SpeedSparse: Exploiting Dense General Matrix Multiplication for Practical Sparse Neural Networks

Shreya Alladi, Alberto Ros, Alexandra Jimborean
University of Murcia
Murcia, Spain
shreya.a.s@um.es, aros@ditec.um.es, alexandra.jimborean@um.es

Abstract

SpeedSparse exploits semi-structured sparsity to provide the memory benefits of sparse representations and the performance of dense–dense BLIS kernels. It selectively avoids storing zeros in memory and performing computations on zeros while maintaining consecutive, vectorization- and tiling-friendly data layout. SpeedSparse achieves a 40% reduction in execution time and a 27% reduction in static memory usage on average compared to Intel's MKL dense–dense General Matrix Multiplication (GEMM) runtime for real-world Neural Networks (NN) and Large Language Models (LLMs).

1 Introduction

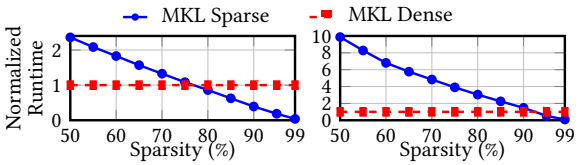
Transformers and LLMs have rapidly grown in size, increasing latency, memory usage, and power consumption. Sparsity, introduced through pruning, masked training, or sparse-weight generation, reduces memory usage while preserving model functionality.

However, different sparsity schemes, such as structured and unstructured sparsity, present distinct trade-offs. Unstructured pruning preserves accuracy but fails to take full advantage of modern hardware features such as vector units, memory buses, and data prefetchers, resulting in no inference speedups. Highly structured sparsity improves computational efficiency but at the cost of accuracy. Semi-structured sparsity provides a balance between these approaches by introducing regularity that can be efficiently exploited by hardware while maintaining relatively high accuracy [19].

Sparse formats [22] such as Coordinate Format (COO), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), ELLPACK (ELL), and Sliced ELLPACK (SELL) do not fully exploit the regularity introduced by semi-structured sparsity. This limits the performance benefits of semi-structured pruning techniques.

Prior approaches [11, 23] exploit semi-structured sparsity patterns by improving locality through clustering and matrix reordering, but rely on CSR/CSC representations leading to poor performance, particularly at moderate sparsity levels (from 40% to 60%) observed in real-world NN and LLMs [8, 10, 12, 15].

Figure 1 shows normalized execution time comparing Intel MKL's sparse(CSR)–dense GEMM kernels with its dense–dense kernels as baseline for varying sparsity levels as the baseline. Our analysis indicate that sparse(CSR)–dense GEMM kernels outperform dense–dense GEMM only at very high sparsity above 75% for 768×768 matrices (see Figure 1a) and above 90% for 4096×4096 matrices (see Figure 1b). At lower sparsity levels (< 75%), sparse formats typically yield no performance benefits due to indirect memory accesses, non-contiguous data layouts, reduced vectorization opportunities, and lack of efficient tiling strategies, leading to performance trade-offs prioritizing memory savings.



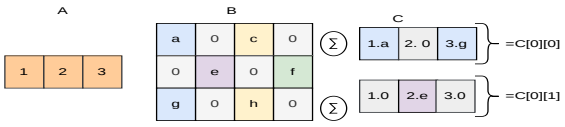
(a) 768×768 **matrix.** **(b)** 4096×4096 **matrix.**

Figure 1: MKL dense-sparse vs. dense-dense GEMM runtime

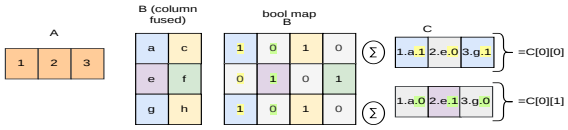
Hence, for real-world LLMs (from 40% to 80% sparsity) an alternative approach is to store weights in a compressed sparse format and decompress them into a dense representation at runtime (or on the fly) [12, 17, 18, 21]. This approach trades additional memory bandwidth and runtime decompression overhead for the higher computational efficiency of dense–dense matmul execution. However, the additional memory traffic required for decompression overhead limits its effectiveness, particularly for memory-bound operations such as matmul. Prior work Kung et al., [16] addresses this problem using a column-combining technique. However, this approach sacrifices accuracy by retaining only the largest-magnitude weight in each combined column group. Moreover, it requires modifications to the systolic array architecture to support column combining.

