2026 Design, Automation & Test in Europe Conference (DATE 2026)

RISC-V ISA Extensions for Vectorized Unstructured Sparse SpMM in LLM Inference

Tengfei Xia[1,2], Zhihua Fan[1,*], Jing Xue[1,2], Shantian Qin[1,2], Xiaochun Ye[1,2], Wenming Li[1,2],
[1]State Key Lab of Processors, Institute of Computing Technology, CAS, Beijing, China
[2]School of Computer Science and Technology, UCAS, Beijing, China
{xiatengfei24s, fanzhihua, xuejing25e, qinshantian23s, yexiaochun, liwenming}@ict.ac.cn

***Abstract*—Unstructured sparsity has emerged as a key enabler for pruning large language models (LLMs) while preserving accuracy. However, its highly irregular pattern makes it notoriously difficult to accelerate, creating severe bottlenecks in vectorization and memory access that prevent efficient deployment on edge hardware with tight power and area constraints. We present SCG, a vectorizable sparse matrix format designed to unlock high-performance unstructured sparse matrix–matrix multiplication (SpMM), the dominant kernel in LLM feed-forward networks and Q/K/V/O projections. To exploit SCG, we introduce custom RISC-V instructions and extend the BOOM processor with two lightweight pipelines for efficient parallel execution. This format–instruction–hardware co-design directly addresses the long-standing challenge of unstructured sparse acceleration in general-purpose processors. On real LLM workloads, our design achieves 11.9×, 12.7×, and 13.4× average speedups over baseline BOOM on LLaMA2-7B, OPT-1.3B, and TinyLLaMA-1.1B, respectively, with negligible hardware overhead. Compared to state-of-the-art sparse accelerators, it delivers up to 1.72× higher area efficiency.**

***Index Terms*—LLM, RISC-V, Unstructured Sparse Pattern**

I. Introduction

Large language models such as GPT and LLaMA [1, 2] have demonstrated remarkable capabilities across AI tasks but pose substantial computational and storage challenges, especially for edge deployment. Model pruning [3–5] mitigates this burden by removing redundant weights, introducing sparsity in linear layers, and converting the core kernel from dense GEMM to sparse matrix–dense matrix multiplication (SpMM).

Sparsity can be broadly categorized into structured and unstructured types. Structured sparsity enforces regular patterns (e.g., N:M), which are vectorization-friendly but may degrade accuracy. In contrast, unstructured sparsity has no fixed pattern: zeros can appear arbitrarily across the matrix, offering fine-grained flexibility and often preserving model accuracy. However, its irregular layout breaks vectorization, causes inefficient memory access, and severely limits parallelism, creating major obstacles for general-purpose processors and accelerators.

Existing sparse accelerators [6–9] attempt to handle this irregularity by stacking abundant compute and memory, yet under unstructured sparsity, they achieve low utilization and high overhead, making them unsuitable for edge deployment. This motivates lightweight, programmable processor extensions that can efficiently handle unstructured sparsity.

* Corresponding author: Zhihua Fan

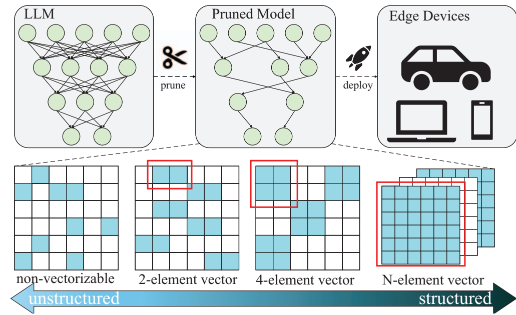

Fig. 1: Pruning in LLMs and the comparison of structured and unstructured sparsity.

On the other hand, inference of large models involves massive computation with a substantial amount of sparse matrix operation, where traditional general-purpose processors exhibit low execution efficiency. Therefore, efficiently supporting sparse computation of large models on edge devices urgently requires lightweight, programmable, and highly resource-reusable processor solutions. In this context, RISC-V emerges as a compelling choice due to its open-source nature, modular design, and support for custom instruction, which collectively enable flexible trade-offs between performance and resource footprint. Prior studies [10, 11] have focused on using RISC-V to address structured sparsity issue with promising results, whereas efficient acceleration of unstructured sparsity remains underexplored.

