Dynamic Partitioning Method for Near-Memory Parallel Processing of Sparse Matrix-Vector Multiplication

Dae-Eun Wi
Department of Electronic Engineering
Hanyang University
Seoul, Korea
sung661116@hanyang.ac.kr

Kwangrae Kim
Department of Electronic Engineering
Hanyang University
Seoul, Korea
kksilver91@hanyang.ac.kr

Ki-Seok Chung*
Department of Electronic Engineering
Hanyang University
Seoul, Korea
kchung@hanyang.ac.kr

***Abstract*—Near-memory processing (NMP), which places lightweight processing units near the DRAM memory, has been actively studied to speed up the execution of memory-intensive applications by reducing the amount of data traffic between the DRAM and the CPU. Sparse matrix-vector multiplication (SpMV) is a representative memory-bound kernel used in various applications such as graph analytics, scientific computing, and machine learning. There are prior works to accelerate SpMV by NMP employing a fixed partitioning scheme that hides random access of SpMV using a parallel NMP core. However, due to the various distributions of the matrix, the fixed partitioning of prior works causes a load imbalance in which a sparse matrix is unevenly allocated to processing units for NMP. To resolve this, dynamic partitioning methods to distribute matrices and vectors to the NMP processing units can be effective.**

In this paper, we propose a dynamic partitioning algorithm (DPA) that analyzes the distribution of non-zero elements in a sparse matrix to classify it into three types (even distribution, skewed distribution, and power-law distribution) and partitions the matrix according to each distribution. Our proposed distribution scheme alleviates load imbalance by up to 73% when compared to static distribution schemes, and such improvement achieves an average speed-up 1.37× (up to 1.84×) over the NMP architecture with static distribution schemes.

***Index Terms*—Sparse Matrix-Vector Multiplication, Near-Memory Processing, Dynamic Partitioning,**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is an important kernel used in various applications such as graph analytics, scientific computing, and machine learning [1]–[7]. SpMV is regarded as a memory-bound operation due to the following reasons: (i) a sparse matrix may have the size from tens of kilobytes to hundreds of megabytes and it has to be transferred from memory to the CPU for processing, and (ii) distribution of non-zero elements in the input sparse matrix and access patterns to the input vector are often irregular leading to a low spatial locality [8]–[11]. To efficiently handle memory-bound operations, including SpMV, many studies have proposed processing-in-memory (PIM) and near-memory processing (NMP) architectures [8], [9], [12]–[15]. Some of these studies proposed PIM architectures on 3D stacked memory structures such as High-bandwidth Memory (HBM) and Hybrid Memory Cube (HMC) [12], [14], [16], [17]. For instance, [13] proposed a method to process irregular sparse data in parallel using SIMD, and [12] proposed a partitioning method to alleviate load imbalance. However, the 3D-stacked-memory-based PIM architecture is a high-cost solution in terms of power consumption and latency, making it difficult to be adopted in commercial computer systems. Due to this reason, DIMM-based NMP is considered an efficient and practical solution [13], [15].

Most existing studies on partitioning input vectors and matrices for SpMV operations and storing into multiple memory ranks employ a static partitioning scheme [8], [12], [18]. The partitioning enables parallel accesses to multiple memory ranks to improve the access latency, especially when irregular memory accesses cause long latency [13], [15], [19], [20]. However, most of these studies do not try to resolve the load-balancing issue meaning that data may be unevenly distributed across multiple memory segments. In many DIMM-based NMP implementations, each rank is equipped with its own processing element to carry out some computation. When the data are unevenly distributed across multiple ranks, some processing elements may stay idle while others are very busy, leading to insufficient exploitation of parallel processing (i.e., load imbalance). This load imbalance often occurs in real SpMV applications. Fig. 1 shows the load-imbalance degree found in some applications of a benchmark suite [10]. The y-axis shows the load-imbalance degree, which is the ratio of the number of non-zero elements of the two partitions when the sparse matrix is divided vertically into two equal-sized partitions. If the degree is 1, we can say that the matrix is balanced. As shown in Fig. 1, matrices in some applications have a significantly-unbalanced distribution (average of 3.53).

