# SpMV for the Cerebras Wafer-Scale Engine


Daniel Renschler[1,3] Jonathan Schäfer[1]
Johannes Gebert[1] Mark Parsons[2]

[1]High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart
[2]Edinburgh Parallel Computing Centre (EPCC), The University of Edinburgh
[3]Karlsruhe Institute of Technology


## Introduction

Sparse Basic Linear Algebra Subprograms (BLAS) routines are a cornerstone of high-performance scientific computing, providing highly optimized, standardized operations for manipulating sparse matrices and vectors. One particularly important BLAS routine is the Sparse Matrix-Vector Product (SpMV).

BLAS routines [1], [2], especially the SpMV, are essential in traditional applications such as finite element analysis (FEA) and computational fluid dynamics (CFD), as well as in applications where large, sparse systems of linear equations dominate the computational workload, including an increasing number of specialized and emerging algorithms.

On the hardware side, today's and tomorrow's high-performance computing architectures are increasingly leaning towards artificial intelligence and machine learning workloads, including systems based on domain-specific accelerators such as those developed by Cerebras and other AI-focused vendors. While these clusters are often marketed for AI applications, computing centers extend their usage beyond deep learning: they also accelerate traditional simulation-driven workloads, hybrid workflows, and data-intensive tasks that benefit from the parallelism and memory hierarchies optimized for AI.

Our poster presents an SpMV implementation for the AI-specific Cerebras hardware that supports arbitrary sparsity patterns without matrix reordering. We present initial scaling results and discuss possible improvements.

## Background

**Krylov Methods:** Many scientific applications require solving a sparse linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ has very few nonzero entries ($\mathrm{nnz}\, A/n^2 \ll 1$). Krylov-subspace solvers are a natural fit for such systems, with the Conjugate Gradient method being the most prominent example.

Since Krylov solvers perform one or more SpMVs per iteration, the efficiency of overall computation relies heavily on the SpMV performance, making it the ideal place to optimize.

**The Cerebras Wafer-Scale Engine:** The Cerebras Wafer-Scale Engine (WSE) is a massively parallel computing accelerator device that holds up to 900000 cores on a single wafer-scale chip. The cores have access only to very fast but limited local memory and communicate via a 2D Network-on-Chip (NoC). The bandwidth of the host ↔ device connection is significantly smaller than the NoC bandwidth [3].

**Sparse Matrix Formats:** Sparse matrices can be stored in a variety of formats. One of the most well-known and applied formats is the "Compressed Sparse Row" (CSR) format [4].

## Implementation

We implement the SpMV operation $Ax =: b$, where the matrix $A$ is stored on the host in the CSR format. The computation is distributed across the WSE's Processing Elements (PEs) via the pipeline illustrated in fig. 1:

1. **(Host)** Problem specifications and runtime parameters are transmitted to the worker node.
2. **(Worker/Distribution)** Matrix $A$ is partitioned to ensure a balanced workload using histogram-based bucketing of nonzero elements. Each processing element (PE) receives its assigned rows, column indices, corresponding elements of vector $x$, and a row index to be able to gather the result later.
3. **(Communication)** Data packets are transmitted from the worker to the WSE.
4. **(WSE/Parallel)** PEs compute partial dot products independently and store local results.
5. **(Communication)** Results are collected from the WSE and returned to the worker.
6. **(Worker/Parallel)** Partial results are aggregated to form the complete vector $b$.
7. **(Communication)** The final result is returned to the host.

