Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow

Zhiyao Li
lizhiyao19@mails.tsinghua.edu.cn
Tsinghua University
China

Jiaxiang Li
jiaxiangli2021@u.northwestern.edu
Northwestern University
USA

Taijie Chen
ctj19@mails.tsinghua.edu.cn
Tsinghua University
China

Dimin Niu
dimin.niu@alibaba-inc.com
Alibaba DAMO Academy
China

Hongzhong Zheng
hongzhong.zheng@alibaba-inc.com
Alibaba DAMO Academy
China

Yuan Xie
y.xie@alibaba-inc.com
Alibaba DAMO Academy
China

Mingyu Gao
gaomy@tsinghua.edu.cn
Tsinghua University
China
Shanghai Qi Zhi Institute
China

ABSTRACT

Sparse matrix-matrix multiplication (SpGEMM) is widely used in many scientific and deep learning applications. The highly irregular structures of SpGEMM limit its performance and efficiency on conventional computation platforms, and thus motivate a large body of specialized hardware designs. Existing SpGEMM accelerators only support specific types of rigid execution dataflow such as inner/output-product or row-based schemes. Each dataflow is only optimized for certain sparse patterns and fails to generalize with robust performance to the widely diverse SpGEMM workloads across various domains. We propose Spada, a combination of three novel techniques for SpGEMM accelerators to efficiently adapt to various sparse patterns. First, we describe a window-based adaptive dataflow that can be flexibly adapted to different modes to best match the data distributions and realize different reuse benefits. Then, our hardware architecture efficiently supports this dataflow template, with flexible, fast, and low-cost reconfigurability and effective load balancing features. Finally, we use a profiling-guided approach to detect the sparse pattern and determine the optimized dataflow mode to use, based on the key observations of sparse pattern similarity in nearby matrix regions. Our evaluation results demonstrate that Spada is able to match or exceed the best among three state-of-the-art SpGEMM accelerators, and avoid the performance degradation of the others if data distribution and dataflow mismatch. It achieves an average 1.44× speedup across a wide range of sparse matrices and compressed neural network models.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575706

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Computing methodologies → Linear algebra algorithms**; • **Computer systems organization** → *Special purpose systems.*

KEYWORDS

sparse matrix multiplication, hardware acceleration, dataflow

ACM Reference Format:
Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575706

1 INTRODUCTION

Sparse matrix-matrix multiplication (SpGEMM) is a vital component in a wide range of important scientific computation fields, from graph analytics [16, 35], linear algebra [30, 39], economic modeling [8], molecule dynamics [10, 18], to machine learning [5, 13, 25]. Compared to the counterpart dense matrix computations, sparse matrices are able to effectively eliminate all ineffectual zero (and sometimes also near-zero) elements to save substantial storage and operation cost, especially when the matrix size is huge and the density is very low due to intrinsic sparse connectivity and interaction in many real-world problems. In particular, the recent emergence of sparse/compressed neural network models and large-scale graph analytics raises increasing demands for higher performance and better efficiency of SpGEMM.

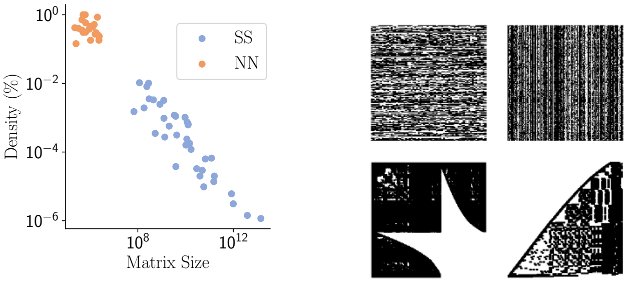
However, actually realizing such high-performance and efficient SpGEMM processing is known to be difficult due to the highly irregular structures, which lead to low utilization of both computation resource and memory bandwidth. As Dennard scaling ends and domain-specific acceleration becomes an increasingly appealing approach to speed up challenging computation tasks, SpGEMM has also been witnessing new architectural innovations to address the inefficiencies in general-purpose platforms [27, 31, 36, 42, 44, 45].

In this work, we observe that, as SpGEMM is applied to more and more application domains, the need for hardware accelerators to efficiently support highly *diverse sparse patterns* is also increasing. For example, real-world graph datasets are extremely sparse and usually have only 0.0001% nonzeros, but could be as large as 10^{13}. In contrast, neural network weight matrices are much smaller, usually on the order of 10^4 in size; even after compressing out both zero and near-zero values, they are still relatively dense with only moderate sparsity around 10% to 90%. Even within one domain or one dataset, different matrices or different sub-regions of a matrix constantly exhibit various sparse patterns in nonzero density and distribution. Unfortunately, existing SpGEMM accelerators are mostly designed to perform efficiently only when data are within certain sparsity ranges that best utilize their underlying hardware architectures. The core reason is that each of these accelerators only uses a fixed execution dataflow that optimizes for either input or output data reuse, but sacrifices the other. Consequently, the performance would suffer if the workload does not fit well with this rigid design assumption.

Therefore, we propose Spada, a combination of hardware and software innovations to efficiently accelerate SpGEMM applications across a widely diverse spectrum of sparse patterns. First, we comprehensively study the tradeoffs between existing SpGEMM dataflow schemes including inner-product [31, 45], outer-product [27, 44], and row-based [36, 42]. We then propose a **window-based adaptive dataflow** (WA). WA supports a spectrum of execution modes into which it can flexibly adapt, in order to realize the input and output reuse benefits of each individual dataflow under different data sparse patterns.

Second, we propose the **Spada hardware architecture**, which supports WA as the dataflow template, and can be flexibly and quickly reconfigured into different execution modes when using different WA configurations. The architecture uses specially designed multiply processing elements, each of which contains multiple lanes that can be dynamically partitioned into independent merge groups to produce individual output partial sum results with minimum hardware overheads. The key to supporting flexible WA schemes is the efficient and fast reconfiguration of such lane groups through specialized hardware components for sorting and reduction. We also use dynamic load balancing within each lane group to reduce idle cycles. In addition, the architecture uses dedicated comparator trees and accumulators to further merge the partial sum results, as well as a global cache with an optimized LRU replacement policy to improve data reuse in the WA dataflow.

Third, we propose an effective yet simple **window shape adaption** algorithm, implemented in the hardware Spada scheduler, to detect the sparse pattern of a specific workload and dynamically determine the best WA mode to use. The key insights are that the sparse pattern remains similar within a local region of the matrix, and that the complex pattern has a strong correlation with the row length of nonzero values. We thus design a profiling-guided approach to use the performance results of previous rows to determine the optimized configurations for future rows, which is quite effective and only has a small hardware cost.



(a) Matrix size and density. **(b) Nonzero distribution.**

Figure 1: Diverse sparse patterns across different workloads. (a) The size and the nonzero density of sparse matrices from compressed NNs using Distiller [47] (NN) and SuiteSparse [7] (SS). (b) The nonzero distribution in two sparse matrices in compressed NNs (top) and two from SuiteSparse (bottom).

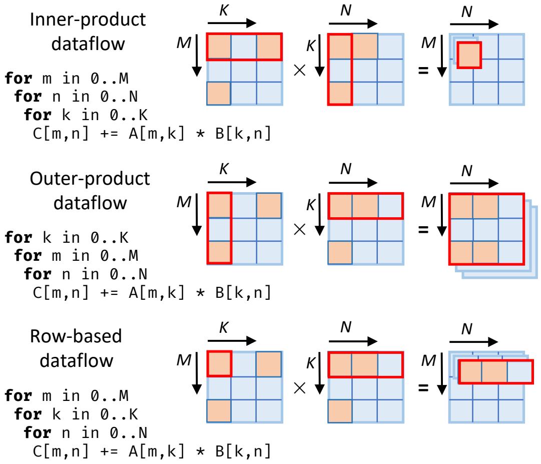
We evaluate Spada against three state-of-the-art SpGEMM accelerators, SIGMA [31] (optimized as in Section 6), SpArch [44], and GAMMA [42], across a wide range of sparse matrices and compressed neural network models with diverse sparse patterns. We observe that Spada can successfully achieve similar or even better performance compared to the best among the three baselines given the same amount of multiplier resources, with average speedups of 38.04× over SIGMA, 1.44× over SpArch, and 1.46× over GAMMA. Further considering the area cost brings the efficiency gains over SpArch and GAMMA to 3.32× and 1.42×, respectively. When the data distribution prefers a certain dataflow scheme, Spada can quickly adapt to the corresponding mode and match the performance of the baseline that is specifically optimized for that dataflow, while avoiding the substantial degradation of the other designs.

2 BACKGROUND AND MOTIVATIONS

Sparse matrices are usually encoded into specific formats, with the most commonly used two as compressed sparse row (CSR) and compressed sparse column (CSC). The CSR format stores a matrix row-by-row using a compressed encoding for only nonzeros in each row. It keeps three arrays, containing the offsets of the beginning of each row (`offsets`), the nonzero values (`vals`), and their column indices (`columns`), respectively. CSC is similar to CSR except that it encodes the matrix by columns. CSR is suitable for row-major traversals while CSC is better for column-major.

