EMBARK: Memory bounded architectural improvement in CSR-CSC Sparse Matrix Multiplication

Shakya Jayakody
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL
Email: shakya@ucf.edu

Jun Wang
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL
Email: Jun.Wang@ucf.edu

***Abstract*—Sparse Matrix Multiplication (SpMM) is a crucial algorithm in modern platforms such as Artificial Intelligence (AI), Graph Neural Network (GNN), Graph Convolutional Network (GCN), and neural network image processing. However, the performance of SpMM is limited by several factors, such as memory storage, data reuse, I/O traffic, and memory access. To address these challenges, we have developed a dynamic memory allocation method called EMBARK, specifically including a new Compressed Sparse Row (CSR) $\times$ Compressed Sparse Column (CSC) matrix multiplication algorithm that reduces significant matrix decompression/compression overhead and optimizes storage allocation.**

Our CSR$\times$CSC algorithm is based on memory partitioning techniques: EMBARK, values, and rowid are together, and colptr is stored separately in the main memory. To improve the performance of our algorithm, the main memory is utilized to store hot data for values, colid, and rowid, while the Non-volatile Memory (NVM) stores partial hot data based on a rank-based page replacement strategy. We have conducted experiments with SuiteSparse matrix real datasets, and our results show CSR$\times$CSC-EMBARK reduced the execution time average by 46.44% compared to the baseline matrix-by-matrix multiplication (M$\times$M).

***Index Terms*—Sparse Matrix Multiplication, Compressed Sparse Row, Compressed Sparse Column, Non-volatile Memory, Memory Allocation**

I. Introduction

Sparse Matrix-Matrix Multiplication (SpMM) is a specialized multiplication operation where both matrices predominantly contain zeros as opposed to non-zero elements [28]. This operation has significant applications in graph algorithms, graph sampling, Graph Neural Networks (GNN), which includes Graph Convolutional Networks (GCN), neural network image processing [3], [17], [22], and the transformer architecture [31]. Within Transformer models, which are foundational to many current deep learning approaches [31], SpMM is instrumental. It facilitates the self-attention mechanism by determining attention scores between different inputs through the dot product of their respective embeddings [12].

In scenarios with limited system memory, out-of-core execution of SpMM is essential. This means that the computation is performed beyond the scope of the primary memory (RAM), utilizing secondary storage devices like hard disk drives or solid-state drives [16]. Such an approach becomes critical when working with expansive datasets or models that surpass the system's RAM capacity [13]. Enhancing the efficiency of the Compressed Sparse Row (CSR) format by discarding zero values and retaining only the non-zero ones has been previously explored. This enhancement typically relies on three components: *values*, *colid*, and *rowid* [32]. Fundamentally, SpMM is integral to a plethora of scientific and high-performance computing tasks.

Non-volatile memory products, particularly ultra-low latency NAND Flash-based Solid State Drives (SSDs), have emerged as promising candidates for constructing a unified memory-storage hierarchy (UMH) [1], [34], or hybrid memory — namely DRAM combined with storage class memory architecture [9], [20], [33], or extended persistent memory [10]. The UMH renders a single ample memory address space for all memories. As a result, the processors can directly access structured data in SSDs and eliminate implicit bulk data copy/swap between devices. Such UMH could mitigate the data movement between NVM and DRAM. However, the translation layer in the memory controller imposes non-negligible memory address mapping overhead.

To address these challenges, we introduce EMBARK, a dynamic memory allocation method. The idea is to allocate CSR and CSC matrices of data and positioning information at runtime according to the dynamic matrix multiplication flow. Hence it can make good use of spatial locality to avoid unnecessary page faults, reduce I/O traffic and expedite computation. Central to EMBARK, we implemented a CSR$\times$CSC multiplication algorithm to reduce the overhead of decompression/compression for large matrices. Intuitively, we improve the execution time without reconstructing the original matrixes on the fly and, therefore reduce the total number of a pair of matrix element multiply operations. However, the CSR format is bottlenecked by data reuse and memory utilization [14] [6]. The main reason is that accessing data and positioning information in different orders from two memory hierarchy levels, compromised fast DRAM and slow Non-Volatile Memory (NVM), can lead to various non-negligible delays. We proposed a storage optimization for NVM and partition theme in the main memory to overcome these issues. Our baseline employs a straightforward matrix multiplication, which allocates matrix A and matrix B, with row-wise data uniformly stored in the main memory. However, this approach incurs non-negligible I/O traffic, resulting from non-sequential data access of matrix B in the main memory. This is especially true in frequent and repeated large matrix multiplications, where out-of-core hot data (i. e., frequently accessed data) are stored in the NVM storage but continuously retrieved by the main memory for reusing the matrix B data in the matrix multiplication. As a result, memory utilization between NVM and main memory could be overwhelmed.

We develop an enhanced dynamic memory allocation method in EMBARK to minimize I/O traffic between NVM storage and main memory-DRAM by saving CSR A and CSC B *values*, *colid* and *rowid* in separate memory blocks. In this method, we minimize the CSR A data storage and maximize CSC B data storage for an extended instruction period, ensuring that all requisite matrix elements for the impending matrix multiplication are retained within the main memory. CSR A undertakes row-wise multiplication across the expansive columns of CSC B. For matrices that are sizable and cannot be accommodated within the main memory's capacity, we judiciously allocate the data and positional information for matrices A and B across separate memory divisions to suit the ensuing vector multiplication. This translates to storing the *values*, *colid*, and *rowid* of two CSR matrix formats in disparate segments of DRAM and NVRAM. By adopting this strategy, we harness the spatial locality inherent in broad matrix access, aligning the data flow with computational instructions. To gauge the efficacy in terms of performance and memory access, we subjected our approach to comprehensive system cycle-level simulations using GEM5 [7]. The results of our simulated experiments revealed that EMBARK enhances storage efficiency and trims memory access, culminating in a 46.44% reduction in matrix multiplication execution time. In essence, our proposed mechanism diminishes I/O interactions between NVM storage and DRAM, streamlining storage and memory access for more agile computations.

