A Technique for Improving Performance of Moderately Sparse Matrix Algorithms

Shashank Adavally, Krishna Kavi and Nagendra Gulur

Abstract. We seek answers to the following: (i) at what sparsity levels is it worth eliminating compressed representation of matrices and use dense representation of data that include both zeros and non-zero values and (ii) even if we use compressed data representation, will it be useful to expand the matrices internally to achieve high degree of parallelism. In this paper we explore the second question using a specialized load/store unit (LSU). Our LSU expands sparse matrices into dense matrices by filling rows (or columns) with zeros as needed, allowing for high degrees of parallelism (such as SIMD). The computational elements use dense matrix algorithms and perform no index computations. We explore the solution within the context of Processing-in-Memory (PIM) where several simple processing elements are included within the logic layer of a 3D-stacked memory. Our studies shows more than 30 percent speedup in performance and 80 percent power savings for sparse matrix multiplication over a baseline consisting of conventional multicore CPUs using sparse matrix programs and these gains are possible when the number of non-zero elements is greater than 30 percent (or sparsity less than 70 percent).

Keywords: Sparse Matrices, SIMD parallelism, Processing-In-Memory, 3D-stacked Memories, prefetching, Load/Store Unit

1 Introduction

Sparse matrices appear in a wide-ranging applications including scientific, graph analytics and deep neural networks. Even though most of the scientific computations involve very sparse matrices (that is, very few non-zero elements), there is evidence that the percent of non-zero elements in some applications can be 30% or higher [1]. In some deep learning algorithms (such as those based on CNN networks), sparsity (number of zero elements) varies from layer to layer, ranging from very low (that is, more dense with 70% non-zero elements or more) to very high sparsities (less than 30% non-zeros) [21]. Sparse matrices are represented using Coordinate (COO) or Compressed Sparse Row (CSR) formats. These structures only store non-zero values (and location of these non-zero values) to save memory space. Sparse matrix algorithms (for example multiplication) are more difficult to parallelize and are not as scalable as dense matrix algorithms[5] because of the computations need to locate the indexes of non-zero elements and uneven load balancing. We want to explore if and when it is

it worthwhile sacrificing memory space in order to achieve higher computational performance (and parallelism). One can either store data as dense matrices (with zero and non-zero values), or internally expand sparse representations into dense data.

We investigate this issue within the context of Processing-in-Memory (PIM) where several simple processing elements are included within the logic layer of a 3D-stacked memory [19], [23], [20], [18], [2], [22]). 3D-stacked DRAMs are becoming widely available to meet the ever increasing demands for memory bandwidths. But previous PIM processing elements still fetched one item (say 32 to 64 bits) at a time and did not fully utilize large amounts of data available in the row buffer of 3D-DRAMs (typically 2KBytes or 4KBytes). If cache memories are used, 64-byte data is fetched on each request. We propose a load/store unit (LSU) that fully exploits the row buffer sizes to "prefetch" data regardless how data elements are distributed within the row buffer. LSU relies on meta-data provided, which describes the nature of data structures involved (for example, the CSR structures). While our approach is similar to other prefetching, we go beyond traditional sequential or strided prefetches. The prefetched data is buffered with PIM processing elements and the computing elements consume the contiguous data, avoiding index computations. Our LSU expands sparse matrices into dense matrices by filling rows (or columns) with zeros as needed by relying on the CSR or COO format provided to LSU as meta data. This permits easy parallel execution of matrix algorithms since individual processing elements do not perform CSR (or COO) based index computations, thus trading-off memory space with computational complexity. Dense matrix algorithms can easily be parallelized and that is the case with dense matrices resulting from our expanded sparse matrices. This approach is likely to be beneficial when the percentage of non-zero elements is moderately large (more than 30 percent, or 70% sparsity). In general, our Load Store Unit (LSU) gains performance partly from prefetching data for computing elements, and partly from parallelism in the compute elements. In this paper we evaluate the benefits of LSU in the context of PIM based accelerators.

The rest of the paper is organized as follows. Our system architecture is discussed in Section 2. Section 3 explains our methodology. We explain our simulation procedure in Section 3.2. In Section 4, we present results and analysis of our experiments. Section 5 provides the overview of research related to this work. Finally, Section 6 contains conclusion and potential extension of this work.