In this paper, we propose a new data partitioning scheme called dynamic partitioning algorithm (DPA), which evenly distributes the matrix partitions into multiple ranks to alleviate the performance reduction due to load imbalance. DPA partitions a matrix into multiple groups with the same column range and classifies the sparse matrix according to the degree of load imbalance of each partition. Based on the distribution type,

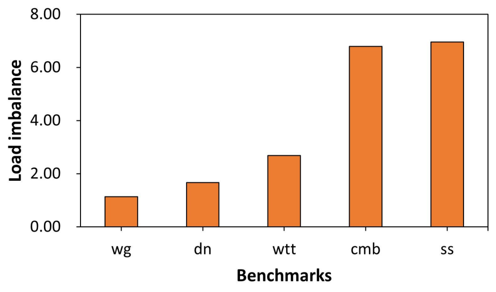


Fig. 1. Load imbalance analysis of four programs in an SpMV benchmark [10].

DPA tries to find a better partitioning by subdividing a partition into multiple row units rather than column units. Next, we suggest an operation flow of the NMP system that performs SpMV based on the DPA mapping scheme. The proposed architecture achieves an average performance improvement of 1.72× (at best 1.96×) compared to the system without partitioning.

II. Background

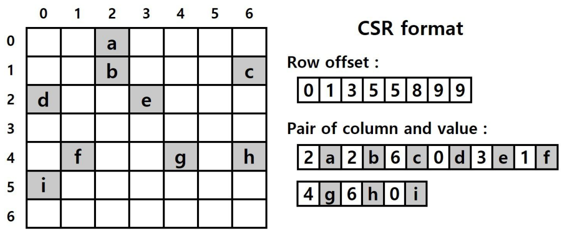
A. SpMV & CSR

SpMV is a key computation that is frequently included in various applications such as graph analytics [2] and machine learning [3]. The input sparse matrix in SpMV operations is commonly represented by a compressed format where only the non-zero elements are represented. One of the commonly used compression formats for sparse matrices is a format called compressed sparse row (CSR). Fig. 2 illustrates how a sparse matrix is compressed using the CSR format.

The N^{th} value in `Row offset` is the cumulative count of non-zero elements up to the N^{th} row, and the `Pair of column and value` stores the column index of a non-zero element and the value of the non-zero element. When carrying out an SpMV operation, the sparse matrix in the CSR format will be multiplied by an input vector without being decompressed into the original form. This paper assumes that all the input sparse matrices are compressed using the CSR format.

B. Parallel NMP architecture for SpMV operations

Fig. 3 shows the multi-rank NMP architecture of this paper and shows roughly how a compressed matrix in the CSR format will be used to carry out SpMV. Each rank has its own processing unit called `NMP core`. We use a similar method in [8] to carry out an SpMV operation when the matrix is compressed using the CSR format as follows. ① Each NMP core loads an input vector and the row offset vector. ② After loading the pair of (column, value), each NMP core carries out a multiplication between the input vector and the non-zero values of the compressed sparse matrix. ③ Each NMP core transfers the output to the host CPU after computing a partial sum. ④ The host CPU merges all the received partial sums to get the final output vector.



	0	1	2	3	4	5	6
0			a				
1			b				c
2	d			e			
3							
4		f			g		h
5	i						
6							

Fig. 2. How to compress a sparse matrix into a CSR format

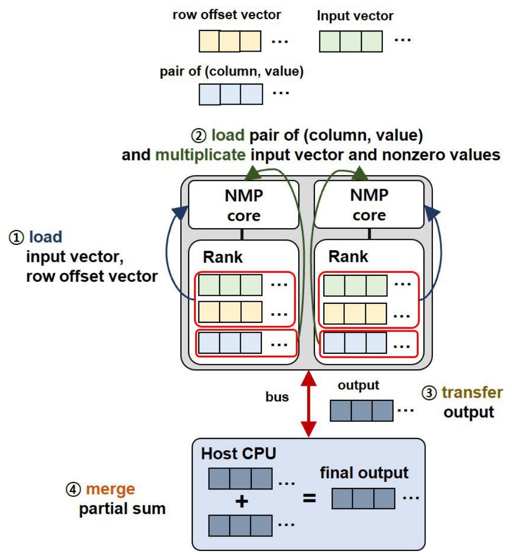
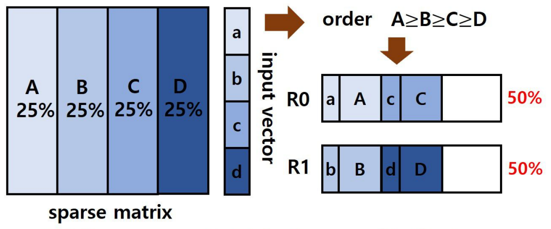