To overcome these challenges, we present Shift-Compaction Grouping (SCG), a novel vectorizable sparse format for unstructured SpMM, together with custom RISC-V instructions and lightweight BOOM pipeline extensions. This format–instruction–hardware co-design bridges the gap between accuracy-preserving unstructured sparsity and efficient execution on general-purpose processors.

The main contributions of this paper are as following:

- We propose a novel sparse matrix format, Shift-Compaction Grouping. It reorganizes nonzeros into groups matching SpMM outer-product access patterns, enabling continuous memory access and vectorization.
- We design a set of custom RISC-V instructions tailored to SCG, providing efficient memory and compute support for sparse kernels.
- We extend lightweight pipeline extensions for partial sum generation and merging, overlapping execution and

978-3-9826741-1-7/DATE26/© 2026 EDAA

reducing intermediate storage.

- Experimental results show that, compared with the original BOOM, our design achieves up to 16.2× performance speedup. Compared with the state-of-the-art, it also delivers up to 1.72× improvement in area efficiency, while incurring only minimal area and power overhead, making it a promising solution for edge inference of LLMs.

II. Background and Motivation

Insight-1: Outer product makes better use of locality in hierarchical memory architectures. Sparse Matrix–Dense Matrix Multiplication (SpMM) multiplies a sparse matrix by a dense matrix. In LLMs, SpMM is the main kernel for FFNs and Q/K/V/O projections, and it is the primary performance bottleneck especially when the sequence length is relatively short [12]. As shown in Fig. 2, each iteration of the inner-product produces a single output element and consumes each input only once, forcing cross-row gathers from B (e.g., $e \to g$ and $f \to h$) that hurt spatial locality and stress caches as B's row dimension grows. By contrast, the outer-product reuses each input multiple times per iteration (e.g., a is reused) and accesses B contiguously in row-major order (e.g., $e \to f$ and $g \to h$). This advantage becomes particularly critical in LLMs, where the dense matrices have relatively large size.

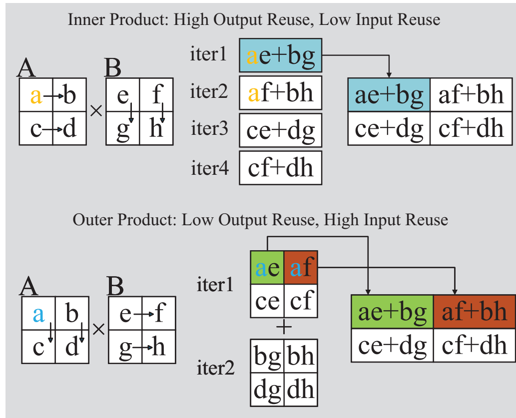

Fig. 2: Comparisons between two different algorithms

Insight-2: LLM matrix scales concentrate in the outer-product sweet spot. We conduct extensive experiments on inner-product versus outer-product SpMM in pruned large-model linear layers. As shown in Fig. 3, red points indicate configurations where the inner-product runs faster, and the boundary of its advantage along the K dimension forms a red plane. The matrix sizes typical of large-model SpMM lie within the green box, where the outer-product implementation is clearly faster. This observation helps explain why many accelerators for large-model matrix operations adopt outer-product–based kernels. For example, SpArch [8] employs hardware-based sparse matrix compression together with a Huffman-tree scheduler and a row-level prefetcher to accelerate SpMM. Spada [6] adopts a reconfigurable architecture, where the hardware organization is determined in a profiling-guided manner. It leverages a bitonic network to efficiently

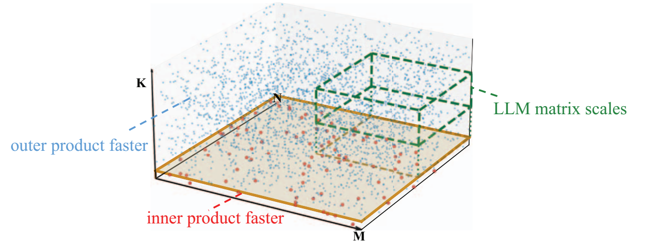

Fig. 3: Performance comparison of inner and outer product

perform partial sum accumulation. Although the aforementioned designs achieve high performance, they rely on heavily stacked compute and storage resources. However, they suffer from poor resource utilization when handling matrices of varying sizes, resulting in substantial energy overhead [13]. In edge scenarios, where both resource availability and power consumption are strictly constrained, these architectures lose their advantages. Therefore, there is an urgent need for edge-end solutions that offer low power consumption and minimal resource usage while still delivering acceptable performance. The extensibility of the RISC-V ISA offers a promising direction for our design.