2 System Architecture

Figure 1 shows the overall architecture of a PIM based system that forms the basis for our work here. The expanded view of the logic layer which contains processing elements and our load/store unit (LSU) is shown in figure 1c. LSU is comprised of 5 units.

1. Adder: Adder is used to calculate the next address of the element requested by incrementing previous address with the stride and data element size.

2. Address Structure: All the addresses of one (or more) element requests which are currently being processed are stored in this unit. Addresses which are calculated by the adder will be stored in this unit at its memory request index (which is based on its actual address). This is based on the meta data (more information will be given in later sections) regarding the data structure being used.

3. Request queue: A request, which is comprised of multiple addresses of one or more data structures will be created and stored in the request queue. The process of a request handling will be explained in a later subsection 2.2 and is similar to memory coalescing in GPUs. This queue will be cleared in FIFO order.

4. LSU Buffer: Data which is received from memory (DDR or 3D DRAM) will be stored in LSU buffer with a tag (tag is created by appending array-request-number, index from the address structure). After the tag creation, related information of this address is erased from address structure unit.

5. Sequencer: Based on the dependencies (which instruction requires this data element), this unit orders data elements and transfers the data to CPU buffer. We use multiple sequencers for Sparse matrix multiplication as shown in figure 1 to compensate for the delay in processing elements, particularly for sparse matrices. The sequencers store non-zero elements along with zeros creating dense rows and columns.

PIM core block comprises of 2 units.

1. PIM Buffer: Data from LSU is copied into PIM core buffer with tags as explained above.

2. PIM Core: This is a very simple in-order processing unit with necessary functional units as ALUs (or specialized array of Multiply-Accumulate or Fused-Multiply-ADD units). Delays used in our simulations for arithmetic operations are inherited from gem5 [4].

2.1 Programming model

Our architecture will require the user to rewrite sparse matrix algorithms to provide the metadata to LSU unit, specify the number of PIM cores to use and other information such as if and how the matrices are blocked/tiled. The metadata is somewhat similar to *Gather* and *Scatter* instructions for vector processors. For example, some vector processors include such instructions as *vector strided load* and *vector indexed load* [9]. These can be extended to include instructions like *vector COO load* by providing pointers to the COO representation of a matrix. PIM cores are simple in-order elements and they can be programmed to perform computations using PIM Instruction Set Architecture. The LSU unit along with PIM cores will rely on parallelism (SIMD) to process the matrix computations as if the data is dense (Figure 1). We assume that the host CPU will set aside buffer space to receive the results from the LSU system. Our LSU system can convert the results into sparse representations (as such CSR) before sending them to the host. Based such a programming model, the host processor will communicate

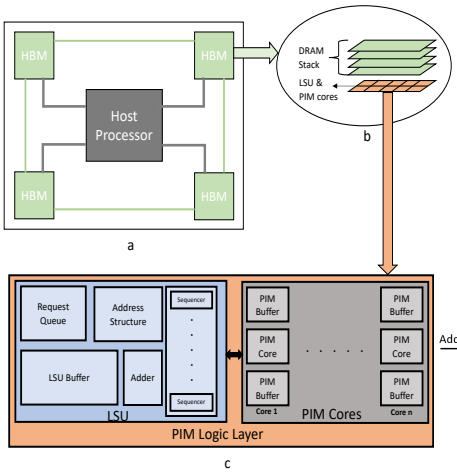

Fig. 1: SMM Architecture

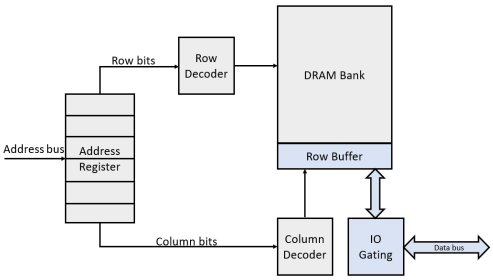

Fig. 2: Modified Address register [12]

Latency type	Description
tBL	Burst Length
tRTP	Read To Precharge
tRAS	Row Access Strobe
tRCD	Row to column Delay
tCCD	Column-to-Column Delay
tCAS	Column Access Delay
tWR	Write to sense amplifiers
tCWD	Column write delay

Table 1: DRAM Latency description [12]

with LSU and PIM cores (or a controller that manages the PIM cores) to initiate the computations. This approach is similar to GPU (or other accelerator) programming.