2.1 Diverse Sparse Patterns

Sparse matrices are commonly used to model many real-world problems from a wide variety of fields. The different intrinsic characteristics of these domains lead to a drastically diverse spectrum of *sparse patterns*, including the matrix size and the density and distribution of nonzeros in the matrix. To study their sparse patterns, we use real-world sparse matrices from the SuiteSparse collection [7] and generate compressed neural network (NN) weight matrices from ResNet50 [14]. Figure 1a shows that these sparse matrices exhibit drastically different sizes and densities, spanning up to seven



```
for m in 0..M
 for n in 0..N
  for k in 0..K
   C[m,n] += A[m,k] * B[k,n]
```

```
for k in 0..K
 for m in 0..M
  for n in 0..N
   C[m,n] += A[m,k] * B[k,n]
```

```
for m in 0..M
 for k in 0..K
  for n in 0..N
   C[m,n] += A[m,k] * B[k,n]
```

	InP	OutP	ROW
Input reuse (**B**)	Poor	Excellent	Poor
Output reuse (**C**)	Excellent	Poor	Good
Index intersection	Inefficient	Efficient	Efficient
Psum granularity	Scalar	Matrix	Vector

Figure 2: Comparison of three SpGEMM dataflow schemes: InP, OutP, ROW. Modified from [42]. Different hardware implementations may parallelize different for loops. E.g., both GAMMA [42] and MatRaptor [36] adopted ROW but parallelized dimension k and m, respectively.

orders of magnitude. NN compression techniques [5, 13, 25] usually generate relatively small (10^4) matrices with moderate sparsity (10% to 90%). In contrast, real-world graph structures such as social networks have much larger (10^{13}) and highly sparse (1% to 0.0001%) matrices. Furthermore, as in Figure 1b, across domains or even within a domain, the nonzero distributions could significantly differ due to various connectivity and locality behaviors, implying completely different sparse patterns.

2.2 Hardware Dataflow for SpGEMM

Similar to the large body of hardware dataflow proposals for dense matrix/tensor computations [4, 11, 12, 19, 24], SpGEMM also exhibits several possible dataflow choices that strongly impact performance and efficiency when used on different sparse patterns. Previously proposed SpGEMM dataflow schemes mainly fall into three categories: *inner-product* (InP) [31, 45], *outer-product* (OutP) [27, 44], and *row-based* (ROW) [36, 42]. Assume in Figure 2 we multiply matrices **A** and **B** to produce matrix **C**, with shapes of $M\times K$, $K\times N$ and $M\times N$, respectively. InP sequentially computes each $\mathbf{C}[m,n]$ element, as the inner-product of the corresponding pair of $\mathbf{A}[m,:]$ row and $\mathbf{B}[:,n]$ column. In contrast, OutP each time generates one of the K partial sum (psum) matrices of **C**, which is the outer-product result of the corresponding $\mathbf{A}[:,k]$ column and $\mathbf{B}[k,:]$ row. Finally, as a middle-ground between the above two, ROW divides **C** into rows, and computes one of the K psum rows of $\mathbf{C}[m,:]$ at a time through a scalar-vector multiplication between an $\mathbf{A}[m,k]$ element and the matching $\mathbf{B}[k,:]$ row. Essentially, if we use loop transformations [28, 40] to analyze, the three dataflow schemes correspond to three different loop orders as in Figure 2, with the reduction dimension k at the innermost, the outermost, and the middle.

These dataflow schemes differ in several aspects [36, 42]. First, they place different requirements on *encoding formats*: InP requires **A** in CSR and **B** in CSC; OutP requires **A** in CSC and **B** in CSR; and ROW requires both operand matrices in CSR and also generates the output in CSR. This makes ROW a more preferred scheme because the input and output matrix formats can be kept consistent in one representation, allowing for convenient chaining of multiple SpGEMM operations. Second, when the on-chip buffer capacity is insufficient to trivially hold all data, the three dataflow schemes would exhibit different *data reuse* behaviors (Figure 2). For example, assume **A** is always streamed, either along the rows (dim m) in InP and ROW, or the columns (dim k) in OutP. With InP, **C** achieves full reuse as all psums to each element are immediately accumulated, but each **B** column must be refetched multiple times, once for an **A** row, resulting in poor reuse. OutP, in contrast, only fetches each **B** row once to match with the corresponding **A** column. But the psums of **C**, if cannot fit on-chip, must be written to off-chip memory and read back later for accumulation, incurring significant traffic. Lastly, ROW only produces and stores a small amount of psums at a time for a single output row, which are easy to keep on-chip and achieve good reuse of **C** similar to InP. On the other hand, the rows in **B** are fetched irregularly based on the nonzero distribution of each **A** row (the **B** row index k should match the **A** element column index), leading to low reuse.

Third, OutP and ROW have another benefit of more efficient *index intersection* between the two input matrices. With InP, after fetching the entire arrays of $\mathbf{A}[m,:]$ and $\mathbf{B}[:,n]$, only their nonzeros with a matched k index could produce a psum to $\mathbf{C}[m,n]$. When the matrices are highly sparse, these effectual intersections are much smaller than the total array size, resulting in excessive data fetch that is useless. On the other hand, OutP fetches matched $\mathbf{A}[:,k]$ and $\mathbf{B}[k,:]$, and ROW fetches $\mathbf{A}[m,k]$ and $\mathbf{B}[k,:]$, both of which are already matched and contain effectual input pairs. Forth, the three schemes produce psums in different *granularities*, which affect loop blocking and parallelization when mapped to hardware.

2.3 Design Motivations

With the facts that different applications use sparse matrices with largely diverse patterns, and that different dataflow choices exhibit various tradeoffs, no single dataflow could always outperform the others under all circumstances. For example, with OutP, the sparse pattern would affect the resulting psum matrix size and thus how critical it is to optimize for output data reuse. In ROW, the sparse pattern (e.g., the nonzero distribution) also influences how much the column indices overlap between neighboring rows of **A**, in which case if two nonzeros $\mathbf{A}[m,k]$ and $\mathbf{A}[m',k]$ have the same column index, they would reuse the same input row $\mathbf{B}[k,:]$.

Therefore, for a hardware SpGEMM accelerator to efficiently support all sparse matrices in different domains, it should be able to *exploit the workload sparse pattern and accordingly adjust the execution dataflow.* To achieve this goal, we identify three key design challenges. First, we need to design **an adaptive dataflow** that could be easily adjusted into different modes when processing input matrices with different sparse patterns. Its different modes should cover a sufficiently large space of input and output reuse tradeoffs. Second, we need **an efficient and flexibly reconfigurable architecture**, which can support this adaptive dataflow and fast switch among its modes, with minimum hardware overheads. Third, we also need **an effective and simple runtime algorithm** to detect the sparse pattern of a specific workload, and then dynamically determine which mode to use. We address each of these challenges in the following sections.

3 WINDOW-BASED ADAPTIVE DATAFLOW

We first propose a *window-based adaptive dataflow* (WA) that can be adjusted into different *modes* to exploit the local sparse patterns of the input matrix. WA simplifies our hardware architecture design (Section 4), while still allowing for flexible adaption between different modes (Section 5).

A good adaptive dataflow should satisfy two conflicting goals. On one hand, it should be flexible to cover diverse reuse characteristics, so we could have a rich space for adaption. On the other hand, it is preferred to have the different modes share similar execution behaviors, so that we can keep hardware reconfiguration simple and cheap. This means that a naive design that tries to directly combine InP, OutP, and ROW is unlikely to work well, as they have drastically different execution behaviors. For example, InP traverses **A** and **B** by rows and columns, respectively, while the case is exactly the opposite for OutP.

We take a different approach. Instead of *precisely resembling the exact execution behaviors* of all dataflow schemes, we only aim to *capture their key benefits of data reuse* on different sparse patterns. We start from ROW, as its overall data reuse has been demonstrated in prior work to be better than others in many cases [36, 42]. Its key drawback is the poor locality to access **B**, whose row indices are determined by the column indices of the randomly distributed **A** nonzero elements. We then leverage loop blocking techniques, an effective way to improve data reuse in dense linear algebra [40], to fix this issue.

From Figure 2, we observe that OutP has the best input reuse on **B**. The key difference is on how to traverse **A**, by rows in ROW with a loop nest order `k`-`m`-`n` vs. by columns in OutP with `m`-`k`-`n`. Therefore we block the outer two dimensions with a loop nest order as `mo`-`ko`-`ki`-`mi`-`n`. By adjusting the block sizes, we can flexibly adapt to the modes that optimize for input and output reuse, respectively. Actually, if we set `mi` and `ki` to 1, it reduces to ROW; if we set `mo` and `ko` to 1, it reduces to OutP. Algorithm 1 formalizes this WA dataflow. Effectively, the inner two dimensions `mi` and `ki` form a 2D *window* on **A** (see below). When mapped to hardware, we spatially unroll this window on parallel processing units.

WA execution flow. WA fetches a small 2D *window* of **A** each time as the basic execution unit, and calculates the product of this window with their corresponding **B** rows. This is in contrast to ROW which fetches a single element of **A** each time. The window is defined directly on the compressed sparse rows (i.e., the CSR representation) rather than the dense format, so it does not contain zero values in a row or all-zero rows. As shown in Figure 3, we fetch a 2×2 window from **A** with four nonzeros, $a_{0,2}, a_{0,3}, a_{1,0}, a_{1,2}$. They are processed in parallel through four scalar-vector multiplications, with the corresponding **B** rows $\boldsymbol{b}_2, \boldsymbol{b}_3, \boldsymbol{b}_0$. Note that $a_{0,2}$ and $a_{1,2}$ reuse the same row $\boldsymbol{b}_2$. The products along the window width dimension (k) are merged into one psum of a **C** row, e.g., $a_{0,2}\times\boldsymbol{b}_2 + a_{0,3}\times\boldsymbol{b}_3 = \boldsymbol{c}_0^{(0)}$, which is accumulated to $\boldsymbol{c}_0$. Essentially, with a window of $\alpha\times\beta$, each row of β nonzeros from **A** are multiplied with the **B** rows matching with their column indices, and are then merged into one psum of a **C** row. These generate α psum rows.

