A 16-bit Floating-Point Near-SRAM Architecture for Low-power Sparse Matrix-Vector Multiplication

Grégoire Eggermann*, Marco Rios*, Giovanni Ansaloni*, Sani Nassif† and David Atienza*
*Embedded Systems Laboratory (ESL), École polytechnique fédérale de Lausanne (EPFL), Switzerland, †Radyalis, Austin, USA
Email: *{gregoire.eggermann, marco.rios, giovanni.ansaloni, david.atienza}@epfl.ch, †srn@radyalis.com

***Abstract*—State-of-the-art Artificial Intelligence (AI) algorithms, such as graph neural networks and recommendation systems, require floating-point computation of very large matrix multiplications over sparse data. Their execution in resource-constrained scenarios, like edge AI systems, requires a) careful optimization of computing patterns, leveraging sparsity as an opportunity to lower computational requirements, and b) using dedicated hardware. In this paper, we introduce a novel near-memory floating-point computing architecture dedicated to the parallel processing of sparse matrix-vector multiplication (SpMV). This architecture can be integrated at the periphery of memory arrays to exploit the inherent parallelism of memory structures to speed up computation. In addition, it uses its proximity to memory to achieve high computational capability and very low latency. The illustrated implementation, operating at 1GHz, can compute up to 370 MFLOPS (millions of floating-point operations per second) while computing SpMV multiplications, while incurring a modest 17% area overhead when interfaced with a 4KB SRAM array.**

***Index Terms*—Near-memory computing, Sparse matrix-vector multiplication, Edge computing.**

I. Introduction

Machine learning (ML) is fostering a revolution in many fields, ranging from implanted devices for healthcare [1] to location-based recommendation applications [2]. Among ML algorithms, Graph Neural Networks (GNNs) are particularly interesting because of their ability to capture dependencies across large problems. GNNs find application in a wide range of scenarios, from physics modeling to text processing [3]. Although GNN models reach state-of-the-art performance in these fields, their large size poses a challenge for computing systems.

Hence, optimizing GNNs is currently a very active area of investigation [4]. Their prevalent computing pattern is that of Sparse Matrix-Vector (SpMV) multiplication because graphs can be represented as (very large) sparse matrices with non-zero elements indicating edges connecting nodes. Hence, efficient implementations of this arithmetic pattern are key for reducing the GNNs' computational and memory requirements. [5].

Several works focus on exploiting the parallelism of a memory structure to speed up and increase the energy efficiency of SpMV multiplication [6] [7] [8]. We review them in Section II. Our work takes a similar approach but differentiates from related efforts by proposing a dedicated near-memory processing unit for SpMV, operating on a 16-bit floating-point data representation. Being specialized for SpMV multiplication, our design is much more area-efficient than solutions for general-purpose near-memory processors [9]. Moreover, the floating-point capability of our architecture allows for larger dynamic ranges in data representation than fixed-point alternatives [8], which is a key requirement for GNNs.

We show that floating-point arithmetic (addition and multiplication) can be added to our design with a marginal increase in area cost, allowing our near-memory SpMV multiplication accelerator to be integrated with SRAM sub-arrays as small as 4KB in size, inducing an overhead of only 17%. Such fine granularity poses a challenge when partitioning and mapping large matrices. Addressing this problem, we show a novel solution that integrates fixed row mapping and row reordering. Our approach effectively limits data replication across memory arrays, allowing for a high degree of computational parallelism.

The paper's contributions are as follows:

- We present a novel near-memory architecture dedicated to computing floating-point SpMV multiplication.
- We explore its integration at the periphery of SRAM banks. Considering a collection of benchmark sparse matrices, we discuss the resource and performance implications of different architectural arrangements.
- We show how SpMV multiplication can be effectively mapped in our proposed near-memory architecture by data partitioning and reordering, with the aim of maximizing computational efficiency.

We summarize concepts and research work related to our contribution in Section II. Then, we detail the near-SRAM architecture in Section III. Our novel strategy for efficiently partitioning and mapping data on compute memories is discussed in Section IV. Experiments are introduced and discussed in Sections V and VI, respectively. Finally, Section VII summarizes the main findings of the paper.