It is worth noting that the Processing-In-Memory (PIM) architecture [2] can be integrated with our EMBARK memory controller to reduce data movement further. However, it is not fully commercialized. This paper focuses on general architecture and does not consider the PIM architecture. We solely concentrate on memory utilization in the main memory and the EMBARK memory controller for CSR$\times$CSC multiplication. Our contributions can be summarized as follows:

- We developed an algorithm for CSR$\times$CSC multiplication (Algorithm 1).
- We explored and determined the optimal memory allocation for CSR A and CSC B to maximize memory utilization based on rank based page replacement strategy
- We introduced the EMBARK memory management system, which aims to minimize data movement between DRAM and NVM and exploits fast memory access for SpMM computation.

II. Background And Motivation

The use of graph algorithms and GNNs in real-world applications has increased significantly over the years [22] [17] [35]. However, processing large graph datasets has become a challenge due to their size and sparsity [27]. SpMM is a fundamental operation in many transformer modules, graph algorithms, and GNNs. It requires high memory capacity and storage [18]. To address this issue, CSR and CSC formats were introduced to reduce the data elements needed to represent the original sparse matrix. The advantage of CSR and CSC formats is that they reduce storage and memory requirements, especially when the sparsity rate is high. Decompression, on the other hand, involves restoring this compactly stored matrix back to its original or a usable format for computational tasks. This two-step process ensures efficient use of storage space without compromising the ability to perform matrix operations when needed.

Although current optimization studies focus on CSR$\times$Dense or Dense$\times$Dense matrix multiplication, we studied and implemented CSR$\times$CSC multiplication to reduce the decompression/compression overhead. However, large CSR$\times$CSC multiplication faces the challenge of data reuse and LLC performance issues due to constant memory replacement in CSC B. To improve performance, we implemented the EMBARK memory manager for hybrid memory that partitions and utilizes memory system access for CSR$\times$CSC multiplication. The EMBARK memory controller with the SpMM data controller is responsible for allocating hot data into the main memory from NVM for SpMM. We explored the performance and memory access using the gem5 full system simulation [7]. The results are shown in Section VII.

III. CSR$\times$CSC and CSR multiplication

The CSR and CSC compression techniques represent non-zero elements using three formats: *values*, *colid*, and *rowid*. In the CSR format, the *values* signifies the value elements of the matrix, the *colid* indicates the column numbers of non-zero elements, and the *rowptr* points to the starting value of each row. Conversely, in the CSC format, the *values* denote the dense elements of the matrix, the *rowid* indicates the row numbers of non-zero elements, and the *colptr* points to the starting value of each column. Fig. 1 illustrates the CSR and CSC formats for the two matrices.

In an effort to minimize decompression and compression overhead, we implemented CSR$\times$CSC multiplication. The multiplication of matrix A in CSR format with matrix B in CSC format is depicted in Algorithm 1. For matrix A in CSR format, all requisite data for computation are sequentially arrayed; similarly, data access in CSC B is sequential. This presents a distinct advantage of CSR$\times$CSC multiplication over traditional matrix multiplication or CSR$\times$CSR multiplication.

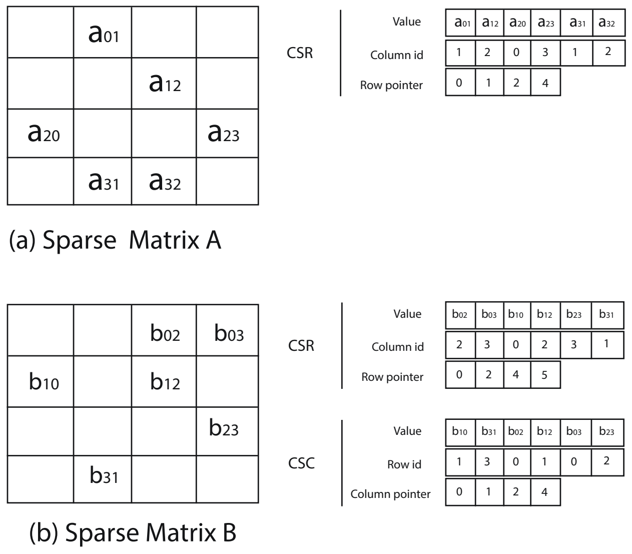

Fig. 1: Compressed format of matrix A and matrix B.

The sequential nature of memory access for CSR×CSC multiplication is illustrated in Fig. 2. Conversely, Fig. 3 displays the non-sequential data access pattern inherent to CSR×CSR multiplication.

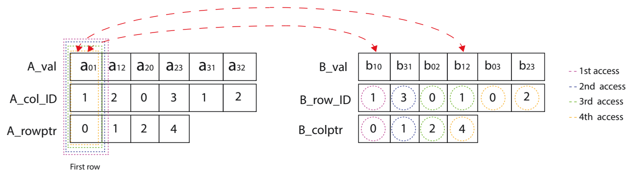

Fig. 2: CSR x CSC multiplication

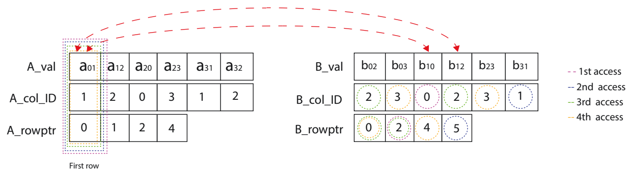

Fig. 3: CSR x CSR multiplication

$$Val_j = a\left(\left\lfloor\frac{j-1}{N}\right\rfloor, j - N * \left\lfloor\frac{j-1}{N}\right\rfloor\right) \tag{1}$$

$$Col_j = j - N * \left\lfloor\frac{j-1}{N}\right\rfloor \tag{2}$$

$$Rowptr_j = (j-1) * N \tag{3}$$

To identify the given position of a CSR matrix when CSR A *colid* is equal to CSC B *rowid*, we derived three equations for *values*, *colid*, and *rowid*. These equations are represented by notation: *j* for iteration and *N* for the number of columns. Equation 1 calculates the value at a specific matrix position. This is achieved by using the floor division method to determine the row and column indices for the non-zero element based on j, a unique identifier for every non-zero element. Equation 2 computes the column index for the non-zero element corresponding to j. This index is crucial for accessing and manipulating values efficiently in sparse matrices, as column positions give insights about the structure of data in matrix operations. Equation 3 is the calculation of a row pointer or index for a specific row. This index essentially acts as a marker, pointing to the beginning of each row in the compressed data structure. It plays a pivotal role in navigating through the matrix, especially during operations that necessitate row-wise traversals.