Algorithm 1 Window-based adaptive dataflow (WA).

```
for each mo in 0 → $M/\alpha$ do
    ▷ One pass
    for each ko in 0 → $K/\beta$ do
        ▷ One window of shape mi × ki
        for each ki in 0 → $\beta$ do                    ▷ Spatially unrolled
            for each mi in 0 → $\alpha$ do               ▷ Spatially unrolled
                Calculate m and k from mo, mi, ko, ki
                for each n in 0 → $N$ do
                    C[m, n]+ = A[m, k] × B[k, n]
```

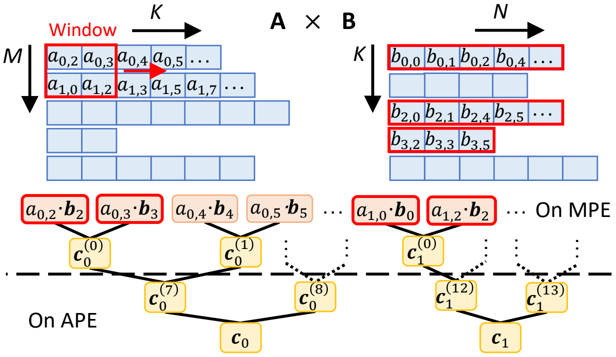


Figure 3: The window-based adaptive dataflow (WA) in Spada. Matrices A and B are shown in their CSR forms. MPE and APE are described in Section 4.

Similar to ROW, the window itself first moves along dim k (*not* sliding, but with stride equal to its width). We define a *pass* as the window moves across the current rows of **A** along k, after which the next pass starts by moving down along dim m. To be adaptive to data sparse patterns, WA allows to *change the window shape when starting a new pass* (Section 5). A pass contains multiple windows, each generating α psum rows. These psum rows need to be merged into the final α output rows of **C**, e.g., all $\boldsymbol{c}_0^{(t)}$ merge to $\boldsymbol{c}_0$ and all $\boldsymbol{c}_1^{(t)}$ merge to $\boldsymbol{c}_1$ as in Figure 3 bottom. The merge of each $\boldsymbol{c}_m$ runs independently from each other, as separate reduction trees.

WA fetches all the three matrices in memory-friendly manners *without* random accesses. On matrix **A**, WA simultaneously fetches α compressed rows sequentially as the window moves. All nonzeros are fully utilized. Each row can also be prefetched in relatively large chunks given a small buffer space, e.g., a few hudrends of bytes for 8 rows. The access patterns of **B** and **C** are the same as in ROW, where a full compressed **B** row is fetched for each nonzero in **A**, and generates a psum row sequentially.

WA advantages. First, using a window with multiple **A** elements rather than a single one in ROW could naturally match the parallelism offered by the underlying hardware. In Section 4 we set the window size $\alpha\times\beta$ to be equal to the number of multipliers within a processing element.

Second and more importantly, by adjusting the window shape of $\alpha\times\beta$, we can effectively achieve different modes with various data reuse benefits. Specifically, setting $\alpha = 1$ (traversing along k in **A**) realizes high output reuse, as the large window width β allows more psum rows of **C** to be immediately merged into one, resembling the behavior of ROW. On the other hand, setting $\beta = 1$ (using a column of **A**) may achieve better input reuse of **B**. This is because a large window height α enables more reuse opportunities of **B** rows across multiple **A** elements in the same column but different rows (e.g., $a_{0,2}$ and $a_{1,2}$ reuse the same row $\boldsymbol{b}_2$ in Figure 3). Moreover, between these two extremes, WA also enables many more modes with different α and β values that exhibit potentially better performance with more balanced tradeoffs between the input and output reuse.

Finally, WA inherits many advantages from ROW, such as the consistent CSR format across all matrices, the moderate granularity for psum generation, and the avoidance of ineffectual index intersection in InP. In fact, by including ROW as one of the many modes, WA has the potential to at least match ROW and further outperform it when input reuse is more critical, assuming low hardware overheads from supporting adaption and accurate policies to select modes, which we address next.

4 SPADA HARDWARE ARCHITECTURE

Having an adaptive dataflow WA, we next design the hardware architecture of our SpGEMM accelerator, Spada. Instead of implementing multiple individual dataflow schemes, Spada supports WA as a *dataflow template*, and allows *efficient, flexible, and fast reconfiguration* among different modes, to best exploit data reuse opportunities on different sparse patterns.

Figure 4 top right shows the Spada architecture overview. Spada consists of multiple multiplication and addition processing elements (MPEs and APEs), a scheduler, and a global cache in front of the off-chip memory. Each MPE executes the multiple scalar-vector multiplications (between **A** elements and **B** rows) in one window of WA, and conducts the first level of psum reduction within the window (called a *multiply* task; Section 4.1). The APEs further complete the inter-window reduction to produce the final output rows (called *merge* tasks; Section 4.2). This division is illustrated in Figure 3 bottom. The scheduler tracks the task execution on the two types of PEs (Section 4.3), and also adaptively adjusts the window shape for each new WA pass (Section 5). The global cache stores and reuses **B** and **C** data using an improved replacement policy (Section 4.4), while **A** is only streamed from memory.

4.1 Multiply Tasks on MPEs

The Spada accelerator contains multiple MPEs, and each MPE has several multiplier *lanes*. The number of lanes per MPE is set to match the supported WA window size, $\alpha\times\beta = 8$ in our design. The bottom part of Figure 4 illustrates the internal MPE components. Each lane has a multiplier, a B fetcher, and a P queue. Each pair of lanes use a sort array for load balancing. And all lanes are connected to a final bitonic network [3] and a flexible reduction tree [31].

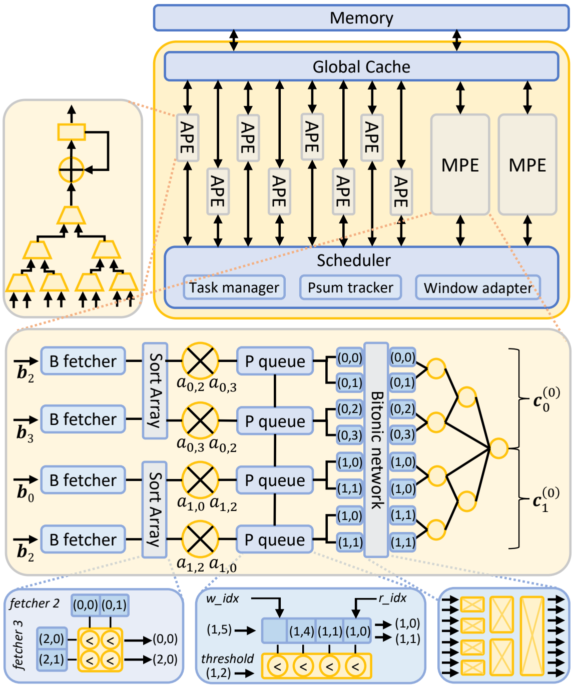


Figure 4: Spada hardware architecture. Top right: overall design. Top left: addition PE (APE). Middle: multiplication PE (MPE). Bottom left: sort array. Bottom middle: P queue. Bottom right: bitonic network.

Each MPE is assigned a *multiply task* to execute, which corresponds to one window in WA. The multiply task executes as follows. ❶ Load a window of $\alpha\times\beta$ elements from **A**, and put each $a_{m,k}$ into each lane's multiplier as well as its paired lane for load balancing. ❷ Each lane's B fetcher uses the column index of its $a_{m,k}$ to fetch the corresponding **B** row $\boldsymbol{b}_k$ from the global cache, and sequentially feeds the nonzeros into the multiplier. ❸ Execute the scalar-vector multiplications in parallel, one per lane with the scalar $a_{m,k}$ and the row $\boldsymbol{b}_k$. The sort array for each pair of lanes dynamically balances their progress by allowing both multipliers to temporarily process the loads of the lagging-behind lane (discussed shortly). The P queues are used to buffer the products before merging, further tolerating mismatched progress across lanes. ❹ Merge the products into α psum rows of **C**. Such merge happens independently among each β lanes, so we call every β lanes as a *group*, and there are α groups in an MPE. Lane groups are dynamically reconfigurable based on the window shape to flexibly support different WA modes. This is realized with the bitonic network and the reduction tree. The group size β can be any power-of-two between 1 and all lanes. ❺ Finally, psum rows are written back to the global cache, and later further merged by the APEs (Section 4.2).

Figure 4 shows the example of MPE execution with the window in Figure 3. The four lanes form two groups, with $a_{0,2}$, $a_{0,3}$ to the first two lanes and $a_{1,0}$, $a_{1,2}$ to the other two. Corresponding **B** rows are fetched by B fetchers. And the two groups respectively merge into two psum rows $\boldsymbol{c}_0^{(0)}$ and $\boldsymbol{c}_1^{(0)}$.

Dynamic lane grouping and merge within each group. We notice that each lane produces a sorted psum row with ascending column indices, buffered in the P queue. Multiple such psum rows from different lanes in one group should be merged (and reduced for those with the same column indices). The key to supporting flexible WA window shapes in the MPE is through the dynamic reconfiguration of lane groups. The grouping does not affect the scalar-vector multiplications, but mainly changes how their products are merged. We use specialized P queues, a bitonic network [3], and a flexible reduction tree [31] to achieve such reconfigurability.

The **P queue** (P for psum) in each lane buffers the sequentially generated products from the multiplier. It acts similarly to a normal FIFO, but with the following enhancement at its output port. To correctly merge psum rows from multiple P queues, we must ensure *no queue can output an index that is smaller than the outputs of any queue in previous cycles*, or equivalently, in each cycle, the smallest set of indices across *all* queues (not just within each single queue) are sent to the later stage for sorting and merging. For example, if two P queues have their current smallest indices as 0 and 3, the second queue cannot output 3 unless it knows the next smallest index in the first queue is no less than 3. If that index is not available now, the second queue output must stall.

To ensure the above requirement, we associate a *threshold* to each P queue, as in Figure 4 bottom middle. At each cycle, only the indices at the head of the queue that are *smaller than* the threshold can output. For example, in the figure with the current threshold $(1, 2)$, only $(1, 0)$ and $(1, 1)$ can output, but not $(1, 4)$. The multiplier is usually able to constantly push one element into the queue per cycle, but the threshold may result in no outputs in some cycles. Therefore we design each P queue to be capable of popping out up to two elements per cycle to compensate. We empirically find in Section 7.4 that this slight over-provisioning tolerates most stalls and achieves matched throughput on average.