2.2 LSU interface with Memory

A memory request packet contains request for multiple data elements. A request is created by the address structure module in LSU, by including all data elements' addresses for one (or multiple) main request(s) that fall to the same row buffer of a memory bank, rank and channel. Memory Controller (MC) contains queues with nested entries to store memory requests in the index node or first column node and actual addresses in its corresponding index nested nodes.

To implement LSU features, we discuss minor modifications to the DRAM. As shown in Figure 2, address register has multiple entries, representing all the data items comprising non-zero elements of one DRAM row (compared to traditional systems that process one request at a time). Once a DRAM is activated, the

column-addresses from each entry of the address register will be directed to column decoder and the selected data element is pushed onto the bus. While the first data element is being pushed onto bus, the second element (at next column-address) will be decoded from the row buffer. This way, data transfer is pipelined, saving "tRAS" cycles by fetching multiple data elements, "tCCD" cycles by pipelining. In case of PIM implementation of our LSU, there is no reason to transfer columns of data one at a time on the bus since LSU can directly access the entire row buffer. Table 1 describes basic DRAM timing parameters. For our implementation, we use the following timings for accessing data.

1. Read and Precharge row [12]:

Original Latency to read one cache line and close row: tRTP + tBL - tCCD

Modified latency to read "n" data elements and precharge: tRTP + (n x (tBURST - tCCD)), if tBURST != tCCD [n is the number of elements that are needed from that row]

(or)

Modified Latency: tRTP + tCCD + (n x tBURST), if tBURST = tCCD [n is the number of elements that are needed from that row]

2. Multiple Reads:

Regular latency to read one cache line: tCAS + tCCD

Modified latency to read "n" data elements: tCAS + (n x (tCCD)) tBURST is pipelined

3. Multiple Writes:

Regular latency to write one cache line: tCWD + tBURST + tWR

Modified latency to write "n" data elements: tCWD + (n x (tBURST + tWR))

4. Write and Precharge row:

Regular latency to write and precharge: tCWD + tBURST + tWR + tRTP

Modified latency to write "n" data elements and precharge: tCWD + (n x (tBURST + tWR))

Since the combined latency is greater than RAS, no need to add precharge latency

3 Methodolody

In this section we will explain how we expand sparse matrices (using COO representation) into dense matrices so that the (PIM) cores can behave like SIMD and process computations in parallel.

3.1 Sparse Matrix Multiplication

There have been other approaches for improving performance of sparse matrix algorithms including changes to data representation, load balancing among parallel threads and GPU threads [8]. Our approach is different: we expand sparse matrices into dense matrices by filling rows and columns with zeros as needed, thus trading off space with ease of parallelization. The performance gains are

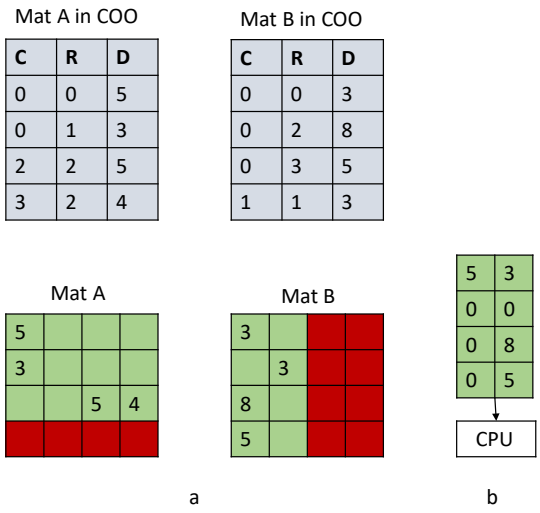

Fig. 3: SMM Expansion

due to the elimination of index computations by individual compute elements and the prefetching of data by the LSU. To minimize the size of buffers and sequencers, we expand a fixed number of rows and columns at a time and pass the data to compute elements. *We can also rely on tiling or blocking of matrices to limit the size of the buffers.* Figure 3a, shows an example where sparse matrices A, B are shown in Coordinate representation (COO).

Rows highlighted in green color of matrix A will be expanded along with the green colored column of matrix B. Figure 3b shows the expanded row-0 of matrix A and column-0 matrix B. LSU continues this process by expanding one row and one column at a time and copies the data into PIM buffers. A PIM core can start the inner-product computation when its buffer is loaded with data. At the same time, the LSU expands row-1 of matrix A and column-0 of matrix B and copies them to the next available PIM core. In this manner, the sparse matrix multiplication takes advantage of multicore parallelism and overlapping data fetch and data expansion with computations performed by PIM cores. Detailed explanation will be given in later sections.