II. Background and Related Work

A. Sparse matrix vector multiplication

SpMV involves multiplying a $N \times M$ matrix with very few non-zero elements by a dense input vector of N elements,

This research was partially supported by EC H2020 FVLLMONTI project (GA No. 101016776) and by the ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR.

979-8-3503-2599-7/23/$31.00 ©2023

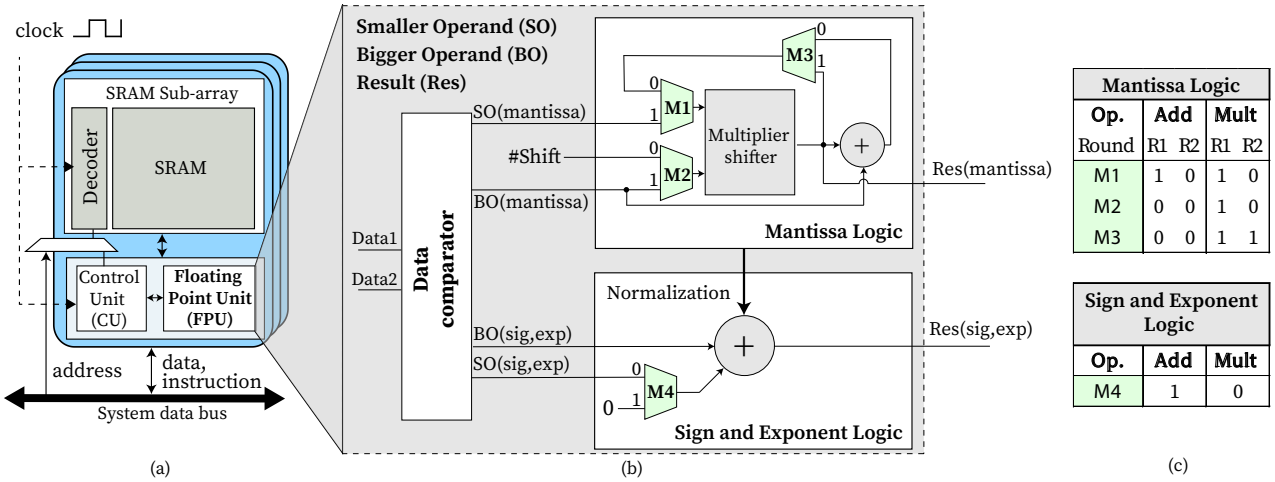

Fig. 1. Block diagram of the implemented design showing (a) the integration of the system in a CPU platform and focusing on (b) the near-memory floating-point unit. c) Multiplexers route the inputs to the multiplier/shifter in the two rounds (R1 and R2) required for addition and multiplication.

resulting in a dense output vector of M elements. SpMV is the fundamental computational kernel in graph-related algorithms such as GNNs, a class of deep learning networks that has recently attracted considerable attention in academia due to their versatility. GNNs can generalize other neural networks, such as convolutional and recurrent neural networks [3]. For example, GNNs have been shown to learn from non-Euclidean data structures [10]. Moreover, they have been applied to a wide range of tasks that benefit from the ability of graph-structured data to represent rich relationships among elements, from cancer classification [11] to text generation [12].

Naïve dense matrix-vector multiplication algorithms are ill-suited for the sparse matrices in GNNs, as they incur a large number of multiplications by zero (that do not influence results), leading to computational and storage inefficiencies. To address this limitation, specialized representations, such as COOrdinate (COO) and Compressed Sparse Row (CSR), have been proposed [13], exploiting the sparsity of the matrices to encode them efficiently. The COO format lists the non-zero elements of a matrix along with their row and column indices. Therefore, each non-zero element of the matrix is represented by a triplet (n, m, non-zero value), where n denotes the row index, and m denotes the column index. CSR instead lists the number of non-zero elements in each row and their column position. We adopt a COO scheme in our work, as has been shown in [9] to lead to greater efficiency in distributed computing. Moreover, it results in a simpler hardware implementation for controlling the execution of SpMV multiplication.