Algorithm 2 Setting P queue threshold for a lane group.

```
Input: P queues in a lane group: p_queues[]
Output: threshold for the lane group: threshold
threshold = MAX
for each q in p_queues do
    if q.len() $\geq$ 3 then
        threshold = min{threshold, q[2].index}
    else if not q.complete() then
        threshold = min{threshold, q.tail().index}
return threshold
```

The threshold is maintained per group, i.e., the same threshold applies to all lanes in a group, denoting an index that is guaranteed to be smaller than any future P queue outputs. Algorithm 2 shows how to update the threshold. Because each queue can pop out up to two elements in each cycle, we set the threshold to be the minimum index among either the third index or the tail of each queue. If a lane has completed all elements in its psum row, it does not restrict the group threshold anymore. Also because we only pop out elements that are strictly smaller than the threshold, the queue never becomes empty in the middle of the processing.

The outputs from all P queues, as an unordered set of elements for each group, are next sent to the **bitonic network**. The P queues and the bitonic network together form a two-phase sorting, where the P queues guarantee the order between different cycles, and the bitonic network ensures the order in the same cycle. With up to two elements from each P queue, the width of the bitonic network inputs is 2× of the number of lanes. As Figure 4 bottom right shows, the bitonic network implements the hardware-friendly bitonic sort algorithm [3], with a logarithmic number of block levels and multiple sorting blocks per level. Each sorting block sorts its inputs and produces ordered outputs. To accommodate flexible lane grouping, we modify the original bitonic network to allow for getting the outputs from any level. For example, to individually sort each group of 2 lanes, we should retrieve outputs just after the first block level and bypass the rest.

The outputs from the bitonic network have been sorted, but those elements with the same column indices are not yet reduced. We finally use a **flexible reduction tree**, similar to the forwarding adder network in SIGMA [31]. It is designed to arbitrarily separate its sub-trees to independently merge each variable-length sub-range. In our case, these sub-ranges are split according to the column indices. As in Figure 4 middle, the reduction tree merges the two groups of sorted products {(0, 0), (0, 1), (0, 2), (0, 3)} and {(1, 0), (1, 0), (1, 1), (1, 1)} into {(0, 0), (0, 1), (0, 2), (0, 3)} and {(1, 0), (1, 1)}. The reduced sums from the sub-trees are pipelined to the final stage as the eventual result $\boldsymbol{c}_m^{(t)}$ with ordered column indices, and are written back to the global cache.

Dynamic load balancing. One potential bottleneck in this design is the imbalance among lanes in a group. Because only the smallest indices can be sent for merging in each cycle, if one lane largely lags behind with many small indices, then the other lanes must wait, eventually filling up the P queues and stalling the multipliers. Note that the column index distribution of each lane's psum row depends on its input row $\boldsymbol{b}_k$, which could vary widely.

To alleviate this issue, Spada MPEs support dynamic load balancing between each pair of neighboring lanes, using their shared **sort array**. If the two lanes are in the same group, the sort array compares the next two column indices of $\boldsymbol{b}_k$ elements from each lane's B fetcher, and selects the smallest two among the four to be sent to the multipliers. For example, in Figure 4 bottom left, B fetcher 2 has $(0, 0)$, $(0, 1)$, and B fetcher 3 has $(2, 0)$, $(2, 1)$. The two with the smallest column indices, i.e., $(0, 0)$, $(2, 0)$, are sent out in this cycle. Recall that the **A** elements are loaded into both multipliers. The row indices are used to select which $a_{m,k}$ to use.

In Spada MPEs, the sort arrays only re-balance between each two lanes, regardless of the group size. This is a tradeoff between performance and cost. Each sort array needs $O(n^2)$ resources, and would grow too large if covering too many lanes [44].

Impact of empty rows and imbalanced row lengths. Any empty row in **A**, which can be detected from the CSR `offsets` array, is simply skipped without being included in the WA window, and would not occupy any resource or introduce any load imbalance. However, if some compressed **A** rows in a pass are shorter than the others, towards the end of the pass the window would contain fewer elements and thus cannot fully occupy all multiplier lanes. Besides, when encountering an empty row in **B**, the B fetcher gets

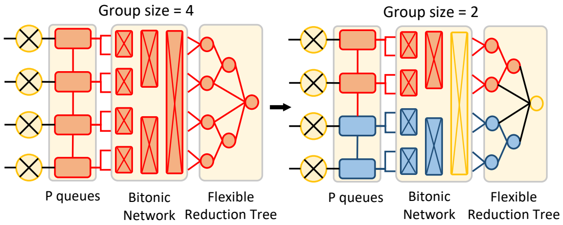


Figure 5: MPE reconfiguration, which only affects the P queues, the bitonic network, and the flexible reduction tree.

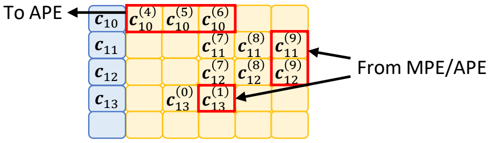
no element and the lane immediately finishes its work. In both cases, load imbalance can be alleviated by the sort array, which allows the idle lane to process the workload of its neighbor lane.

Reconfiguration and task pipelining. The MPEs support different WA window shapes $\alpha\times\beta$ through the flexible lane grouping mechanism described above. We allow the MPEs to be reconfigured to a different window shape only after completing a pass, i.e., when finishing the current set of **A**/**C** rows and switching to the next rows. So reconfiguration happens much less frequently than previous work [31]. The new window shape is set by the window adapter in the scheduler (Sections 4.3 and 5). Reconfiguration of an MPE does not affect the first few stages, but only changes the P queue threshold logic, the bitonic network output level, and the flexible reduction tree. As an example in Figure 5, when the window shape is changed from 1×4 to 2×2, the group size is updated from 4 to 2. The P queue threshold is updated to track the smallest index among two lanes in each group. The bitonic network is configured to sort only within each group, and to output at the second level. The flexible reduction tree is also configured to only merge within each group of two lanes. The two outputs from the two groups become two psum rows for different **C** rows.

Across different tasks (of the same window shape) in one pass, the different stages in MPE effectively form a pipeline, so the next task can start immediately (e.g., multipliers load **A** elements, B fetchers start fetching) after the end of the last task without fully draining the pipeline. The P queues are aware of different tasks, so the products belonging to different tasks are not popped in the same cycle to ensure correct merging. For consecutive tasks between passes (of different window shapes), additional stalls at the P queue output ports are needed, in order to wait for reconfiguring the later units. Specifically, we cannot let any element of the new pass pop out of a P queue, until all the elements of the previous pass completely leave all the P queues; the lanes that finish draining early need to stall and wait. The reconfiguration itself also adds a few pipeline stall cycles to change the configuration bits. Nevertheless, during these stall cycles, the B fetchers can start prefetching the **B** rows of the next task and reduce later cache misses.

4.2 Merge Tasks on APEs

While the psum rows from each WA window are fully merged inside the MPE, those across windows in one pass are separately written back to the global cache in different cycles and need further merging. We use dedicated APEs in Spada for these *merge tasks*. Each APE is a lightweight unit, as in Figure 4 top left, composed of a comparator tree of radix R and an accumulator [42]. It takes R psum rows as input and streams them in, and in each cycle, it outputs the element with the smallest column index among them. The accumulator reduces the output elements with the same index. If the number of psum rows is more than R, the output streams would need further merge rounds, essentially as a tree of merge tasks [42]. Our implementation uses $R = 8$.



$\boldsymbol{c}_{10}$ $\boldsymbol{c}_{10}^{(4)}$ $\boldsymbol{c}_{10}^{(5)}$ $\boldsymbol{c}_{10}^{(6)}$
$\boldsymbol{c}_{11}$ $\boldsymbol{c}_{11}^{(7)}$ $\boldsymbol{c}_{11}^{(8)}$ $\boldsymbol{c}_{11}^{(9)}$
$\boldsymbol{c}_{12}$ $\boldsymbol{c}_{12}^{(7)}$ $\boldsymbol{c}_{12}^{(8)}$ $\boldsymbol{c}_{12}^{(9)}$
$\boldsymbol{c}_{13}$ $\boldsymbol{c}_{13}^{(0)}$ $\boldsymbol{c}_{13}^{(1)}$

Figure 6: Psum tracker table structure in Spada scheduler.

MPEs and APEs in Spada work in a macro-pipelined way, coordinated by the scheduler (Section 4.3). They communicate the psum rows through the shared global cache. To balance their throughput, we provision MPEs and APEs by having enough numbers of APEs to match the peak throughput of the MPEs. MPEs are the most expensive resource so we would like to keep them fully utilized; APEs consume much smaller area and power so using more APEs has minor overheads. For example, in Figure 4 there are two MPEs with four lanes each, so they produce up to eight psum elements per cycle in the worst case (when group size is 1 and no reduction in each group happens). Hence we use eight APEs where each consumes one psum element per cycle.

4.3 Scheduler

The Spada scheduler contains several components, for assigning tasks to MPEs/APEs, tracking task execution states and psum rows, as well as collecting performance information and reconfiguring WA dataflow modes.

The **task manager** generates new multiply and merge tasks and tracks their execution states. For each pass (i.e., fully traversing the window along dim k), Spada uses a fixed window shape determined as in Section 5. The task manager records the current pass, the current window shape, and the current position of the window in this pass; it generates new multiply tasks by moving the window. In addition, it generates new merge tasks when enough psum rows have been produced (e.g., fully utilizing the APE radix R) after notified by the psum tracker (see below), or when all multiply tasks in a pass are complete. It also keeps the states of ongoing tasks, including the window shape, the MPE/APE ID, the input matrix row addresses, and the input/output psum row addresses.

