MeNDA: A Near-Memory Multi-way Merge Solution for Sparse Transposition and Dataflows

Siying Feng
fengsy@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Xin He
xinhe@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Kuan-Yu Chen
knyuchen@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Liu Ke
ke.l@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

Xuan Zhang
xuan.zhang@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

David Blaauw
blaauw@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Trevor Mudge
tnm@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Ronald Dreslinski
rdreslin@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

ABSTRACT

Near-memory processing has been extensively studied to optimize memory intensive workloads. However, none of the proposed designs address sparse matrix transposition, an important building block in sparse linear algebra applications. Prior work shows that sparse matrix transposition does not scale as well as other sparse primitives such as sparse matrix vector multiplication (SpMV) and hence has become a growing bottleneck in common applications. Sparse matrix transposition is highly memory intensive but low in computational intensity, making it a promising candidate for near-memory processing. In this work, we propose MeNDA, a scalable near-DRAM multi-way merge accelerator that eliminates the off-chip memory interface bottleneck and exposes the high internal memory bandwidth to improve performance and reduce energy consumption for sparse matrix transposition. MeNDA adopts a merge sort based algorithm, exploiting spatial locality, and proposes a near-memory processing unit (PU) featuring a high-performance hardware merge tree. Because of the wide application of merge sort in sparse linear algebra, MeNDA is an extensible solution that can be easily adapted to support other sparse primitives such as SpMV. Techniques including seamless back-to-back merge sort, stall reducing prefetching and request coalescing are further explored to take full advantage of the increased system memory bandwidth. Compared to two state-of-the-art implementations of sparse matrix transposition on a CPU and a sparse library on a GPU, MeNDA is able to achieve a speedup of 19.1×, 12.0×, and 7.7×, respectively. MeNDA also shows an efficiency gain of 3.8× over a recent SpMV accelerator integrated with HBM. Incurring a power consumption of only 78.6 mW, a MeNDA PU can be easily accommodated by commodity DIMMs.

1 INTRODUCTION

As a fundamental primitive in many important application domains such as graph analytics, machine learning, and scientific computation [3, 8, 17, 20, 22, 38, 41, 45, 47, 55, 58], Sparse Basic Linear Algebra Subprograms (SpBLAS) are notoriously memory intensive due to the irregular memory access pattern. Recently, there has been a surge in customizing hardware accelerators near memory to tackle sparse BLAS applications such as sparse gathering [2, 24, 30], sparse matrix vector multiplication (SpMV) [2, 42, 52], and graph analytics [1, 12, 36, 57, 60]. However, none of these works address sparse matrix transposition.

Sparse matrix transposition converts a sparse matrix stored in the column-major order to the row-major order or vice versa. It is an essential building block for a wide range of applications, such as biconjugate gradient [18], standard quasi-minimal residual [19] and algebraic multigrid methods [53]. In addition, many recent graph analytics frameworks adaptively switch between different

representations of the dataflow, which requires either frequent sparse matrix transposition on-the-fly, or multiple copies of the input graph in different orders [43].

Merge sort is a common approach for sparse matrix transposition [49]. Some recent near-memory processing (NMP) proposals implement outer product based SpMV by adopting a reduction tree to merge sort the partial columns [2, 42]. But these designs targeting SpMV cannot perform sparse matrix transposition for two reasons. First, reduction trees in these systems usually perform merge sort based on the row indices of matrix elements, and thus do not care about the order of the column indices, while sparse matrix transposition needs to take into account the order of both indices. Second, unlike SpMV, which outputs a dense vector, sparse matrix transposition outputs a sparse matrix, which is irregular and requires much higher output bandwidth. To support sparse matrix transposition, all these issues need to be addressed.

In contrast to many other sparse primitives, sparse matrix transposition involves no arithmetic operations but integer comparisons to reorder the nonzero elements. Therefore, the performance of sparse matrix transposition highly depends on the attainable memory bandwidth. However, the effective system bandwidth that can be utilized by transposition is restricted both by the theoretical peak bandwidth that the memory interface can provide and the contention at the memory interface, which is confirmed by the roofline model and the scalability analysis presented in Sec. 2.2. All these constraints make sparse matrix transposition a promising candidate for NMP because NMP exposes the high internal memory bandwidth of memory devices and avoids the contention bottleneck at the memory interface. Instead of integrating accelerators with 3D/2.5D-stacked memory devices, in this paper we focus on a DIMM-based design for its cost-efficient capacity scaling, which is critical for workloads involving large datasets.

Designing a near-DRAM solution for sparse matrix transposition poses four unique challenges. First, for lack of reduction, sparse matrix transposition requires high bandwidth for both input fetching and output streaming. Hence, sending the output directly to the host like prior sparse gathering proposals [2, 24] is not feasible. Second, due to the large dataset size and the limited on-chip storage, recent CPU implementations [49] for transposition transfer intermediate data back-and-forth between the host and the main memory, exhibiting a significant amount of memory traffic. Because near-DRAM accelerators have more strict area restrictions and consequently even less area for SRAM, reducing the amount of intermediate data transfer is more difficult. Third, performing parallel transposition on multiple concurrent processing units (PUs) is non-trivial. To exploit the high internal bandwidth, accelerators are usually employed in the buffer chip of a DIMM beside each rank. Thus communications across PUs in different DIMMs need to go through the off-chip memory interface, which is prohibitively expensive and can easily become the performance bottleneck [46]. Finally, near memory transposition puts additional requirements on the data layout. The transposition process should not change the data representation and should allow easy access to the matrix non-zero elements (NZs) as the standard compressed formats after transposition. These requirements together make designing an efficient and scalable PU with minimal modifications to the commodity DIMM hardware a challenging task.

To tackle these challenges, we propose MeNDA, an scalable NMP solution for sparse matrix transposition. The key component of MeNDA is a lightweight PU featuring a hardware merge tree deployed in the buffer chip of a DIMM. The merge tree is designed to be very wide to reduce the number of merge sort iterations, which is proportional to the amount of intermediate data transfer, and supports seamless execution of multiple rounds of merge sort to minimize stalls in execution. Techniques including stall reducing prefetching and request coalescing are also explored to further improve the memory bandwidth utilization. MeNDA proposes a novel data layout to avoid communications between PUs and keep a consistent compressed format for both the input and output matrix, enabling a software-agnostic transposition backend. The data layout also considers workload balancing to maximize parallelism and memory bandwidth utilization.

Merge sort is widely employed in sparse linear algebra applications, making MeNDA an efficient solution for many sparse dataflows. Finally, to showcase its applicability to other sparse dataflows, we illustrate how MeNDA can be adapted to perform SpMV, which is a fundamental kernel for machine learning and graph analytics [2, 17, 38, 42].

Specifically, this paper makes the following contributions:

(1) An in-depth characterization of sparse matrix transposition which unveils the memory-bound nature and the request contention bottleneck at the memory interface, motivating the adoption of NMP.
(2) A scalable NMP solution for sparse matrix transposition, MeNDA, which explores DIMM- and rank-level parallelism by placing custom PUs beside each DRAM rank. The PUs feature lightweight hardware merge trees and are enhanced with techniques including seamless back-to-back merge sort, stall reducing prefetching, request coalescing and workload balancing to fully utilize the exposed high internal memory bandwidth.
(3) Adaptation of MeNDA to SpMV, demonstrating that MeNDA is an extensible and efficient solution to multi-way merge dataflows in SpBLAS.
(4) A heterogeneous programming model to completely hide the implementation details of MeNDA from the host and enhance ease of adoption.
(5) Qualitative and quantitative analyses of the benefits and overhead of integrating MeNDA into existing designs for sparse linear algebra applications.

MeNDA is an efficient solution that can be easily integrated into the buffer chip of a commodity DIMM. Experiments show that MeNDA achieves an average speedup of 19.1× and 12.0× over scanTrans and mergeTrans on CPU, respectively, and 7.7× over cuSPARSE on GPU. Compared to a recent near-memory SpMV accelerator based on HBM, MeNDA shows an efficiency gain of 3.8×.

2 BACKGROUND AND MOTIVATION

Sparse matrix transposition is widely used in SpBLAS applications, but has received much less attention than many other sparse kernels, such as sparse matrix matrix multiplication (SpMM) and SpMV [49]. Based on the roofline model and the thread scaling analysis, sparse

matrix transposition can potentially achieve great performance benefits and energy savings from NMP since it has low computational intensity while being heavily memory bandwidth bound.

2.1 Preliminaries on Sparse Matrix Formats and Sparse Matrix Transposition

Sparse matrices are often stored in compressed formats to save storage and avoid computations on zero elements. Commonly used formats are compressed sparse row (CSR) and compressed sparse column (CSC). As shown in Fig. 1, CSR(/CSC) stores a sparse matrix in three arrays: (1) an index array for the column(/row) index of each NZ, (2) a value array for the value of each NZ, and (3) a pointer array for the start pointer of NZs of each row(/column).
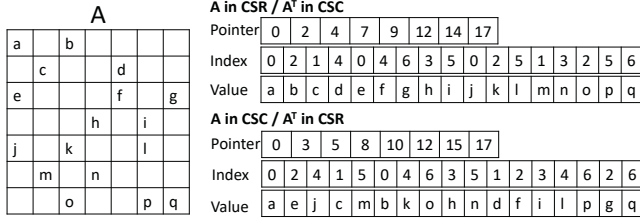
Figure 1: Sparse matrix transposition and compressed storage formats for sparse matrices.