This work seeks to combine the memory savings of sparse representations with the performance benefits of optimized dense–dense GEMM implementations for *semi-structured sparsity* found in LLMs [8, 10, 12, 15], without runtime decompressionor modifications to the hardware architecture. We selectively eliminate zero-valued weight computations. At the same time, we preserve consecutive, vectorization- and tiling-friendly memory accesses. We leverage the robust infrastructure and highly optimized dense–dense GEMM algorithms provided by libraries such as Basic Linear Algebra Subprograms (BLAS) [13] and BLAS-like Algebra Subprograms (BLIS) [4].This approach relies on algorithmic modifications while preserving compatibility with conventional hardware architectures.

Our key insights are as follows: (i) For input matrices A, B and output matrix C ($C = A \times B$), multiple output elements in C are computed against the same row of A and different columns of B, (e.g., elements C_{00} and C_{01} both use row A_0 as illustrated in Figure 2a); (ii) For NN and LLM workloads, sparse(CSR)–dense GEMM approaches provide limited memory and no performance benefits. Consequently, a dense–dense GEMM approach is more efficient; (iii) At extremely high sparsity levels (e.g., 99% sparsity in a 4096×4096 matrix), the execution time of sparse(CSR)–dense GEMM becomes negligible compared with the dense–dense GEMM, as only a small fraction of the matrix elements are non-zero (nz); And (iv) two or more columns can be combined into a single column without loss of nz elements if the nz elements occur in different row positions, producing the disjoint columns (e.g., B_0 and B_1 in Figure 2a).



(a) Reuse of a single row of A in $C = A \times B$, illustrating the original matrix multiplication where the same row of A is reused with two different columns of B to compute $C[0][0]$ and $C[0][1]$.



(b) After SpeedSparse column fusion: B is the fused matrix formed by combining columns (1&2) and (3&4), with a boolean map B encoding the locations of the original elements. C illustrates the masked fma operation on the 0th column of the fused B matrix, using the boolean map to recover the correct results in $C[0][0]$ and $C[0][1]$.

Figure 2: GEMM before and after SpeedSparse column fusion

Our proposal, SpeedSparse, exploits our previous observations by fusing disjoint columns (or rows) to maximize reuse of the other matrices, as illustrated in Figure 2b. Fusion reduces matrix passes and the total number of required multiplications (see section 2.1.2), while still maintaining consecutive memory accesses and taking full advantage of BLIS tiling and micro-kernels. This is in contrast to prior approaches [11, 23], which rely on indirect memory accesses and cannot leverage efficient tiling.

Specifically, this work offers the following contributions: (i) A novel *column/ row fusion* technique (pre-processing technique) to reduce memory usage and execution time; (ii) A SpeedSparse-GEMM approach that reuses highly optimized BLIS algorithms; (iii) By fusing disjoint columns (or rows), SpeedSparse enables values loaded into vector registers to be reused across multiple output-element computations, improving register-level data reuse and reducing redundant memory accesses; And (iv) SpeedSparse selectively avoids storing zero values in memory and loading zero values into vector registers, thereby reducing unnecessary memory traffic; (v) Evaluation of the memory savings and runtime performance of SpeedSparse across different matrix dimensions and sparsity levels.

2 SpeedSparse

SpeedSparse is designed to reduce memory usage and enable faster and more efficient execution of sparse–dense GEMM, thereby improving inference performance. It consists of two key components: (i) A *weight matrix fusion* technique that compresses the sparse weight matrix during compile-time preprocessing and (ii) an optimized *micro-kernel* algorithm for efficient sparse–dense GEMM that leverages the compression scheme. Based on the sparsity patterns of the weight matrix, we classify the fusion process into two categories: (i) *Truly Complementary Matrix (TCM)*, and (ii) *Non-TCM with residuals*. We first describe the compressed format and then present the corresponding algorithm for each case.