The ordering of the matrix rows also plays a role in the efficiency of computing SpMV multiplications. The recursive algorithm in [14], based on nested dissection, [15], is particularly effective in reordering rows to increase run-time efficiency. We explore its benefits when applied to our design in Section IV.

B. Near-memory computing

SpMV multiplication is intrinsically a memory-bounded kernel, as it requires a very large amount of memory bandwidth. Therefore, processor-centric systems, such as CPUs, poorly support this kernel. It is similarly challenging for GPUs, as their high level of computational parallelism is hampered by sparse and irregular access patterns. In particular, the lack of spatial locality in sparse matrices leads to a poor temporal locality in input and output vectors (since contiguous elements may be accessed at distant moments in time), resulting in poor utilization of local memories, which lowers performance [7].

This challenge motivated the development of hardware accelerators dedicated to sparse arithmetic [16]. In this light, Near-Memory Computing (NMC) solutions are particularly promising, as they combine the high bandwidth present at the memory periphery with the high degree of parallelism derived by the regular structure of memory arrays.

Most works in SpMV multiplication in NMC consider high-performance computing solutions. The authors of [9] and [6] propose to integrate SpMV computing units into DRAM banks on a 3D integration using Through Silicon Vias (TSV).

With a focus on low-power edge systems, we propose a near-SRAM computing unit instead. The investigated scenario is similar to the one in [8]. However, the NMC design in that work only supports fixed-point arithmetics, which has a considerably lower dynamic range than floating-point notation. Moreover, their solution is based on analog computing, which requires energy- and area-hungry domain converters. We address these deficiencies by a fully digital near-SRAM architecture that performs 16-bit floating-point arithmetic, showing that these operations can be implemented within tight energy and area envelopes while incurring negligible accuracy drops [17]. Finally, being entirely CMOS-based, our design can be readily integrated with SRAM arrays.

III. System Architecture

In this section, we first present the components of the proposed Floating-Point Unit (FPU), illustrating how these are

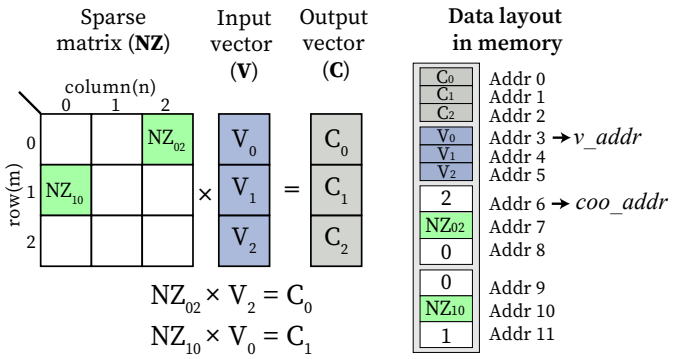

Fig. 2. Diagram showing data mapping of a portion of matrix inside the memory

employed to perform addition/subtraction and multiplication operations. Then, we describe how these basic operations are managed by a dedicated Control Unit (CU) to implement SpMV. The FPU is interfaced with the sub-arrays composing conventional SRAM arrays, while the CU is connected to the system bus (as shown in Fig. 1-a) and programmed via configuration registers. Our design does not require modifying standard bus protocols or re-designing the memory sub-arrays. Since the FPU and CU operate at the same speed as the memory accesses, they do not affect their performance. They are entirely transparent when a sub-array is solely used for storing data (i.e., no near-memory computation is performed). To parallelize the run-time execution, multiple sub-arrays can be employed, each having its own near-memory FPU and CU.

A. Near-SRAM floating-point unit

The FPU structure comprises three main blocks, as shown in Fig. 1-b. First, the data comparator selects the Bigger Operand (*BO*) and the Smaller (*SO*) one (in absolute value). Then, the Mantissa Logic, which is employed twice in both multiplications and additions, is built with a combinational multiplier and an adder. Shifts are carried out in the multiplier, by multiplying one operand with a one-hot code representing the number of shifts (named *#shift* in Fig. 1). Finally, the Sign and Exponent logic comprises a three-input adder.