The **psum tracker** manages all intermediate psum rows using a hardware table as illustrated in Figure 6. Each entry in the table corresponds to a final output row $\boldsymbol{c}_m$, and contains a list of addresses for the psum rows $\boldsymbol{c}_m^{(t)}$ that should be merged into it. This address list is managed as a circular buffer, where newly produced psum rows (from either multiply tasks or merge tasks) are pushed in, and existing ones are popped out to generate a merge task. When all multiply tasks in this pass are complete and there is only one psum row remaining in the entry, that psum row becomes the final result. Both the number of table entries and the list length per entry are design parameters that are well bounded in our design.

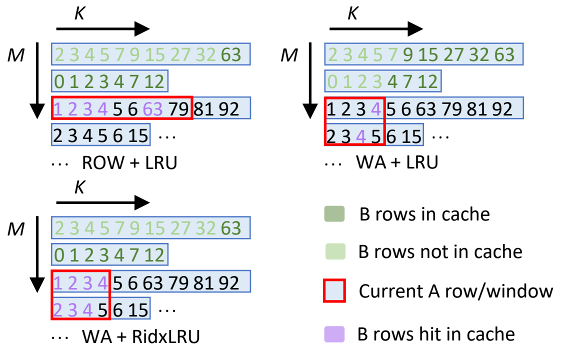


Figure 7: Comparison of global cache replacement policies. The figure shows the CSR format of the matrix A, where numbers are the column indices of A nonzeros, which are equal to the indices of B rows that are multiplied and also cached. The global cache can store eight B rows. Currently processing A rows or windows are highlighted in red boxes.

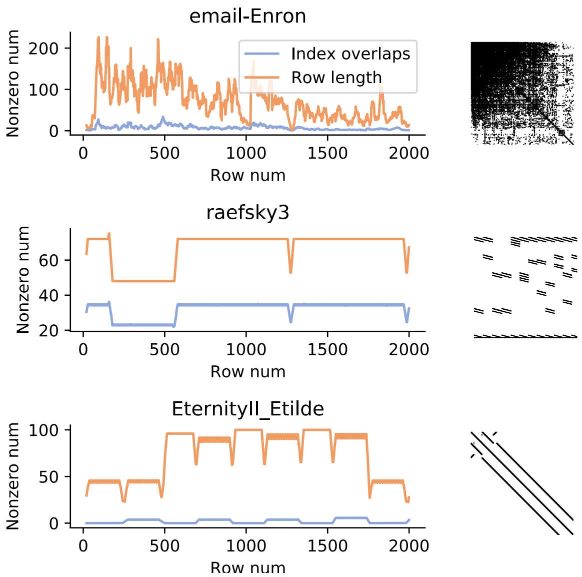


Figure 8: Index distribution similarity and correlation with row length. We sample 2000 contiguous rows from several sparse matrices to plot the number of column index overlaps between adjacent rows and the row length of nonzeros. The sparse pattern of each matrix is shown on the right.

On average, the number of table entries should cover all $\boldsymbol{c}_m$ rows that are currently under processing, which is equal to the number of MPEs times the rows per MPE (i.e., the window height α). The address list needs to be long enough for all intermediate psum rows for a specific output row. Because we use sufficient APEs, as long as R psum rows are produced, it should be quick to find an idle APE to merge them. For both parameters, we further apply small factors of over-provisioning. Consequently, a table with 16 entries and 10 psum rows per entry avoids almost all stalls.

Finally, the **window adapter** collects runtime information and determines an optimized window shape for each pass. We defer to Section 5 for its detailed algorithm.

4.4 Global Cache and Replacement Policy

Spada uses a unified global cache for input rows $\boldsymbol{b}_k$ from **B** and psum rows $\boldsymbol{c}_m^{(t)}$ for C. Both are accessed in streaming manners. We adopt the FiberCache design in GAMMA [42]. A shared cache could exhibit better utilization than separate ones when the input and output reuse trades off in different dataflow modes.

To better support WA, we also apply a small optimization on the cache replacement policy. In the original ROW dataflow, a simple LRU policy would effectively leverage sparse pattern locality between neighboring rows. In Figure 7 top left, $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, $\boldsymbol{b}_3$, etc. used by the previous **A** row remain in the cache and can be reused by the current row, while those fetched by the older **A** rows are evicted. However, with WA that processes multiple **A** rows simultaneously, the cache may not be large enough to cache all rows. As a result, only the tail parts of these rows could remain in the cache, while the head parts are completely evicted. When the current window starts from the beginning of the following rows, few **B** rows could be reused (Figure 7 top right).

To improve reuse, we use a RidxLRU policy, which associates each **B** row in the global cache with the most recent **A** row index that used it. When replacing, **B** rows with smaller **A** row indices are prioritized for evicting (Figure 7 bottom). RidxLRU in WA effectively approaches the effects of LRU in ROW, allowing us to reuse more **B** rows fetched by the previous **A** rows.

5 WINDOW SHAPE ADAPTION

As discussed in Section 3, the WA window height α determines the input reuse of **B** rows across multiple **A** rows, and the window width β determines the number of intermediate psum rows for output reuse. In Spada in Section 4, the total window size $\alpha \times \beta$ is fixed to the number of lanes in an MPE. Therefore, the runtime algorithm for determining optimized window shapes should properly adjust between α and β to best balance the reuse of both input and output.

However, accurately capturing these effects in an analytical model is almost impossible. The reuse of **B** rows highly depends on the index distribution between **A** rows. If neighboring **A** rows have many overlapped column indices, these nonzeros could reuse the same **B** rows. But such information cannot be extracted without actually scanning through each CSR row of **A**, leading to unacceptable overheads. Furthermore, even if we could somehow estimate such similarity, the actually achievable reuse on hardware still depends on the complex cache behaviors and the interference among MPEs.

Therefore, we design our window shape adaption algorithm in a profiling-guided way. The key idea is that, *within a local matrix region, we can use the profiling performance results on the first few rows to determine the optimized window shape for the following rows.* We rely on two key insights from empirical studies in Figure 8. The first one, which we call *index distribution similarity*, says that within a local region of contiguous rows, the overlapped column indices between adjacent rows[1] are relatively stable. We call such a region a *band*. Within a band, the similar index overlapping would lead to similar reuse behaviors, and thus the same choice of the optimized dataflow. Naturally, the band serves as the granularity of our profiling-based method. The band size could be large, e.g., up to several hundred rows in `EternityII_Etilde`, or relatively small, e.g., dozens of rows in `dbir2`.

The other insight further answers how we partition bands. Inspecting all column indices in each row is obviously impractical. Instead, we observe *strong correlation between index distribution and row length*, i.e., in a band with similar index distribution, the row lengths also stay relatively stable. This can be seen from the similar shapes between the two curves in Figure 8: when one curve sees a sharp variation, the other curve follows. Note that their relative ratios *across* bands are not necessarily constant; we only need the sharp changes to detect the band boundaries. We suspect that such behavior is due to the regular local patterns (e.g., diagonal bands, local dense regions) that widely exist in sparse matrices. As a result, we can partition a matrix into bands by only looking at the row lengths, i.e., the CSR `offsets` array, which is much faster.

To determine *what to profile*, recall that each band contains multiple rows and thus multiple WA passes. The end-to-end performance of executing one pass is an intuitive candidate. However, due to the asynchronous execution between multiply and merge tasks (Section 4), the total runtime is difficult to accurately measure. Instead, we leverage the architectural features in Spada that there are sufficient APEs to match the maximum throughput of MPEs and the two phases form a macro-pipeline, so the overall performance is mostly determined by the multiply tasks. Moreover, the index distribution mostly affects the reuse of **B** rows which are the input to multiply tasks. Therefore, we track *average runtime of multiply tasks* in the Spada scheduler window adapter.

Finally, for *how to profile*, we further apply an optimization to handle large and small bands separately. For a large band with many rows, we divide its execution into a profiling phase, during which we try different window shapes on the first few rows, and a stable phase, during which we simply apply the best window shape to the rest rows. However, for small bands, the profiling phase may dominate the execution or even cannot finish due to too few rows. So instead we use a more adaptive cost-descent approach. When trying different window shapes, as soon as we realize a shape results in decreased performance, we immediately stop profiling and use the best window shape found so far. This method adapts more rapidly, but may be sub-optimal due to insufficient profiling.

Put it all together. We summarize our window shape adaption algorithm in Spada. (1) The window adapter loads the CSR `offsets` array of **A**, and partitions the rows into bands. We empirically use an absolute threshold $T_{\text{abs}} = 5$ and a relative threshold $T_{\text{rel}} = 2$. If the difference (ratio, respectively) between $\text{len}[i]$ (i.e., $\texttt{offsets}[i] - \texttt{offsets}[i-1]$) and $\text{len}[i-1]$ is larger than T_{abs} (T_{rel}, respectively), we start a new band from row i. (2) Each band is categorized as large or small, using an empirical threshold of $T_{\text{band}} = 128$ rows. (3) For a large band, we first execute four passes with window shapes 1×8, 2×4, 4×2, 8×1 in the profiling phase, and respectively track the number of multiply tasks and their runtime sum in each pass. Then we use the window shape with the best average runtime for the rest passes in this band in the stable phase. (4) For a small band, we start with a 1×8 window, and then 2×4, etc. When a new window shape causes decreased performance, we stop trying new ones and use the currently obtained best. We keep tracking and updating the performance of the most recently executed pass for each window shape, and change the applied one to always use the best.

Table 1: Hardware configurations of Spada.

MPEs	2 8-lane MPEs; 8-slot P queue; 16-width net/tree
APEs	16 APEs; 8-radix binary comparator tree
Global cache	1.5 MB, 16 banks, 16-way associative
Crossbars	16×16 and 16×16, swizzle-switch based
Main memory	128 GB/s over 16 64-bit HBM channels

Table 2: Area breakdown of Spada and one MPE.