2.1 Case 1: TCM ($weight_matrix_{TCM}$)

A TCM is a matrix in which every column (or row) belongs to at least one group of mutually disjoint columns (or rows) called Maximal Fusion Group (MFG). This is the case of matrix B in Figure 2a.

2.1.1 TCM fusion. Leveraging the TCM structure, we begin by identifying the most favorable fusion pattern in the sparse weight matrix and form an MFG by fusing the maximum number of columns (or rows) permitted by the sparsity pattern. This complementary structure allows two or more columns (or rows) to be safely merged into a single fused column (or rows) while preserving all nonzero values and omitting zeros, thereby reducing memory usage. The fused representation is maintained in all phases, without decompression. Unlike prior approaches [12, 18] that decompress the weight matrix at runtime, the proposed approach does not expand the weight matrix; moreover, operating on the compressed SpeedSparse format enables data reuse. For example, at 50% sparsity with two columns fused, SpeedSparse loads the matrix's B columns once and computes two values of the result matrix C (see subsection A.2 lines 8-12 in algorithm 2 and section 2.1.2).

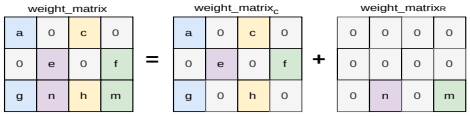
We maintain metadata describing (i) the number of fused columns (or rows), and (ii) a Boolean map of the positions of the elements to guide the GEMM algorithm to process elements in the correct position. At 50% sparsity, each fused column is associated with two corresponding column in the boolean map. Eg., in Figure 2b, the 0th and 1st columns of the bool map B correspond to the 0th column of the fused matrix B. Applying the mask from the 0th column of the bool map B to the 0th column of the fused matrix B reconstructs the 0th column of the original matrix B (as seen in Figure 2a).

2.1.2 TCM GEMM. Next, we perform dense–dense GEMM following the BLIS algorithm for micro-kernel code generation as shown in algorithm 2 (see subsection A.2). SpeedSparse-GEMM differs from BLIS-GEMM with the following changes. (i) Total iterations along the fusion dimension are reduced by the number of fused columns or rows (see subsection A.2 lines 6–7 in algorithm 2). (ii) Instead of using a conventional Fused Multiply Add (FMA) operation, we employ a masked FMA operation (see subsection A.2 lines 9–11 in algorithm 2, Figure 2b), using the generated Boolean map as the mask. This information guides the microkernel and lets the fused representation be processed with vectorization and tiling-friendly computation while ensuring each element contributes to the correct position in the result matrix. (iii) Data reuse along the fusion dimension increases by the number of fused columns or rows (see subsection A.2 lines 8–12 in algorithm 2).

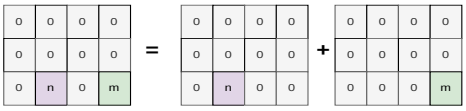
2.2 Case 2: Non TCM

A non TCM is a matrix in which columns (or rows) contain overlapping nonzero elements, and cannot be included in the same MFG.

2.2.1 Not TCM fusion. We split the $weight_matrix$ into two matrices, $weight_matrix_{TCM}$, and the residual elements matrix, i.e., $weight_matrix_R$ (see Figure 3a and algorithm 3 in subsection A.3); both matrices have the same dimensions as the $weight_matrix$. $weight_matrix_{TCM}$ contains elements that have disjoint columns (or rows) forming a TCM and follows the same process as Case-1. Overlapping elements are preserved in the $weight_matrix_R$, along



(a) step 1: Split Weight matrix for fusing two columns



(b) step 2: Split weight matrix for fusing 4 columns

Figure 3: Illustration of a two-steps weight matrix fusion

with their positions. If the residual sparsity is below the Low-Density Threshold (LDT), a matrix-dimension-dependent sparsity threshold above which sparse(CSR)–dense GEMM execution time becomes negligible compared to BLAS/BLIS, we recursively apply Case-2 to $weight_matrix_R$ (e.g., see Figure 3b). For example, for a 4096×4096 matrix, the LDT is 98%, based on empirical measurements (see Figure 1). This process continues until the residual sparsity is above the LDT.