Algorithm 1 CSRxCSC multiplication

```
1:  for $i = 0$ to $A\_array\_size - 1$ do
2:      for $j = 0$ to $B\_array\_size - 1$ do
3:          if $A\_coltp[i] == B\_rowptrtp[j]$ then
4:              $C\_valtp$.push_back($A\_valtp[i] * B\_valtp[j]$)
5:              $C\_coltp = B\_coltp[j]$
6:              $C\_rowtp = A\_rowptrtp[i]$
7:          end if
8:      end for
9:  end for
10: if $C\_valtp \neq 0$ then
11:     for $i = 0$ to $C\_valtp\_size - 1$ do
12:         $add\_C\_val+ = C\_valtp[i]$
13:     end for
14:     $C\_val$.push_back($add\_C\_val$)
15:     $C\_col$.push_back($C\_coltp$)
16:     $C\_row$.push_back($C\_rowtp$)
17: end if=0
```

IV. Memory allocation for CSR and CSC matrices

In this section, we discussed the limitations of memory allocation for CSR and CSC formats. Furthermore, we propose a dynamic memory allocation technique tailored for these matrix representations.

A. Limitation of memory allocation for CSR matrices

Memory allocation stands as a pivotal component in the optimization of programs, with the compiler and the memory controller being instrumental in this endeavor [29] [21] [19]. Nevertheless, compilers often face challenges in efficaciously allocating data within the main memory. Fig. 4 illustrates general memory allocation for CSR A and CSC B, detailing how the compiler and main memory apportion memory to the *values*, *colid*, and *rowptr* arrays.

To refine memory partitioning, thereby diminishing cycle access and I/O traffic between DRAM and NVM, it is paramount to account for the capacity and bandwidth restrictions of the main memory, particularly when manipulating expansive, sparse matrices in the CSR format. As illustrated in Algorithm 1 for CSR×CSC multiplication, the initial row of

CSR A necessitates multiplication with every column of matrix B. Yet, when confronted with an expansive matrix, the bulk of the column data for matrix B might not be accommodated in the main memory, leading to performance bottlenecks due to heightened cycle access and I/O traffic.

B. Proposed memory allocation for CSR and CSC

To mitigate these challenges, it's essential to optimize and partition the data slated for storage in the main memory. As illustrated in Fig. 2, for CSR A, only a single row needs to be held in the main memory at a time, and this remains true until all column accesses in CSC B are finished. This data access pattern can be harnessed to refine memory partitioning, facilitating more rapid computation. Achieving this necessitates the efficient collaboration of the memory controller and the EMBARK memory management system.

Furthermore, when addressing the dilemma of CSC B's data not being accommodated within the main memory's hot data, we have enhanced our approach. Given the data-intensive nature of this situation, there's a pronounced need for advanced job scheduling by the EMBARK memory controller to oversee the data transfer between the main memory and the NVM.

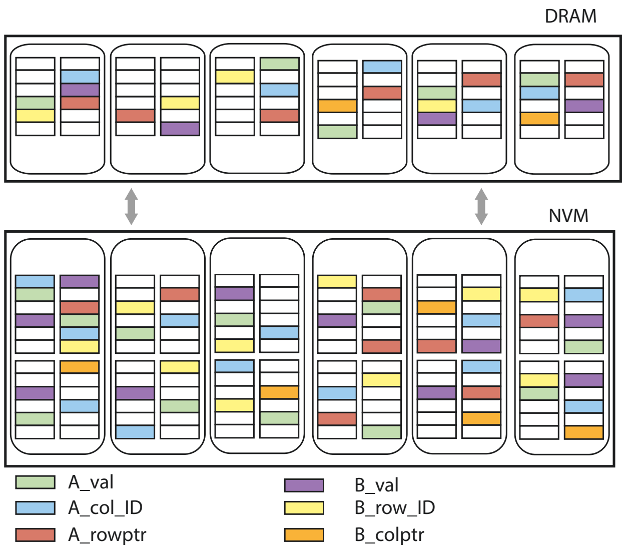

Fig. 4: General memory allocation for CSR A and CSC B

To diminish I/O access, we introduce a dynamic memory partitioning strategy for CSR A and CSC B, depicted in Fig. 5. Our method allocates the least amount of row-wise data from CSR A in memory, ensuring more available space for the column-wise data of CSC B within the main memory. The memory requirement for CSR A is contingent upon matrix A's sparsity rate and the maximum number of dense elements in each row. Subsequent data can be stored in the NVM or secondary storage, keeping it there until the multiplication for each row with all of CSR B's columns is finalized for a particular cycle.

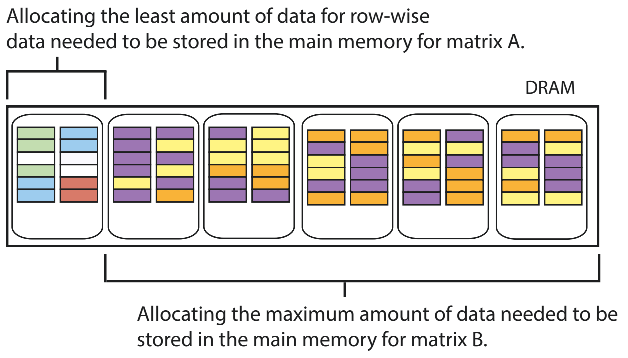
Allocating the least amount of data for row-wise data needed to be stored in the main memory for matrix A.
DRAM
Allocating the maximum amount of data needed to be stored in the main memory for matrix B.