Fig. 3. An architecture for parallel NMP of SpMV operations

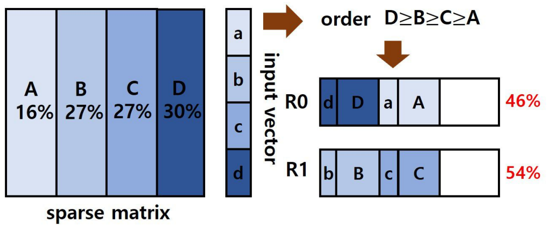
III. Proposed Method

A. Dynamic Partitioning Algorithm

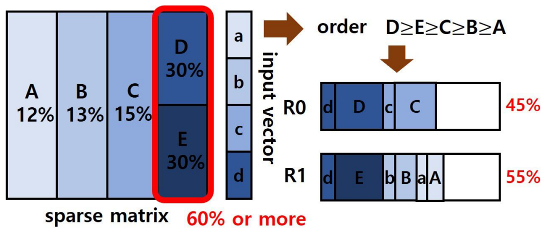
This section describes the proposed method called dynamic partitioning algorithm (DPA). Partitioning is a method for dividing the input matrix and the input vector and storing them into different memory segments so that they can be loaded to the processing units in parallel [8], [12], [15]. There are two ways to partition the input matrix and the input vector: static partitioning and dynamic partitioning. Static partitioning means the number of partitions is predetermined, and typically, the size of each partition is the same [8], [12], [18]. On the other hand, dynamic partitioning may apply a different partitioning method when sparse matrices and input vectors are saved in different ranks. If the sparse matrix has an irregular and unbalanced distribution of non-zero elements, static partitioning may suffer performance degradation due to load imbalance [12]. To apply the most appropriate partitioning method according to the degree of load imbalance of the input sparse matrix, DPA first divides the matrix into four partitions of equal size and classifies the sparse matrix according to the load imbalance degree of each partition. The types of distribution are as follows:



(a) The process of DPA in the even-distribution



(b) The process of DPA in the skewed-distribution



(c) The process of DPA in the power-law distribution which exists a group that accounts for more than 60% of the matrix

Fig. 4. The DPA method

Algorithm 1: grouping&mapping

```
1  Input: GCSR_rowoffset[0:4][], GCSR_pair[0:4][], Gnnz[0:4],
     vec[0:numRows-1], power-law_flag
2  Output: write operations
   /*Generating Group consists of vec, GCSR offset and
     pair
3  for $i = 0$ to 4 do
4      Group[i].vec, Group[i].offset, Group[i].pair
5      ← partitioned vec, GCSR_rowoffset[i], GCSR_pair[i]

   /*Ordering group and finding rank number to map each
     group
6  if $flag = 1$ then
       /*Ordering Gnnz and group index
7      order_Gnnz[] ← ordering Gnnz []
8      order_group[] ← group index based on code 7
       /*num_Rx: number of data in rank x
9      num_R0 , num_R1 ← 0
       /*First ordered group is mapped to R0
10     num_R0 ←order_Gnnz[0]
11     map_R[0] ← 0
12     for $i = 1$ to 4 do
13         if num_R0 ≥num_R1 then
14             num_R1 ←order_Gnnz[i]
15             map_R[i] ← 1

16         else
17             num_R0 ←order_Gnnz[i]
18             map_R[i] ← 0

19 else
20     except for group 4, do like codes 6-18

   /*mapping group into each rank
21 for $j = 0$ to length of map_R[] do
22     if $map\_R[j] = 0$ then
23         write 16words of group[] according to order_group[]
             sequentially R0

24     else
25         write 16words of group[] according to order_group[]
             sequentially R1
```