2.2.2 LDT GEMM. Residual elements matrix with sparsity exceeding the LDT, represented using CSR and processed using the sparse(CSR)–dense GEMM kernel [7]. This recursive strategy reduces the number of residual elements while minimizing indirect memory accesses and maintaining a vectorization-friendly data layout without introducing any runtime overhead, as the recursive decomposition is performed during compile-time preprocessing.

2.2.3 Calculation of final results: The final result matrix is obtained using the left distributive property of matrix multiplication.

$$C = A \times \text{weight_matrix} = A \times (\text{weight_matrix}_C + \text{weight_matrix}_R)$$

$$C = A \times \text{weight_matrix}_C + A \times \text{weight_matrix}_R$$

Thus, the contribution from the TCM and the contribution from the residual matrix can be computed independently and subsequently accumulated to obtain the final result matrix (C).

2.3 Discussion

Following the BLIS micro-kernel algorithm [4] for computing ($C = A \times B$), the appropriate fusion direction (i.e., row-fusion or column-fusion) depends on the storage order of the output matrix (C). We select row or column fusion to maximize data reuse and align the fusion direction with the memory-access pattern. Specifically, the following two cases arise:

- **OrderC Row-major (R):** When C is stored in row-major, then (A) is the weight matrix and (B) is the input matrix; (A) is accessed row-wise by the micro-kernel. Therefore, we apply *row fusion* to the weight matrix (A) to align with the access pattern and maximize data reuse.
- **OrderC Column-major (C):** When C is stored in column-major order, then B is the weight matrix and A is the input matrix. The micro-kernel accesses B column-wise; therefore, we apply *column fusion* to the weight matrix B to match the access pattern and maximize data reuse.

SpeedSparse exploits the transpose property of matrix multiplication, where $C^T = B^T A^T$ is equivalent to $C = AB$. Thus, the weight matrix is treated as the first input matrix by using its transpose as matrix A (i.e. first input matrix). The following cases are possible:

(1) Row fusion OrderC-R: Row fusion applied to the weight matrix, results are stored in row-major order.
(2) Row fusion OrderC-C: Row fusion applied to the transposed weight matrix, results are stored in column-major order.
(3) Column fusion OrderC-C: Column fusion applied to the weight matrix, results are stored in column-major order.
(4) Column fusion OrderC-R: Column fusion applied to the transposed weight matrix, results are stored in column-major order.

Concretely, SpeedSparse selectively omits operations involving zero-valued operands while preserving contiguous data layouts; this maintains cache locality and vectorization, increases data reuse, and yields practical sparsity-induced speedups without the runtime or storage overheads of conventional sparse formats or the need for runtime weight decompression.

3 Methodology

We evaluate *SpeedSparse* on an Intel® Xeon® Silver 4314 CPU operating at 2.40 GHz - 3.1 GHz and supporting the AVX-512 vector Instruction Set Architecture (ISA) with 2 vector units. Our evaluation considers both randomly generated and real-world sparse workloads to characterize the effectiveness of SpeedSparse across different sparsity patterns and matrix dimensions. All experiments are performed using single-threaded execution on a single CPU core.

Workloads. We consider the following workload configurations. (i) Randomly generated workloads, which are randomly generated matrices with 50% sparsity following a 2:4 sparsity pattern with varying dimensions from 256×256 to 4096×4096. This represents an ideal case in which the sparsity pattern enables complete fusion without residual elements. (ii) Real-world workloads, which are model weights from LLaMA [24], GPT [14], and NSNet2 [9] with 50% sparsity following a 2:4 sparsity pattern, with residual elements. We verified the correctness of all configurations discussed below, with a Mean Absolute Error (MAE) of less than 10^{-3}.

Configurations. We compare SpeedSparse against BLIS [1] and Intel MKL [1] implementations. The evaluated configurations for $C = AxB$ are as follows. (i) *BLIS-Order-R:* BLIS GEMM kernel optimized for AVX512 with row-major storage for C. (ii) *SpeedSparse-Row-Fusion(A)-Order-R:* SpeedSparse applies row fusion to the matrix A (see section subsection 2.3), with the C stored in row-major order. (iii) *BLIS-Order-C:* BLIS GEMM kernel optimized for AVX512 with column-major C storage. (iv) *SpeedSparse-Column-Fusion(B)-Order-C:* SpeedSparse with column fusion applied to weight matrix (B) and column-major C storage. (v) *SpeedSparse-Row-Fusion(A)-Order-C:* SpeedSparse with row fusion applied to weight matrix (A) (see section subsection 2.3) under column-major C storage. (vi) *BLAS MKL Dense:* Intel MKL dense–dense GEMM [3]. (vii) *BLAS MKL Sparse:* Intel MKL sparse–dense GEMM [7].