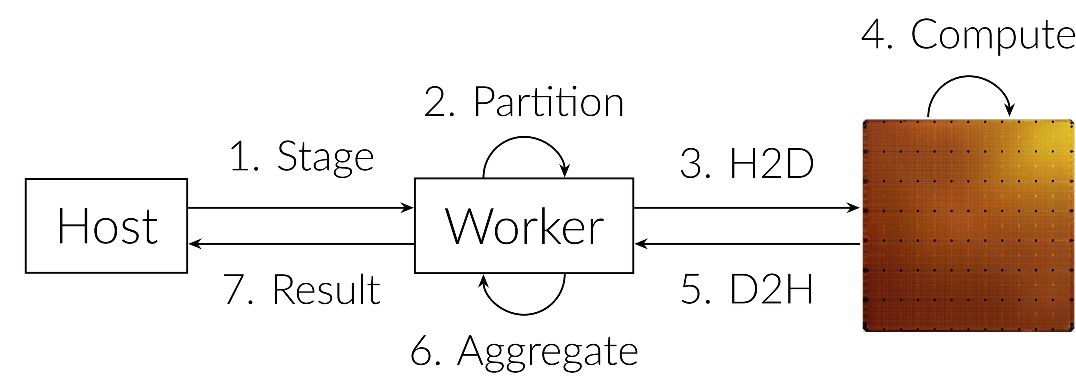


Fig. 1. SpMV execution pipeline

## Results and Discussion

In the following, we present strong and weak scaling analysis, coupled with our interpretation.

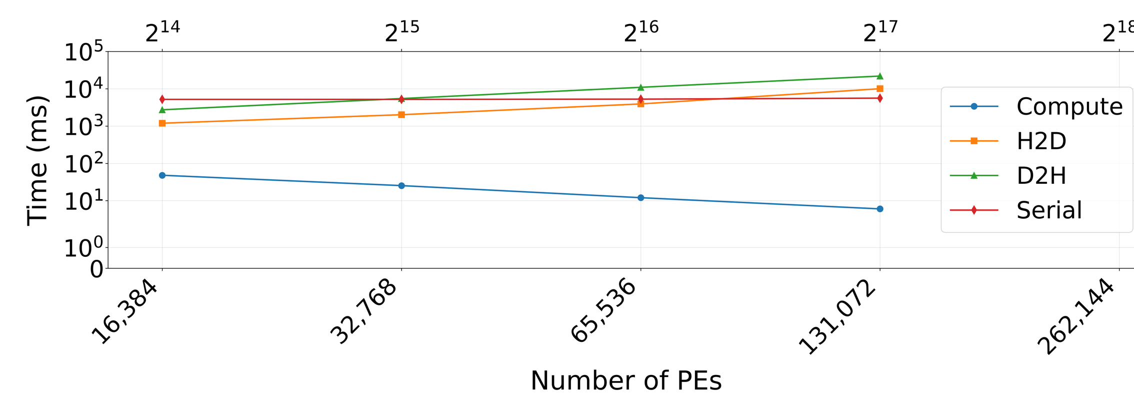


Fig. 2. Strong Scaling: Run Time by Components.

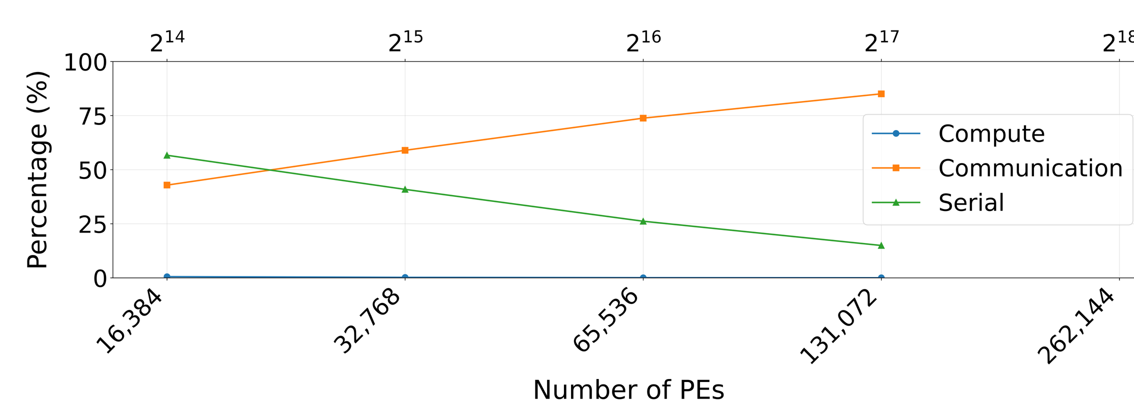


Fig. 3. Strong Scaling: Relative Run Time by Component (%).