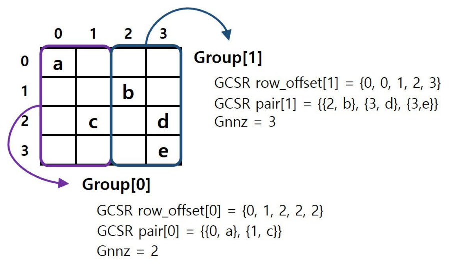

0
1
2
3
0
1
2
3
a
b
c
d
e
Group[1]
GCSR row_offset[1] = {0, 0, 1, 2, 3}
GCSR pair[1] = {{2, b}, {3, d}, {3,e}}
Gnnz = 3
Group[0]
GCSR row_offset[0] = {0, 1, 2, 2, 2}
GCSR pair[0] = {{0, a}, {1, c}}
Gnnz = 2

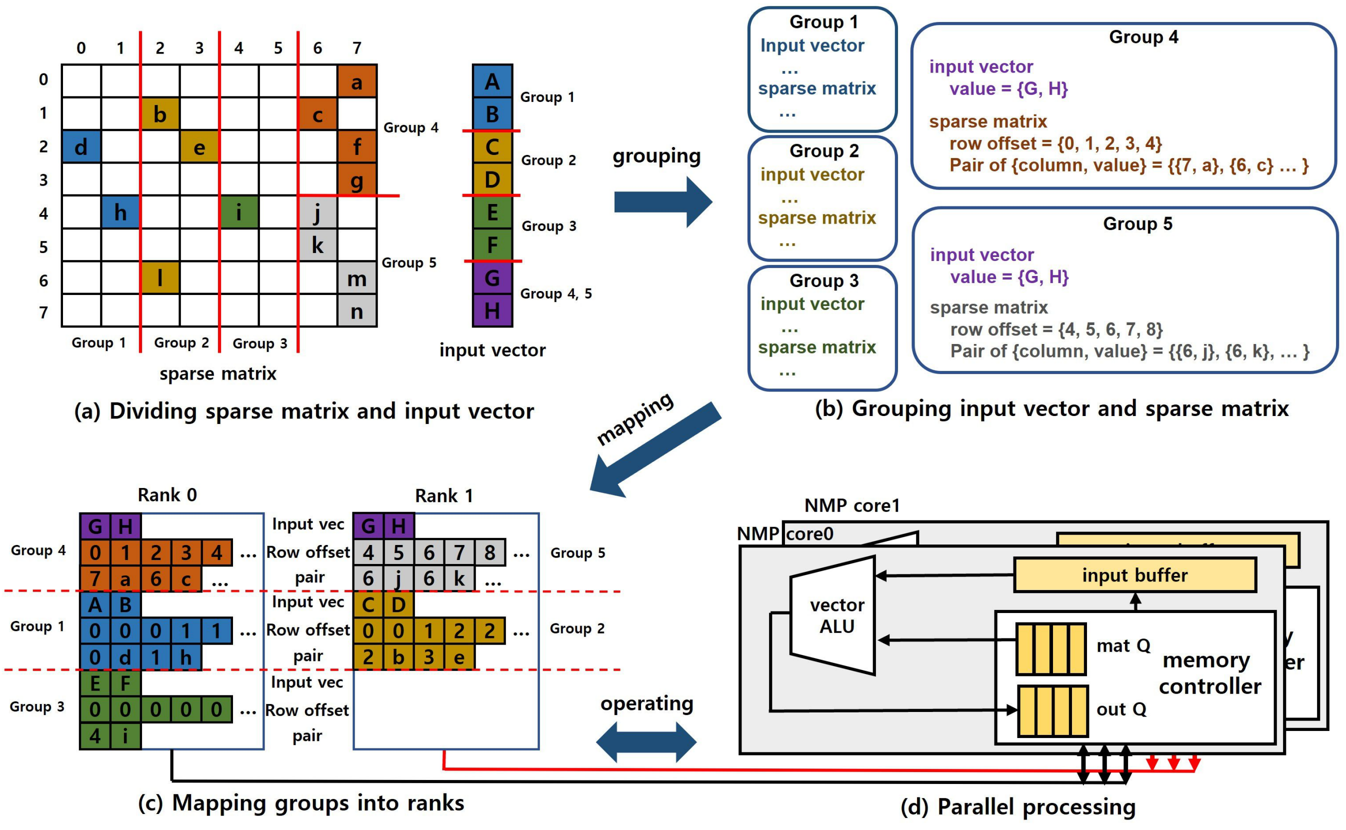
Fig. 5. GCSR format

(i) Even distribution: The number of non-zero elements in each partition is about the same.

(ii) Skewed distribution: There is at least one partition that has a significantly different number of non-zero elements from the others.

(iii) Power-law distribution: There is at least one partition where the percentage of non-zero elements is above 60%.

Dynamic Partitioning Method: Fig. 4 shows examples of dynamic partitioning for each distribution. `A` to `D` and `a` to `d` are four partitions of the sparse matrix and those of the input vector of equal size, respectively. The ratio represents the percentage of non-zero elements in each partition. DPA sorts the partitions according to the ratios and maps a partition group of the sparse matrix and the input vector to a rank. In this example, we assume that there are two ranks, so a partition group is mapped to either the first rank (R0) or the second rank (R1). Fig. 4 (a) shows the process of partitioning and mapping in the even distribution case, in which case, a partition group is alternately stored in R0 and R1. In the case of skewed distribution, as shown in Fig. 4 (b), the order of mapping changes according to the ratios of each matrix partition. If static partitioning is used to store each partition into two ranks evenly in the case of Fig. 4 (b), the ratio of the number of non-zero elements of R0 and R1 will be 43 vs. 57. However, if DPA is applied, the ratio will be 46 vs. 54, mitigating the load imbalance by 1.12×. Fig. 4 (c) shows a case where the ratio of non-zero elements is higher than 60% when the matrix is