Discussion: We attempted to integrate the CPU-specific sparsity-aware tools such as DeepSparse for comparison, but encountered

[1]We optimized the BLIS GEMM kernel for AVX-512 by leveraging Intel AVX-512 vector intrinsics and tuning the tiling factors.

build and compatibility issues. Furthermore, these tool is no longer actively maintained or supported [20]. Therefore, we exclude it from our evaluation.

4 Results

Figure 4 shows the performance of SpeedSparse for a synthetic workload with 50% sparsity following a 2:4 sparsity pattern with no residual elements on the left side, with *Avg RG* representing the geometric mean for randomly generated inputs. Real-world NN and LLMs workloads are on the right, with *Avg RW* representing the geometric mean for real-world workloads. The *y*-axis shows execution time normalized to BLIS-OrderC-C, while the *x*-axis represents the weight matrix dimension for randomly generated workloads.

BLIS and BLAS kernels. BLIS GEMM (BLIS-OrderC-R, BLIS-OrderC-C) and BLAS MKL dense GEMM cannot exploit these semi-structured sparsity patterns and retain zero-valued weights during computation. Consequently, their execution times are similar.

BLAS MKL sparse implementation relies on CSR representation for the sparse matrix. At 50% sparsity, this approach incurs substantial overhead from indirect memory accesses and scatter/gather vector loads and stores, as verified through disassembly using objdump. Consequently, we see performance overhead compared to dense–dense matmul variations, with the overhead increasing as the matrix dimensions increase.

SpeedSparse kernels. SpeedSparse kernels are designed to exploit the semi-structured sparsity in 2:4-pruned LLM models. Through column/ row fusion, SpeedSparse halves the number of micro-kernel iterations and reduces memory load operations by 50% at 50% sparsity, while doubling data reuse within the micro-kernel (see algorithm 2 and *Other performance metrics* section). Although SpeedSparse performs the same number of FMA operations as BLIS and BLAS kernels, its improved data reuse and reduced memory operations enhances performance, particularly when memory is the primary bottleneck as seen in GEMM kernels.

SpeedSparse-Column-Fusion(B)-OrderC-C reduces execution time by an average of 20% compared with BLIS-OrderC-C, primarily by reducing the number of loop iterations and fp32 load operations through weight matrix fusion, as shown in algorithm 2 (see subsection A.2). Fused-column computation introduces additional overhead in the micro-kernel due to mask loading. Specifically, the SpeedSparse micro-kernel requires the mask to be tiled and performs an additional `load_mask` operation. For column fusion on matrix B, data reuse in the BLIS micro-kernel is also limited to 2 for 50% sparsity thereby doubling the floating-point operations per load operation, while two mask operations must be loaded [2] for each matrix B load, resulting in a total of eight mask loads per micro-kernel iteration. These additional operations increase the micro-kernel latency and partially offset the performance benefits of the reduced data load operation. Consequently, despite reducing data loads by 50%, *SpeedSparse-Column-Fusion(B)-OrderC-C* achieves an average execution-time reduction of only 20%.

SpeedSparse-Row-Fusion(A)-OrderC-C reduces execution time by an average of 47% compared with BLIS-OrderC-C, primarily by reducing the number of loop iterations and load operations through weight matrix fusion, as shown in 2 (see subsection A.2). Since the fusion is applied to rows of matrix *A*, each fused row is loaded once for every two micro-kernel iterations, effectively reducing the number of loads. Moreover, the fused computation requires only two `load_mask` operations per micro-kernel iteration, compared with the 8 `load_mask` operations introduced by column fusion of *SpeedSparse-Column-Fusion(B)-OrderC-C*. Consequently, the latency of the innermost micro-kernel loop remains approximately unchanged despite doubling the FMA operations compared to the baseline. Therefore, the approximately 50% reduction in loop iterations translates to an average execution-time reduction of 47%.

