 8-bit weights achieves 2.47×, 3.75×, and 1.99× speedups over ANT, Olive, and BitVert, respectively, while maintaining similar energy consumption.

Iso-Accuracy Comparison. Due to the generality of TransArray, it can employ state-of-the-art (SOTA) algorithms to further enhance its performance. For 4-bit quantization, TransArray can theoretically provide 16× and 8× improvements over 8-bit quantization-based and bit-sparsity-based architectures, respectively. TransArray with 4-bit weights achieves 4.91×, 7.46×, and 3.97× speedups and 1.65×, 2.31×, and 1.65× energy efficiency improvements over ANT, Olive, and BitVert, respectively. These significant improvements are also attributed to the generalized design of TransArray. Other architectures find it difficult to benefit from algorithmic advancements.

5.6 Energy Breakdown Analysis

Fig. 11 presents the energy breakdown of TransArray for the first fully connected (FC) layer of the LLaMA-1-7B model. We observe that buffer operations consume the majority of the energy, primarily due to frequent accesses to the prefix buffer, which is essential for enabling efficient TranSparsity. Although our design incurs significant on-chip buffer access overhead, the high efficiency of TranSparsity significantly reduces overall execution time. As a result, TransArray achieves lower DRAM static energy consumption compared to other baselines. Consequently, despite increased buffer access, TransArray maintains high energy efficiency relative to these baselines.

Essentially, the TransArray design enhances computational efficiency at the expense of increased buffer energy consumption. Furthermore, TransArray resembles SIMD [16] or in-memory processing [11, 28, 31] architectures. We may implement TransArray using GPU tensor cores [44] to explore broader scenarios and leverage advanced technologies for more energy-efficient solutions.

5.7 Performance on Attention Layers

Fig. 12 presents the performance results of the Attention layers in LLaMA-1-7B, LLaMA-2-2B, and LLaMA-3-8B. All data are collected from real model inference with a sequence length of 2048. For Attention layers, we treat the K and V cache as weight tensors. Since Attention is challenging to quantize, we adopt 8-bit group-wise quantization for both ANT and TransArray, while using 16-bit quantization for BitFusion. TransArray achieves a 1.54× speedup over ANT under the same quantization settings and a 3.97× speedup over BitFusion.

Furthermore, prior designs [8, 21, 36]—such as Olive, Tender, and BitVert—do not support Attention layers but only focus on fully connected (FC) layers. These approaches rely on specific offline quantization methods, making them incompatible with dynamic Attention processing. For instance, BitVert depends on complex bit-level binary pruning and channel reordering techniques, which do not support online processing. In contrast, TransArray with the dynamic Scoreboard is a general method that can be applied to various integer quantization techniques. With the help of the Scoreboard unit, TransArray seamlessly processes all GEMM operations in DNNs while remaining transparent to users.

5.8 Static and Dynamic Scoreboard Comparison

In this subsection, we compare the static and dynamic Scoreboards on the first FC layer of LLaMA-1-7B model using 8-bit TranSparsity, as illustrated in Fig. 13. Clearly, the dynamic Scoreboard achieves significantly lower density (higher speedup) than the static Scoreboard for smaller tiling row sizes (< 512), while achieving comparable results for larger sizes (> 512).

A static Scoreboard Information (SI) is shared across an entire tensor. When the tile row size is small (e.g., 64), the Hasse graph becomes significantly sparse under 8-bit TranSparsity, containing 256 nodes, of which fewer than 25% are present. This means that different tiles are likely to contain distinct nodes, generating entirely different forests (multiple trees) for SI. As a result, static SI leads to frequent **SI Misses**, causing substantial performance degradation. Despite this limitation, the static Scoreboard remains significantly more efficient than bit sparsity. However, as the row size increases beyond 256, the static Scoreboard achieves comparable performance to the dynamic Scoreboard, reaching equivalent performance at a row size of 1024. Additionally, since the static Scoreboard does not require a dedicated hardware Scoreboard unit, it reduces area overhead by approximately 25%, leading to potentially better overall performance than the dynamic Scoreboard in some cases.

In contrast, the dynamic Scoreboard generates SI for each subtile, allowing it to achieve near-optimal sparsity and demonstrating strong applicability. More importantly, the dynamic Scoreboard is transparent and seamlessly compatible with existing frameworks.