Sparse matrix transposition transforms a M×N sparse matrix A to a N×M matrix A^T by swapping the row index and column index of each NZ. Therefore, transposing a sparse matrix is in essence equivalent to converting a sparse matrix from the CSR format to the CSC format, or the opposite. As can be seen from Fig. 1, the CSC representation of a sparse matrix A is equivalent to the CSR representation of its transpose A^T. For simplicity, we will use converting a matrix from CSR to CSC to denote general sparse matrix transposition from this point of the paper.

Sparse matrix transposition is an essential building block in both the processing and pre-processing stages of sparse linear algebra applications [49]. Typical examples are linear system solvers such as biconjugate gradient [18] and standard quasi-minimal residual [19]. Despite the fact that considering the scenario of consuming a fixed sparse matrix, the overhead of pre-processing (including sparse matrix transposition) can be amortized by iterative execution, many recent works have shown that this overhead is becoming no longer negligible as the dataset size grows and have taken this overhead into account in the evaluation [34, 54]. There are also applications that are not iterative enough to amortize the transposition overhead or have to transpose a changed sparse matrix each iteration. For example, the simultaneous localization and mapping problem requires a new information matrix at each step, and performing $A^T A$ on the new matrix dominates the execution time [15, 32].

Since Beamer *et al.* [5] first proposed a hybrid approach for Breadth First Search, many recent graph analytics frameworks have built upon this work and adopted dynamic reconfiguration between a sparse and a dense representation of the dataflow based on the active vertex set [9, 13, 17, 21, 33, 37, 43, 48, 50, 56, 59]. The dynamic reconfiguration greatly improves performance but requires the original graph A for one representation and its transpose A^T for

the other representation during execution. A common misconception regarding the transposition overhead is shown in the top bar in Fig. 2(a), *i.e.* the transposition overhead is minor compared to the execution time of an end-to-end workload and can be easily amortized. However, the reality (middle bar in Fig. 2(a)) is that recent breakthroughs in algorithms and architectures have significantly improved the performance of graph processing. Consequently, runtime transposition using a state-of-the-art implementation [49] can introduce a 126% performance overhead to a recently proposed graph framework [17]. Therefore, graph frameworks usually store more than one copy of the input graph in different formats to avoid the performance overhead of transposing the graph on-the-fly.

Although many recent efforts have been spent on optimizing sparse primitives, sparse matrix transposition has not received as much attention. As shown in Fig. 2(b), the execution time of SpMM has been improved from being comparable to that of sparse matrix transposition (OuterSPACE, 2018) to being much less than that of transposition (SpArch, 2020). These efforts only further increase the percentage of time taken by sparse matrix transposition in a workload, making it a more noteworthy bottleneck. Therefore, coming up with an efficient solution for sparse matrix transposition has become increasingly important.
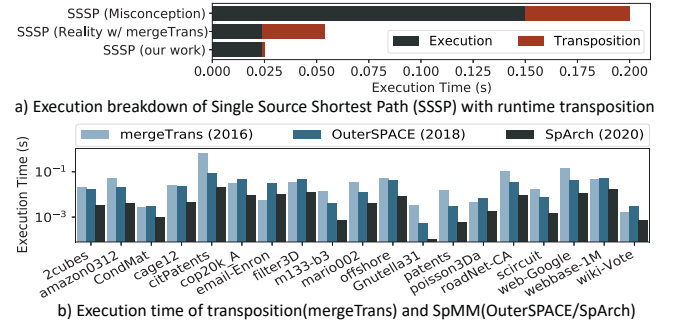
Figure 2: (a) Breakdown of SSSP execution time on CoSPARSE[17] for graph `amazon` based on common misconceptions, using `mergeTrans`[49], and using our work. (b) Execution time comparison of recent proposals for transposition (`mergeTrans`) and SpMM (OuterSPACE[38] / SpArch[58]). Recent hardware breakthroughs have greatly optimized sparse applications, *e.g.* SpMM and SpMV, whereas little research effort has been spent on accelerating sparse matrix transposition, making transposition a more evident bottleneck.

2.2 Characterizations on Sparse Matrix Transposition

To understand the bottleneck of sparse matrix transposition, we performed characterizations on `mergeTrans` [49], a merge sort based sparse matrix transposition implementation on CPUs. The methodology for these experiments is detailed in section 5.

2.2.1 Roofline Analysis. A roofline mode [51] of sparse matrix transposition is built and presented in Fig. 3(a). The throughput is measured through the number of NZs generated per second (NNZ/s), which is a metric introduced in [40]. The roofline model

Number/Color Coding and Specifications of Matrices used for Characterizations
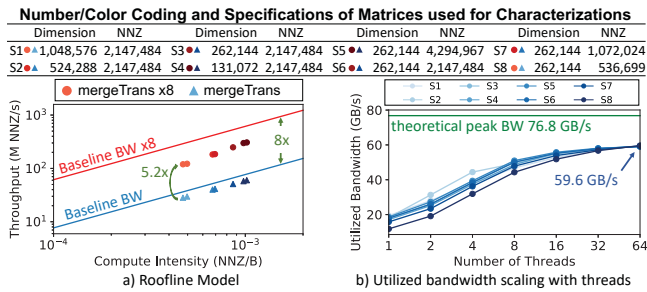
	Dimension	NNZ		Dimension	NNZ		Dimension	NNZ		Dimension	NNZ
S1●▲	1,048,576	2,147,484	S3●▲	262,144	2,147,484	S5●▲	262,144	4,294,967	S7●▲	262,144	1,072,024
S2●▲	524,288	2,147,484	S4●▲	131,072	2,147,484	S6●▲	262,144	2,147,484	S8●▲	262,144	536,699

Figure 3: (a) Roofline model of `mergeTrans` [49] running with 64 threads. Sparse matrix transposition is memory bandwidth bound because the data points are close to the "roof", *i.e.* the red and blue lines that label the peak throughputs which can be achieved when the system memory bandwidth is fully utilized. (b) Memory bandwidth utilized by `mergeTrans` with an increasing number of threads. The memory bandwidth utilization saturates before reaching maximum due to the bottleneck at the memory interface.

shows that sparse matrix transposition lies in the memory bound region. Specifically, the throughput achieved is within only 25% of the theoretical maximum and bottlenecked by the system memory bandwidth. The impact of exposing the high internal memory bandwidth on throughput is revealed by lifting the roofline by $8\times$ [24]. The throughput is improved by 4.1-$5.2\times$, which shows the potential benefit of applying NMP on sparse matrix transposition. Meanwhile, sparse matrix transposition has much lower computational intensity than common sparse routines such as SpMM and SpMV because no floating point operations are involved. *The high memory requirement and low arithmetic intensity make sparse matrix transposition a promising candidate for NMP* [11].

2.2.2   Thread Scaling Analysis. Prior work shows that the performance of state-of-the-art sparse matrix transposition implementations does not scale well with increasing number of threads [49]. To further analyze the scalability of sparse matrix transposition, we measured the utilized bandwidth with an increasing number of threads, as shown in Fig. 3(b). While the theoretical peak bandwidth, represented by the green horizontal line, is at 76.8 GB/s, the achievable maximum bandwidth is at around 62 GB/s [24]. In Fig. 3(b), the utilized memory bandwidth starts to saturate at 16 threads and reaches the maximum at 64 threads at 59.6 GB/s. In practice, little performance benefit is observed beyond 16 threads and further bandwidth saturation is undesirable due to significantly increased memory latency. *What efficient sparse matrix transposition will most benefit from is an approach that reduces memory latency and relieves the contention at the off-chip memory interface by avoiding transferring data back-and-forth between the host and the memory device.*

3   MENDA SYSTEM ARCHITECTURE

Prior work proposed two algorithms for parallel sparse matrix transposition - a count sort based algorithm (`scanTrans`) and a merge sort based algorithm (`mergeTrans`) [49]. In this work, we adopted the merge sort algorithm not only because merge sort presents

higher spatial locality but also because merge sort is widely used in sparse linear algebra [17, 38, 42]. Inspired by prior near-DRAM accelerators [4, 24], the near-memory processing units (PUs) are embedded in the buffer chips of DIMMs to minimize the modifications to commodity DRAM devices. The proposed solution is scalable as a higher throughput can be achieved by populating a memory channel with multiple MeNDA enabled DIMMs. To take full advantage of the exposed high internal memory bandwidth, the custom PU features a very wide multi-way merge tree supporting seamless back-to-back merge sort, stall reducing prefetching and request coalescing. To further improve parallelism, a novel data layout is proposed to eliminate communications and balance workloads among PUs.