Insight-3: Traditional sparse formats are not friendly to unstructured sparse vectorization. Numerous sparse formats have been proposed to reduce storage and accelerate sparse computation, each with its own advantages and limitations. COO explicitly stores the coordinates of each nonzero element, which leads to excessive storage overhead and random memory accesses. CSR organizes nonzeros in row order with a row pointer array, making it efficient for inner-product traversals, but its row-oriented layout fundamentally mismatches the column-oriented access required by outer-product algorithms. CSC mirrors CSR along columns and is better aligned with outer-product traversal, yet the irregular and highly skewed number of nonzeros per column severely hampers vectorized execution. ELL enforces equal-length rows to enable regular vectorization and simplified indexing, but under unstructured sparsity this results in massive padding overhead and wasted computation. Collectively, these formats highlight a common tension: optimizing for storage efficiency often sacrifices regularity needed for vectorization, while enforcing regularity incurs prohibitive overhead under unstructured sparsity.

III. Our Design

A. SCG: Vectorizable Sparse Matrix format

Unlike traditional sparse formats, SCG is tailored to the memory access patterns of the SpMM outer-product algorithm. It compacts nonzeros within rows and aggregates across columns, producing a structured, contiguous, and compact layout that significantly enhances memory locality and vectorization efficiency.

The SCG format first compacts nonzeros within each row, preserving their column indices required for outer-product SpMM. Compacted elements from different rows are then aggregated upward along columns to form a Group, representing a complete or partial compressed column. Within each Group,

Fig. 4: Illustrate of SCG format

nonzero values are stored sequentially in a val array in column-major order, and their original column indices are stored in a col_idx array to index dense matrix rows. To recover the original row of each nonzero, a two-level metadata structure is employed:

- **group_nnz**: records the number of nonzeros in each group. If a group is full (i.e., its size equals the group height), a special marker -1 is used. The original row index of a nonzero can then be inferred from its position within the group and the group number, eliminating the need for explicit row indices.
- **group_idx**: used only for non-full groups (i.e., **group_nnz** $\neq -1$) to explicitly store the original row index of each nonzero element.

As shown in Fig. 4, the first column of the unstructured sparse matrix A contains three nonzeros: {e, l, p}. For offline-pruned weights, the matrix can be preprocessed into SCG format. A compressed column{a, c, e, g, j, l, n, p} is temporarily treated as a single group; in practice, a column may span multiple groups to match the vector register width. Similarly, the second group {b, d, f, h, k, m, o, q} (green) and the third group {i, r} (black) are generated. Nonzeros are stored sequentially in the val array, with original column indices in col_idx. Full groups (first and second) have group_nnz = -1, allowing original row indices to be inferred from element positions. For example, in the first group, element g at position 3 maps to the 3rd row of the partial sum matrix. The third group is non-full (group_nnz = 2), so element i at position 0 has its original row explicitly stored in group_idx.

The SCG format offers several advantages: ① Using the special marker **group_nnz** $\neq -1$, SCG preserves row positions without redundant storage, improving compression ratio and saving memory. ② Nonzeros from different original columns are merged into the same compressed column, reducing the number of intermediate partial sum matrices and overall storage requirements. ③ By packing nonzeros into column-wise compact groups, the outer-product algorithm can sequentially access the val and col_idx arrays, reducing cross-row accesses and improving cache efficiency. ④ Each group naturally forms a vector, enabling efficient use of vector extension instructions, reducing packing/unpacking overhead, and improving instruction throughput.