BLIS-OrderC-R exhibits a 10-20% execution-time overhead relative to BLIS-OrderC-C (baseline). Hence, the version SpeedSparse-Row-Fusion(mat A)-OrderC-R also incurs this overhead compared to BLIS-OrderC-C variants such as SpeedSparse-column-Fusion(mat B)-OrderC-C and SpeedSparse-Row-Fusion(mat A)-OrderC-C.

Real-world workloads. *SpeedSparse-Row-Fusion(mat A)-OrderC-C* incurs an additional 8–10% execution-time overhead due to the computation required for residual elements that cannot be fused. The *NSNet2-cm* workload represents a special pruning case in which the pruning pattern is selected such that there are no residual elements. Consequently, SpeedSparse does not incur the additional residual-element computation overhead for this workload, resulting in the lower execution time compared to the nsnet2.

Normalized execution time versus memory usage. Figure 5 shows the normalized execution time (geometric mean) on the y-axis and normalized static memory usage on the x-axis for randomly generated workloads and for real-world workloads.

BLIS GEMM (BLIS-OrderC-R, BLIS-OrderC-C) and BLAS MKL dense GEMM cannot exploit these semi-structured sparsity patterns, and therefore both approaches store the zero and nonzero weights, providing no memory savings compared with the corresponding dense, unpruned models. BLAS MKL sparse stores the weights using the CSR format. At 50% sparsity, CSR requires approximately the same storage as a dense representation.

In contrast, our proposed work, the SpeedSparse approach, reduces the memory usage by approximately 41% for randomly generated workloads compared to BLIS GEMM. Without residual elements, SpeedSparse stores 50% weight matrix values, retaining only the non-zero values after fusion, consuming 50% of the memory required by BLIS GEMM. Although the Boolmap must store the full matrix to reconstruct the original positions, SpeedSparse represents it using AVX masking bits [2], where 1 byte encodes 8 mask elements. Consequently, the Boolmap requires only approximately 4% of the memory, with an additional 3% used for metadata.

For real-world workloads following 2:4 (50%) sparsity, the first split of the TCM matrix requires 59% of the memory used by BLIS GEMM, as discussed above. An additional 8% is required to store the residual weight matrix after fusing 12 columns or rows. The Boolmap requires approximately 4% of the baseline memory, while metadata accounts for an additional 3%. Overall, SpeedSparse requires only 73% of the memory used by BLIS GEMM.

Other performance metrics: We evaluate Intel-MKL, BLIS-OrderC-C, and SpeedSparse-Row-Fusion (mat A)-OrderC-C using three metrics: GFLOP/s, floating-point operations per load (FP ops/load), and floating-point operations per store (FP ops/store). O. Intel-MKL and BLIS achieve on average 80 GFLOP/s, 1.65 FP ops/load, and 47 FP ops/store, whereas SpeedSparse achieves 128

[2] throughput of vector mask load is half of vector fp32 load [5, 6]

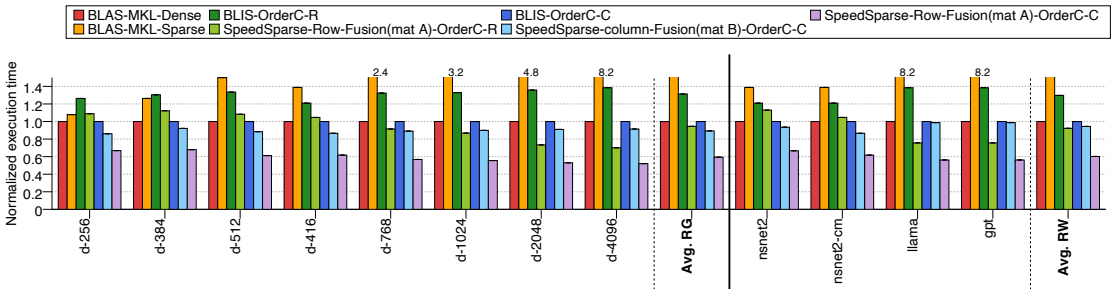


Figure 4: SpeedSparse normalized execution time

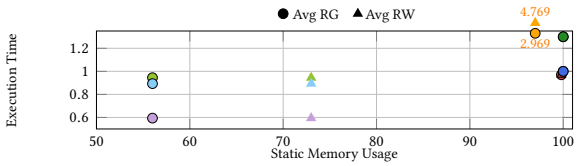


Figure 5: Normalized runtime versus static memory usage

GFLOP/s, 3 FP ops/load, and 71 FP ops/store on average. These results indicate that SpeedSparse delivers higher computational throughput while performing more floating-point operations per memory operations, thereby reducing the execution time.