3.1   Algorithm and Dataflow

MeNDA applies the merge sort algorithm to perform sparse matrix transposition. Fig. 4 demonstrates the dataflow of transposing the matrix in Fig. 1 using a 4-leaf hardware merge tree. An l-leaf merge tree merges l incoming sorted streams into a single sorted stream in a round. Since the 4-leaf merge tree does not have enough hardware resources to merge sort all matrix rows, more than one iteration is needed. As shown in Fig. 4, in iteration 0, the first four rows and the last three rows are merged subsequently. Then in iteration 1, the two sorted streams are merged into the final output. In practice, the number of iterations required to finish transposition equals $log_l N$, where l refers to the number of leaves in the merge tree and N refers to the number of non-empty matrix rows.

The input and output data are both stored in the compressed format, *i.e.* the input in CSR and the output in CSC. If the algorithm needs more than one iteration to finish, the intermediate data are stored in the coordinate format (COO). COO stores the row index, column index, and value of each NZ in three separate arrays so that accesses to the intermediate data can exploit bank-level parallelism. Due to matrix sparsity, an intermediate sorted stream may contain numerous empty rows/columns. Therefore, COO tends to take up less storage than CSR/CSC and is also easier to decode. The memory space for the input sorted streams are freed immediately after they are processed. Therefore, a runtime storage overhead of $O(l \cdot N)$ is required, where $l << N$. In contrast, storing a second copy of the matrix requires an overhead of $O(N^2)$.

3.2   Processing Unit (PU) Microarchitecture

MeNDA places PUs in the data buffer chips of DIMMs beside each rank to minimize modifications to DRAM devices and to explore DIMM- and rank-level parallelism. Each PU concurrently transposes a partition of the matrix and issues memory requests to the corresponding ranks in parallel. The effective memory bandwidth available to MeNDA thus scales with the total number of ranks.

A MeNDA PU consists of a merge tree, prefetch buffers, a controller, a request queue, and a memory interface unit (Fig. 5). In the merge tree, each processing element (PE) is connected to two child PEs through a FIFO unless it is a leaf node. An l-leaf merge tree thus has $l-1$ PEs and $log_2 l$ levels, *i.e.* at least $log_2 l$ cycles are required for data to travel from a leaf PE to the root PE. The existence of FIFOs allows each PE to pop one data packet every cycle without a critical path from the root to the leaf PEs. The root PE is connected to an
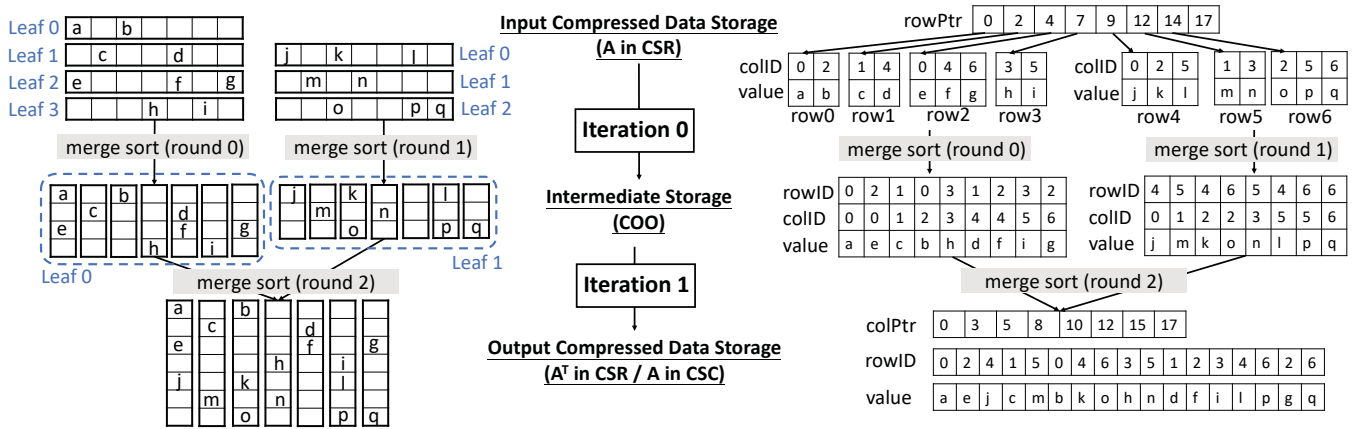
Figure 4: Dataflow of MeNDA performing transposition on the sparse matrix in Fig. 1. Each round of merge sort is executed sequentially on a 4-way merge tree. Left: The outcome of each round in the dense data structure. Right: The real data input and output of each round that are stored in memory. The input and output data are stored in the compressed data storage formats (CSR/CSC), and the intermediate data are stored in the coordinate format (COO).
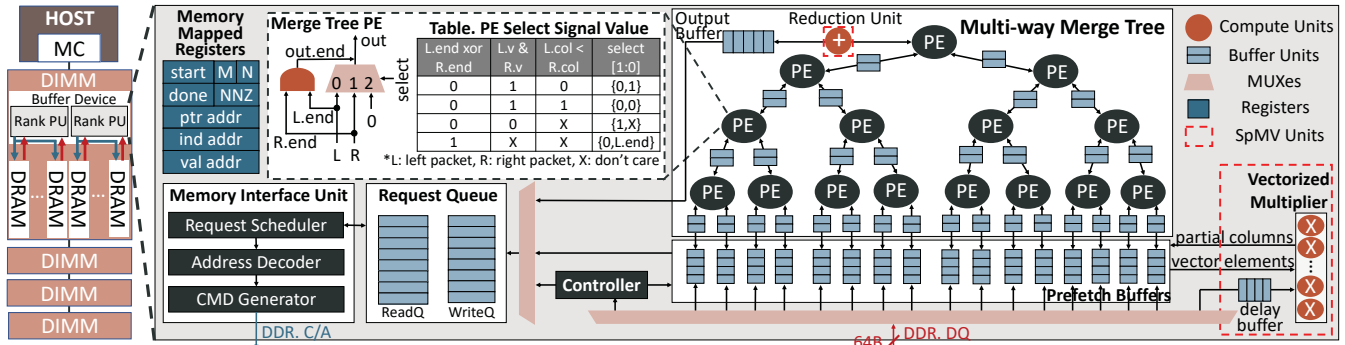

Figure 5: Architecture of MeNDA (left) and a MeNDA PU (right). A PU consists of a merge tree, prefetch buffers, a controller, a request queue, and a memory interface unit. The extra units required to support SpMV, *i.e.* a delay buffer and floating point adders and multipliers, are labelled in red.

output buffer, which allows store requests to be sent at memory block granularity (64B). Each leaf PE is connected to two prefetch buffers through FIFOs. Prefetch buffers are in charge of sending memory load requests and feeding the leaves with correct data. The controller is an FSM that assigns each prefetch buffer the start and end addresses of the corresponding sorted streams. Theoretically, in each cycle, only one load request is sent to the prefetch buffers because only one element is popped from the root PE. Similarly, only one store request is sent to the prefetch buffers to fill in the data from the memory bus because only one memory response can return each cycle. Therefore, to reduce power consumption, the prefetch buffers are implemented as multi-bank SRAM. The design goal of the merge tree is to saturate the internal memory bandwidth while fitting in the buffer chip, which, according to the evaluation, is satisfied by the current design.

Data are transferred among PEs through data packets containing a 1-bit valid signal and the 32-bit row index, the 32-bit column index, and the 32-bit value of a NZ. Only when both child PEs provide valid

packets will a PE pop the data packet with the smaller column index and send to its parent PE or the output buffer if it is the root PE. All the memory requests are sent to a request queue with separate queues for loads and stores and processed by a memory interface unit, which mimics a memory controller. The memory interface unit consists of a request scheduler that selects the request with the highest priority from the request queue, an address decoder that translates the incoming physical address to a DRAM address, and a command generator that generates DRAM commands for the chosen request. The request scheduler selects requests based on a first come first serve first ready (FCFS-FR) policy that prioritizes requests ready to launch and DRAM row hits.

3.3 Seamless Back-to-back Merge Sort

Real-world sparse matrices tend to be extremely large and sparse, causing each iteration of sparse matrix transposition, especially the first iteration, to handle many rounds of merge sort of short input streams. Hence, it is important to reduce the stalls between

different rounds of merge sort. An end-of-line signal is added to the data packet to signify the end of a sorted stream and allow seamless execution of multiple rounds of merge sort. The prefetch buffer sets the end-of-line signal when the last element of a sorted stream is sent. The PEs propagate the end-of-line signal when both child PEs set the end-of-line signal. Instead of starting a new round of merge sort after the current round of merge sort has finished, the prefetch buffers feed their PEs with data for the next round immediately after the end-of-line signal is set.
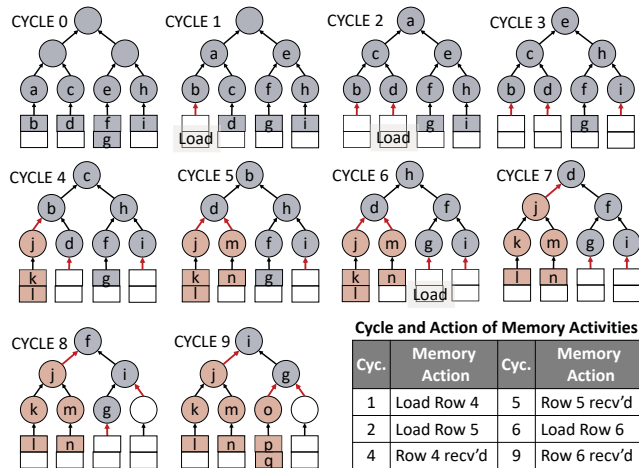

Figure 6: Timing diagram of data propagation for merge sort shown in Fig. 1 on a 4-leaf merge tree assuming a memory latency of 3 cycles. The cycle number and the corresponding memory activities are shown in the bottom right table. End-of-line signal propagation is shown with red arrows.