5.9 Random and Real Data Comparison

Fig. 13 also compares the impact of data distribution on performance. We collect real data from the LLaMA-1-7B model and generate 0-1 uniform random data for both static and dynamic Scoreboards. Interestingly, TransArray achieves slightly better performance on real data compared to random data. After a thorough examination and detailed analysis, we identify two key reasons for this performance improvement: (1) Although most original tensors of real data follow a Gaussian distribution, the binary 0-1 distribution after bit-slicing

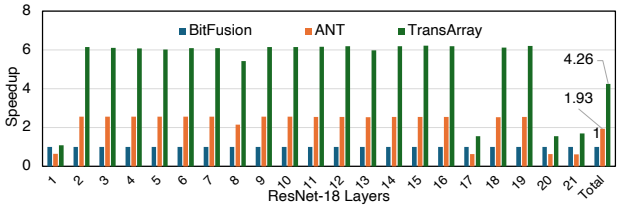


Figure 14: Speedup comparison on ResNet18 models.

resembles a uniform distribution. (2) Certain patterns may exist in the weight tensor of DNN models. For uniform random data, the occurrence of repeated values follows the classical problem of discrete uniform distributions. The expected number of unique values in 256 random 8-bit TransRows is approximately 162. However, in real data, this number is slightly lower than 162, suggesting the presence of structural patterns in DNN weight tensors.

5.10 TransArray in DNN Models

We evaluate TransArray on the ResNet-18 [27] model using the ImageNet dataset [13]. Both BitFusion and ANT are optimized for CNN models with mixed precision. For TransArray, we adopt 4-bit quantization using MQBench [37]. TransArray supports mixed-precision computation for 4-bit and 8-bit activations, as discussed in Sec. 4.5. We configure the first convolution layer and the final fully connected (FC) layer with 8-bit quantization. Following prior work [23], we also employ `im2col` [10] to transform convolution layers into GEMM operations. Fig. 14 presents the speedup results of all ResNet-18 layers for BitFusion, ANT, and TransArray. TransArray demonstrates superior performance, achieving a 4.26× speedup over BitFusion and a 2.21× speedup over ANT.

6 Related Works

6.1 DNNs and Quantization

Deep Neural Networks (DNNs) and Large Language Models (LLMs) have achieved state-of-the-art performance across various domains. However, their high computational and memory demands pose challenges in resource-constrained environments. Quantization techniques [2, 18, 20, 38, 39, 41, 47, 61, 62] mitigate these issues by reducing the precision of weights and activations, enhancing computational and memory efficiency with minimal performance loss. Notable methods include SmoothQuant [61], which introduces per-channel scaling; AWQ [38], which refines this with activation-aware scaling. These techniques collectively address key quantization challenges in LLMs.

Quantization-based accelerators [21, 23, 29, 40, 48, 63] leverage reduced bit-width data representations to minimize memory bandwidth and enhance computational efficiency, making them suitable for resource-constrained environments. BitFusion [48] introduces a flexible PE array that supports various bit-widths dynamically. OLAccel [45] employs a heterogeneous quantization strategy with 16-bit MAC units for the first layer and 4-bit MAC units for subsequent layers, optimizing energy efficiency without significant accuracy loss. GOBO [63] focuses on outlier-aware quantization by quantizing outlier weights with higher precision. ANT [23] proposes a fixed-length adaptive quantization framework but overlooks outliers, limiting its effectiveness. OliVe [21] introduces an outlier-victim pair quantization technique that efficiently handles outliers with minimal hardware overhead, achieving negligible accuracy loss with 4-bit quantization. However, this datatype-based quantization accelerator struggles with LLM models, highlighting the need for improved quantization approaches.

6.2 Bit-slicing and Sparsity

Therefore, recent techniques [1, 42, 49] have focused on exploiting bit-level sparsity to enhance quantization efficiency. Bit-slice accelerators specifically optimize computations in sparse-bit scenarios by skipping ineffectual zero bits, thus significantly improving efficiency. Pragmatic [1] pioneered this approach by selectively processing non-zero bits using variable shifters to align bit significances, simplifying computations but introducing higher hardware overhead. However, such methods are inherently limited to exploiting intrinsic sparsity, typically constrained to approximately 50–60% due to inherent data characteristics [9].

Consequently, many accelerators have turned their attention to sparsity-aware processing [19, 22, 24, 25, 58–60, 64–66], though often encountering significant irregularities in memory access patterns. To address these irregularities, substantial efforts have been made to regularize sparsity patterns into hardware-friendly formats, such as sparse tensor cores [66], thereby achieving greater speedups on sparse DNN models. BitVert [9], a recent bit-slice and sparsity co-design accelerator, dynamically balances workload distribution by selectively skipping zero bits in sparse bit-columns, at the cost of more intricate PE architectures. Prosperity [59] can directly support the bit-level sparsity on spiking neural networks. In our study, we leverage insights from structured sparsity processing to regularize transitive sparsity, resulting in higher hardware utilization and efficiency gains in our proposed design.