Figure 4 shows the flow of operations for the example. Initially, matrices in sparse format are fetched with the help of metadata to LSU memory (or buffers), where they are expanded to dense format using the sequencers (In the figure we show 2 sequencers). The data from sequencers is transferred to PIM buffers for processing by the PIM cores. PIM cores (or arrays of specialized Fused-Multiply Add (FMA) units) perform computations and the final result is sent back to the LSU memory. We assume that these processing elements can skip the computation if one or both inputs are zeros. LSU converts results back to sparse format and stores them in (host) memory. The LSU and PIM cores use

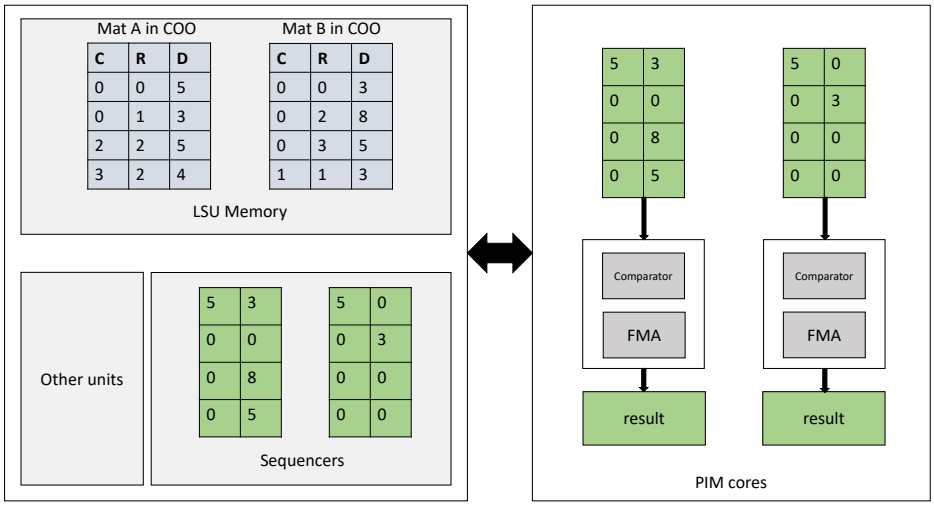

Fig. 4: Methodology

synchronizing flags that indicate when expanded data in sequencers is available for processing (see Figure 4).

Although in this paper, we place our LSU architecture in a PIM-like context, our LSU can be used as an accelerator integrated on a multi-core processors die. The only difference is that in a PIM context, LSU unit benefits from direct access to DRAM row buffers and in an accelerator context, LSU unit will have to rely on a narrow communication channel with DRAM memory.

3.2 Simulation

To evaluate the potential performance gains of our LSU architecture, we used Ramulator [15]. We used randomly generated sparse matrices with varying sizes and number of non-zero elements (or sparsities). The baseline processor configuration (based on X86) used for comaprisions is shown in Table 2. We chose Intel MKL Sparse Matrix multiplication routine [10] for comparing our approach. Using Intel PIN tool[17], we captured memory traces with randomly generated input matrices. We processed these traces through the simulator configured to implement the base configuration. To simulate our LSU system, we modified Ramulator, implementing the new memory and compute instructions for the LSU and CPU cores using new timings as described previously in Section 2. We used McPAT [16] to obtain power ratings and silicon area needed for our architecture. Our area and power analyses include all components of our LSU unit (including PIM cores and buffers). The LSU configurations used in our work are shown in table 3. The LSU requires large amounts of memory to buffer non-zero elements and then expand them into dense rows and columns. However, our PIM cores

Specification	Values
Baseline Core type	128-entry instruction window, 4 wide issue
Core count	8
Frequency	3.2 GHz
Caches	L1 - 32 KB L2 - 128 KB L3 - 2 MB

Table 2: Base configuration

Specification	Values
PIM Core type	Simple Inorder
Core count	8 - 32
Frequency	3.2 GHz
LSU Buffer	2 MB
PIM Buffers	16 KB