Fig. 6 illustrates the seamless execution of the first and second rounds of merge sort in Fig. 4. Load requests for the second round of merge sort are sent in cycle 1, 2 and 6, *i.e.* as soon as the prefetch buffers become empty. Propagating the end-of-line signals enables the merge tree to produce effective results without stalls. If the merge sort is executed one after another, in the scenario presented in Fig. 6, the first round of the merge sort ends at cycle 10 and then the three load requests for the second round are sent. The second round of the merge sort is not able to start until cycle 15 due to memory stalls, and the merge tree thus remains idle for 5 cycles. The use of the end-of-line signals not only maximizes the hardware resource utilization but also help distribute burst memory requests at the start of a new round of merge sort evenly over time.

3.4 Memory Bandwidth Utilization Optimizations

The prefetch buffers aim to make the best use of the fetched memory blocks and reduce merge tree stalls. However, even launching load requests as soon as the prefetch buffers become empty would cause the merge tree to stall while waiting for the memory responses. Therefore, **stall reducing prefetching** is proposed so that memory load requests are sent whenever a prefetch buffer can fit the requested data. Assuming a prefetch buffer can fit 16 NZs and 4

NZs have been popped to the leaf PE, if the number of NZs left in the current sorted stream is less than or equal to 4, the memory requests for the subsequent NZs will be issued. However, a prefetch buffer is not allowed to send memory requests for more NZs when there are outstanding memory requests even if the prefetch buffer can accommodate the NZs. This is because, to reduce merge tree stalls, it is more desirable to keep all prefetch buffers non-empty than serially filling each prefetch buffer until full.

While stall reducing prefetching aims at taking full advantage of the available memory bandwidth, **request coalescing** is designed to reduce the total memory traffic. Due to matrix sparsity, multiple matrix rows can be co-located in the same memory block. In this case, memory load requests for the same memory block can be sent from different prefetch buffers in the first iteration. Request coalescing avoids sending these duplicate memory requests to the memory device by checking the read request queue each time a new load request is enqueued. If a load request to the same memory block is found, the incoming request will be merged into the same request queue slot. Since the memory response is broadcast to all the prefetch buffers, merging the duplicate memory requests does not affect the functional correctness of the design and there is no need to keep track of the requesters. Because the prefetch buffers are implemented as multi-bank SRAM and the prefetch buffers that send the same memory requests are usually neighbors, the memory response from a merged request can fill multiple prefetch buffers in one cycle by interleaving neighboring prefetch buffers to different SRAM banks. Minimal additional hardware is required to support request coalescing. Specifically, a comparator is added to each entry of the read request queue to enable parallel address matching, similar to a content-addressable memory (CAM). Synthesis of the RTL model shows that the additional hardware has neglible impact on the frequency and the area of PUs.

Taking the example in Fig. 6, if row 6 of the input matrix has only one element *o*, stall reducing prefetching allows the load request for *o* to be issued in cycle 1 instead of cycle 6. Request coalescing merges this request into the prior request for row 4, making *o* available in cycle 4 instead of cycle 9.

3.5 Input Operand Co-location and Workload Balancing

In near-DRAM accelerators, communications between PUs, especially those across DIMMs, need to go through the off-chip memory interface and thus are prohibitively expensive. A common challenge is to keep all the input operands local in a single rank for a rank-level PU [11]. To avoid communications between MeNDA PUs, each PU is assigned a contiguous chunk of the sparse matrix, *i.e.* each PU is responsible for transposing a horizontal partition of the input sparse matrix. The original CSR format can then be directly used without preprocessing, and it is also easy to locate an NZ after transposition.

A naïve way to partition the sparse matrix is to use the most significant bits (MSBs) of the address to assign NZs to a rank. However, this could cause severe workload imbalance. For example, assuming a total of 8 ranks, if the 3 MSBs of the input array ranges from 000 to 100, only rank 0 to rank 4 will be assigned work while rank 5 to rank 7 remain idle throughout the execution. Since the execution

time of a PU is roughly proportional to the number of NZs (NNZ) assigned to it, an NNZ based partitioning technique is desired.
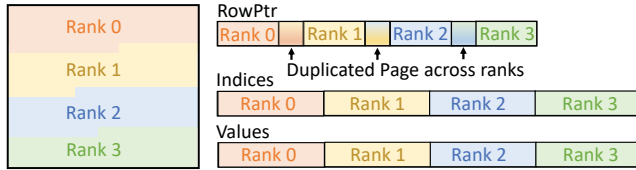

Figure 7: Matrix partitioning across 4 ranks.

The workload balancing takes place during data allocation using the technique proposed in [11]. The host first uses the number of MeNDA PUs and the NNZ of the input matrix to determine the NNZ assigned to each PU, and then allocates contiguous chunks of physical memory accordingly. To ensure that the index and value of each NZ assigned to a PU are mapped to the corresponding rank, page coloring is used to specify the rank a physical page belongs to, and thus the data assigned to a PU needs to be aligned by page. However, the same technique does not apply to the row pointer array because the rank that a row pointer belongs to depends on the matrix distribution. Therefore, the host needs to calculate the start and end row indices of the NZs assigned to a PU and then assigns the corresponding pages of the row pointer array to the target rank using page coloring. In the case that one page of the row pointer array is needed by two ranks, the page will be duplicated and each rank will have a private copy, leading to a maximum total storage overhead of $page\_size \times \#ranks$, which is negligible for typical datasets. Fig. 7 shows a partitioned sparse matrix given 4 ranks. The start and end addresses of the row pointer, index, and value arrays of each rank are written to specific memory mapped registers for PUs to calculate the target addresses during computation.

3.6 Adaptation to SpMV

Merge sort is widely used in SpBLAS. A typical example is outer product based SpMV. The merge phase of SpMV has the same dataflow as sparse matrix transposition, and thus can be implemented directly on MeNDA. As transposition does not involve floating point computations, to support SpMV, a reduction unit consisting of three pipelined floating point adders is inserted between the root PE and the output buffer. In addition, a vectorized floating point multiplier is placed next to the prefetch buffers. The additional hardware units required to support SpMV are highlighted with red rectangles in Fig. 5. When executing sparse matrix transposition, these units will be gated and incur no power overhead.

The input matrix is stored in a partitioned CSC format, which matches the format of the transposed matrix generated by our work. The reason to apply horizontal partitioning to the input matrix is that each PU would generate a partition of the final vector instead of a partial result vector. Due to the irregular distribution of sparse matrices, the horizontal matrix partition processed by a PU can have numerous empty columns. To reduce the memory loads to the pointers and vector elements that correspond to the empty columns, an auxiliary pointer array is constructed to label the memory blocks in the pointer array that contain non-empty columns.

Each time the controller sends a load request for the column pointers based on the auxiliary array, it also issues a request to fetch the vector elements that need to be multiplied with these columns. In contrast to sparse matrix transposition, the column indices are not needed for computation because all columns are eventually merged into a single vector. Hence, the space in the prefetch buffers aimed to store the column indices for transposition is now reused to store the vector elements instead. When a read request for the matrix values returns, the data is sent to the multiplier. Meanwhile, the prefetch buffers that are waiting for this memory response snoop the memory bus and send the stored vector elements to the multiplier. However, the needed vector elements could be unavailable at the moment because the load request for the vector elements is still outstanding. This is very likely due to request reordering caused by the scheduling policy and request coalescing. To deal with this situation, a delay buffer is designed to register the response and notify the request scheduler to prioritize requests for vector elements until the request needed by the registered response is served. The outputs of the multiplier are broadcasted and stored into the prefetch buffers. Note that the multiplication is only performed in the first iteration, *i.e.* the multiplier is disabled starting from the second iteration. When an element with the smallest row index is popped from the root PE, the root PE compares its index with prior outputs and merges the elements with the same index using the reduction unit. The intermediate vectors are stored in (index, value) pairs, and the output vector is stored in a dense array.

4 PROGRAMMING MODEL AND INTERFACE

MeNDA adopts a heterogeneous programming model, similar to prior NMP proposals [4, 24]. The host is responsible for memory allocation and initialization for tasks offloaded to PUs. Fig. 8(a) shows the pseudo-code of a sample graph analytics workload based on the CoSPARSE implementation [17]. In line 0-2, the host performs memory allocation and workload balancing partitioning as described in Sec. 3.5 for the input sparse matrix. The allocation functions also write the necessary metadata to the corresponding memory-mapped registers. The host can access the allocated data structures with no modifications to the original implementation because the allocation functions have taken care of the virtual to physical address mapping, which is hidden from the host.