7 Conclusion

In this study, we introduced a novel sparsity paradigm, **Transitive Sparsity**, which leverages the reuse of previously computed results to significantly reduce GEMM computations, thereby decreasing operational overhead. To effectively exploit this sparsity, we designed the **Transitive Array**, a multiplication-free accelerator that addresses challenges related to execution order and parallelism while also supporting on-the-fly quantization of Attention layers. Comprehensive evaluations demonstrated that the Transitive Array achieves significant speedup compared to state-of-the-art accelerators, maintaining comparable model accuracy on LLaMA models. These improvements are attributed to the generalized design of TransArray, which allows broad support for state-of-the-art quantization frameworks without specialized requirements, and its streamlined PE design that eliminates traditional multiplication operations. Our work provides a novel perspective on sparsity optimization in GEMM computation.

Acknowledgments

This work was supported in part by NSF-2112562 and ARO W911NF-23-2-0224. The authors sincerely thank the anonymous reviewers for their constructive feedback and valuable suggestions that greatly improved the quality of this work.

References

[1] Jorge Albericio, Alberto Delmas, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-Pragmatic Deep Neural Network Computing. *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017).

[2] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024. QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs. arXiv:2404.00456 [cs.LG] https://arxiv.org/abs/2404.00456

[3] Kenneth E Batcher. 1968. Sorting Networks and Their Applications. *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (1968), 307–314.

[4] Vaclav E. Benes. 1964. Mathematical Theory of Connecting Networks and Telephone Traffic. *Mathematics in Science and Engineering* 17 (1964).

[5] Peter F Brown, Stephen A Della Pietra, Vincent J Della Pietra, Jennifer C Lai, and Robert L Mercer. 1992. An estimate of an upper bound for the entropy of English. *Computational Linguistics* 18, 1 (1992), 31–40.

[6] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[8] Yuzong Chen, Jian Meng, Jae-sun Seo, and Mohamed S Abdelfattah. 2024. BBS: Bi-directional Bit-level Sparsity for Deep Learning Acceleration. *arXiv preprint arXiv:2409.05227* (2024).

[9] Yuzong Chen, Jian Meng, Jae sun Seo, and Mohamed S. Abdelfattah. 2024. BBS: Bi-directional Bit-level Sparsity for Deep Learning Acceleration. arXiv:2409.05227 [cs.LG] https://arxiv.org/abs/2409.05227

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759 [cs.NE] https://arxiv.org/abs/1410.0759

[11] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 27–39.

[12] Wikipedia contributors. [n. d.]. Hasse diagram — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Hasse_diagram

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[14] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzmán, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vítor Albiero, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[16] Michael J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.* C-21 (1972), 948–960. https://api.semanticscholar.org/CorpusID:18573685

[17] Ian Goodfellow. 2016. Deep learning.

[18] Cong Guo, Feng Cheng, Zhixu Du, James Kiessling, Jonathan Ku, Shiyu Li, Ziru Li, Mingyuan Ma, Tergel Molom-Ochir, Benjamin Morris, et al. 2025. A Survey: Collaborative Hardware and Software Design in the Era of Large Language Models. *IEEE Circuits and Systems Magazine* 25, 1 (2025), 35–57.

[19] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[20] Cong Guo, Yuxian Qiu, Jingwen Leng, Xiaotian Gao, Chen Zhang, Yunxin Liu, Fan Yang, Yuhao Zhu, and Minyi Guo. 2022. SQuant: On-the-Fly Data-Free Quantization via Diagonal Hessian Approximation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=JXhROKNZzOc

[21] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[22] Cong Guo, Fengchen Xue, Jingwen Leng, Yuxian Qiu, Yue Guan, Weihao Cui, Quan Chen, and Minyi Guo. 2024. Accelerating sparse dnns based on tiled gemm. *IEEE Trans. Comput.* (2024).

[23] Cong Guo, Chen Zhang, Jingwen Leng, Zihan Liu, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2022. Ant: Exploiting adaptive numerical data type for low-bit deep neural network quantization. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1414–1433.

[24] 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. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[25] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[26] Helmut Hasse. 1967. Grundlagen der Mathematik in historischer Entwicklung. *Mathematisch-Naturwissenschaftliche Bibliothek* 31 (1967).

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[28] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent offloading and mapping (TOM) enabling programmer-transparent near-data processing in GPU systems. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 204–216.

[29] Weiming Hu, Haoyan Zhang, Cong Guo, Yu Feng, Renyang Guan, Zhendong Hua, Zihan Liu, Yue Guan, Minyi Guo, and Jingwen Leng. 2025. M-ANT: Efficient Low-bit Group Quantization for LLMs via Mathematically Adaptive Numerical Type. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1112–1126.