Components	Area (mm^2)	MPE components	Area (mm^2)
2 MPEs	0.86 (14%)	8 B fetchers & SAs	0.11 (26%)
16 APEs	0.40 (6%)	8 Multipliers	0.05 (12%)
Scheduler	0.07 (1%)	8 P queues	0.09 (21%)
Global cache	4.80 (76%)	Bitonic network	0.07 (16%)
Crossbars	0.19 (3%)	Reduction tree	0.11 (25%)
Total	6.32	MPE total	0.43

6 METHODOLOGY

Configurations. We evaluate Spada with the default configurations shown in Table 1. We use 2 MPEs each with 8 lanes, 16 APEs, and a 1.5 MB global cache. We implement the key components, i.e., MPE, APE, and scheduler, in RTL, and synthesize them using Synopsys DC on the TSMC 28 nm technology. The Spada chip runs at 1 GHz, and is connected to a 128 GB/s High-Bandwidth Memory (HBM) module. We use CACTI 7.0 [2] to model the global cache and use the swizzle-switch network for the crossbars [33]. To measure performance and memory traffic, we further build a cycle-accurate simulator of Spada in Rust. The simulator carefully models the interactions between hardware components and implements the window shape adaption algorithm in Section 5. The simulator is open-sourced at https://github.com/tsinghua-ideal/spada-sim.

Workloads. We use 27 sparse matrices with a large variation of sparse patterns, from both the SuiteSparse collection [7] and several compressed NN models, as summarized in Table 3. We select 18 matrices from SuiteSparse, with the goal of having their densities cover the full range in Figure 1a, i.e., 10^{-7} to 10^{-1} (sorted in Table 3). Besides, their nonzeros per row vary widely from 4 to 5162, further ensuring diversity and validating the need for flexibly supporting different dataflow schemes. To construct SpGEMM workloads, we follow the same method in GAMMA [42], where a square matrix is multiplied with itself and a non-square matrix is multiplied with its transpose. To include compressed NNs, we train and compress ResNet50 [14] and AlexNet [21] using open-source toolchains [47]. Furthermore, a pruned BERT-Base is trained as in [32]. We select

[1] Index overlap between two consecutive rows $i, i+1$ is defined as the number of their identical column indices, $|\texttt{columns}[\texttt{offsets}[i] : \texttt{offsets}[i+1]] \cap \texttt{columns}[\texttt{offsets}[i+1] : \texttt{offsets}[i+2]]|$.

Table 3: Characteristics of evaluated workloads.

Workload	Density	Workload	Density
hugetrace-0010	2.48e-07	msc10848	1.04e-02
cit-Patents	1.16e-06	lpi_forest6	2.85e-02
kkt_power	3.00e-06	cari	3.18e-01
web-Google	6.08e-06	lp_fit2d	4.90e-01
Hardesty2	1.42e-05	—	—
ldoor	4.69e-05	alexnetfc2	2.49e-01
email-Enron	2.73e-04	resnet50fc	1.13e-01
ca-CondMat	3.35e-04	resnet50b3_c1	3.38e-01
EternityII_Etilde	5.70e-04	resnet50b4_c3	3.16e-01
dbir2	1.33e-03	resnet50b2_c1	4.17e-01
poisson-3Da	1.93e-03	resnet50b1_c2	6.76e-01
ship_001	3.20e-03	bertl0_key	6.25e-02
raefsky3	3.31e-03	bertl0_query	7.14e-02
nemsemm1	3.54e-03	bertl0_ffn	7.22e-02

representative convolutional and fully-connected layers from different residual blocks of ResNet50, the second fully-connected layer of AlexNet, and the query, key, and feed-forward network from layer 0 of BERT-Base.

Baselines. We use three state-of-the-art SpGEMM accelerators that use InP, OutP, and ROW, respectively, i.e., SIGMA [31], SpArch [44], and GAMMA [42]. To make fair comparisons, we set all designs to have the same number of multipliers, i.e., sixteen. SpArch and Spada are already in such a size, while we scale down GAMMA by half, e.g., using half of its original 32 PEs, and correspondingly half the size and banks for the original 3 MB FiberCache which is now at the same capacity as Spada (1.5 MB). For SIGMA, we scale a Flex-DPE down to the width of 16, shrink the SRAM buffer to 1.5 MB, and increase its frequency to 1 GHz. Since its original bitmap format scales poorly to very sparse workloads, we modify it to use the CSR format and adopt the content-addressable-memory implemented in ExTensor [15] to accelerate index matching. We also evaluate scaled-up versions of these designs in Figure 14. All designs use the same off-chip HBM module as Spada. All designs use the 64-bit double-precision data type that is commonly used in scientific computing, following the same setting as SpArch and GAMMA for fair comparisons. A common practice of compressed NNs is to use lower precision for inference. We leave such flexible precision support as future work, which is similar to how GPUs support both workload types [6, 9] and other academic proposals [20, 22, 34]. Finally, we also compare with CPU and GPU platforms. For CPU, we use `mkl_sparse_spmm` in Intel MKL [17], on a server with dual Xeon 6240 processors (each has 36 hyper-threads and a 24.75 MB last-level cache) and 8 DDR4-2933 channels. For GPU, we measure `cusparseXcsrgemm2Nnz` followed by `cusparseDcsrgemm2` in cuSPARSE [26] on one NVIDIA RTX 3090 card.

7 EVALUATION

7.1 Area and Power

Table 2 shows the area breakdown of Spada with the configurations in Table 1. Same as previous work [42, 44], most of the chip area in Spada is used by the large 1.5 MB global cache, as SpGEMM is mostly memory-bound and requires large caches to alleviate the off-chip data transfer bottleneck. Among the logic components, the MPEs dominate the area. The buffers for input and output data (B fetchers and P queues) as well as the merge network and reduction tree consume most of the space in each MPE. All the APEs, despite being 8× more in numbers, occupy 2.2× less area than the MPEs. These small area overheads justify the choice of using dedicated APEs for the merge tasks and making MPEs exclusively focus on the critical multiply tasks. The scheduler only introduces a minor extra area to the chip.

Compared to the baselines, if both (scaled) GAMMA and SpArch are scaled from 45 nm to 28 nm as Spada, we see that GAMMA and Spada have similar areas (6.13 mm^2 and 6.32 mm^2), while SpArch is about 2× larger (13.96 mm^2). Notice the three designs have the same number of multipliers, therefore Spada and GAMMA are more area-efficient. We exclude concrete area comparison with SIGMA. Its reported area was under 500 MHz with much more resources [31], and thus it is hard to accurately estimate the scaled-down area.

We also report the power consumption of Spada from the synthesize results. The MPEs, APEs, and scheduler together consume 1.66 W. The whole Spada chip power is about 4.84 W, dominated by the global cache.

7.2 Performance

Figure 9 uses different workloads to compare the four accelerators: SIGMA, SpArch, GAMMA, and Spada. We see that performance varies a lot across workloads due to their diverse sparse patterns, and no single accelerator always achieves the best performance for all workloads. This result validates our motivation that different SpGEMM workloads exhibit different sparse patterns that prefer different execution dataflow choices.

Specifically, for the sparse workloads from SuiteSparse towards the left, InP has poor performance in most cases, which is mainly affected by the inefficient index intersection even if we applied extra optimizations [15]. InP performs relatively well on `lpi_forest6`, `cari`, and `lp_fit2d`, as they are either small enough to fit in the cache or significantly denser than other workloads. For the other two baselines, SpArch (OutP) outperforms GAMMA (ROW) on 4 SuiteSparse workloads `lpi_forest6`, `dbir2`, `cari`, and `lp_fit2d`, with nearly 20× difference on `cari`. ROW performs particularly bad with `cari`, because the many long rows in **A** and **B** thrash the cache. ROW is better on the rest SuiteSparse workloads with up to 6.2× on `email-Enron`. `email-Enron` is most efficient with ROW due to limited overlapped column indices across **A** rows and thus limited input reuse opportunities in OutP.

With the denser NN workloads on the right side, though still inefficient on many workloads, SIGMA has more comparable performance and achieves an 8.2× speedup on `resnetb4_c3`. SpArch outperforms GAMMA on the two compressed fully-connected layers `resnet50fc` and `alexnetfc2`, while GAMMA is better on the rest convolutional and self-attention layers.

In contrast, due to the capability of flexibly adapting its WA dataflow mode between exploiting input or output reuse to dynamically match the sparse pattern, Spada is able to outperform (e.g., `kkt_power` and `bertl0_ffn`) or perform close to the best among the baselines on almost all evaluated workloads. The two fully-connected layers are exceptions, where there remains a gap

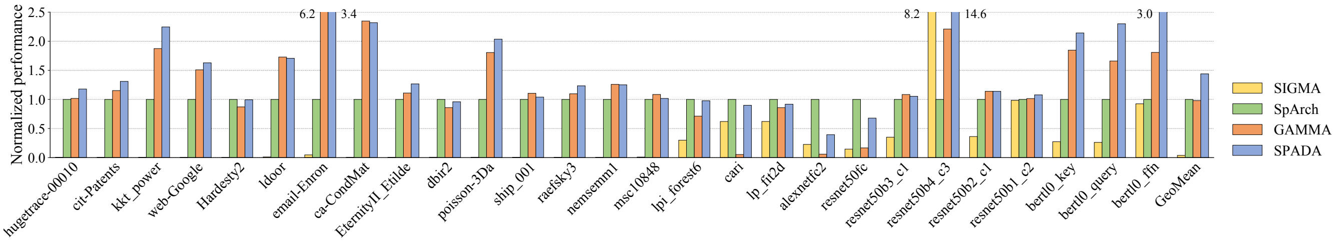


Figure 9: Performance comparison among SIGMA, SpArch, GAMMA, and Spada.