5 Conclusion

SpeedSparse demonstrates that sparsity can be exploited while retaining the efficiency of dense–dense GEMM and storing only nonzero weights. Through column fusion, sparse weights maintain vectorization-friendly data layouts and enable efficient tiling, and eliminating zero operations without runtime decompression. By leveraging highly optimized BLIS GEMM kernels, SpeedSparse combines the memory benefits of sparse storage with the computational efficiency of dense kernels. In our evaluation, SpeedSparse reduces memory usage by 27% and execution time by 40% at 50% sparsity on average for real-world workloads.

Acknowledgment

This work has been partially funded by the EU's Horizon 2021 research and innovation program (CONVOLVE, g.a. no. 101070374 under HORIZON-CL4-2021-DIGITAL-EMERGING-01), the Spanish Ministry of Science, Innovation and Universities (MCIN) MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (DAMAS project, grant PID2022-136315OB-I00), and by the FSRM/10.13039/100007801
(SCALE project, grant 23039/GERM/25), Spain.

References

[1] 2026. Accelerate Fast Math with Intel® oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
[2] 2026. Accelerating Compute-Intensive Workloads with Intel® AVX 512. https://www.intel.com/content/www/us/en/developer/articles/technical/accelerating-compute-intensive-workloads-with-intel-avx-512-using-microsoft-visual-studio.html
[3] 2026. cblas_sgemm. https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-0/cblas-gemm-001.html
[4] 2026. gemm_blis_family/src/AVX2 at main · martineh/gemm_blis_family. https://github.com/martineh/gemm_blis_family/tree/main/src/AVX2
[5] 2026. *Intel® Intrinsics Guide _load_mask16.* https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_load_mask16
[6] 2026. *Intel® Intrinsics Guide _mm512_loadu_ps.* https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm512_loadu_ps
[7] 2026. mkl_sparse_d_mm. https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2025-0/mkl-sparse-mm.html
[8] 2026. RedHatAI/SparseLlama. https://shorturl.at/N1Tm5
[9] 2024. claroche1/nsnet2-sparse-rowfusion · Hugging Face. https://huggingface.co/claroche1/nsnet2-sparse-rowfusion
[10] 2026. Intel/bert-sparse-80. https://shorturl.at/lcDCe
[11] A. A. R. Islam et al. 2025. Improving SpGEMM Performance Through Matrix-Reordering and Cluster-wise Computation. doi:10.1145/3712285.3759885
[12] D. Joo et al. 2025. Coruscant: Co-Designing GPU Kernel and Sparse Tensor Core Advocate Unstructured Sparsity in LLM Inference. doi:10.1145/3725843.3756065
[13] Ragesh Hajela, Masato Nakagawa, Atsushi Nukariya, Kuninori Ishii, Masahiro Doteguchi, Ikuo Miyoshi, and Priyanka Sharma. 2025. Optimizing BLAS GEMM Kernel Performance: Scalable Multi-Core and Vectorized Design. In *2025 IEEE 32nd International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*. 297–298. doi:10.1109/HiPCW66559.2025.00103
[14] Intel. 2024. gpt-j-6b-sparse. https://huggingface.co/Intel/gpt-j-6b-sparse. Hugging Face model card, accessed 2026-06-22.
[15] Intel Labs. 2025. sparsepeft-mistral. https://shorturl.at/37dQa.
[16] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 821–834. doi:10.1145/3297858.3304028
[17] Eldar Kurtic, Denis Kuznedelev, Elias Frantar, Michael Goin, and Dan Alistarh. 2023. Sparse Fine-tuning for Inference Acceleration of Large Language Models. arXiv:2310.06927 [cs.CL] https://arxiv.org/abs/2310.06927
[18] M. A. Qureshi et al. 2021. Sparse-PE: A Performance-Efficient Processing Engine Core for Sparse Convolutional Neural Networks. *IEEE Access* 9 (2021).
[19] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. arXiv:2104.08378 [cs.LG] https://arxiv.org/abs/2104.08378
[20] Neural Magic. 2021. *DeepSparse.* Sparsity-aware inference software. https://github.com/neuralmagic/deepsparse
[21] NVIDIA. 2026. *Sparse Matrix Formats — NVPL Dense.* https://docs.nvidia.com/nvpl/latest/sparse/storage_format/dense_matrix.html
[22] NVIDIA. 2026. *Sparse Matrix Formats — NVPL SPARSE.* https://docs.nvidia.com/nvpl/latest/sparse/storage_format/sparse_matrix.html
[23] Q. Wang et al. 2025. HR-SpMM: Adaptive Row Partitioning and Hybrid Kernel Design for Sparse Matrix Multiplication. doi:10.1145/3721145.3725770
[24] Red Hat AI. 2024. SparseLlama-3-8B-pruned_50.2of4. https://huggingface.co/RedHatAI/SparseLlama-3-8B-pruned_50.2of4. 2:4 sparse Meta-Llama-3-8B model, Hugging Face model card, accessed 2026-06-22.