When performing **additions**, the mantissa of the *SO* is right-shifted by the multiplier/shifter to align its exponent with the *BO*'s exponent, as indicated by the comparator block [1]. The adder in the Mantissa Logic then calculates the addition of the mantissa. The result is again fed to the multiplier/shifter so that its most significant bit is properly aligned, since a right or left shift may be required to align the most significant bit of the mantissa output, depending on the operand values. Finally, the Mantissa Logic forwards the number of shifts (*normalization*) to the adder in the sign and exponent logic. As the last step in the floating-point addition, the exponent of *BO* is added with the *Normalization* value.

When executing **multiplications**, the multiplier/shifter is instead used to multiply the two mantissas. The Mantissa Logic's adder is bypassed, and the result is looped back to the multiplier/shifter, where, as in the addition case, such block is employed to align the mantissa output. However, differently from additions, the Sign and Exponent Logic adds the exponents of *BO* and *SO*, as well as the *Normalization* value.

For multiplication and addition, the specialized logic (residing in the comparator) handles special values such as infinity, zero, and not a number. Our design performs floating-point additions and multiplications in only **5** clock cycles.

[1]For simplicity, the logic responsible for the one-hot coding of the *#shifts* signal is omitted from Fig. 1-b.

Algorithm 1 The SpMV algorithm implemented at the compute sub-array controller. It performs $NZ_{mn} \times V_n = C_m$ in each **while** loop. The sparse-matrix column and input index are represented by n, while m represents the sparse-matrix row and the output index.

```
1: v_addr ← base address of input vector
2: coo_addr ← address of current non-zeros (COO format)
3: last_addr ← last sub-array address
4: while coo_addr ≥ last_addr do
5:     n = read(coo_addr + +)
6:     V_n = read(n + v_addr)
7:     NZ_mn = read(coo_addr + +)
8:     m = read(coo_addr + +)
9:     C_m ← C_m + NZ_mn × V_n                ▷ MAC op.
10: end while
```

B. SpMV algorithm

Multiplication and addition between floating-point numbers are used to implement the MACs (Multiply-ACcumulates) required for SpMV. To this end, the architecture supports COO encoding, which, as illustrated in Section II-A, enables the compression of the sparse matrix representation by considering only non-zero values. Each sub-array is logically partitioned into three regions to maximize computation efficiency, as shown in the example in Fig. 2. The top region stores a slice of the output vector being computed. The middle region stores a slice of the input vector. Finally, the bottom region stores the non-zero values of the matrix tile corresponding to the input and output slices, represented in COO format. Notice that 3 memory words are required for each non-zero matrix element (stating its column index, value, and row index, respectively).

Three registers are used to keep track of the SpMV computation at run-time:

- coo_addr : Stores the address of the next word to be read in the region storing non-zero matrix values. This register is initialized at the beginning of the matrix tile region.
- v_addr: Stores the offset address corresponding to the beginning of the input region.
- $last\_addr$: Stores the address of the last non-zero value.

As dictated by COO coordinates and the offset register value, the CU orchestrates the memory accesses to read/write operands and results, while activating the FPU to perform arithmetic operations. It performs a **while** loop over the sub-array addresses until all MAC operations are performed, as shown in Algorithm 1.

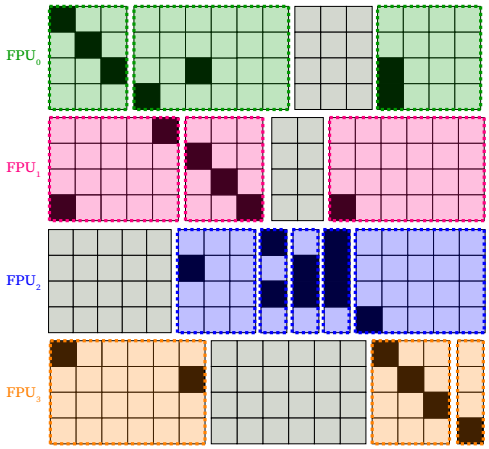

Fig. 3. Example of fixed-row tiling for an architecture composed of 4 sub-arrays with 16-word memory capacity. The black squares represent the non-zero values, the dotted rectangles are the tiles, and each color corresponds to the sub-array to which the tile is assigned. Skipped portions of stripes (containing only zero values) are shaded in grey.