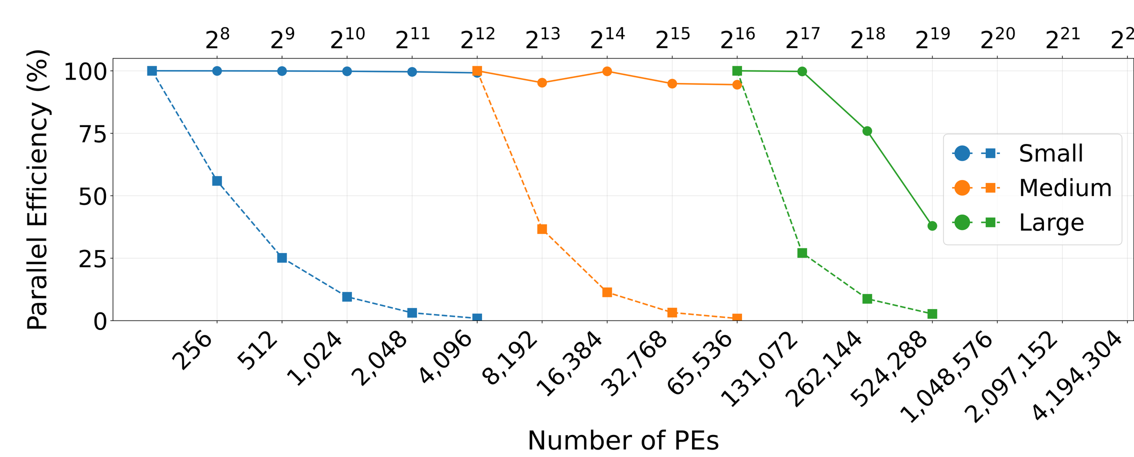


Fig. 4. Strong-scaling efficiency. Solid: only PE calculations; dotted: iterative performance.

Figures 2 and 3 illustrate a strong scaling sweep for the "Strong-1" configuration to be found in table 1. Results show that compute time is negligible compared to SDK overhead. The orders-of-magnitude gap between computation and communication highlights significant optimization potential; specifically, the current implementation redundantly sends data over the worker-WSE interconnect, which lacks the bandwidth of the WSE NoC [3]. Offloading partition and aggregate steps directly to the WSE should substantially improve runtime.

In fig. 4, we conduct a strong scaling experiment with three problem sizes. The setup is given in table 1 by the rows "Strong-Small" to "Strong-Large". We use these three scenarios to highlight the efficiency scaling across a wider range of grid configurations on the WSE. The graph compares PE-grid sizes to the parallel efficiency, normalized to the first run of each sweep. We see a moderate decrease in efficiency for the compute kernel on the WSE alone. Combining the measurements with all other components of the pipeline, especially the worker↔WSE communication, we observe a significant decrease in efficiency, as communication time dominates. This is expected, because communication remains constant with strong scaling, whereas the computational load decreases.

| Sweep | Range | nnz range |
|---|---|---|
| Strong-1 | 16 384 to 131 072 | 40 000 000 |
| Strong-Small | 128 to 4096 | 500 000 |
| Strong-Medium | 4096 to 65 536 | 10 000 000 |
| Strong-Large | 65 536 to 524 288 | 40 000 000 |
| Weak-1 | 256 to 65 536 | 256 000 to 65 536 000 |