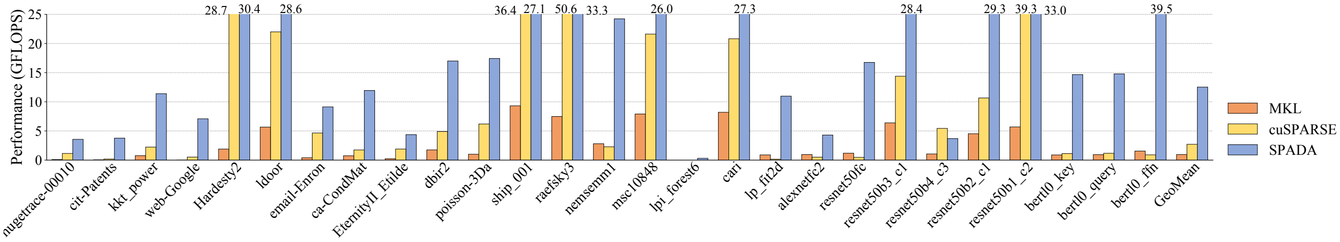


Figure 10: Performance comparison among CPU (MKL), GPU (cuSPARSE), and Spada.

between WA and the optimal OutP. This is because there are few **A** rows (e.g., 16) that are insufficient to profile to get optimized window shapes. Spada could achieve even higher performance than the baselines which are specifically optimized for one dataflow, because those workloads have diverse sparse patterns even within one matrix, and prefer different schemes when processing different regions. Spada is able to dynamically adapt in these cases.

Overall, the WA-based Spada accelerator significantly improves the performance of SpGEMM across diverse sparse patterns, on average 38.04×, 1.44×, and 1.46× faster than SIGMA, SpArch, and GAMMA, respectively. On performance/area, Spada outperforms SpArch and GAMMA by 3.32× and 1.42×, respectively.

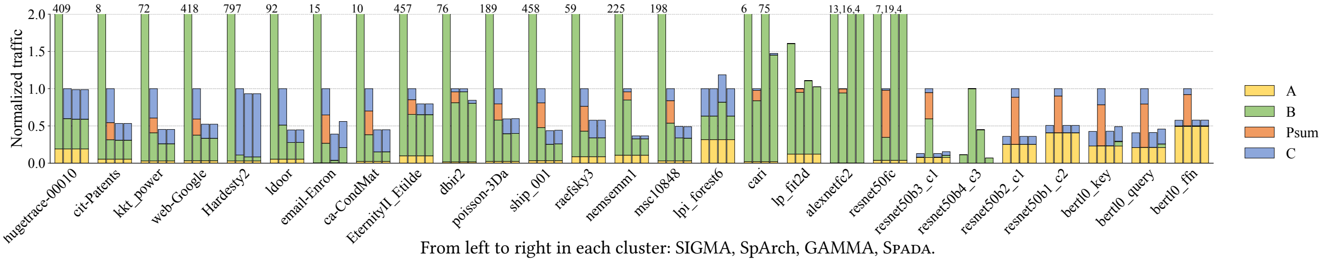
Figure 10 compares Spada with the CPU and GPU baselines. MKL is the slowest in almost all workloads despite using a high-end CPU. In contrast, cuSPARSE is faster than MKL with an average 2.84× speedup. Finally, Spada achieves 12.52 GFLOPS on average, 13.10× and 4.61× faster than the CPU and the GPU, respectively. Note that the GPU is able to slightly outperform Spada on a few workloads, e.g., `raefsky3`, but this performance gain is at the cost of much higher power and area.

7.3 Detailed Analysis

We next explain the high performance gains of Spada. Figure 11 shows for the four accelerators on different workloads, the off-chip memory traffic of four types of data: **A**, **B**, psum, and **C**. Different workloads exhibit different dominant memory traffic types. Since OutP used by SpArch has better input reuse and ROW used by GAMMA mostly reuses outputs (Figure 2), GAMMA performs better on workloads dominated by psum and **C** traffic, e.g., `ca-CondMat` and `email-Enron`, while SpArch performs better on workloads dominated by **B** traffic, e.g., `cari` and `resnet50fc`. In some cases like `nemsemm1` and `resnetb4_c3`, though **B** traffic dominates, GAMMA still has lower **B** traffic than SpArch. This is because the sparse patterns provide few **B** reuse chances only in small local regions, in which OutP fetches too many **B** rows and thus thrashes the cache. InP-based SIGMA has good output reuse, but its input memory traffic is huge on very sparse workloads, due to significant index intersection inefficiency. Spada is capable of better balancing both types of reuse and thus achieves similarly low total memory traffic to the best of the three baselines. Overall, Spada on average saves about 21.9× memory traffic over SIGMA, 1.69× over SpArch, and 1.39× over GAMMA.

Figure 12 further compares the multiplier utilization among the four accelerators. Overall, Spada has multiplier utilization on average 50.7×, 1.41×, and 1.42× better than that of SIGMA, SpArch, and GAMMA, respectively. The multiplier utilization improvement closely correlates to the traffic reduction in Figure 11 as most workloads are memory-bound. In some cases like `poisson-3Da`, Spada has slightly higher multiplier utilization than GAMMA, though the two have similar traffic. This is because due to the variation of local sparse patterns, some parts of the workloads are actually compute-bound if the data are well reused from the cache. To better understand the underlying reasons, we also include two hypothetical designs in Figure 12, Spada with an ideal memory (infinite bandwidth and zero latency), and Spada with an ideal memory and no pipeline stalls. We see that an ideal memory brings on average 36% utilization improvements and an ideal pipeline brings another 13%, validating that the utilization is mostly hindered by memory stalls. A few workloads suffer from significant pipeline stalls: `EternityII_Etilde` has short **B** rows that cause frequent pass switching; `lpi_forest6` is a small matrix on which pipeline draining/filling dominates.

The remaining loss on multiplier utilization of Spada when excluding memory and pipeline stalls can be attributed mainly to load imbalance among multiplier lanes. This loss is only 12% on average, indicating that Spada has a reasonable load balance



From left to right in each cluster: SIGMA, SpArch, GAMMA, Spada.

Figure 11: Memory traffic comparison among SIGMA, SpArch, GAMMA, and Spada.

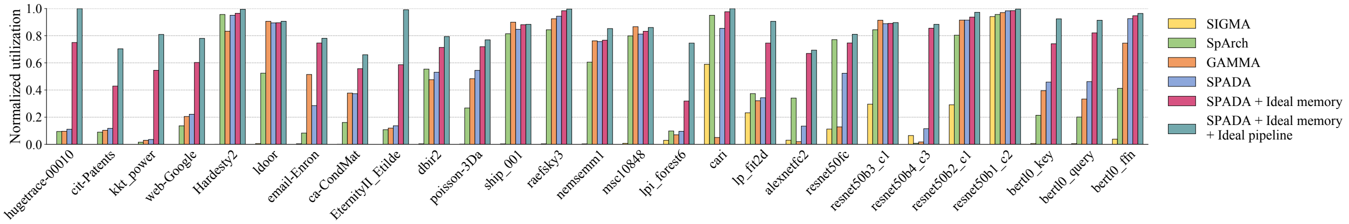


Figure 12: Multiplier utilization comparison among SIGMA, SpArch, GAMMA, Spada.

behavior with the help of the sort arrays. Among all workloads, `ca-CondMat` has the biggest remaining utilization loss. Its average **A** row length is around 9, only a little larger than the number of lanes, 8. Each pass thus includes two windows with 8 and 1 nonzeros, respectively, where the hardware resources are severely underutilized during the second window.

To illustrate the effectiveness of our window adaption algorithm in Section 5, Figure 13 compares the performance of Spada with static WA schemes, where we fix the window height at 1, 2, 4, and 8. Different window heights represent different tradeoff points between the reuse of **B** and **C**. Again due to the diverse sparse patterns, `ca-CondMat`, `dbir2`, and `resnetb4_c3` prefer shorter windows, while `raefsky3`, `cari`, and `lpi_forest6` prefer taller windows. In many cases like `Hardesty2`, `email-Enron`, and `resnet50fc`, the performance even varies non-monotonically. Spada is able to adapt to the preferred window shapes and achieves close to the best static or even better performance, by exploiting index distribution similarities. The profiling-based approach additionally helps Spada to quickly converge, even in non-monotonic cases.

7.4 Sensitivity and Scalability Studies

Table 4 conducts sensitivity studies on several key configuration parameters in Spada. We first investigate different configurations between the numbers of MPEs and lanes per MPE. Having more lanes in an MPE enlarges the window size of WA and thus its flexibility, but increases the overheads of merge/reduction at the end of the pipeline and also decreases the lane utilization. Our default setting of two 8-lane MPEs achieves a good tradeoff. Second, inside an MPE, we explore the detailed P queue design, including its length and the maximum number of elements popped out per cycle. Longer P queues can tolerate more index distribution imbalance and eliminate stalls, at the cost of additional area. A P queue of length 8 is sufficient. Supporting more elements popped out per cycle improves the throughput because it quickly drains the queue after a stall, but also requires larger bitonic networks and reduction trees. We find a design with maximum 2 pops per cycle is reasonable. Third, for window adoption, we empirically find 128 as a good threshold for categorizing large and small bands. Finally, we evaluate the effectiveness of the sort array, which brings an average 9% speedup.

Table 4: Sensitivity studies of Spada configurations.

Configurations		(Default)	
# MPEs × # lanes Performance vs. default	4 × 4 0.92	2 × 8 1.00	1 × 16 0.72
P queue length Performance vs. default	4 0.83	8 1.00	16 1.01
P queue max pops/cycle Performance vs. default	1 0.72	2 1.00	4 1.02
Large/small band threshold Performance vs. default	64 0.79	128 1.00	256 0.84
Sort arrays Performance vs. default	Without 0.91	With 1.00	

Figure 14 scales up the four accelerators and evaluates their performance at different scales, from 16 multipliers with 1.5 MB of SRAM, to 128 multipliers with 12 MB. All accelerators are scalable as the available hardware resource increases, while Spada keeps achieving the best performance at all scales. The speedup of SpArch from 64 multipliers to 128 is higher than that from 32 to 64. This is because at the largest configuration the dominant psum data start to fit in the on-chip cache. SIGMA actually has the highest scaling factor among the four, but the absolute performance is quite low due to the inefficiencies described before.