In each MAC operation, the column index n of a non-zero element of the sparse matrix tile is read (line 5) then this index is translated to a sub-array address offset by v_addr, allowing the fetching of the corresponding input vector element. The non-zero matrix value is accessed on the subsequent address of coo_addr (line 7). At this point, both NZ_{mn} and V_n are available at the FPU and so they are multiplied. In parallel, the row index m is fetched from the sub-array by accessing the next coo_addr (line 8). Once the multiplication has reached completion, the product accumulates in C_m (line 9). A complete multiply-accumulate operation requires **14** clock cycles.

SpMV computations are executed independently on different matrix tiles and can therefore be distributed among several sub-arrays operating in parallel. Note also that the SpMV multiplication approach outlined above can accommodate rectangular matrix tiles (input and output slices with different dimensions). We exploit this feature to optimize data mapping, as discussed in the next Section.

IV. Data partitioning and mapping

The sparse matrices used in ML applications are, of course, far larger (even when using compressed representations) than the storage capacity of sub-arrays. Therefore, to perform SpMV efficiently, it is key to properly partition the data into matrix tiles and the inputs/outputs into vector slices.

Of the various tiling strategies proposed in the literature [9], fixed-row tiling is particularly appealing in our scenario. Such a mapping partitions the sparse matrix into 2D rectangular tiles, which all have the same height and a variable width (depending on the matrix sparsity). Consequently, the output vector slice computed in each sub-array has a constant size throughout the sub-arrays, while the input vector slice size varies with the tile width. In our architecture, such a strategy eliminates the need for a computation phase to aggregate partial results. Instead, each sub-array is assigned to compute a horizontal *stripe*, i.e., all tiles in a row. Using this strategy, inputs must still be transferred tile-wise to the sub-array according to the available storage. However, the output remains resident in memory and thus accumulates the results of all the relevant tiles, minimizing the number of data transfers.

On the other hand the fixed row tiling requires the replication of input vector slices in multiple sub-arrays. This redundancy can be reduced by maximizing the size of the output vector slice (hence, the height of the matrix stripe) computed in each sub-array. Such a strategy is particularly effective in the common case when each input vector element only contributes to the computation of a few output elements, for example, in sparse matrices that are locally dense along the diagonal.

Our implementation of fixed-row tiling is illustrated in Fig. 3 for a small illustrative example. In the figure, we consider a 16×16 matrix and an architecture composed of 4 sub-arrays, each composed of 16 words storing matrix tiles, inputs, and outputs according to the scheme in Fig. 2. First, the height of the matrix stripe is fixed to four in the example. Then, the tile widths, different for each tile, are determined to maximize the number of non-zero matrix values per tile while not exceeding the sub-array size constraint. The tiles in a stripe are then processed sequentially. Leading columns in a tile are completely skipped if they only contain zeros, reducing the redundancy in the input vector data and the number of required matrix tiles. In the example, the first 5 input vector elements do not have to be transferred to FPU_2 since the corresponding matrix columns are skipped.

TABLE I
Benchmark sparse matrices, from [18].

Benchmarks	Matrix size	Non-zeros	Density
c-61	43,618 × 43,618	310,016	1.63×10^{-4}
roadNet-TX	1,393,383 × 1,393,383	3,843,320	1.98×10^{-6}
delaunay_n19	524,288 × 524,288	3,145,646	1.14×10^{-5}
fe_ocean	143,437 × 143,437	819,186	3.98×10^{-5}
gridgena	48,962 × 48,962	512,084	2.14×10^{-4}
k49_norm_10NN	38,547 × 38,547	618,158	4.16×10^{-4}
worms20_10NN	20,055 × 20,055	240,826	5.99×10^{-4}
amazon0601	403,394 × 403,394	3,387,388	2.08×10^{-5}
webbase-1M	1,000,005 × 1,000,005	3,105,536	3.11×10^{-5}

V. Experimental set-up

Implementation. We designed the near-memory unit composed of the floating-point computing block and the SpMV controller in 28nm CMOS technology. We appended the near-memory units to high-density single-port SRAM sub-arrays ranging from 4kB (2048 16-bit words) to 32kB (16384 16-bit words). The total storage capacity was kept constant at 32kB. Hence, we instantiate one, two, four, and eight FPUs to 32kB, 16kB, 8kB, and 4kB SRAM sub-arrays, respectively. Area and energy extractions were performed post-synthesis for a 1GHz timing constraint.