Table 3: LSU configuration running SMM

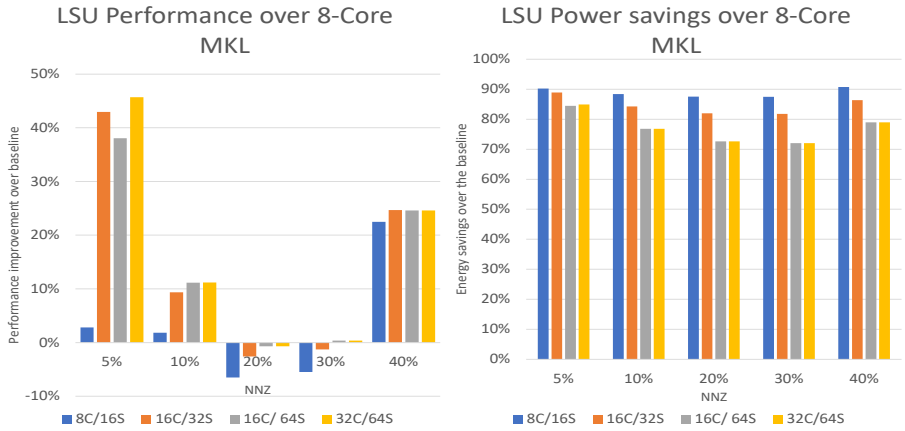

Fig. 5: 200 x 200 Performance speedup (left), Energy savings in percent (right) over baseline. (We accounted for power consumed by all LSU components)

do not use cache memories and the total buffer space needed is significantly less than the space occupied by L1 and L2 caches in conventional multicore systems. Since our goal is to show that the expanded matrices can be parallelized, we simulated with multiple PIM cores as shown in Table 3. The baseline CPU core configuration is shown in Table 2).

4 Results and Analysis

In this section we show the results of our simulations and provide an analysis of these results on our proposed architecture.

4.1 LSU for Sparse Matrix Algorithms

Our goal for sparse matrix algorithms is to utilize multiple cores and easily parallelize the matrix multiplication code (and other sparse matrix algorithms). For this purpose, our LSU expands sparse matrices (represented using either

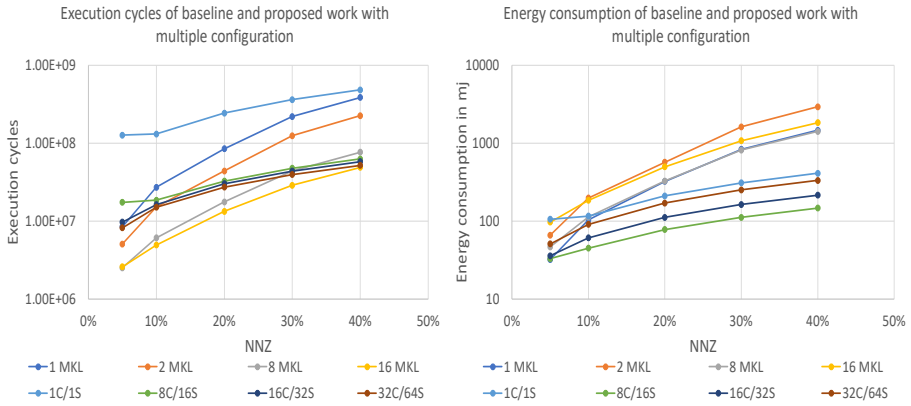

Fig. 6: 500 x 500 Execution cycles (left), Energy consumption (nj) (right) of baseline and proposed work with multiple configuration. Note that both the y-axis are in log10 scale

coordinate form or compressed sparse rows formats) by filling the matrices with both zero and non-zero values. These "dense" or expanded matrices can then easily be parallelized and individual cores do not perform index computations that involve indirect memory accesses, to determine the location of non-zero elements. We compare results obtained on our design with those obtained using Intel MKL Sparse Matrix multiplication routines [10] running on a multicore system in the base configuration. Figure 5 (left portion) shows the performance improvement for 200 x 200 matrices with different number of non-zero(nnz) elements. At 5 and 10 percent non-zero elements, LSU architecture with 16, 32 cores performed 40% and 10% better than the baseline configuration respectively. This gain in performance is due to two reasons. 1) The host system's inability to parallelize efficiently at low percent of nnz elements and threads spent most of the time on index computations to access data. 2) In LSU architecture, the amount of computations wasted due to the zeros is small. At 20 and 30 percent nnz, MKL threads will have more work to do (more non-zero values to multiply-accumulate), thus performing better than our architecture. At 40 percent nnz, even though baseline system (or host) could use parallelism, each thread is still expending time on index computations. The benefit of our methodology becomes more visible at this level of sparsity (40% non-zero values, or 60% sparsity). Each PIM core has more non-zero values to compute and fewer zero values to skip. We can see more than 20% performance gain from LSU-based system over the baseline. Figure 5 shows energy saving results. The figure shows that, even when our architecture does not show performance gains, our system shows as much as 80% energy savings over the baseline. The energy savings in in part due to the simple in-order PIM cores and in part due to eliminating index computation (and several memory accesses) by each PIM core.