initially partitioned into four partitions (Power-law distribution). In this case, DPA conducts an additional partitioning on the last partition, splitting it into two sub-partitions of equal size. During this process, the vector `d` is duplicated and included in both sub-partition groups because these sub-partitions may be mapped to a different rank. Duplicating a vector segment does not cause any significant overhead because the size of a segment of an input vector is typically small. After this additional partitioning, sorting and mapping are iterated with the updated set of partitions as in Fig. 4 (a) and (b). If a static



(a) Dividing sparse matrix and input vector
sparse matrix
input vector
Group 1
Group 2
Group 3
Group 4
Group 5
Group 4, 5
grouping
(b) Grouping input vector and sparse matrix
Input vector
sparse matrix
value = {G, H}
row offset = {0, 1, 2, 3, 4}
Pair of {column, value} = {{7, a}, {6, c} … }
row offset = {4, 5, 6, 7, 8}
Pair of {column, value} = {{6, j}, {6, k}, … }
mapping
Rank 0
Rank 1
Input vec
Row offset
pair
(c) Mapping groups into ranks
operating
NMP core1
NMP core0
vector ALU
input buffer
mat Q
out Q
memory controller
(d) Parallel processing

Fig. 6. Overall execution flow

partitioning scheme is adopted in case of Fig. 4 (c), the ratio of the number of non-zero elements in R0 and R1 will be 27 vs. 73, which means the degree of load imbalance is 2.21× bigger than the case where DPA is applied (45 vs. 55).

Algorithm: In DPA, the sparse matrix is compressed using the CSR format. Once the matrix is partitioned into multiple partitions, the CSR representation should be modified accordingly. We call this modification Group CSR (GCSR). Fig. 5 shows the GCSR format when the matrix is partitioned into two groups. The GCSR row offset is additionally generated for each group. The GCSR pair represents a pair of the column index and the non-zero value.