A Appendix

A.1 Algorithm: GEMM

Algorithm 1 General Matrix Multiplication

Input: Matrices $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$
Output: Matrix $C \in \mathbb{R}^{m \times p}$
for $i = 0$ to m **do**
  for $j = 0$ to p **do**
    $C_{ij} \leftarrow 0$
    for $k = 0$ to n **do**
      $C_{ij} \leftarrow C_{ij} + A_{ik} \cdot B_{kj}$
    end for
  end for
end for
return C

A.2 Algorithm: SpeedSparse GEMM after Column Fusion

Algorithm 2 General Matrix Multiplication after Column Fusion

Input: Matrices $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, and the number of fused columns $fused_n$
Output: Matrix $C \in \mathbb{R}^{m \times p}$
for $i = 0; i < m$ **do**
  for $j = 0$ to p **do**
    $C_{ij} \leftarrow 0$
    $fused_n \leftarrow number_of_fused_columns$
    for $k = 0, f = 0;$ to $f < n/fused_n; k = k + fused_n, f = f + 1$ **do**
      {# Unrolled to factor of $fused_n$}
      $C_{ij} \leftarrow C_{ij} + A_{ik} \cdot B_{fj} \cdot \text{mask}_{ik}$
      $C_{ij} \leftarrow C_{ij} + A_{ik+1} \cdot B_{fj} \cdot \text{mask}_{ik+1}$
      ...
      $C_{ij} \leftarrow C_{ij} + A_{ik+fused_n} \cdot B_{fj} \cdot \text{mask}_{ik+fused_n}$
      {# B is reused $fused_n$ times in registers}
    end for
  end for
end for
return C

A.3 Algorithm: SpeedSparse Weight Matrix Fusion

Algorithm 3 Weight Matrix Fusion

Function TCM_Fusion($weight_matrix_c$)
$disjoint_columns \leftarrow$ GetDisjointColumns($weight_matrix_c$)
$weight_matrix_{TCM} \leftarrow$ FuseDisjointColumns($weight_matrix_c, disjoint_columns$)
return $weight_matrix_{TCM}$

Function Not_TCM_Fusion($weight_matrix$)
$[weight_matrix_C, weight_matrix_R] \leftarrow$ DecomposeNotTCM($weight_matrix$)
$weight_matrix_{TCM} \leftarrow$ TCM_Fusion($weight_matrix_C$)
if $weight_matrix_R \neq \emptyset$ **then**
  if sparsity($weight_matrix_R$) < LDT **then**
    $[weight_matrix'_C, weight_matrix'_R] \leftarrow$ Not_TCM_Fusion($weight_matrix_R$)
  else
    getCSR($weight_matrix_R$)
  end if
end if
return $weight_matrix_{TCM}, weight_matrix_R$