Fig. 5: Memory partitioning for CSR A and CSC B

C. Optimization for CSC B memory space

To fine-tune the memory allocation for CSR B, we examined three scenarios based on the row count N and column count P of matrix B. In Case 1, where $N > P$, there's a need for a more substantial memory allocation for *colptr* due to an increase in its elements. In Case 2, with $N = P$, an equal distribution of storage for *values*, *rowid*, and *colptr* is ideal. For Case 3, where $N < P$, the *colptr* elements decrease, allowing an even allocation of memory for *values* and *rowid*. Dynamic memory allocation, tailored to these cases, promises reduced I/O access and improved execution time compared to the general memory allocation for CSR A and CSC B shown in Fig. 4.

Our advanced dynamic memory partitioning strategy aims to condense the row-wise data of CSR A in the primary memory while amplifying the columns of CSR B, thus curtailing I/O access. Moreover, refining the memory allocation for CSC B, contingent upon its rows and columns, can further accelerate the execution time.

Hot data: Highest number of column data for CSC B.
Data replacment

Fig. 6: Hot data and memory replacement for CSC B

The sparsity rate influences the element count in either the CSR or CSC format. For instance, when the row count surpasses the column count ($N > P$), there are more *colptr* elements compared to *rowid* and *value*. Such a scenario presents an opportunity to refine CSC B's memory utilization. However, given the algorithm's necessity to access all of CSC B's columns, we can apply the same optimization approach as with CSR A. If the entirety of CSC B's data cannot reside within the primary memory, hot data prioritization is essential, achievable through a rank-based page replacement strategy [26], depicted in Fig. 6.

Both CSR A and CSC B benefit from sequential data access; applying a replacement policy becomes unviable due to substantial data reuse. To efficiently preserve hot data, elements in columns with higher priority are retained in the main memory. This method minimizes the volume of data transferred from NVM back to DRAM, leading to performance gains. Our memory allocation strategy is rooted in these insights, determining the proportion of *values*, *rowid*, and *colptr* for CSC B to be stored in the main memory based on the original matrix's varied row and column counts.

V. Hybrid memory system

Research into emerging memory systems is becoming increasingly important due to the high demand for data-intensive applications. However, the limitations of transistor scalability have led to a stagnation in the annual improvement of main memory performance [25].

A. Hybrid and NVM memory technology

In the past decade, various memory technologies have been developed to bridge the gap between main memory and secondary memory. One such technology is storage class memory (SCM), which uses 3D stack technology to provide a lower cost per bit [20] [33]. However, SCM's performance is not up to main memory standards [4]. To address this issue, Hybrid memory technology has been extensively researched and implemented to bridge the gap between main memory and secondary memory. It offers a lower cost per bit compared to traditional main memory technologies [20], [33]. However, its performance is not up to the standard of main memory [4], which allowed for an increase in main memory capacity at a lower cost and reduced cycle access to the memory system by integrating the memory bus data path inside the DIMM chip [30]. The most popular hybrid memory architecture is DRAM+NVM, which is integrated with a migration policy and a memory management system [23]. The Fig. 7 represents the hybrid memory architecture.

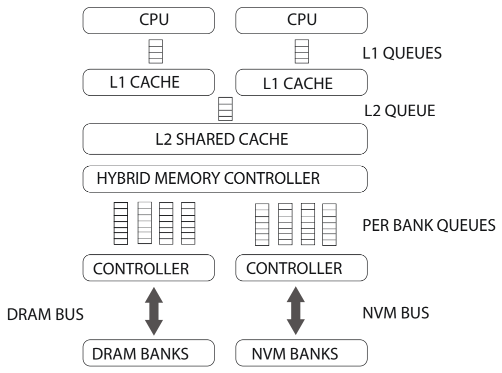

Fig. 7: Hybrid memory architecture [8]

B. Integrating the EMBARK memory controller for hybrid memory

In our EMBARK memory design, we've integrated hybrid memory technology with an aim to optimize both performance and cost-effectiveness. The architecture is structured with a memory controller dedicated to each bank queue, facilitating seamless data transfer between the DRAM and NVM banks. Moreover, distinct controllers for DRAM and NVM banks are incorporated, ensuring adept data management and migration, which in turn enhances performance and reliability. This hybrid memory setup is especially tailored for CSR$\times$CSC multiplication, addressing the demands of data-intensive applications by enhancing the performance-to-cost ratio.

VI. Embark Memory Manager

To effectively construct a memory controller suitable for this scenario, multiple factors need to be taken into account.

A. Behavior of Embark Memory Controller

1. **Memory Partitioning**: One of the primary tasks is to optimize memory partitioning to curtail I/O and cycle access. This necessitates storing minimal row-wise data of CSR A in the main memory while finetuning CSC B columns. Depending on the relationship between N and P:

- If $N > P$: Greater storage is essential for *colptr* with values and *rowid* being allocated equally.
- If $N = P$: *Values*, *rowid*, and *colptr* are allocated proportionally.
- If $N < P$: A larger allocation is required for *rowid* with equal allocation for *values* and *colptr*.

2. **Memory Allocation Optimization**: For CSR A values, colid arrays, and rowptr arrays of CSR B, it's not enough to rely on initial memory allocations made by compilers and main memory. An additional layer of optimization is crucial to mitigate I/O accesses and cycle access, thereby cutting system overheads.

3. **Handling CSC B's Data Overflows**: Provisions must be in place for instances where CSC B's hot data outstrips main memory's capacity. Such scenarios demand enhanced job scheduling by memory controllers to efficiently shuttle data between main memory and NVM.