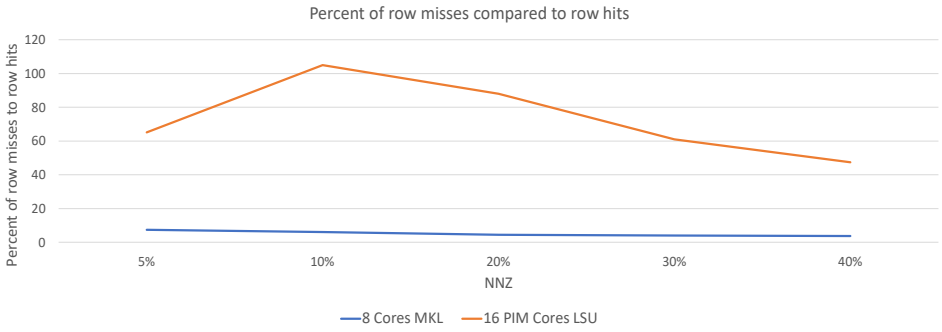

Fig. 7: Percent of row misses to row hits at different nnz for 500 x 500 matrix

Figure 6 shows the results for a 500 x 500 matrices with different number of non-zero elements (5 percent, 10 percent, 20 percent, 30 percent and 40 percent). Since the location of non-zero elements is random, the distribution of work among cores is also random. When the number of non-zero elements reaches 30 percent, LSU architecture shows better performance with 16 cores when compared to baseline with 8 X86 cores, and shows 10% performance gains. With 40 percent non-zero elements, 16 PIM core configuration achieves more than 24% performance gain over the baseline with 8 cores. Performance improvements achieved by our architecture show similar trend as the number of non-zero elements are increased beyond 40%. It should be noted that even though we use 16 PIM cores, these are very simple in-order cores, compared to the 8 out-of-order host cores.

Figure 7 show the percent of DRAM row buffer misses versus row buffer hits. As can be seen, the LSU system shows significantly lower row buffer misses when compared to that for baseline. This is the second major reason for the performance gains of our architecture (in addition to highly parallel computations of PIM cores using expanded matrices). Our LSU extracts all possible non-zero values contained in the DRAM row buffer so that the row need not have to be reactivated multiple times to satisfy requests from (baseline) CPU cores. In LSU system, the row buffer hits increases with the number of non-zero elements (or as the sparsity decreases). For such inputs, (baseline) CPU cores make more requests for data that in turn may increase the number of row activations (since it is likely that row may have been closed to satisfy other requests).

Figure 6 shows energy consumption results for 500 x 500 matrices based on our LSU configurations shown in table 3. With 5 percent non-zero elements, 16 and 32 PIM core configurations consumed slightly higher energy than baseline with 8-cores. As the number of non-zeros increase (or sparsity decreases), the LSU system shows better energy consumption over the baseline. Higher PIM core counts increase the hardware complexity and increase power consumption but the difference in the power savings reduces as the sparsity decreases (or non-

zeros increase) due to the higher degrees of parallelism that can be exploited. It should be noted that the energy is measured in terms of joules, a measurement of energy per unit time. It is directly proportional to the execution time.

Summary. Although not shown in this paper due to space limitations, we observed that with larger matrix dimensions, our LSU based architecture outperforms the baseline at higher percentage of non-zeros (for example, for 1000x1000 our architecture outperforms the baseline with 40% nonzero values). Even when there is no performance gain (with higher sparsities or very few non-zero elements), our architecture shows energy savings over the baseline. The savings are in part because PIM cores are simpler than host cores. In comparing the energy savings, our LSU unit with 16 PIM cores and 32 sequencers occupies about 21% less area than the baseline with 8-cores. Given that PIM systems are limited to 10W [23], it is possible to fit 25 PIM cores and 25 sequencers. All the buffer space needed for our LSU system is significantly smaller than the space used by L1 and L2 private caches in a modern multicore host systems.