In line 10, the host launches the sparse matrix transposition through a non-blocking function call NMP::transpose(), which sets the start signals of PUs by writing to the memory-mapped registers. While the PUs are transposing the matrix, the host can concurrently execute other kernels. Prior work has proposed techniques to efficiently allow concurrent accesses from both the host and NMP PUs [11]. However, it is still undesirable for the host to execute memory intensive workloads because sparse matrix transposition is already heavily memory bandwidth bound. Since sparse matrix transposition can easily saturate the memory bandwidth, executing another memory intensive workload on the host will only severely hurt the performance of both tasks.

Upon finishing transposition, a PU sets the finish signal and updates the addresses of the transposed matrix in the memory-mapped registers. In the case that the transposed matrix is required for subsequent code execution, NMP::wait() can be used to block

```
    // memory allocation and initialization
0   NMP::array<int> colID(NNZ, #ranks, type::NNZ);
1   NMP::array<float> values(NNZ, #ranks, type::NNZ);
2   NMP::array<int> rowPtr(N, #ranks, type::PTR);
3   array<float> vector(N), result(N);
    // CoSPARSE dense iteration
4   rowStart, rowEnd= NMP::getRowPartition(tileID());
5   for (i = rowStart; i < rowEnd; i++) {
6     for (j = rowPtr[i]; j < rowPtr[i+1]; j++)
7       partial_sum += values[j] * vector[colID[j]];
8     result[i] += partial_sum; partial_sum = 0;
9   }
10  NMP::transpose(); // sparse matrix transposition
11  NMP::wait(); // wait for transposition to finish
    // CoSPARSE sparse iteration
12  colPtr, rowID, values = NMP::getAddr(tileID());
13  v_tasks = CoSPARSE::getVectorTasks();
14  for (v : v_tasks) {
15    head = colPtr[v];
16    headList.push(rowID[head], values[head]);
17  } // construct a sorted list of head elements
    ......
```

(a) Sample pseudo-code of CoSPARSE　　　(b) CoSPARSE Microarchitecture
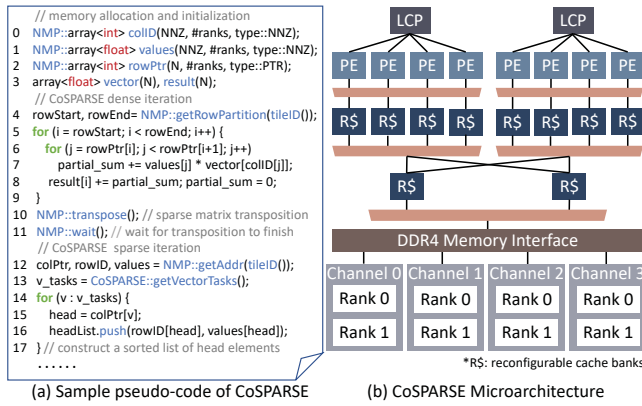
*R$: reconfigurable cache banks

Figure 8: (a) Sample pseudo-code of CoSPARSE using the programming interface of MeNDA and (b) the microarchitecture of the hardware substrate of CoSPARSE with 2 processing tiles and 4 PEs per tile.

the host execution until the transposition finishes, as shown in line 11. NMP::wait() is implemented similar to a conditional variable, which gets notified to resume the host execution as soon as the finish signals of all the PUs are set. After transposition, each rank will hold a horizontal partition of the sparse matrix stored in CSC. To access the data in a column, NMP::getAddr(i) is used to obtain the start addresses of the data arrays in rank i (line 12).

4.1　Integrating MeNDA with Existing Platforms

The programming interface of MeNDA aims at minimizing the modifications to the standard compressed storage format of sparse matrices so that minimal code changes are required to integrate MeNDA. The potential performance overhead of integrating MeNDA comes in two ways. First, the proposed data layout assigns each rank with a contiguous chunk of the sparse matrix with the same NNZ. This requires modifications to the address mapping and support from the page table of the operating system. Second, after transposition, the sparse matrix is stored in multiple horizontal partitions in CSC, which needs the host implementation to adapt to the partitioned data storage. To access an entire column, the host needs to access the sub-column in each rank.

To analyze the performance overhead, we implemented MeNDA on CoSPARSE [17], a recent graph analytics framework on a reconfigurable hardware substrate [39]. An architecture overview of CoSPARSE is shown in Fig. 8(b). CoSPARSE performs SpMV in inner product using row-major COO for the dense iterations, and outer product using CSC for the sparse iterations. To apply MeNDA, the dense iteration implementations are the same except that the memory address mapping is different. For the sparse iterations, since CoSPARSE uses preprocessing that performs horizontal partitioning based on NNZ, CoSPARSE can directly use the post-transposition data format and save preprocessing overhead with minor modifications to the implementation. Assuming a CoSPARSE system of A tiles and B PEs per tile and where there are R DRAM ranks in total, for simplicity, we let tile $A/R \times i$ to $A/R \times (i+1)-1$ work on the horizontal partition in rank i.
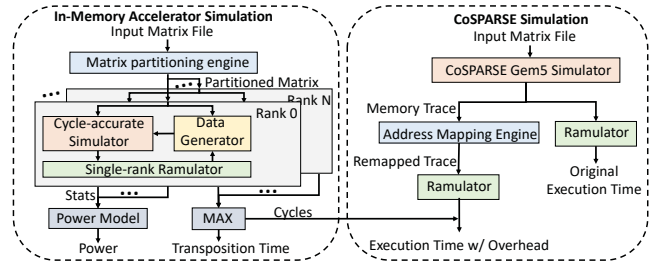

Figure 9: Experimental methodology for MeNDA.

Table 1: Parameters of Ramulator and MeNDA.

Ramulator CPU Parameters					
L1	32KB	L2	256KB	LLC	3MB
Cache	64B block size, 8-way associative, 16 MSHR entries				

Ramulator DRAM Parameters	
Standard	DDR4_2400R
Orginization	4Gb_x8
Scheduling	32-entry RD/WR queue, FRFCFS_PriorHit
Timing Parameters	tRC=55, tRCD=16, tCL=16, tRP=16, tBL=4, tCCDS=4, tCCDL=6, tRRDS=4, tRRDL=6, tFAW=26

Processing Unit Parameters			
Frequency	800 MHz	Number of Leaves	1024
No. FIFO Entry	2	No. Prefetch Buffer Entry	32
No. Read/Write Queue Entry			32
FP Units (SpMV only)	16 3-stage FP Mult, 3 2-stage FP Add		

Many recent designs use NNZ based partitioning [17, 42] and thus similar implementations can apply. Even if the host needs to access each DRAM rank to access and process a column, for graph analytics workloads, the sparse iterations access only a small subset of columns, and the dense iterations usually take up the majority of the total execution time (Fig. 11). Therefore, there are many use cases that would benefit from MeNDA without introducing a significant performance overhead.

5　EXPERIMENTAL METHODOLOGY

This section details the experimental methodology that is used to characterize mergeTrans and evaluate MeNDA.

5.1　Simulation Methodology

To model the performance of MeNDA, we designed a cycle-accurate simulator and connected the memory interface to Ramulator [29], as shown in Fig. 9 (left). The system parameters are shown in Tab. 1. The area and power estimations are based on the synthesis of an RTL model of the PU in 40nm using Synopsys design compiler.

Characterizations on mergeTrans The roofline model and the thread scaling analysis (Fig. 3) are built through trace simulation of mergeTrans[49] on Ramulator. We created a trace generator that collects the memory trace and ran the traces in cpu mode of Ramulator with a custom implementation of barrier synchronization to improve simulation accuracy. The parameters used in Ramulator are shown in Tab. 1.

Table 2: Specifications of CPU and GPU baselines.

Platform	Specifications
CPU	AMD Ryzen Threadripper 2990WX, 32 cores/64 threads at 3.0-4.2 GHz, 128 GB DDR4 memory @ 68.3 GB/s, 213 mm^2 (12 nm)
GPU	NVIDIA Tesla V100, 5120 CUDA cores at 1.25 GHz, 16 GB HBM2 memory at 900 GB/s, 815 mm^2 (12 nm)

Table 3: Specifications of Synthetic Uniform* (N#) and Power-law†(p#) Matrices.

Matrix	Dimension	NNZ	Matrix	Dimension	NNZ
N1/P1	262,144	3,435,973	N5/P5	524,288	8,388,608
N2/P2	262,144	1,717,986	N6/P6	1,048,576	8,388,608
N3/P3	262,144	858,993	N7/P7	2,097,152	8,388,608
N4/P4	262,144	429,496	N8/P8	4,194,304	8,388,608

*Generated by randomly sampling NZs until NNZ is reached.
†Generated using GenRMat(Dimension,NNZ,0.1,0.2,0.3) (snap.py).

Integration with CoSPARSE The performance impact of integrating MeNDA is estimated on CoSPARSE [17] assuming a system size of 8×16, *i.e.* 8 tiles with 16 PEs per tile. As shown in Fig. 9 (right), the memory trace is collected using the gem5 simulator [6, 7] and then processed by a memory re-mapping engine based on the strategy described in Sec. 3.5. Both the original and the re-mapped memory trace are then executed on Ramulator in dram mode to obtain the performance of CoSPARSE after integrating MeNDA.