4. **Data Access Patterns**: For optimal computation, CSR A should ideally store a solitary row until every column access is completed by CSC B, ensuring a more strategic memory allocation.

5. **Data Reuse Patterns**: With CSC B following a sequential data access model, it becomes pivotal to prioritize a greater chunk of high-reuse column elements in the main memory. This approach serves to diminish NVM to DRAM data rewrites, subsequently boosting performance.

To encapsulate these requirements, a memory controller must incorporate:

- **Dynamic Memory Allocation**: Tailoring memory resources allocation based on sparsity rate, matrix dimensions, and dense element counts.
- **Prioritized Data Storage**: Giving precedence to high-reuse data in the main memory, thus minimizing access delays.
- **Job Scheduling**: Orchestrating data transfers between main memory and NVM based on data attributes and access patterns.

- **Data Compression**: Shrinking data footprints to optimize main memory usage.
- **Cache Management**: Frequently accessed data should be strategically cached to boost performance.

Memory controllers should finesse memory partitioning, streamline data transfers between memory platforms, and elevate data reuse patterns, all aiming to reduce access overheads and uplift performance.

The EMBARK memory manager stands at the forefront of this operation, ensuring the seamless interplay between DRAM and NVM. At its core, it assesses the non-zero entries in CSR A and CSC B matrices in relation to N, making informed decisions about memory allocations for various scenarios.

B. Design of Embark memory controller

EMBARK proficiently manages the primary memory allocation for both CSR A and CSC B. Although CSR A requires a limited amount of data, CSC B demands a significant portion of hot data. Once the row-wise computation for CSR A is finalized, EMBARK smoothly transitions its data in the main memory for new row-wise data. Fig. 8 showcases the amalgamation of the hybrid memory architecture with the memory controller, further augmented by the SpMM data manager.

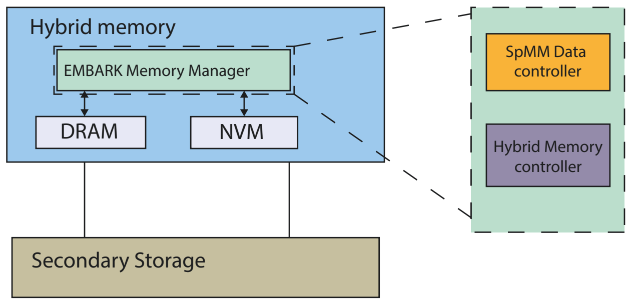

Fig. 8: Hybrid memory architecture with EMBARK memory manager

For successful implementation, the EMBARK memory manager emphasizes the parallelization of data movement with CPU computations, facilitated by the EMBARK memory controller. This controller integrates DRAM and NVM bus schedulers, a power manager, and an SpMM data controller. To adeptly oversee the memory banks, it's imperative to have two distinct hardware memory controllers: one dedicated to NVM and another to DRAM.

Although a conventional hybrid memory controller is adequate for the DRAM bank, a specialized memory controller was devised for the NVM bank. Equipped with these components, EMBARK adeptly orchestrates memory allocation and data movement, guaranteeing peak performance for extensive sparse matrix computations in CSR$\times$CSC multiplication.

VII. Evaluation

Initially, we evaluated the performance of the three algorithms on CPU clusters, with the M$\times$M specifically tested on a GPU cluster. To augment the memory efficiency of the CSR$\times$CSC multiplication, we integrated the Embark memory controller. For this enhancement, the Gem5 full system simulator [7] was employed to emulate scenarios and gather performance metrics, thereby confirming our contribution.

A. Methodology

To create an out-of-core scenario in the gem5 simulation environment, begin by establishing a physical memory system with a restricted capacity that is less than the working set of the benchmark workload. Subsequent to this, proceed to construct a disk image that integrates both an operating system and a designated swap partition. Ensure that the operating system is set up to utilize the swap space automatically once the available physical memory has been fully allocated.

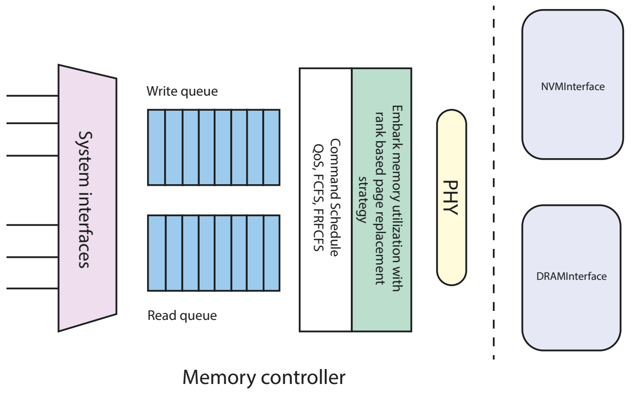
System interfaces
Write queue
Read queue
Command Schedule QoS, FCFS, FRFCFS
Embark memory utilization with rank based page replacement strategy
PHY
NVMInterface
DRAMInterface
Memory controller

Fig. 9: Gem5 architecture modification for memory controller

We employed the default parameters of Gem5's NVM for our simulation by deliberately reducing the memory capacity, we created an out-of-core scenario. This decision was made to rigorously test the efficacy of our proposed memory controller in managing its resources. Fig. 9 illustrates the modifications made to the gem5 architecture for the memory controller. In this setup, an Embark memory utilization scheme featuring a rank-based page replacement strategy has been integrated into the default memory controller.