5 Related Work

There are several studies on data prefetching. In [7], prefetching is based on calculating the stride from previous accesses and prefetching is limited to tracking the stride for one data structure. In our work, we prefetch data based on the meta data provided about each data structure. The meta-data can provide the number of elements needed and stride. The meta data may also specifiy non-uniform stride. Streambuffers [14] prefetch sequential streams of cache lines even if the fetched data is not utilized, while our work prefetches only useful data elements and supplies them to processing elements. Markov prefetching [13] is address correlation-based prefetching and stores the history of missed address streams. Based on the history of previous miss pattern, future misses are predicted and prefetched. This method does not maintain any knowledge about specific data structure strides or keep track of multiple structures simultaneously. In [11], authors prefetched data based on distance prefetching from slower memory into on-chip buffer in a heterogeneous memory architecture (consisting of faster 3D DRAMs and slower non-volatile devices such as PCM). The distance is measured in terms reuse distance. The authors propose to prefetch heavily used pages from slower non-volatile memories into faster DRAM based memories. This is in lieu of migrating pages completely into faster memories. Our prefetching focuses on gathering data elements based on the request that fall to the same DRAM row buffer. We also prefetch into buffers contained within the LSU and not into processor cache memories.

There are several works that attempt to improve the performance of sparse matrices. One paper [6] proposed a parallel sparse matrix algorithm based on SUMMA used in BLAS library and parallelized the sparse matrix multiplication, while we expanded the sparse matrices into dense representation and parallelized the expanded matrices.

Greathouse [8] proposed an algorithm, CSR-Stream to compute sparse matrix-vector multiplication for smaller rows and CSR-Adaptive algorithm to choose CSR-Stream instead traditional CSR compared to expansion from COO format and parallelizing dense matrix workloads. In [3], authors proposed a parallel Sparse Matrix-Sparse vector (SpMSpV) algorithm that stores the product of Sparse matrix-vector based on the row indices and later accumulates it, all by using buckets. In our approach, we initially expand the matrix into denser format row and column wise and compute the product.

6 Conclusions

We set out to see if there are any benefits in expanding sparse matrix data into dense data (including both zero and non-zero values) so that the compute elements can avoid complex index operations. We explored this idea within the context of Processing-In-Memory (PIM) to benefit from high bandwidth and wide row buffers available in 3D-stacked memories. We use a special hardware Load/Store Unit that is provided with meta data about the matrices. The meta data allows LSU to extract all useful data items available in an open DRAM row buffer. LSU also expands the sparse matrices into dense matrices by filling in missing zeros (along with non-zero values). This buffered and expanded dense data is then transferred to the processing elements. We evaluated our architecture for matrix multiplication. We compared our approach with a conventional multicore host implementing sparse matrix algorithms (the baseline).

Our experiments show that our approach results in performance gains over the baseline when the percentage of non-zero values exceeds 30%. For larger matrices, this threshold may be higher (say 40% non-zeros). In addition to performance gains, our approach also shows a reduction in energy consumed. The performance gains stem from both the expansion (and parallel execution of the expanded data) and prefetching of data, fully utilizing all the data available in a DRAM row buffer. The latter is shown by the significantly higher DRAM row buffer hits with our architecture when compared to the row buffer hits using conventional host processors implementing sparse matrix algorithms.

While scientific applications deal with very large matrices and the percentage of non-zeros is very low, many other emerging applications contain higher percentages of non-zeros. For example, many input sets for deep learning applications show the percentage of non-zeros exceeding 30%, and this percentage varies between 30% and 70%. Additionally, the data sets can be easily organized into smaller blocks. Thus, we feel that our approach is applicable for such application domains. We are currently investigating our LSU approach to deep learning applications. Although we placed our design in a Processing-In-Memory context, the LSU can be used as an accelerator, residing on the same CPU die. The only difference will be in terms of direct access to DRAM row buffers offered in PIM context, compared to the limited communication width offered by an off-chip DRAM.