When the matrix is compressed using the GCSR format, the number and the distribution of non-zero elements can be estimated by the row offset value. As described in Algorithm 1, the partitioned matrices are represented using GCSR, and the input vector is also partitioned accordingly, and a group of a matrix partition and a related vector partition is mapped to a rank. First, DPA generates `Group[]` structure consisting of matrix partitions in GCSR and input vector partitions (described in lines 3-5). The partitions in `Group[]` are sorted. (described in lines 6-20). Lines 6-18 describe the sorting method when the power-law flag is on. If the power-law flag is on, additional partitions will be generated. The updated set of partition groups are sorted in `order_group[]` according to the `order_Gnnz` where `Gnnz` represents the number of non-zero values in each group. (described in line 7 and 8) In lines 9-18, steps to determine `map_R[]` using `order_Gnnz[]` and `order_group[]` are shown. The `map_R[]` array will store the mapping information between a partition group and a rank.

B. Execution flow of the NMP architecture with DPA

Fig. 6 describes the overall execution flow of the NMP system when DPA is applied. Let's suppose that the input sparse matrix has a distribution of non-zero elements that satisfies the power-law distribution criterion. Fig. 6 (a) and (b) show a partitioning of the sparse matrix and the input vector. In Fig. 6 (c) and (d), each partition group is mapped to a rank after ordering and sent to a NMP core for SpMV operation. A detailed description of these steps is given as follows.

Partitioning & Grouping: Fig. 6 (a) and (b) illustrate this step. Initially, a sparse matrix and an input vector are divided equally. Columns 6 and 7 of the matrix are further divided into sub-partitions row-wisely because more than 60% of the total non-zeros belong to columns 6 and 7, so Group 4 and Group 5 are created as described in Fig. 6 (b). In this case, groups 4 and 5 have the duplicated input vector (G and H). After the grouping (Fig. 6 (b)), each group consists of an input vector partition and a sparse matrix partition in the GCSR format.

Mapping: Fig. 6 (c) illustrates this step. After the partitioning, partitions are sorted according to the percentage of the non-zero elements in each partition. Next, they are mapped to a

TABLE I
EVALUATED SYSTEM CONFIGURATION

Memory Controller	FCFS, 64 entry request queue
Memory Mapping Scheme	row (15b) || column (7b) || bank a (2b) || $bankg(2b)$ || rank (1b)
Main Memory	32 Gb device, x8 DDR4-2400, 1 channel, 2 rank, 4 banks/rank, 64 K rows/bank, 1 KB page

certain rank one by one. As shown in Fig. 6 (c), groups 4 and 5 are first mapped because they have the most non-zero values, and the rest of the partitions will be mapped to appropriate ranks by the DPA.

Parallel processing: Fig. 6 (d) shows an example of the parallel processing of the NMP architecture. Each rank has its own NMP core. An NMP core consists of a vector ALU, a queue for the output vector (`out Q`), a queue for matrix (`mat Q`), an input buffer, and a memory controller. The input vector is first loaded into the input buffer. Subsequently, the matrix values in GCSR are loaded into `mat Q` and `vector ALU` will carry out multiplications between the input vector and the load matrix values. The multiplication result is stored to the corresponding rank using `out Q`. Eventually, a merge operation of all the partial sums is performed by the CPU to complete an SpMV operation.

IV. PERFORMANCE EVALUATION

To evaluate the performance of the proposed method, a DRAM simulator called Ramulator [21] is used with some necessary modifications. Table I shows the system configuration used in the evaluation. A DDR4 memory module with two ranks is used. We compare our proposed idea with two other NMP models: one without partitioning (baseline) and the other with static partitioning. For static partitioning, two different numbers of partitions are tested: two partitions and four partitions. They will be called Group-2 and Group-4, respectively. We extract each memory trace based on the memory mapping method described in Table I and provide these traces to the modified Ramulater as inputs to evaluate the performance.

We evaluate the performance of our proposed method with respect to various sparse matrices in a benchmark suite called SuiteSparse [10]. For fair evaluation, matrices with various distributions (even distribution, skewed distribution, and power-law distribution) are selected, and Table II summarizes the detailed information of the selected matrices, including the degree of load imbalance.

