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 scienti c and deep learning applications. The highly irregular structures of SpGEMM limit its performance and e ciency on conventional computation platforms, and thus motivate a large body of specialized hardware designs. Existing SpGEMM accelerators only support speci c types of rigid execution data ow such as inner/output-product or row-based schemes. Each data ow 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 e ciently adapt to various sparse patterns. First, we describe a window-based adaptive data ow that can be exibly adapted to di erent modes to best match the data distributions and realize di erent reuse bene ts. Then, our hardware architecture e ciently supports this data ow template, with exible, fast, and low-cost recon gurability and effective load balancing features. Finally, we use a pro ling-guided approach to detect the sparse pattern and determine the optimized data ow 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 data ow mismatch. It achieves an average 1.4× speedup across a wide range of sparse matrices and compressed neural network models.

This work is licensed under a Creative Commons Attribution 4.0 International License.

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, data ow

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 Data ow. 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 scienti c computation elds, 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 e ectively eliminate all ine ectual 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 e ciency of SpGEMM.

However, actually realizing such high-performance and e cient SpGEMM processing is known to be di cult due to the highly irregular structures, which lead to low utilization of both computation resource and memory bandwidth. As Dennard scaling ends and domain-speci c acceleration becomes an increasingly appealing approach to speed up challenging computation tasks, SpGEMM has

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 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 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 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 inefficient 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.

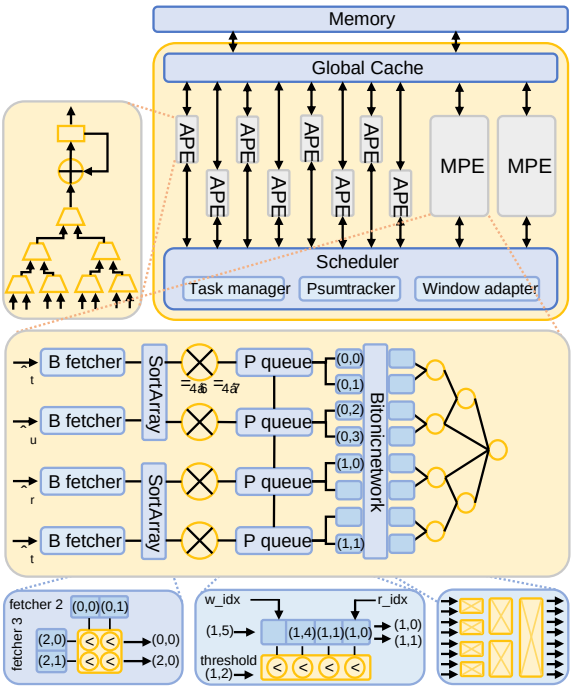

Figure 4: Spada hardware architecture. Top right: overall design. Top left: addition PE (APE). Middle: multiplication PE (MPE). Bottom left: sort array. Bottom middle: P queue. Bottom right: bitonic network.

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].

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 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 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 $c_0^{"0"}$ and $c_1^{"0"}$.

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

The P queue (P for psum) in each lane bu ers 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 rst 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 gure 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 nd 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() ≥ 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 di erent 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 exible 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 rst 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 nally use a exible 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 nal stage as the eventual result $c_m^{"t"}$ 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 lling 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 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 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 tradeo between performance and cost. Each sort array needs „2^2" 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

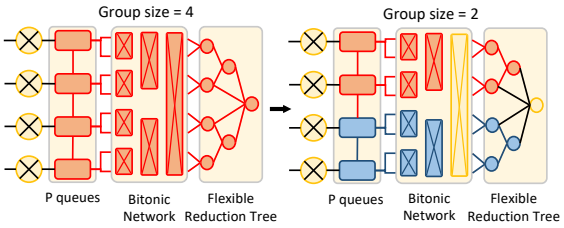

Figure 5: MPE reconfiguration, which only affects the PE queues, the bitonic network, and the flexible reduction tree.

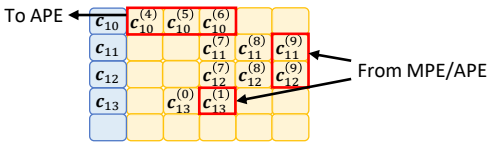

Figure 6: Psum tracker table structure in Spada scheduler.

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 PE 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 PE 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 PE 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 PE 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 PE queue, until all the elements of the previous pass completely leave all the PE 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 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 $R = 8$.

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 data flow 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 c_m, and contains a list of addresses for the psum rows $c_m^{(t)}$ 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 $\mathbf{A}$, where numbers are the column indices of $\mathbf{A}$ nonzeros, which are equal to the indices of $\mathbf{B}$ rows that are multiplied and also cached. The global cache can store eight $\mathbf{B}$ rows. Currently processing $\mathbf{A}$ rows or windows are highlighted in red boxes.