Baselines. We compared our FPU to a similar architecture capable of adding and multiplication in a fixed-point format. The fixed-point architecture features a 16-bit multiplier instead

TABLE II
COMPARISON OF THE AREA PERFORMANCE OF THE FLOATING-POINT ARCHITECTURE WITH RESPECT TO FIX-POINT ARCHITECTURE. AREA VALUES ARE IN μm^2.

	Fixed-point		**Floating-point**	
	Area	Overhead	Area	Overhead
Memory (4kB)	7927.2	-	7927.2	-
Near-memory unit	622.8	7.9%	874.7	11.0%
Controller	442.1	5.6%	475.3	6.0%
Total	**1064.9**	**13.5%**	**1350.0**	**17.0%**

of the 11-bit multiplier in the Mantissas Logic of our proposed design. On the other hand, it does not require logic to deal with exponents. It can perform a MAC operation in 11 clock cycles because the multiplier must only be traversed once (as no normalization step has to be performed). Hence, for a fair comparison, we set its timing constraint to 787MHz (a setting that puts the integer implementation at an advantage), so that both floating- and fixed-point architectures can perform a MAC in the same time span.

Benchmarks. We tested the performance of our designs on the collection of sparse matrices of the University of Florida [18]. We selected a representative sample of matrices that were related to general graph applications. Their characteristics are reported in Tab. I. All benchmark matrices are square, with sizes varying between 20 thousand and 1.4 million lines. The number of non-zero values ranges from 240 thousand to 3.8 million. Thus, the density of the matrices is on the order of magnitude of 10^{-4} to 10^{-6}.

VI. EXPERIMENTAL RESULTS

A. Floating-point compared to fixed-point

The area breakdown of the implemented design is detailed in Tab. II. The first line of the table reports the area of a 4KB SRAM sub-array, which is independent of the FPU design. The second and third lines report the area (and the related overhead) of the arithmetic unit and the controller. The total area overhead of the floating-point unit in this implementation is 17%. We regard this as a worst-case scenario of using very small sub-arrays. Indeed, increasing the sub-array size reduces the ratio of the area dedicated to FPUs.

The fixed-point baseline incurs a slightly lower area overhead of 13.5%. The difference between the two implementations is due to the additional complexity (comparator, exponent logic) required by floating-point arithmetic and the more stringent timing constraint employed to synthesize this design. Such effects are partially compensated for by the larger bit width of the fixed-point multiplier, which requires 50% more area. The energy consumption of our near-memory block is only 8.5pJ/MAC, representing an increase of 14.9% of the energy compared to the fixed-point architecture, which consumes 7.4pJ/MAC. The overall performance of our floating-point architecture is thus competitive with the fixed-point implementation. We have slightly increased the area and power of our design to benefit the dynamic range of the represented values.

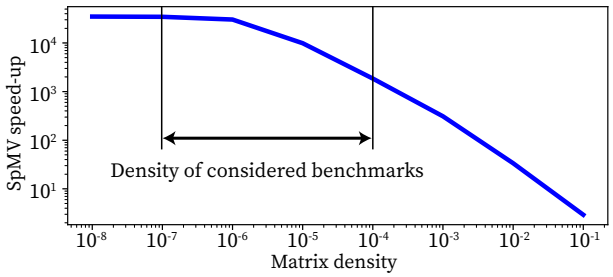

Fig. 4. Speedup of SpMV multiplication, with respect to a conventional algorithm, on a 40,000 × 40,000 matrix, varying the density of non-zero values.