[30] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. arXiv:1712.05877 [cs.LG] https://arxiv.org/abs/1712.05877

[31] Dong-Ik Jeon, Kyeong-Bin Park, and Ki-Seok Chung. 2017. HMC-MAC: Processing-in memory architecture for multiply-accumulate operations with hybrid memory cube. *IEEE Computer Architecture Letters* 17, 1 (2017), 5–8.

[32] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. arXiv:1704.04760 [cs.AR] https://arxiv.org/abs/1704.04760

[33] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG] https://arxiv.org/abs/2001.08361

[34] Donald E. Knuth. 2009. The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1. In *The Art of Computer Programming*. Addison-Wesley Professional.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[36] Jungi Lee, Wonbeom Lee, and Jaewoong Sim. 2024. Tender: Accelerating Large Language Models via Tensor Decomposition and Runtime Requantization. *arXiv preprint arXiv:2406.12930* (2024).

[37] Yuhang Li, Mingzhu Shen, Jian Ma, Yan Ren, Mingxin Zhao, Qi Zhang, Ruihao Gong, Fengwei Yu, and Junjie Yan. 2021. Mqbench: Towards reproducible and deployable model quantization benchmark. *arXiv preprint arXiv:2111.03759* (2021).

[38] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. arXiv:2306.00978 [cs.CL] https://arxiv.org/abs/2306.00978

[39] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving. arXiv:2405.04532 [cs.CL] https://arxiv.org/abs/2405.04532

[40] Zihan Liu, Xinhao Luo, Junxian Guo, Wentao Ni, Yangjie Zhou, Yue Guan, Cong Guo, Weihao Cui, Yu Feng, Minyi Guo, et al. 2025. VQ-LLM: High-performance Code Generation for Vector Quantization Augmented LLM Inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1496–1509.

[41] Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. 2024. SpinQuant: LLM quantization with learned rotations. arXiv:2405.16406 [cs.LG] https://arxiv.org/abs/2405.16406

[42] Hang Lu, Liang Chang, Chenglong Li, Zixuan Zhu, Shengjian Lu, Yanhuan Liu, and Mingzhe Zhang. 2021. Distilling Bit-level Sparsity Parallelism for General Purpose Deep Learning Acceleration. *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2021).

[43] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[44] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. Technical Report. NVIDIA Corporation. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf Accessed: 2024-11-23.

[45] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-efficient neural network 'accel'erator based on outlier-aware low-precision computation. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 688–698.

[46] 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)*. IEEE, 58–70.

[47] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. 2024. OmniQuant: Omnidirectionally Calibrated Quantization for Large Language Models. arXiv:2308.13137 [cs.LG] https://arxiv.org/abs/2308.13137

[48] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 764–775.

[49] Man Shi, Vikram Jain, Antony Joseph, Maurice Meijer, and Marian Verhelst. 2024. BitWave: Exploiting Column-Based Bit-Level Sparsity for Deep Learning Acceleration. *Proceedings of the 30th IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2024).

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[52] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[53] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[55] Abraham Waksman. 1968. A permutation network. In *8th Annual Symposium on Switching and Automata Theory*. IEEE, 111–117.

[56] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. 2024. Efficient Large Language Models: A Survey. arXiv:2312.03863 [cs.CL] https://arxiv.org/abs/2312.03863

[57] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. BitNet: Scaling 1-bit Transformers for Large Language Models. arXiv:2310.11453 [cs.CL] https://arxiv.org/abs/2310.11453

[58] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1083–1095.

[59] Chiyue Wei, Cong Guo, Feng Cheng, Shiyu Li, Hao Frank Yang, Hai Helen Li, and Yiran Chen. 2025. Prosperity: Accelerating Spiking Neural Networks via Product Sparsity. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 806–820.

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

[61] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2024. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. arXiv:2211.10438 [cs.CL] https://arxiv.org/abs/2211.10438

[62] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 27168–27183. https://proceedings.neurips.cc/paper_files/paper/2022/file/adf7fa39d65e2983d724ff7da57f00ac-Paper-Conference.pdf

[63] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 811–824.

[64] Chen Zhang, Yang Wang, Zhiqiang Xie, Cong Guo, Yunxin Liu, Jingwen Leng, Guangyu Sun, Zhigang Ji, Runsheng Wang, Yuan Xie, et al. 2024. DSTC: Dual-Side Sparsity Tensor Core for DNNs Acceleration on Modern GPU Architectures. *IEEE Trans. Comput.* (2024).

[65] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. [n. d.]. H$_2$O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. arXiv:2306.14048 [cs] http://arxiv.org/abs/2306.14048

[66] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 359–371.