B. RISC-V Custom Instruction Extensions

To support the proposed SCG format and enable efficient SpMM computation, we propose a set of custom RISC-V extension instructions. We choose 0x0b as Opcode to implement custom instructions, and employ Funct3 to distinguish each instruction. The encoding of all custom instructions is summarized in Table I.

- **LDVALIDX** is specifically designed to load the nonzero values and their original column indices of the sparse matrix, compressed in SCG format, into vector registers. Since the value array and the column index array are stored contiguously in memory, the instruction only requires one input address.
- **VSMV** moves data from a specified position in a vector register to a scalar register (i.e., a general-purpose register). The aim of this instruction is to extract row indice for accessing the dense matrix from the vector register that holds the **col_idx** values.
- **LDPRF** is responsible for loading the row data from the dense matrix into a vector register, based on the column index extracted by **VSMV**. Since a full row of B is typically larger than the length of a vector register (VLEN), this instruction incorporates a prefetching mechanism. While loading the values in the range [addr, addr+VLEN), it simultaneously prefetches the segment [prf_addr, prf_addr+VLEN) into the cache. This allows subsequent accesses to retrieve data directly from the cache rather than fetching it from main memory.
- **VSMUL** is the primary computation instruction. The equivalent expression for this instruction is: $vdest[:] = vsrc0[index] \times vsrc1[:]$, where $index$ is the register placed in the Funct7 field. Semantically, it performs the operation of extracting a scalar element within a Group of sparse matrix, and multiplying it with a row vector from the dense matrix, generating a partial sum block.
- **STPS** stores the result generated by **VSMUL** from the vector register into a dedicated buffer for holding partial sum matrix blocks, facilitating subsequent merge.
- **MERGE** performs the accumulation of two partial sum blocks stored in the buffer and writes the combined result

TABLE I: The encoding of custom instructions

Instruction	Funct7[31:25]	Rs2[24:20]	Rs1[19:15]	Funct3[14:12]	Rd[11:7]	Opcode[6:0]
LDVALIDX	/	vector dest0	vector dest1	0	input addr	0x0b
VSMV	index	/	vector src0	1	scala dest	0x0b
LDPRF	/	prefetch addr	input addr	2	vector dest	0x0b
VSMUL	index	vector src1	vector src0	3	vector dest	0x0b
STPS	/	/	vector src	4	buffer addr	0x0b
MERGE	/	buffer addr1	buffer addr0	5	buffer addr2	0x0b
STRES	/	/	buffer addr	6	output addr	0x0b

back to the buffer, enabling progressive reduction of intermediate results.

- **STRES** writes a fully merged partial sum matrix block from the buffer back to main memory. The block size is implicitly determined by the vector length (VLEN), so no explicit size specification is required.

C. Hardware Design and Acceleration Strategy

The processor architecture is adapted to efficiently support our custom instructions. As shown in Fig. 5(b), the frontend pipeline is slightly modified with a decoder for the new instructions. The main change is in the backend, where two pipelines are extended via the RoCC interface [14] to integrate acceleration modules efficiently. The Vector Register File (VRF) temporarily holds input matrix data. The Partial Sum Unit (PSU) computes partial sum matrix blocks using only multipliers. Simple control logic handles memory requests and VRF data access. The Partial Sum Buffer (PSB) stores partial sums and intermediate merged results. The Merge Unit (MU) reads two partial sum blocks from the PSB, merges them using adders, and writes both intermediate and final results back to the PSB or main memory.

The PSU and MU occupy two separate extended pipelines, dedicated to partial sum generation and merging, respectively. Fig. 5(a) illustrates the SpMM implementation using our custom instructions. Corresponding to the pipelines, the code is divided into generation blocks (e.g., Gen P11) for computing partial sums, and merge blocks (e.g., Merge P11 and P21) for accumulation. In Gen P11, the LDVALIDX instruction loads a vector of nonzero values and their column indices from the SCG-formatted sparse matrix into vector registers vr0 and vr1. Column addresses are computed implicitly via simple arithmetic on the instruction-encoded base address. The VSMV instruction with index 0 extracts the 0-th element from vr1 into the general-purpose register r3, identifying the corresponding row of dense matrix B. This row index is then used by LDPRF to load the required element and prefetch subsequent data into cache, reducing future memory accesses. With operands ready, VSMUL performs scalar-vector multiplication to generate the partial sum block P_{11}, which is stored in the PSB using STPS. The generation of remaining partial sum blocks follows the same procedure and is omitted for brevity.