Tab. 1. Weak and Strong Scaling Setup

## Results and Discussion (continued)

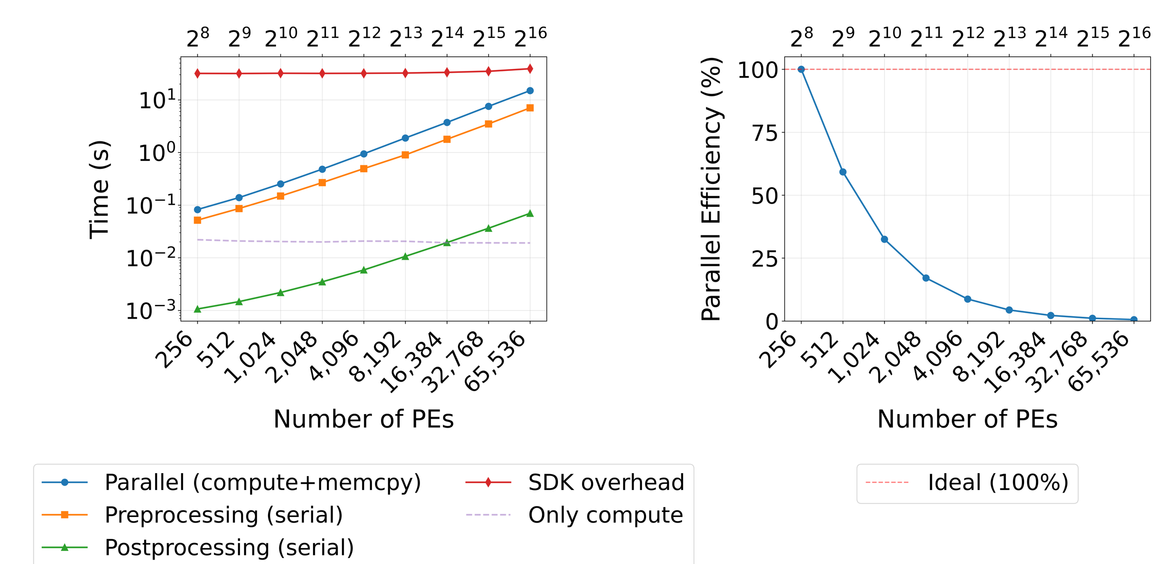


Fig. 5. Weak Scaling: Time Breakdown (left), Efficiency (right)

Figure 5 shows the results of the conducted weak scaling experiment. We fix the load per PE to 1000 $\mathrm{nnz}$ /PE, and scale the global problem with the number of PEs. The relevant run in table 1 is "Weak-1".

The left graph shows the absolute runtime of different components for the scaled global problem. Here *pre-processing* refers to the partition operation on the worker node, *post-processing* to the aggregate operation and SDK overhead to step 1 of the algorithm, namely initializing and loading the program onto the WSE. The compute and memcpy curve shows the combined time of the computation on the WSE and the worker-WSE communication. The computation alone is shown as a dotted line. We observe perfect weak scaling for the computation alone, because computing the local partial dot products is embarrassingly parallel, and SDK overhead is constant as code size is PE-independent. Communication between worker and WSE, partitioning, and aggregation are the main bottlenecks of the algorithm and need to be optimized.

The complete efficiency can be seen in the right plot. Although the computation on the WSE scales perfectly, the large communication and worker node computation overhead completely overshadow this property from the whole algorithm.

## Conclusion and Future Work

The presented initial implementation is a first step towards a Krylov-subspace solver on the WSE that supports arbitrary sparsity. It is currently not competitive compared to highly optimized GPU implementations, because two key capabilities of the WSE are not yet exploited: explicit NoC communication between PEs, and minimizing the worker-WSE communication. We plan to optimize the communication bottleneck by moving the partitioning and aggregation steps of vectors $b$ and $x$ to the WSE. This will most importantly decrease the communication load on the interconnect between worker and WSE, but it will also eliminate partitioning and aggregation on the worker node.

To the best of the authors' knowledge, we presented the first SpMV implementation for arbitrary sparsity patterns on the Cerebras WSE that requires no matrix reordering. Our results show that this implementation is heavily host↔WSE communication bound, and therefore we suggest optimization strategies to reduce this bottleneck. In this effort, the current algorithm serves as a foundation for a fully on-chip SpMV on the WSE, which we are actively researching. The source code is openly available [5].

## References


[1] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, Sep. 1979.

[2] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–17, Mar. 1988.

[3] S. Lie, "Cerebras architecture deep dive: First look inside the hardware/software co-design for deep learning," *IEEE Micro*, vol. 43, no. 3, pp. 18–30, May 2023.

[4] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, "Yale sparse matrix package i: The symmetric codes," *International Journal for Numerical Methods in Engineering*, vol. 18, no. 8, pp. 1145–1151, Aug. 1982.

[5] D. Renschler, *Cerebras SpMV*, *Github repository*, version 1.0, https://github.com/hlrs-fcg/cerebras-spmv/releases/tag/v1.0, 2026.