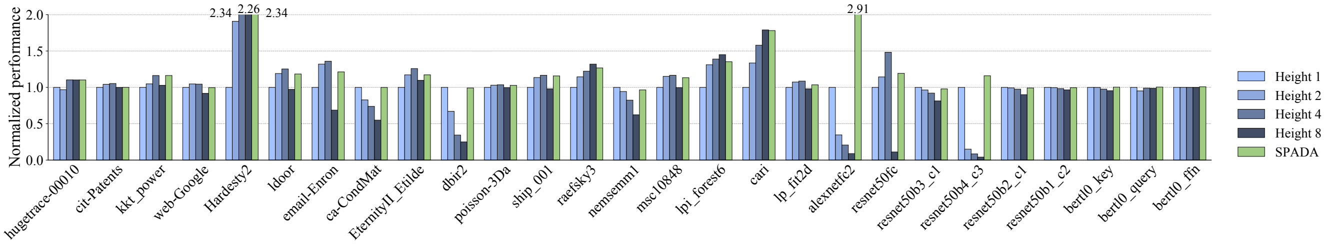


Figure 13: Performance comparison between static window heights and Spada dynamic adaption.

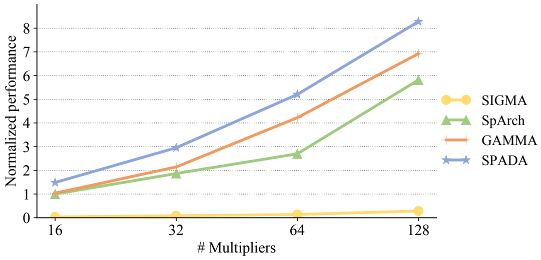


Figure 14: Scalability of SIGMA, SpArch, GAMMA, and Spada. Both the number of multipliers and the SRAM cache capacity are scaled together, from 16 multipliers, 1.5 MB to 128 multipliers, 12 MB.

8 OTHER RELATED WORK

Performance optimization of SpGEMM has been well studied on CPUs and GPUs [17, 23, 26, 38]. For example, Intel MKL [17] is the most commonly used library for SpGEMM on their CPUs, and cuSPARSE [26] is widely adopted on GPUs. Applying WA in software on CPUs and GPUs may enjoy some but not all the benefits of using specialized hardware. Software-based profiling and monitoring for adaptation may incur non-negligible overheads; thread initialization and load balancing are expensive; and general-purpose caches lack optimizations for sparse rows.

Recent work has used domain-specific hardware to accelerate SpGEMM in the machine learning, graph analysis, and linear algebra domains [15, 27, 31, 36, 42, 44]. Different from Spada, most of these accelerators supported only a fixed dataflow to match their specific target applications. For example, as categorized in Section 2.2, ExTensor [15] and SIGMA [31] used InP, OuterSPACE [27] and SpArch [44] used OutP, and MatRaptor [36] and GAMMA [42] used ROW. Another army of accelerators [1, 29, 37, 43, 45, 46] targeted specific SpGEMM in compressed deep learning models. Some of them were designed for unstructured SpGEMM [1, 29, 43], which were agnostic to nonzero value distribution and skipped unnecessary computations dynamically. For example, SCNN [29] skipped computations when either model weights or feature map values are zero. On the other hand, structured sparse accelerators [37, 45, 46] required regular sparse patterns generated by the co-designed pruning algorithms. For example, Cambricon-S [45] relied on a specially designed pruning algorithm and an encoding format to improve over its unstructured predecessor, Cambricon-X [43].

Nevertheless, few prior designs have considered different sparsity levels or sparsity distribution diversity within and across workloads. In contrast, Spada is able to adapt to the sparse pattern and reconfigure the hardware accordingly. STICKER [41] was designed to be aware of the sparsity distribution and encoded matrices into different sparse formats. It mostly focused on the metadata benefits of different encoding formats, and restricted to deep learning models without considering the much sparser scientific computing datasets. Furthermore, it did not exploit the data reuse tradeoffs provided by different sparse patterns for higher performance.

9 CONCLUSIONS

We proposed Spada, a hardware-software co-design to accelerate SpGEMM through adaptive reconfiguration based on the diverse sparse patterns across different scientific computing and deep learning applications. The key innovations of Spada include a window-based adaptive dataflow template, an efficient and reconfigurable hardware architecture to realize data reuse benefits under different data distributions, and an effective yet simple dynamic window shape adaption algorithm to determine the best configuration to use. Spada is able to match or exceed the performance of three optimized baseline designs and achieves 1.44× speedups on average.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by an Alibaba Innovative Research (AIR) program and the National Natural Science Foundation of China (62072262). Mingyu Gao is the corresponding author.

REFERENCES

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. 1–13.

[2] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017).

[3] Gianfranco Bilardi and Alexandru Nicolau. 1989. Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines. *SIAM J. Comput.* 18, 2 (1989), 216–228.

[4] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*. 367–379.

[5] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2020. Universal Deep Neural Network Compression. *IEEE Journal of Selected Topics in Signal Processing* 14, 4 (2020), 715–726.

[6] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and Programmability. *IEEE Micro* 38, 2 (2018), 42–52.

[7] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[8] Mr Hamid Faruqee, Mr Peter Isard, Mr Douglas Laxton, Mr Eswar Prasad, and Mr Bart Turtelboom. 1998. *Multimod Mark III: The Core Dynamic And Steady State Model*. International Monetary Fund.

[9] Denis Foley and John Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17.

[10] Giulia Galli. 1996. Linear Scaling Methods for Electronic Structure Calculations and Quantum Molecular Dynamics Simulations. *Current Opinion in Solid State and Materials Science* 1, 6 (1996), 864–874.

[11] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 751–764.

[12] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 807–820.

[13] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*.

[14] 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 (CVPR)*. 770–778.

[15] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 319–333.

[16] Torsten Hoefler and Marc Snir. 2011. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In *Proceedings of the International Conference on Supercomputing (ICS)*. 75–84.

[17] Intel. 2022. *Accelerate Fast Math with Intel ONEAPI Math Kernel Library*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[18] Satoshi Itoh, Pablo Ordejón, and Richard M Martin. 1995. Order-N Tight-Binding Molecular Dynamics on Parallel Computers. *Computer Physics Communications* 88, 2 (1995), 173–185.

[19] 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, Daniel Killebrew, 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. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12.

[20] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-Serial Deep Neural Network Computing. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NeurIPS)*. 1097–1105.

[22] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. 2018. UNPU: An Energy-Efficient Deep Neural Network Accelerator with Fully Variable Weight Bit Precision. *IEEE Journal of Solid-State Circuits (JSSC)* 54, 1 (2018), 173–185.

[23] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*. 370–381.

[24] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA)*. 553–564.

[25] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 5058–5066.

[26] NVIDIA. 2022. *cuSPARSE :: CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/cusparse/index.html

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

[28] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *Proceedings of the 2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. 304–315.

[29] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. 27–40.

[30] Cosmin G Petra, Olaf Schenk, Miles Lubin, and Klaus Gärtner. 2014. An Augmented Incomplete Factorization Approach for Computing the Schur Complement in Stochastic Optimization. *SIAM Journal on Scientific Computing* 36, 2 (2014), C139–C162.

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

[32] Victor Sanh, Thomas Wolf, and Alexander M. Rush. 2020. Movement Pruning: Adaptive Sparsity by Fine-Tuning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS)*. 20378–20389.

[33] Korey Sewell, Ronald G. Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F. Wenisch, Dennis Sylvester, David Blaauw, and Trevor Mudge. 2012. Swizzle-Switch Networks for Many-Core Systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2, 2 (2012), 278–294.

[34] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. 764–775.

[35] Julian Shun and Kanat Tangwongsan. 2015. Multicore Triangle Computations without Tuning. In *Proceedings of the 31st International Conference on Data Engineering (ICDE)*. 149–160.

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

[37] Zhanhong Tan, Jiebo Song, Xiaolong Ma, Sia-Huat Tan, Hongyang Chen, Yuanqing Miao, Yifu Wu, Shaokai Ye, Yanzhi Wang, Dehui Li, and Kaisheng Ma. 2020. PCNN: Pattern-Based Fine-Grained Regular Pruning Towards Optimizing CNN Accelerators. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[38] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An Input-Aware Auto-Tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*. 94–105.

[39] Ichitaro Yamazaki and Xiaoye S Li. 2010. On Techniques to Improve Robustness and Scalability of A Parallel Hybrid Linear Solver. In *Proceedings of the International Conference on High Performance Computing for Computational Science*. 421–434.

[40] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 369–383.

[41] Zhe Yuan, Yongpan Liu, Jinshan Yue, Yixiong Yang, Jingyu Wang, Xiaoyu Feng, Jian Zhao, Xueqing Li, and Huazhong Yang. 2019. STICKER: An Energy-Efficient Multi-Sparsity Compatible Accelerator for Convolutional Neural Networks in 65-nm CMOS. *IEEE Journal of Solid-State Circuits (JSSC)* 55, 2 (2019), 465–477.

[42] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 687–701.

[43] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An Accelerator for Sparse Neural Networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[44] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *Proceedings of the*

2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). 261–274.

[45] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28.

[46] 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 (MICRO)*. 359–371.

[47] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. 2019. Neural Network Distiller: A Python Package For DNN Compression Research. *arXiv preprint arXiv:1910.12232* (2019).

Received 2022-07-07; accepted 2022-09-22