In a conventional sequential process, partial sum blocks are generated and accumulated serially, with total execution time equal to the sum of generation and merging time. In contrast, our dual-pipeline design (Fig. 5(d)) decouples generation and merging to enable overlapping execution. During initialization, STAGE1 generates P11 and STAGE2 generates P21. In subsequent stages, generation and merging proceed concurrently: for example, as STAGE3 generates P31, P11 and P21 are merged on the other pipeline. The enlarged view in Fig. 5(b) shows the extended backend, where three wiring styles denote data flows for the last, current, and next stages. While the PSU writes P31 into the PSB, the MU simultaneously reads P11 and P21, merges them, and writes the result back. The PSB acts as a staging buffer, storing at most four blocks at a time (e.g., P11, P21, P31, and $\mathrm{I} = \mathrm{Merge}(\mathrm{P11}, \mathrm{P21})$). In subsequent cycles, new blocks such as P41 and $\mathrm{J} = \mathrm{Merge}(\mathrm{P}_{31}, \mathrm{I})$ overwrite the slots of already consumed blocks to save storage overhead.

To enable overlap between generation and merging, the Merge P11 and P21 process is implemented as a single MERGE instruction. Superscalar processors exploit a limited dispatch window. If merging required multiple instructions, they could contend with subsequent generation instructions. Condensing the merge phase into one instruction increases the likelihood of ideal overlapped execution with, e.g., Gen P_{31}.

IV. Evaluation

A. Experiment Setup

Methodology: For performance evaluation, we construct a cycle-accurate gem5 simulator [15] based on the BOOM core [16] and extend it. The detailed configuration of the BOOM core is summarized in Table II. For power and area evaluation, we implement our hardware modules in RTL and integrate them into the BOOM core. We synthesize the extended core using a 12nm process, meeting timing at a 1GHz clock frequency consistent with that described in [14].

TABLE II: Simulated configurations of the extended BOOM core

Baseline	RV64GC, 4 ALUs, 2 FPUs; 8-way-issue out-of-order; 128 physical integer registers; 128 physical floating-point registers; 32KB 8-way L1D cache; 32-entry store buffer; 32KB 8-way L1I cache; 512KB 8-way L2 cache
Extensions	custom-instruction decoder; PSU with 8 multipliers; MU with 8 adders; 4×128-bit vector registers; 64B PSB (SPM)

Baseline: To evaluate the contribution of each component in our work, we perform a series of ablation studies. The baseline configuration uses the original BOOM core with the CSC format and no custom instruction extensions. To assess