5.2 Baseline and Benchmarks

We evaluate MeNDA against scanTrans and mergeTrans from [49] on the CPU and cusparseCsr2cscEx2 from cuSPARSE v11.4.0 on the GPU. The specifications of the CPU and GPU are detailed in Tab. 2. The CPU and GPU power are measured using AMDuProf and nvidia-smi, respectively. The specifications of the evaluated synthetic and real-world matrices are shown in Tab. 3 and Tab. 4, respectively. The power-law matrices are generated using SNAP RMat generator GenRMat. The real-world matrices are selected from the SuiteSparse Matrix Collection [14].

**Table 4: Specifications of SuiteSparse Matrices [14].**

Matrix Dimension,NNZ Kind	Plot	Matrix Dimension,NNZ Kind	Plot	Matrix Dimension,NNZ Kind	Plot
amazon 262K,1.23M Directed graph		ASIC_320K 321K,1.93M Circuit simulatio...		bcsstk32 44K,2.01M Structural proble...	
language 399K,1.22M Directed graph		mac_econ 206K,1.27M Economic proble...		parabolic 525K,3.67M Fluid dynamics	
rajat21 411K,1.88M Circuit simulatio...		sme3Dc 43K,3.15M Structural proble...		Slashdot0902 82K,948K Directed graph	
stomach 213K,3.02M 2D/3D problem		transient 178K,961K Circuit simulatio...		twotone 120K,1.21M Circuit simulatio...	
venkat01 62K,1.72M Fluid Dynamics		webbase-1M 1.00M,3.11M Directed graph		wiki-Talk 2.39M,5.02M Directed graph	

6 EVALUATION AND ANALYSIS

This section evaluates the performance, area and power of MeNDA for sparse matrix transposition and SpMV. In addition, the benefits of integrating MeNDA with existing designs and the optimizations proposed in Sec. 3.4 are presented. Finally, the performance impact of the matrix properties and the system size and frequency on MeNDA are studied.

6.1 Comparison with CPU and GPU Baselines

MeNDA is compared to state-of-the-art sparse matrix transposition implementations on CPU and GPU in Fig. 10. *The speedup of MeNDA over baselines comes from both the reduction in memory traffic and the improvement in memory bandwidth utilization.* Taking wiki-Talk as an example, compared to mergeTrans, MeNDA reduces the memory traffic by 11.2× while exhibiting 2.7× higher bandwidth utilization. These result from both the exposed high internal memory bandwidth and the optimizations in Sec. 3.4. *In general, MeNDA achieves higher throughput on large, less sparse matrices.* MeNDA performs better on less sparse matrices because less memory bandwidth is then spent on accessing and updating the pointer array, which does not contribute to the throughput, which is measured in NNZ/s. In the case that the number of iterations to finish transposition remains the same, MeNDA favors larger matrices as bank-level parallelism can be better exploited when there are more sorted streams to merge in the last iteration. mergeTrans and scanTrans, however, do not scale as well for large, sparse matrices, and perform the worst on wiki-Talk. Accordingly, MeNDA shows the most speedup over mergeTrans and scanTrans on this matrix.
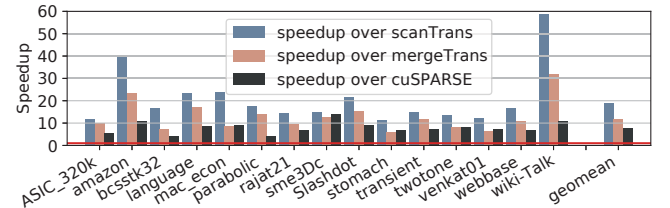
Figure 10: Speedup of MeNDA over scanTrans and mergeTrans on CPU [49] and cuSPARSE on GPU. The red line labels the speedup of 1.

The performance of cuSPARSE also favors less sparse matrices, and is sensitive to matrix distribution. bcsstk32 and sme3dc have similar dimensions and densities, but the throughput of cuSPARSE on bcsstk32 is much higher than sme3dc. Because the performance of MeNDA is not affected by matrix distribution, which is further proved in Sec. 6.6, MeNDA achieves the highest speedup over cuSPARSE on sme3dc and the lowest speedup for bcsstk32. *Overall, MeNDA achieves an average speedup of 19.1×, 12.0× and 7.7× compared to* scanTrans, mergeTrans *and* cuSPARSE, *respectively.*

6.2 Area and Power Analysis

A MeNDA PU consumes 78.6 mW at 800 MHz and takes up 7.1 mm^2 in 40 nm. The extra logic required to support SpMV adds negligible area and up to 13.8 mW power consumption. Given the estimations of prior works [4, 24] and that a typical data buffer chip takes up

100 mm^2 [35], the PU is within the power constraint and can be integrated into the buffer chip of a DIMM, introducing a small area and power overhead.

6.3 Benefits and Overhead Analysis on End-to-end Workloads

To analyze the performance benefits and overhead of integrating MeNDA into existing designs, the execution time of CoSPARSE performing SSSP algorithm on the graph amazon with and without MeNDA is illustrated in Fig. 11. Though the number of the sparse iterations is twice that of the dense iterations, the majority (87%) of execution time is taken up by the dense iterations. The potential performance overhead of MeNDA comes from two sources – the additional execution time due to the memory mapping required by MeNDA and the execution time of the transposition.
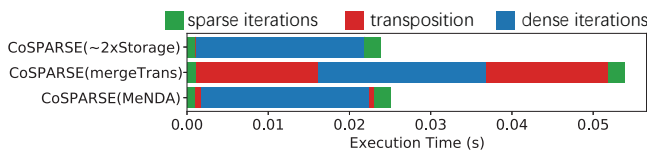

Figure 11: Execution time of SSSP on CoSPARSE for amazon without runtime transposition, with runtime transposition using mergeTrans, and with runtime transposition using MeNDA. CoSPARSE(~2xStorage) avoids runtime transposition at the cost of storing two copies of the graph [17].

Although integrating MeNDA requires the matrix partition assigned to a PU to reside in a rank, as shown in Fig. 11, the change in memory mapping has negligible impact on the execution time of the SSSP algorithm. This is because the PEs in CoSPARSE work on all matrix partitions concurrently to exploit memory-level parallelism, resulting in all the DRAM ranks being accessed in parallel. Therefore, rank-level parallelism is still well exploited. Sparse matrix transposition is launched each time CoSPARSE switches from the dense dataflow to the sparse dataflow or the opposite. In practice, sparse matrix transposition is commonly performed at most twice for a graph algorithm execution. As shown in Fig. 11, integrating MeNDA for dynamic matrix transposition decreases the transposition overhead from 126% to 5% while allowing CoSPARSE to store only one copy of the graph in DRAM, reducing the required storage by almost half, thus supporting a larger graph within a fixed DRAM size. As dataset sizes keep growing, MeNDA can prevent designs like CoSPARSE from expensive disk accesses when the DRAM devices can only fit a single copy of the graph, at the cost of introducing a minor transposition latency.

6.4 Memory Bandwidth Utilization Optimization Analysis

The execution time of MeNDA with different optimizations enabled and prefetch buffer sizes is shown in Fig. 12. A key observation is that *request coalescing greatly benefits the first iteration by reducing total memory traffic while stall reducing prefetching improves the performance of the following iterations by increasing memory bandwidth utilization.*
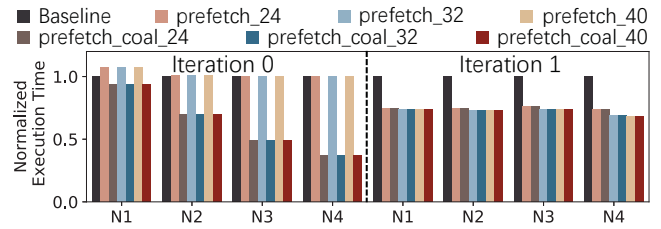

Figure 12: The execution time of MeNDA applying different optimizations normalized to that of the baseline implementation. In the legend, "prefetch" refers to stall reducing prefetching enabled, "coal" refers to request coalescing enabled, and the number refers to the size of the prefetch buffers.

Stall reducing prefetching fetches data needed in the future in advance to keep the prefetch buffers non-empty and thus reduce the stalls of the merge tree. Although stall reducing prefetching has little impact on the total amount of memory traffic, it improves the memory bandwidth utilization by 8-16%, leading to 12-16% better performance. Larger prefetch buffers enable the merge tree to send out more prefetch requests. However, little performance improvement is seen after the size of the prefetch buffer reaches 32. This is because the memory bandwidth is already saturated and the prefetch buffers are not able to send out more requests even if there are vacancies. This is also demonstrated by the fact that, when request coalescing is not enabled, stall reducing prefetching can sometimes worsen the performance of the first iteration. The reason is that the excessive prefetching requests block the critical read requests on demand, resulting in performance degradation.