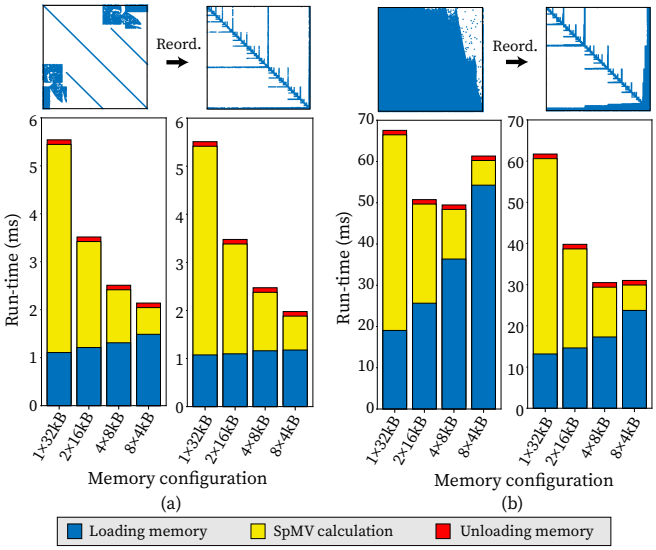

Fig. 5. Execution time of the SpMV algorithm before and after reordering of the matrices for a total memory size of 32kB split in 1, 2, 4, and 8 FPUs with (a) c-61 and (b) amazon0601.

B. SpMV algorithm performance

Fig. 4 showcases the speed-up obtained by our SpMV strategy, compared to a conventional matrix-vector multiplication. It considers a square matrix of 40 thousand lines with random placement of non-zero values and varying degrees of sparsity. Very large speedups are reached for high sparsity levels (up to 35,000× for densities of 10^{-6}). SpMV multiplication performs worse than a conventional dense computation (with speed-ups less than 1) only when the matrix densities are higher than 10%. At these sparsity levels, the overhead of storing each non-zero value's coordinates is no longer compensated by the gain induced by skipping multiplications by zero. Notice that, for the benchmark matrices considered further in this section, the speedups obtained are always greater than 10^3.

Often, sparse matrices from graph applications are locally denser around the diagonal, the edges, and the corners. This characteristic is beneficial in our NMC scenario, as few input vector elements have to be replicated across sub-arrays. Such a feature can also be emphasized (or enforced) by reordering the matrix rows. We highlight such an effect, employing the

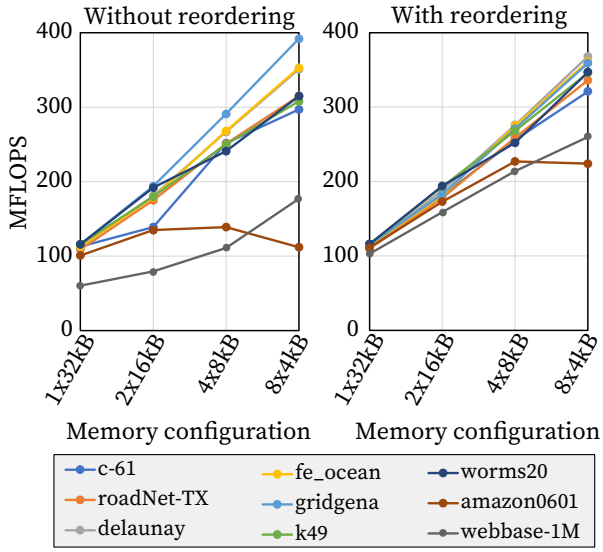

Fig. 6. Performance of the SpMV algorithm with and without matrix reordering for a 32KB compute memory, having 1, 2, 4, or 8 FPUs.

nested dissection algorithm illustrated in [15], in Fig. 5. When sparse matrices are already well-formed, as is the case of the *c-61* benchmark in Fig. 5-a, reordering does not have a major effect, resulting in marginal benefits, especially when employing small sub-arrays. Instead, sparse but irregularly scattered matrices such as *amazon0601* presented in Fig. 5-b are highly impacted by reordering. In particular, in such a benchmark, increasing the number of FPUs (e.g., from 4 to 8) can lead to a performance drop in the non-reordered case. Such an effect is due to the high increase in data transfers due to the replication of input vector elements.