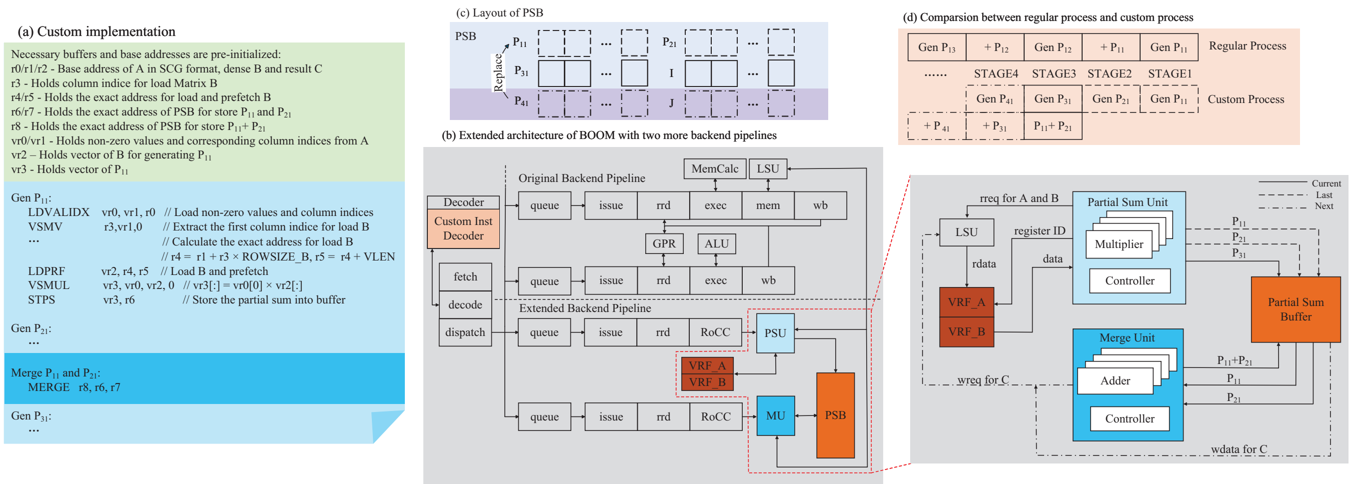

Fig. 5: Extended architecture of BOOM RISC-V processor

the efficiency of our proposed SCG format, we replace CSC with SCG while keeping the BOOM core and instruction set unchanged. Finally, we enable the full configuration, integrating the BOOM core with both the SCG format and our custom instructions, to further demonstrate the effectiveness of the proposed instructions and hardware extensions.

To highlight the advantages of our work, we also compare it with state-of-the-art accelerators, using Trapezoid [17], TPU [18] and Sigma [19]. These designs differ substantially in computational capability, target deployment scenarios, and optimization goals, making direct performance comparisons unfair. Accordingly, we focus on area efficiency (performance per area), which better reflects practical value and deployment potential in resource-constrained edge scenarios.

Benchmark: We evaluate our design using three representative transformer models: TinyLLaMA-1.1B, OPT-1.3B, and LLaMA2-7B. These models have relatively small parameter scales, making them common choices for edge deployment, and they are also frequently used in pruning-related research. The specific model configurations are summarized in Table III. Our evaluation mainly focuses on the SpMM kernels in the feed-forward networks and the Q/K/V/O projections, which dominate Transformer computation [12, 20]. For benchmarking, we employ the ShareGPT [21] dataset, a widely used corpus for chatbot evaluation, with an average input sequence length of 783 tokens. All models are then pruned using the advanced Wanda technique [4], with unstructured sparsity levels of 0.4, 0.5, and 0.6, allowing us to assess the performance of our acceleration strategy under realistic sparsity patterns while preserving model accuracy.

B. Experiment results

To demonstrate the effectiveness of our design, we conduct ablation studies to present the normalized speedup contributed by different components of the proposed system, as shown in Fig. 6. The custom vector register length (VLEN) is set to 8, corresponding to a capacity of eight FP16 elements, which

TABLE III: Model configurations used for evaluation

Model	Layer	Kernel	Shape
LLaMA2-7B	1/16/32	Up proj.	11008×4096
		Down proj.	4096×11008
		Q/K/V/O proj.	4096×4096
OPT-1.3B	1/12/24	Up proj.	8192×2048
		Down proj.	2048×8192
		Q/K/V/O proj.	2048×2048
TinyLLaMA-1.1B	1/11/22	Up proj.	5632×2048
		Down proj.	2048×5632
		Q/K/V/O proj.	2048×2048

is later shown to be the most energy efficient choice. The experimental results indicate that the incomplete system using only the SCG format achieves a modest average speedup of around 1.3× across all sparsity levels. Notably, the maximum speedup of 1.50× occurs at 0.5 sparsity in the Q/K/V/O projection of OPT-1.3B middle layer, demonstrating that even without custom instructions and hardware extensions, the SCG format alone provides tangible acceleration due to improved cache locality and reduced metadata.

In comparison, the complete configuration demonstrates substantial speedup across all layers and sparsity levels. For instance, the Up projection of TinyLLaMA-1.1B last layer at 0.6 sparsity achieves the highest speedup of 16.2×. Furthermore, similar speedups across sparsity levels indicate broad sparsity compatibility. With SCG, unstructured matrices at any sparsity are compacted into vectorizable dense arrays, so hardware resource utilization remains stable across sparsity levels.