References

1. Suitesparse matrix collection formerly the university of florida sparse matrix collection https://sparse.tamu.edu/
2. Ahn, J., Hong, S., Yoo, S., Mutlu, O., Choi, K.: A scalable processing-in-memory accelerator for parallel graph processing. In: 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). pp. 105–117 (June 2015). https://doi.org/10.1145/2749469.2750386
3. Azad, A., Buluç, A.: A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017. pp. 688–697 (2017). https://doi.org/10.1109/IPDPS.2017.76, https://doi.org/10.1109/IPDPS.2017.76
4. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (Aug 2011). https://doi.org/10.1145/2024716.2024718, http://doi.acm.org/10.1145/2024716.2024718
5. Buluc, A., Gilbert, J.R.: Challenges and advances in parallel sparse matrix-matrix multiplication. In: 2008 37th International Conference on Parallel Processing. pp. 503–510 (Sep 2008). https://doi.org/10.1109/ICPP.2008.45
6. Buluç, A., Gilbert, J.R.: Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. SIAM J. Scientific Computing **34** (2012)
7. Fu, J.W.C., Patel, J.H., Janssens, B.L.: Stride directed prefetching in scalar processors. In: Proceedings of the 25th Annual International Symposium on Microarchitecture. pp. 102–110. MICRO 25, IEEE Computer Society Press, Los Alamitos, CA, USA (1992), http://dl.acm.org/citation.cfm?id=144953.145006
8. Greathouse, J.L., Daga, M.: Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 769–780. SC '14, IEEE Press, Piscataway, NJ, USA (2014). https://doi.org/10.1109/SC.2014.68, https://doi.org/10.1109/SC.2014.68
9. Hennessy, J., Patterson, D.:
10. Intel: Intel Math Kernel Library (2003), https://software.intel.com/en-us/mkl
11. Islam, M., Banerjee, S., Meswani, M., Kavi, K.: Prefetching as a potentially effective technique for hybrid memory optimization. In: Proceedings of the Second International Symposium on Memory Systems. pp. 220–231. MEMSYS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2989081.2989129, http://doi.acm.org/10.1145/2989081.2989129
12. Jacob, B., Ng, S., Wang, D.: Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
13. Joseph, D., Grunwald, D.: Prefetching using markov predictors. IEEE Transactions on Computers **48**(2), 121–133 (Feb 1999). https://doi.org/10.1109/12.752653
14. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: [1990] Proceedings. The 17th Annual International Symposium on Computer Architecture. pp. 364–373 (May 1990). https://doi.org/10.1109/ISCA.1990.134547
15. Kim, Y., Yang, W., Mutlu, O.: Ramulator: A fast and extensible dram simulator. IEEE Comput. Archit. Lett. **15**(1), 45–49 (Jan 2016). https://doi.org/10.1109/LCA.2015.2414456, https://doi.org/10.1109/LCA.2015.2414456

16. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P.: Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In: Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 469–480. MICRO 42, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1669112.1669172, http://doi.acm.org/10.1145/1669112.1669172
17. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Languagen Design and Implementation. pp. 190–200. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065034, http://doi.acm.org/10.1145/1065010.1065034
18. Scrbak, M., Greathouse, J.L., Jayasena, N., Kavi, K.M.: Dvfs space exploration in power constrained processing-in-memory systems. In: ARCS (2017)
19. Scrbak, M., Islam, M., Kavi, K.M., Ignatowski, M., Jayasena, N.: Exploring the processing-in-memory design space. J. Syst. Archit. **75**(C), 59–67 (Apr 2017). https://doi.org/10.1016/j.sysarc.2016.08.001, https://doi.org/10.1016/j.sysarc.2016.08.001
20. Shelor, C., Kavi, K.M.: Dataflow based near data computing achieves excellent energy efficiency. In: HEART (2017)
21. Vardan Papyan, Jeremias Sulam, Y.R.M.E.: Sparsity in convolutional neural networks http://stanford.edu/ qysun/Lecture04$_S$*unQY.pdf*
22. Zhang, D.P., Jayasena, N., Lyashevsky, A., Greathouse, J., Meswani, M., Nutter, M., Ignatowski, M.: A new perspective on processing-in-memory architecture design. In: Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness. pp. 7:1–7:3. MSPC '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2492408.2492418, http://doi.acm.org/10.1145/2492408.2492418
23. Zhang, D.P., Jayasena, N., Lyashevsky, A., Greathouse, J.L., Xu, L., Ignatowski, M.: Top-pim: throughput-oriented programmable processing in memory. In: HPDC (2014)