TABLE I: SuiteSparse matrix [11] datasets

Matrix	dimension	no. of non-zero(nnz)
fxm3_6	5K X 5K	94,026
bcsstk17	10.9K X 10.9K	428,650
bcsstm25	15.4K X 15.4K	15,439
t3dl_a	20.3K X 20.3K	509,866
epb2	25.2K X 25.2K	175,027
bcsstk35	30.2K X 30.2K	1,450,163
case39	40.2K X 40.2K	144,945
ecl32	51.9K X 51.9K	380,415

This setup evaluates the system's resilience under memory-intensive tasks. The real-world matrix benchmarks used in our experiments are detailed in Table I. This paper posits that smaller datasets constrained by limited memory will

TABLE II: CPU and GPU in-core memory execution time in seconds

Matrix	CPU-CSR×CSC	CPU-CSR×CSR	CPU-M×M	GPU-M×M
fxm3_6	2.25	1.67	1556.787	119.28
bcsstk17	207.67	0.187E6	18528.05	1128.9
bcsstm25	30.61	13334.86	68628.36	3312.26
t3dl_a	1597.61	0.765E6	0.169E6	7187.73
epb2	372.74	0.51E6	0.27E6	14474.44
bcsstk35	5866.74	0.481E6	0.49E6	23617.6
case39	0.166E6	6.17E6	1.421E6	2004.14
ecl32	1684.37	3.94E6	3.13E6	56393.3

CSR×CSC - Compressed Sparse Row × Compressed Sparse Column
CSR×CSR - Compressed Sparse Row × Compressed Sparse Row
M×M - Matrix × Matrix

demonstrate behaviors akin to larger datasets within expansive memory environments.

To streamline the engineering process, we modified the simulation instructions to mimic the EMBARK memory manager's operations. This adaptation facilitated a simulation behavior closely mirroring the actual functionality of the EMBARK memory manager. All the workload matrices are stored in a compressed matrix format. To facilitate M×M multiplication, we incorporated an additional step to handle the overhead associated with converting the compressed matrix to a row-column-wise matrix, and this overhead has been added to our experiment results.

B. Evaluated Schemes

In our experiment, we evaluate the following algorithms in CPU, GPU, and Gem5 simulator:

- ***CPU-CSR×CSC, CPU-CSR×CSR, and CPU-M×M***: These tests utilized a CPU Intel® Xeon® Gold 6240R operating at a frequency of 2.40GHz. The CPU features 48 cores, an in-core memory capacity of 128GB, 32K L1 Instruction/Data cheches, 1024K L2 cache, and 36608K L3 cache. Tests were performed on the UCF Stoke cluster.
- ***GPU-M×M***: In these tests, matrix-by-matrix multiplication (M×M) evaluations were conducted on a Tesla V100-PCIE-16GB GPU using CUDA version 12.2. The tests took place on the high-performance UCF Newton cluster.
- ***M×M-Baseline***: This baseline was established using a conventional setup in the gem5 simulator with the x86-64 ISA architecture. The setup features an out-of-order configuration with 4 cores and uses the NVMInterface NVM_2400_1x64 for out-of-core operations. The cache is configured with 64K L1 Instruction/Data cheches and 128K L2 cache.
- ***CSR×CSC-EMBARK***: This configuration utilized the gem5 simulator with x86-64 ISA architecture in an out-of-order configuration with 4 cores. It operates with an NVMInterface NVM_2400_1x64 for out-of-core operations, integrated with the behavior of the EMBARK memory manager. The cache is configured with 64K L1 Instruction/Data cheches and 128K L2 cache.
- ***CSR×CSR***: Tests under this category were conducted using the gem5 simulator featuring x86-64 ISA architecture in an out-of-order setup with 4 cores. It operates with an NVMInterface NVM_2400_1x64 for out-of-core operations. The cache is configured with 64K L1 Instruction/Data cheches and 128K L2 cache.

C. Experimental Results and Analyses

In Table II, we evaluate the in-core memory performance of the CSR×CSC algorithm in comparison with the CSR×CSR algorithm and the M×M algorithm, contrasting their efficiencies on a real-world CPU against the GPU implementation of the M×M. Based on the data presented, the CPU's CSR×CSC method consistently surpasses the GPU's M×M in terms of execution speed.

From Table II, it's evident that the CSR×CSR mutiplication method exhibits below average performance for SpMM operations. This inefficiency can be attributed to the non-sequential data access pattern for CSR B, as depicted in Fig. 3. The CSR×CSC algorithm demonstrates superior performance in comparison to the GPU's M×M execution on CUDA. In GPU-based M×M operations, a significant disadvantage emerges due to the substantial overhead incurred during the transmission of memory between the host and the device throughout the computation process. This overhead can potentially reduce performance. The performance of the CSR×CSC algorithm is dependent on the sparsity of the matrix. Specifically, the more sparse the matrix is, the more performance gains can be achieved with the CSR×CSC method. In contrast, the execution time for the M×M algorithm remains consistent regardless of the matrix sparsity.

TABLE III: Gem5 out-of-core memory execution time in seconds

Matrix	CSR×CSC-EMBARK	CSR×CSR	M×M-baseline
fxm3_6	624.81	635.42	956.30
bcsstk17	5448.56	210828.52	18928.98
bcsstm25	16414.38	296608.49	26595.70
t3dl_a	37372.23	3.99E6	60851.34
epb2	70711.66	11.37E6	0.14E6

The Table III offers insights into the out-of-core execution times of three matrix multiplication algorithms across five distinctive matrices: 'fxm3_6', 'bcsstk17', 'bcsstm25', 't3dl_a', and 'epb2'. Among the methods, CSR×CSC-EMBARK consistently emerges as the most efficient, often registering the shortest execution times. Conversely, the CSR×CSR technique frequently tallies the lengthiest durations, with especially stark disparities observed in matrices like 'bcsstk17', 'bcsstm25', 't3dl_a', and 'epb2', where its execution time towers by orders of magnitude over the others. The M×M algorithm typically occupies an intermediary position, with its execution durations mostly lying between the other two. Notably, in the 'fxm3_6' matrix, the three methods present closely-packed execution times. Furthermore, for the 'epb2' matrix, we observed a notable decrease in execution time when utilizing the M×M method compared to its performance with other matrices. This phenomenon is attributed to the high level of sparsity present in the matrix. While the high sparsity reduces the number of computations needed for the CSR×CSC and CSR×CSR methods, the M×M method always involves the same number of elements. This suggests that CSR×CSC and CSR×CSR are generally better options than M×M, especially for very sparse matrices. In essence, while CSR×CSC-EMBARK exhibits consistent efficiency. For matrices with dimensions below 5K×5K, the performance difference between the CSR×CSC and CSR×CSR multiplication algorithms is negligible.