Request coalescing, instead, benefits the first iteration much more than the other iterations, especially for sparser matrices. Because sparser matrices have fewer NZs per row, *i.e.* each memory block can accommodate more rows, a single memory response can fill more prefetch buffers. On the other hand, after the first iteration, sorted streams are usually much longer than a memory block, so there is little opportunity for request coalescing. Therefore, the following iterations barely benefit from request coalescing. Experiments show that request coalescing reduces the memory traffic of iteration 0 by up to 60%, leading to a maximum speedup of 2×. *Overall, stall reducing prefetching and request merging can achieve a speedup of 1.2× to 2.1× compared to a baseline with no optimizations.*

6.5 Scalability Analysis

MeNDA places PUs at DRAM rank-level, and thus the performance scales with the number of ranks. In the synthetic matrices, $N1 - N4$ have the same matrix dimensions but decreasing densities while $N5 - N8$ have the same NNZs but increasing matrix dimensions. As shown in Fig. 13, the throughput of MeNDA scales almost linearly with the increasing number of channels. The execution time of transposing $N1$ to $N4$ decreases with NNZ while that of $N5$ to $N8$ remains similar. The throughput of MeNDA decreases slightly from $N1$ to $N4$ and from $N5$ to $N8$ under a fixed number of channels. This is because when the size of the pointer array increases with the matrix dimension and becomes even larger compared to the index and value array, accessing and updating the pointer array takes

up a larger portion of the memory bandwidth usage. However, this does not contribute to the throughput, which is defined as NNZ/s, and thus results in a throughput degradation. *In summary, the throughput of MeNDA is proportional to the total number of ranks, and the execution time scales with the NNZ of the input matrix, assuming the number of iterations in the execution is fixed.*
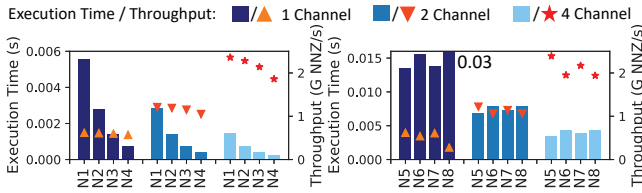

Figure 13: Execution time and throughput of MeNDA sweeping matrix size and density and the number of channels.

Transposing $N8$ on one channel is an outlier because $N8$ is the largest synthetic matrix and requires three iterations to finish while all other matrices finish within two iterations. Adding an iteration to the execution significantly increase the total memory traffic and severely degrades the throughput. Therefore, it is desirable to minimize the number of iterations in the execution. In this work, the nominal number of leaf PEs is 1024, which allows transposition to be finished within two iterations for matrices with a size up to $1024^2 \times R$, where R is the total number of DRAM ranks.

6.6 Matrix Distribution Analysis

Many real-world matrices have irregular distributions, especially those in the graph analytics domain. However, Fig. 14 shows that the performance of MeNDA is barely affected by matrix distribution. *Although in most cases, the power-law matrices take longer to transpose, the differences in execution time remain within 10%.* This can be attributed to the workload balancing strategy (Sec. 3.5), which divides tasks evenly among PUs to improve parallelism, and the seamless back-to-back merge sort feature (Sec. 3.3), which maximizes hardware resource utilization.
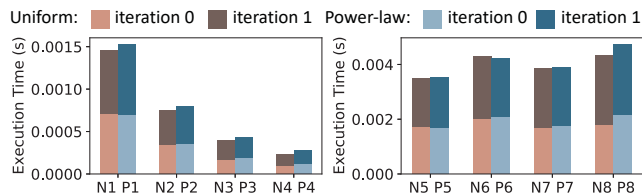

Figure 14: The execution time of the uniform matrices compared with that of the power-law matrices with the same sizes and densities.

6.7 Design Space Exploration

Fig. 15 (left) presents the execution time and energy delay product (EDP) of MeNDA under different frequencies. Because *MeNDA already saturates the memory bandwidth, increasing the system frequency beyond 800 MHz brings little performance benefit and simply*
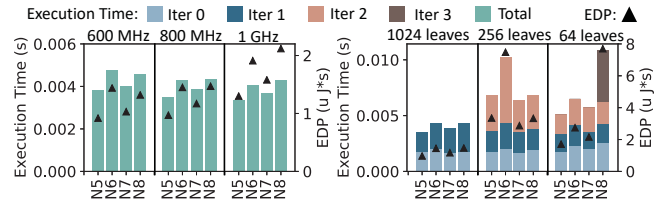

Figure 15: The execution time and energy delay product (EDP) of MeNDA sweeping the accelerator frequency (left) and number of leave PEs (right).

boosts the power consumption, resulting in a higher EDP. Although 600 MHz presents a lower EDP, this work prioritizes performance and selects 800 MHz as the nominal frequency. In a scenario where EDP is the most important metric, a lower frequency can be used at the cost of performance.

The execution time and EDP of MeNDA with merge trees of different sizes are shown in Fig. 15 (right). The size of the merge tree does not affect the throughput, but impacts the number of iterations needed to finish the sparse matrix transposition. A PU with a 1024-leaf merge tree can transpose $N5$ to $N8$ in two iterations. With 256 leaves, three iterations are needed. With only 64 leaves, $N5$ to $N7$ can still finish in three iterations but $N8$ requires four iterations. The reduction in power consumption resulted from using a merge tree with fewer leaf PEs does not offset the performance degradation caused by the increase in the number of iterations. Hence, *the PU with a 1024-leaf merge tree has not only the best performance but also the lowest EDP.*

The execution time of $N6$ is much longer than that of the other matrices on a 256-leaf merge tree. This is because $N6$ does not have enough rows that the third iteration only merges two sorted streams, and loading the two sorted streams induces many row conflicts. Although $N5$ has an even lower number of rows and the third iteration has at most two sorted streams, the majority of the NZs resides in one of the sorted streams. Therefore spatial locality is well exploited when loading the long sorted stream. $N7$ and $N8$, on the other hand, have much more rows than $N6$ and thus have more sorted streams to merge in the third iteration. The percentage of row conflicts in the third iteration is 57% for $N6$ but 43% for $N7$. This is because the bank-level parallelism exploited by loading multiple sorted streams reduces the row conflicts and enables MeNDA to transpose $N7$ and $N8$ faster than $N6$.
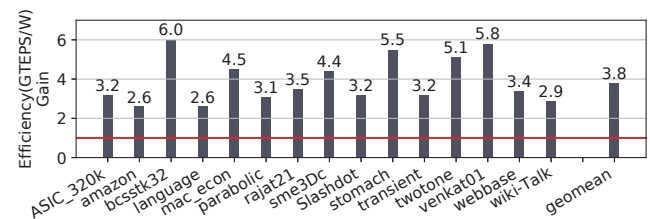

Figure 16: Energy efficiency gain of MeNDA over Sadi *et al.* [42] for SpMV.

6.8 SpMV Analysis

We evaluate SpMV against an HBM-based NMP SpMV accelerator [42]. [42] interleaves the output vector elements among reduction trees to reduce the on-chip buffer to a feasible size, taking advantage of the regular output data. However, sparse matrix transposition outputs an irregular sparse matrix, which has an unknown number of elements per row/column. Therefore, [42] cannot perform sparse matrix transposition without introducing frequent synchronization and large on-chip buffers, which will severely degrade the performance. While [42] is a monolithic design with a high peak throughput saturating the memory bandwidth of four HBM stacks, MeNDA features lightweight PUs that can be integrated into commodity DIMMs, which has better capacity scalability than HBM devices. For a fair comparison, we use giga traversed edges per second (GTEPS) per bandwidth (GB/s) as the performance metric. As [42] achieves 0.049 GTEPS/(GB/s) on average, MeNDA achieves a comparable average iso-bandwidth throughput of 0.043 GTEPS/(GB/s) with a maximum of 0.073 GTEPS/(GB/s). For efficiency gain, we scale our power to match the technology while keeping the performance because the performance of MeNDA is limited by the memory bandwidth instead of the system frequency. Overall, MeNDA presents an average improvement of 3.8× in efficiency (GTEPS/W) (Fig. 16).

7 RELATED WORKS

Near-DRAM Accelerators In recent years, many near-DRAM accelerators have been proposed to accelerate memory bandwidth bound workloads and save data transfer energy. Chameleon integrates coarse-grain reconfigurable architectures (CGRAs) into the data buffer chip on load-reduced DIMMs [4]. Inspired by Chameleon, TensorDIMM [30] and RecNMP [24] place accelerators in the DRAM buffer devices to optimize sparse embedding operations in recommender systems. The performance benefits of RecNMP are further demonstrated on AxDIMM, an FPGA-based NMP prototyping and evaluation platform [25]. Fafnir identifies the limitations of TensorDIMM and RecNMP and proposes a near-DRAM reduction tree consisting of custom PEs for sparse gathering [2]. GraFboost [23] and MetaStrider [44] are sort-reduce accelerators. GraFboost [23] targets datasets that exceed DRAM capacity and reside in flash-based systems. The intermediate data are reduced by more than 80% before written back to improve latency and flash lifetime. MetaStrider [44] deploys merger units and metadata storage at HBM memory controllers and interleaves data by indices at bank-level to achieve memory-level parallelism. In sparse matrix transposition, however, there is no data reduction. More importantly, data interleaving can cause output data fragmentation and create difficulties in quickly locating specific NZs post-merge. In summary, none of the above designs can perform sparse matrix transposition efficiently as is.