These results substantiate the effectiveness of our design and highlight its strong scalability across models of varying sizes and sparsity. The acceleration mainly stems from: 1. novel SCG format, which groups non-zero elements and computes row indices implicitly, reducing memory footprint and improving cache locality. 2. vectorized custom instructions, which enable prefetching and parallel computation, thereby hiding memory access latency. 3. efficient hardware extensions, which overlap the generation and merge processes to reduce overall

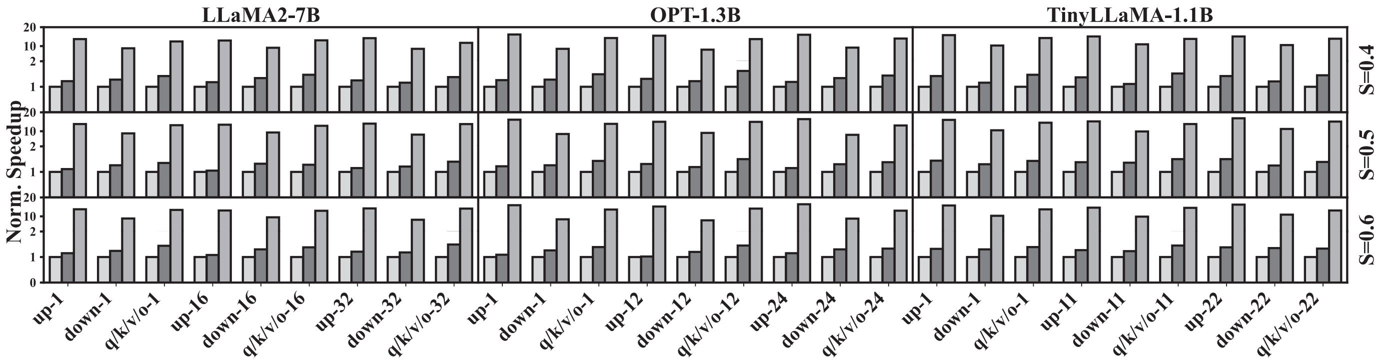

Fig. 6: Performance comparisons between baseline, baseline + SCG, and baseline + SCG + RISC-V extention

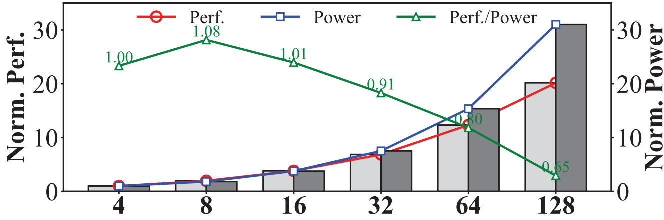

Fig. 7: Design space exploration of VLEN

latency. Overall, the combination of format, instruction, and hardware optimizations enables the system to achieve high, consistent speedups across multiple models and sparsity.

As shown in Fig. 7, we explore the effects of VLEN on performance and power. The number of multipliers and adders in both the PSU and MU are provisioned to equal VLEN, ensuring compute throughput matches the vector width. Increasing VLEN improves performance at the cost of higher power consumption, with VLEN = 8 reaching the best energy efficiency. Moreover, under VLEN scaling, the normalized energy efficiency remains above unity up to VLEN = 32. Beyond 32, the sharp increase in power consumption makes further VLEN scaling less cost-effective.

To demonstrate the efficiency of our design in edge scenarios, we conducted a comparative study of area efficiency against state-of-the-art accelerators, as shown in Fig. 8. The results indicate that at higher sparsity levels of 0.4, 0.5, 0.6, 0.7, and 0.8, our design consistently delivers the best area efficiency, averaging a 1.64× improvement over Trapezoid and up to 30% over Sigma. At a sparsity of 0.3, our design is nearly on par with the best-performing TPU. As expected and acceptable for near-dense workloads, at the low sparsity level of 0.2 our design does not show an advantage over the TPU but remains on par with Trapezoid.

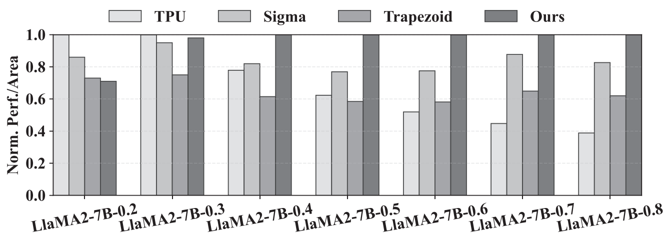

Fig. 8: Area efficiency comparison with SOTA works

TABLE IV: Area and power breakdown.