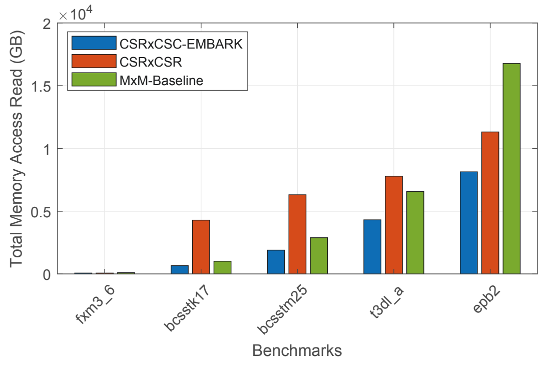

Fig. 10: Total memory access read in gigabytes (GB) for different benchmarks using CSR×CSC-EMBARK, CSR×CSR, and M×M-Baseline multiplication methods. Each bar represents the total memory access read for a specific matrix under the respective multiplication method.

In Fig. 10 for three SpMM techniques: CSR×CSC, CSR×CSR, and M×M, across five benchmark datasets namely 'fxm3_6', 'bcsstk17', 'bcsstm25', 't3dl_a', and 'epb2'. Analyzing the patterns, the CSR×CSC algorithm consistently demands the least memory write access across all benchmarks. In stark contrast, the CSR×CSR algorithm exhibits considerably higher memory write needs for most datasets, with an exceptionally high demand in 'bcsstk17', 'bcsstm25', and 't3dl_a'. The M×M-Baseline memory requirements are generally intermediate, although it outperforms CSR×CSR in the 'epb2' dataset. This occurrence is attributed to the high level of sparsity present in the matrix. While the high sparsity reduces the number of elements that need to be stored and read from the main memory for the CSR×CSC and CSR×CSR methods, the M×M method always involves the same number of elements.

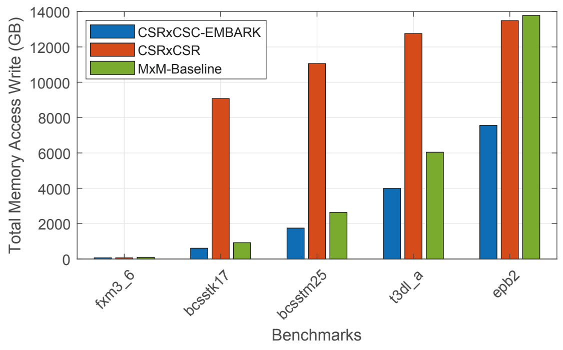

Fig. 11: Total memory access written in gigabytes (GB) for different benchmarks using CSR×CSC-EMBARK, CSR×CSR, and M×M-Baseline multiplication methods. Each bar represents the total memory access write for a specific matrix under the respective multiplication method.

In Fig. 11 data offers a comparative analysis of memory access writes across three distinct algorithms: CSR×CSC-EMBARK, CSR×CSR, and M×M, spread over five benchmarks: 'fxm3_6', 'bcsstk17', 'bcsstm25', 't3dl_a', and 'epb2'. For the 'fxm3_6' benchmark, the memory writes for CSRxCSC-EMBARK and CSR×CSR algorithms are almost on par, while the M×M algorithm exhibits a noticeably higher memory access rate. In the 'bcsstk17' context, the CSR×CSR algorithm shows a pronounced increase in memory usage, substantially outpacing both the CSR×CSC-EMBARK and MxM algorithms. When assessing the 'bcsstm25' dataset, the trend continues with the CSR×CSR algorithm registering the highest memory writes, followed by M×M-Baseline, with CSR×CSC-EMBARK trailing behind. The 't3dl_a' benchmark maintains a similar pattern, with CSR×CSR taking the lead in memory writes, and the other two algorithms showing more modest figures in comparison. For the 'epb2' matrix, the distinctions become less clear-cut, with all three algorithms showcasing closer memory write values. In essence, the CSRxCSR algorithm consistently demonstrates a heightened memory write across the majority of the benchmarks, hinting at potential challenges in its memory efficiency in specific scenarios compared to its counterparts.

D. Limitation of CSR×CSC mutiplication

CSR×CSC multiplication algorithm has a complexity of *O(nnz(A) * nnz(B))*, where *nnz(A)* and *nnz(B)* represent the number of nonzero elements in matrices A and B, respectively. This complexity can be high when both matrices have a large number of nonzero elements, resulting in increased computation time. If the sparsity rate is high, the number of non-zero elements in the resulting matrix is significantly lower than in traditional matrix multiplication. This reduction in the number of non-zero elements leads to faster computation and less memory usage.

However, if the sparsity rate is low, CSR×CSC multiplication may not be the most efficient approach, as the number of non-zero elements in the resulting matrix increases. In this case, alternative methods such as CSR x dense matrix or M×M may be more suitable. Therefore, it is important to consider the sparsity rate and the characteristics of the matrices when selecting the appropriate matrix multiplication method.

VIII. Related work

MatRaptor is a novel solution for sparse-sparse matrix multiplication acceleration, which employs a unique row-wise product approach. This technique enhances the computational speed and efficiency of sparse matrix operations, particularly those featuring two sparse matrices. Primarily focused on hardware accelerators, matrix storage format optimizations, or leveraging graphics processing units(GPU), MatRaptor provides by targeting the specific challenges of sparse-sparse matrix scenarios. This methodology represents a substantial advancement in the field, contributing to improvements in various domains that heavily rely on sparse matrix calculations [28].