The performance of different implementations of our NMC design is comparatively evaluated in Fig. 6 for all the benchmarks in Tab. I. Plots report the MFLOPS (Millions of floating-point Operations per Second) executed by the architectures, with and without considering matrix reordering. The results show that the performance scales gracefully even for very small subarray sizes (such as the 4×8KB configuration). Increasing further the number of FPUs (e.g. in the 8×4KB case) still induces speed-ups in most cases, at the cost of a higher area requirement. As discussed above, *amazon0601* and *webbase-1M* (without reordering) exhibit shallower trends, as their irregular sparsity induces inefficient data replications of input vector elements for small sub-array sizes.

C. Comparison to the state of the art

Tab. III compares the performance of our architecture to the UPMEM PIM architecture [9], with the latter operating on 16-bit integer and 32-bit floating point numbers. The authors do not report data for the 16-bit floating-point case. It can be noticed that our design outperforms UPMEM PIM in terms of throughput per pipeline, thanks in large part to the higher proximity between memory and computing elements in our case.

TABLE III
COMPARISON TO THE ARCHITECTURE IN [9].

	UPMEM PIM		**Our work**
Clock frequency	500 MHz		1 GHz
Memory technology	DRAM		SRAM
Data format	INT16	FP32	FP16
Performance (per pipeline)	6.25 Mop/s	0.53 Mop/s	**46.25 Mop/s**

VII. CONCLUSION

In this work, we have proposed a novel near-memory floating-point architecture designed to seamlessly interface with SRAM sub-arrays and perform sparse matrix-vector multiplications, a key computational kernel in large graph applications such as GNNs. Our design only causes a 17% area overhead when attached to a 4kB SRAM, marginally more than an equivalent fixed-point implementation. Moreover, we have described how very large matrices can be mapped on our near-memory architecture using fixed row mapping and reordering. Our strategy effectively minimizes data transfer overheads caused by replicating input vector values. Finally, we have demonstrated the scalability of our approach on a representative set of sparse-reference matrices.

REFERENCES

[1] M. e. a. Shaeri, "Challenges and opportunities of edge AI for next-generation implantable BMIs," in *AICAS*, 2022.
[2] T. e. a. Zhong, "Hybrid graph convolutional networks with multi-head attention for location recommendation," *World Wide Web*, 2020.
[3] J. e. a. Zhou, "Graph neural networks: A review of methods and applications," *AI open*, 2020.
[4] K. e. a. Huang, "Understanding and bridging the gaps in current GNN performance optimizations," in *SIGPLAN*, 2021.
[5] S. e. a. Qiu, "Optimizing sparse matrix multiplications for graph neural networks," in *LCPC*, 2022.
[6] X. e. a. Xie, "SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator," in *HPCA*, 2021.
[7] D. e. a. Fujiki, "Near-memory data transformation for efficient sparse matrix multi-vector multiplication," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.
[8] S. e. a. Srinivasa, "Trends and opportunities for SRAM based in-memory and near-memory computation," in *ISQED*, 2021.
[9] C. e. a. Giannoula, "SparseP: Towards efficient sparse matrix vector multiplication on real processing-in-memory systems," *arXiv preprint arXiv:2201.05072*, 2022.
[10] M. M. B. et al., "Geometric deep learning: going beyond euclidean data," *CoRR*, 2016.
[11] S. e. a. Rhee, "Hybrid approach of relation network and localized graph convolutional filtering for breast cancer subtype classification," *arXiv preprint arXiv:1711.05859*, 2017.
[12] L. e. a. Song, "A graph-to-sequence model for AMR-to-text generation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018.
[13] V. L. et al., "Matrix computations (Johns Hopkins studies in mathematical sciences)," *Matrix Computations*, 1996.
[14] A. e. a. Pinar, "Improving performance of sparse matrix-vector multiplication," in *SC*, 1999.
[15] R. J. e. a. Lipton, "Generalized nested dissection," 1979.
[16] S. e. a. Dave, "Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights," *Proceedings of the IEEE*, 2021.
[17] S. e. a. Gupta, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015.
[18] T. A. e. a. Davis, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663