Component		Area (μm^2)	Power (mW)
BOOM (baseline)		270000	31.4
Extensions	PSU	1376	0.66
	MU	981	0.64
	PSB	750	0.13
	VRF	650	0.11
	Total	3757	1.54
Overhead		1.4%	4.9%

C. Hardware Overhead

The evaluation of our hardware design is performed in 12nm technology with the frequency of 1GHz, and it is confirmed that there is no decrease in frequency due to the extension of new hardware. The area and power overhead of the baseline BOOM and extra hardware (VLEN = 8) to support ISA extension is shown in Table IV. It can be observed that PSU and MU, the two computational units, constitute the majority of the overhead, accounting for 63% of the additional area and 84% of the additional power. By contrast, the temporary-storage components PSB and VRF impose only a small overhead under our dedicated design. In terms of total overhead, the increases of 1.4% in area and 4.9% in power are negligible compared with the performance improvements over the baseline BOOM core.

V. Conclusion

We present SCG, a vectorizable sparse format tailored to outer-product SpMM, along with custom RISC-V ISA extensions integrated into an out-of-order BOOM core. The design natively supports unstructured sparsity, addressing a gap in prior work focused on structured patterns, and delivers up to 16.2× speedup over baseline BOOM while surpassing prior solutions in area efficiency with minimal hardware cost. In future, we will extend SCG and its ISA to dynamic sparsity and scale the design across multi-core fabrics.

VI. Acknowledgment

This work was supported by National Key R&D Program of China (Grant No.2023YFB4503500)National Natural Science Foundation of China (Grant No.62502498), Beijing Natural Science Foundation (Grant No.L234078).

REFERENCES

[1] J. Achiam *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] H. Touvron *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[3] E. Frantar and D. Alistarh, "Sparsegpt: Massive language models can be accurately pruned in one-shot," in *International Conference on Machine Learning*, pp. 10323–10337, PMLR, 2023.

[4] M. Sun *et al.*, "A simple and effective pruning approach for large language models," *arXiv preprint arXiv:2306.11695*, 2023.

[5] P. Michel *et al.*, "Are sixteen heads really better than one?," *Advances in neural information processing systems*, vol. 32, 2019.

[6] Z. Li *et al.*, "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 2023, (NY, USA), p. 747–761, Association for Computing Machinery, 2023.

[7] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–40, 2017.

[8] Z. Zhang *et al.*, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274, 2020.

[9] G. a. o. Gerogiannis, "Spade: A flexible and scalable accelerator for spmm and sddmm," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, (NY, USA), Association for Computing Machinery, 2023.

[10] P. Vasireddy *et al.*, "Sparse-t: Hardware accelerator thread for unstructured sparse data processing," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–8, 2022.

[11] V. Titopoulos *et al.*, "Indexmac: A custom risc-v vector instruction to accelerate structured-sparse matrix multiplications," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2024.

[12] H. Wang *et al.*, "Sofa: A compute-memory optimized sparsity accelerator via cross-stage coordinated tiling," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1247–1263, 2024.

[13] A. Fuchs and D. Wentzlaff, "The accelerator wall: Limits of chip specialization," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–14, 2019.

[14] J. Zhao *et al.*, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, pp. 1–7, International Symposium on Computer Architecture Valencia, Spain, 2020.

[15] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[16] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu, and M. D. Wong, "Boom-explorer: Risc-v boom microarchitecture design space exploration framework," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2021.

[17] Y. Yang *et al.*, "Trapezoid: A versatile accelerator for dense and sparse matrix multiplications," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 931–945, 2024.

[18] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[19] E. Qin *et al.*, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, 2020.

[20] Y. Tay *et al.*, "Efficient transformers: A survey," 2022.

[21] ShareGPT Teams, "Sharegpt vicuna unfiltered." https://sharegpt.com/, 2023.