GE-SpMM presents a solution for SpMM on GPUs, with specific applicability to GNNs. The method harnesses the power of GPU architectures to perform efficient, scalable, and high-speed SpMM operations, which are critical for the functioning and performance of GNNs. GE-SpMM overcomes the limitations of existing techniques, which often struggle with irregular memory access patterns and workload imbalances associated with sparse computations on GPUs. This approach contributes substantially to the field, improving the speed and effectiveness of GNNs and other applications that heavily rely on SpMM operations [18].

Gustavson's algorithm [15] is a classical approach to SpMM, with numerous improvements and variants proposed over the years. Ballard et. al. [5] presented a communication-minimizing algorithm for parallel SpMM, whereas Liu and Vinter [24] proposed an efficient storage format, CSR5, for cross-platform SpMM. Existing approaches focus on improved computation speed or storage space efficiency for in-memory execution. Unfortunately, many real-world SpMM applications are out-of-core. The system will trigger a page fault if an in-memory matrix element is missed. The subsequent event handler loads data from NVM storage into the DRAM/main memory, which could incur a significant delay from disk access.

IX. Conclusion and Future Work

SpMM has found extensive applications in various domains, including graph analytics, neural networks, electronic control systems, and mathematical analysis. In this work, we introduced a memory architecture tailored to enhance the performance of SpMM, specifically focusing on CSR×CSC matrix multiplication. Our objective was to minimize the overhead associated with compressing and decompressing matrices. The EMBARK memory manager has been instrumental in harnessing the optimal performance of main memory during CSR×CSC multiplication. As a result, we achieved a significant reduction in the execution time of CSR×CSC multiplication, with an average improvement of 46.44%. We employed the Gem5 full system simulator to validate the efficacy of our design, simulating the behavior of the EMBARK memory manager. A performance comparison of CSR×CSC multiplication against traditional matrix multiplication and CSR×CSR multiplication revealed notable findings. Specifically, as the sparsity rate decreases, the performance of CSR×CSC multiplication tends to deteriorate. As part of our future endeavors, we aim to integrate the EMBARK memory controller within the Gem5 full system simulator. Additionally, we are exploring the possibility of developing a GPU-based CUDA version of CSR×CSC multiplication.

Acknowledgments

This work was sponsored in part by the U.S. National Science Foundation (NSF) under Grants 1907765, and 2028481.

References

[1] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 971–985.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 336–348.

[3] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 781–792.

[4] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring storage class memory with key value stores," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013, pp. 1–8.

[5] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 193–204.

[6] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus," *IBM research report RC24704*, no. W0812–047, 2009.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[8] S. Bock, B. R. Childers, R. Melhem, and D. Mossé, "Characterizing the overhead of software-managed hybrid main memory," in *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2015, pp. 33–42.

[9] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 133–146.

[11] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[13] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, “The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale,” *SIAM review*, vol. 60, no. 4, pp. 808–865, 2018.

[14] D. Fujiki, N. Chatterjee, D. Lee, and M. O’Connor, “Near-memory data transformation for efficient sparse matrix multi-vector multiplication,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–17.

[15] F. G. Gustavson, “Two fast algorithms for sparse matrices: Multiplication and permuted transposition,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[16] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré, “Caffe con troll: Shallow ideas to speed up deep learning,” in *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, 2015, pp. 1–4.

[17] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” *Advances in neural information processing systems*, vol. 30, 2017.

[18] G. Huang, G. Dai, Y. Wang, and H. Yang, “Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–12.

[19] B. Jun and D. Shin, “Workload-aware budget compensation scheduling for nvme solid state drives,” in *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*. IEEE, 2015, pp. 1–6.

[20] H. Kim, M. P. Sah, C. Yang, and L. O. Chua, “Memristor-based multilevel memory,” in *2010 12th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA 2010)*. IEEE, 2010, pp. 1–6.

[21] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, “Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers,” in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 2010, pp. 1–12.

[22] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.

[23] D. Knyaginin, G. N. Gaydadjiev, and P. Stenstrom, “Crystal: A design-time resource partitioning method for hybrid main memory,” in *2014 43rd International Conference on Parallel Processing*. IEEE, 2014, pp. 90–100.

[24] W. Liu and B. Vinter, “Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

[25] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, “Challenges and future directions for the scaling of dynamic random-access memory (dram),” *IBM Journal of Research and Development*, vol. 46, no. 2.3, pp. 187–212, 2002.

[26] L. E. Ramos, E. Gorbatov, and R. Bianchini, “Page placement in hybrid memory systems,” in *Proceedings of the international conference on Supercomputing*, 2011, pp. 85–95.

[27] S. Sakr, “Processing large-scale graph data: A guide to current technology,” *IBM Developerworks*, vol. 15, 2013.

[28] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, “Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 766–780.

[29] S. Udayakumaran and R. Barua, “Compiler-decided dynamic memory allocation for scratch-pad based embedded systems,” in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, 2003, pp. 276–286.

[30] D. Ustiugov, A. Daglis, J. Picorel, M. Sutherland, E. Bugnion, B. Falsafi, and D. Pnevmatikatos, “Design guidelines for high-performance scm hierarchies,” in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 3–16.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[32] B. Wheatman and H. Xu, “Packed compressed sparse row: A dynamic graph representation,” in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[33] J. J. Yang, M. D. Pickett, X. Li, D. A. Ohlberg, D. R. Stewart, and R. S. Williams, “Memristive switching mechanism for metal/oxide/metal nanodevices,” *Nature nanotechnology*, vol. 3, no. 7, pp. 429–433, 2008.

[34] J. Zhang, M. Kwon, D. Gouk, S. Koh, N. S. Kim, M. Taylan Kandemir, and M. Jung, “Revamping storage class memory with hardware automated memory-over-storage solution,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 762–775.

[35] T. Zhao, X. Zhang, and S. Wang, “Graphsmote: Imbalanced node classification on graphs with graph neural networks,” in *Proceedings of the 14th ACM international conference on web search and data mining*, 2021, pp. 833–841.