On average, the number of table entries should cover all r_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 $\mathbf{B}$ and psum rows $\boldsymbol{c}_m^{"t"}$ for $\mathbf{C}$. 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 data flow modes.

To better support WA, we also apply a small optimization on the cache replacement policy. In the original ROW data flow, 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 $\mathbf{A}$ row remain in the cache and can be reused by the current row, while those fetched by the older $\mathbf{A}$ rows are evicted. However, with WA that processes multiple $\mathbf{A}$ 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 $\mathbf{B}$ rows could be reused (Figure 7 top right).

To improve reuse, we use a RidxLRU policy, which associates each $\mathbf{B}$ row in the global cache with the most recent $\mathbf{A}$ row index that used it. When replacing $\mathbf{B}$ rows with smaller $\mathbf{A}$ 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 $\mathbf{B}$ rows fetched by the previous $\mathbf{A}$ rows.

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.

5 WINDOW SHAPE ADAPTION

As discussed in Section 3, the WA window height α determines the input reuse of $\mathbf{B}$ rows across multiple $\mathbf{A}$ 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 $\mathbf{B}$ rows highly depends on the index distribution between $\mathbf{A}$ rows. If neighboring $\mathbf{A}$ rows have many overlapped column indices, these nonzeros could reuse the same $\mathbf{B}$ rows. But such information cannot be extracted without actually scanning through each CSR row of $\mathbf{A}$, 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

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

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

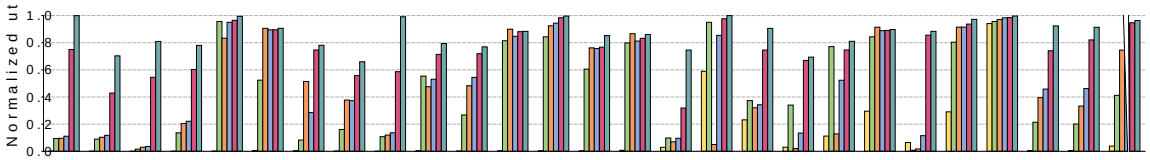

Figure 12: Multiplier utilization comparison among SIGMA, SpArch, GAMMA, Spada.

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.

Table 4: Sensitivity studies of Spada configurations.

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

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 PE queue design, including its length and the maximum number of elements popped out per cycle. Longer PE queues can tolerate more index distribution imbalance and eliminate stalls, at the cost of additional area. A PE 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.

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.

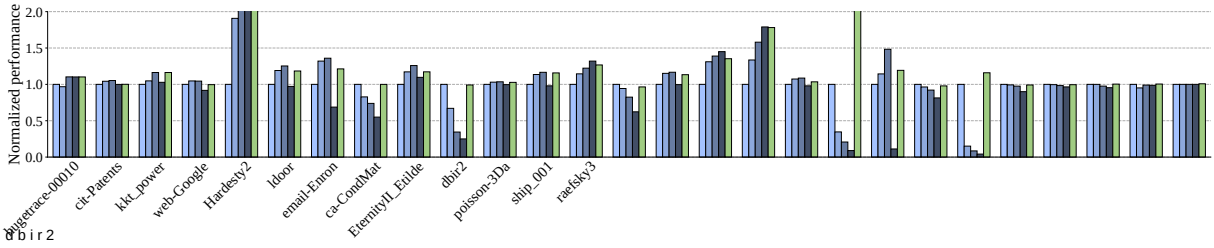

Figure 13: Performance comparison between static window heights and Spada dynamic adaption.

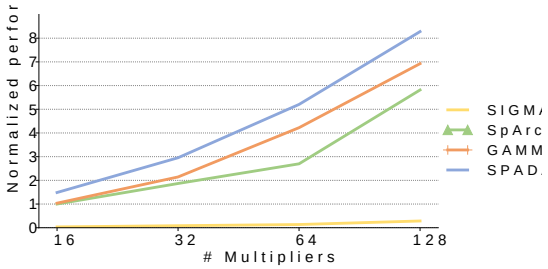

Figure 14: Scalability of SIGMA, SpArch, GAMMA, and Spada. Both the number of multipliers and the SRAM cache capacity are scaled together, from 16 multipliers, 1.5 MB to 128 multipliers, 12 MB.

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 data flow 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 [35] 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 data flow 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 '16). 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 Data flow for Convolutional Neural Networks. In Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA '16). 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 Data flow 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, Geo rey 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-51). 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-52). 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