There are also designs placing accelerators at bank (group) level to further exploit the inherent parallelism in DRAM devices [10, 16, 26–28, 31]. However, these designs are mostly used for element-wise or multiply-and-accumulate operations because they require all input operands to sit within a specific bank (group). This is infeasible for sparse matrix transposition as it would pose challenges not only to restricting the required input operands to reside in a bank (group) but also to locating elements in the output matrix after sparse matrix transposition.

HMC/HBM accelerators for SpMV and graph analytics Apart from near-DRAM accelerators, plenty of designs have been proposed to tightly integrate computation logic with 3D/2.5D-stacked memory devices to optimize sparse linear algebra applications, such as SpMV and graph algorithms [1, 12, 36, 42, 52, 57, 60]. These designs usually involve communications between NMP cores, which are prohibitively expensive for near-DRAM accelerators. Besides, HBM/HMC devices often suffer from limited capacity whereas capacity scalability is critical in sparse linear algebra workloads.

None of the aforehead mentioned works address sparse matrix transposition, nor can they be used to perform sparse matrix transposition, including those designs featuring near-memory reduction trees that can compute SpMV [2, 23, 42, 44]. However, based on the insights in the prior works, we identify sparse matrix transposition as a promising candidate for NMP because of its low arithmetic complexity and high memory bandwidth requirements.

8 CONCLUSION

MeNDA is a scalable solution to near-DRAM multi-way merge for sparse dataflows, including sparse matrix transposition and SpMV. A MeNDA PU features a high-performance merge tree enhanced with techniques to maximize bandwidth utilization. To ease the deployment of MeNDA, a heterogeneous programming model is designed and showcased by integrating MeNDA to a recent graph analytics framework. Overall, MeNDA achieves an average speedup of 19.1× over `scanTrans` and 12.0× over `mergeTrans` on CPU and 7.7× over `cuSPARSE` on GPU for sparse matrix transposition, and shows an average efficiency gain of 3.8× over an HBM-based SpMV accelerator. Incurring a power overhead of 78.6 mW per PU, MeNDA can be accommodated by commodity DIMMs, introducing a small area and power overhead.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[2] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. 2021. FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 908–920. https://doi.org/10.1109/HPCA51647.2021.00080

[3] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. Alrescha: A lightweight reconfigurable sparse-computation accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 249–260.

[4] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783753

[5] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[7] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. 2006. The M5 Simulator: Modeling Networked Systems. *IEEE Micro* 26, 4 (2006), 52–60. https://doi.org/10.1109/MM.2006.82

[8] Azzedine Boukerche and Carl Tropper. 1998. A distributed graph algorithm for the detection of local cycles and knots. *IEEE Transactions on Parallel and Distributed Systems* 9, 8 (1998), 748–757.

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[10] Benjamin Y Cho, Jeageun Jung, and Mattan Erez. 2020. Accelerating Bandwidth-Bound Deep Learning Inference with Main-Memory Accelerators. *arXiv preprint arXiv:2012.00158* (2020).

[11] Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. 2020. Near Data Acceleration with Concurrent Host Access. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 818–831. https://doi.org/10.1109/ISCA45697.2020.00072

[12] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2018), 640–653.

[13] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*. 752–768.

[14] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[15] Frank Dellaert and Michael Kaess. 2006. Square Root SAM: Simultaneous localization and mapping via square root information smoothing. *The International Journal of Robotics Research* 25, 12 (2006), 1181–1203.

[16] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 283–295. https://doi.org/10.1109/HPCA.2015.7056040

[17] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael O'Boyle, Chaitali Chakrabarti, and Ronald Dreslinski. 2021. CoSPARSE: A Software and Hardware Reconfigurable SpMV Framework for Graph Analytics. In *58th Design Automation Conference*. ACM Association for Computing Machinery.

[18] Roger Fletcher. 1976. Conjugate gradient methods for indefinite systems. In *Numerical analysis*. Springer, 73–89.

[19] Roland W Freund and Noël M Nachtigal. 1991. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische mathematik* 60, 1 (1991), 315–339.

[20] Andrew Goldberg and Tomasz Radzik. 1993. *A heuristic improvement of the Bellman-Ford algorithm*. Technical Report. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

[21] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 17–30.

[22] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[23] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFboost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 411–424. https://doi.org/10.1109/ISCA.2018.00042

[24] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 790–803. https://doi.org/10.1109/ISCA45697.2020.00070

[25] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, Yeongon Cho, Jin Hyun Kim, Yongsuk Kwon, Kyungsoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin Sean Lee. 2021. Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM. *IEEE Micro* (2021), 1–1. https://doi.org/10.1109/MM.2021.3097700

[26] Byeongho Kim, Jongwook Chung, Eojin Lee, Wonkyung Jung, Sunjung Lee, Jaewan Choi, Jaehyun Park, Minbok Wi, Sukhan Lee, and Jung Ho Ahn. 2020. MViD: Sparse matrix-vector multiplication in mobile dram for accelerating recurrent neural networks. *IEEE Trans. Comput.* 69, 7 (2020), 955–967.

[27] Byeongho Kim, Jaehyun Park, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. 2021. TRiM: Tensor Reduction in Memory. *IEEE Computer Architecture Letters* 20, 1 (2021), 5–8. https://doi.org/10.1109/LCA.2020.3042805

[28] Heesu Kim, Hanmin Park, Taehyun Kim, Kwanheum Cho, Eojin Lee, Soojung Ryu, Hyuk-Jae Lee, Kiyoung Choi, and Jinho Lee. 2021. GradPIM: A Practical Processing-in-DRAM Architecture for Gradient Descent. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 249–262. https://doi.org/10.1109/HPCA51647.2021.00030

[29] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[30] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 740–753. https://doi.org/10.1145/3352460.3358284

[31] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56. https://doi.org/10.1109/ISCA52012.2021.00013

[32] John J Leonard, Hugh F Durrant-Whyte, and Ingemar J Cox. 1992. Dynamic map building for an autonomous mobile robot. *The International Journal of Robotics Research* 11, 4 (1992), 286–298.

[33] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence* (Catalina Island, CA) *(UAI'10)*. AUAI Press, Arlington, Virginia, USA, 340–349.

[34] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 631–643.

[35] Patrick J Meaney, Lawrence D Curley, Glenn D Gilda, Mark R Hodges, Daniel J Buerkle, Robert D Siegl, and Roger K Dong. 2015. The IBM z13 memory subsystem for big data. *IBM Journal of Research and Development* 59, 4/5 (2015), 4–1.

[36] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 457–468.

[37] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 456–471.

[38] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[39] Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, Kuba Kaszyk, John Magnus Morton, Jiawen Sun, Michael O'Boyle, Murray Cole, Chaitali Chakrabarti, David Blaauw, Hun-Seok Kim, Trevor Mudge, and Ronald Dreslinski. 2020. Transmuter: Bridging the Efficiency Gap Using Memory and Dataflow Reconfiguration. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 175–190. https://doi.org/10.1145/3410463.3414627

[40] Subhankar Pal, Dong-hyeon Park, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, Jonathan Beaumont, Kuan-Yu Chen, Chaitali Chakrabarti, Michael Taylor, Trevor Mudge, David Blaauw, Hun-Seok Kim, and Ronald Dreslinski. 2019. A 7.3 M Output

Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm. In *2019 Symposium on VLSI Circuits*. C150–C151. https://doi.org/10.23919/VLSIC.2019.8778147

[41] CA Philips. 1989. Parallel graph contraction. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. 148–157.

[42] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices Using Scalable Multi-Way Merge Parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 347–358. https://doi.org/10.1145/3352460.3358330

[43] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[44] Sriseshan Srikanth, Anirudh Jain, Joseph M. Lennon, Thomas M. Conte, Erik Debenedictis, and Jeanine Cook. 2019. MetaStrider: Architectures for Scalable Memory-Centric Reduction of Sparse Data Streams. *ACM Trans. Archit. Code Optim.* 16, 4, Article 35 (oct 2019), 26 pages. https://doi.org/10.1145/3355396

[45] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[46] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-based Near-Memory Processing with Inter-DIMM Broadcast. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 237–250.

[47] James Vlasblom and Shoshana J Wodak. 2009. Markov clustering versus affinity propagation for the partitioning of protein interaction graphs. *BMC bioinformatics* 10, 1 (2009), 1–14.

[48] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 38–52.

[49] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel Transposition of Sparse Data Structures *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 33, 13 pages. https://doi.org/10.1145/2925426.2926291

[50] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[51] Samuel Williams. 2009. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore. *ACM Communications* (2009).

[52] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 570–583.

[53] Jinchao Xu and Ludmil Zikatanov. 2017. Algebraic multigrid methods. *Acta Numerica* 26 (2017), 591–721.

[54] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[55] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

[56] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*. 183–193.

[57] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 544–557.

[58] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[59] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 301–316.

[60] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. Graphq: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 712–725.