TABLE II
BENCHMARK

benchmark	**distribution**	**rows**	**nnz**	**density**	imbalance
xenon2(xe)	even	157,464	3,866,688	1.56×10^{-4}	0.00
web-Google (wg)	skewed	916,428	5,105,039	6.10×10^{-6}	0.14
delaunay-n19(dn)	skewed	524,288	3,145,646	5.4×10^{-8}	0.67
wiki-Talk-temporal(wtt)	skewed	1,140,149	3,309,592	2.55×10^{-6}	1.69
com-Youtube(cmb)	power-law	1,134,890	5,975,248	4.64×10^{-6}	5.79
soc-Slashdot0902(ss)	power-law	82,168	948,464	1.4×10^{-4}	5.96

V. EXPERIMENTAL RESULT

A. Performance

Fig. 7 summarizes the results of comparing the performance of DPA with those of the others. The compared systems are an NMP architecture without partitioning (baseline) and NMP architectures with two different static partitioning methods (Group-2 and Group-4). For the baseline case, input vectors and matrices are first divided by the size of one cache line (64 b) [22] and they are mapped to a rank alternately. The performance advantage of DPA compared with the baseline is up to 1.96×, and the average is 1.72×. When compared to the static partitioning methods, DPA achieves speedups of up to 1.84× and on average 1.37× compared with Group-2. Meanwhile, the Group-4 method is almost equivalent to DPA when no partition satisfies the power-law distribution criterion. Consequently, in the case of benchmark `xe`, no significant performance differences are observed among all the partitioning methods because the matrix has evenly distributed non-zero elements. On the contrary, in the case of `cmb` and `ss`, significant performance differences are exhibited because distributions of multiple matrix partitions correspond to the power-law distribution. As a result, DPA achieves up to 1.17× and on average 1.15× speedups over Group-4 in the case of benchmarks where matrices have power-law distributions.

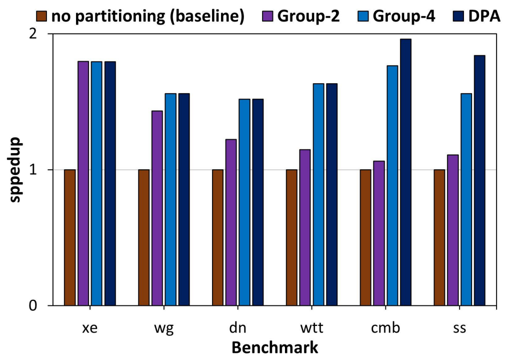


Fig. 7. Speedups of the NMP system with DPA over the other compared systems

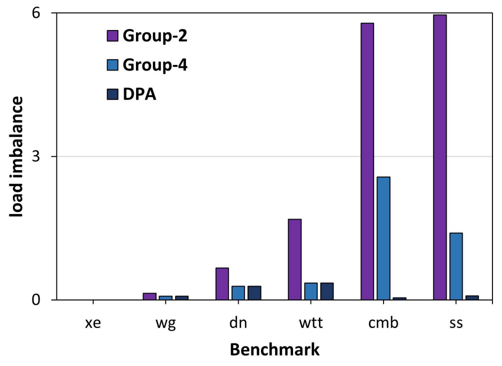


Fig. 8. Load imbalance of DPA compared to static partitioning schemes (Group-2 and Group-4)

B. Load Imbalance

Fig. 8 describes the evaluation result of the load imbalance degree of DPA compared to the other partitioning methods. DPA reduces the load imbalance degree by an average of 74% and 38% compared to Group-2 and Group-4, respectively. The reduced degree of load imbalance is presumably the main reason why DPA achieves the speedups shown in Fig. 7.

VI. Conclusion

Near-memory processing is known to be an effective solution to speed up the execution time of memory-bound applications. Sparse matrix-vector multiplication (SpMV) is a memory-bound operation because it has to deal with matrices of huge size. Prior works have proposed various types of partitioning methods for near-memory parallel processing of SpMV. In this paper, we proposed a new dynamic partitioning method for near-memory parallel processing of SpMV. When there are multiple ranks and each rank has its own processing unit, partitioning the input sparse matrix and the input vector and mapping them evenly to multiple ranks will be important to fully exploit the parallel processing capability. Since many sparse matrices have uneven and irregular distributions of nonzero elements, static partitioning cannot handle the situation where the input matrices have diverse forms of distribution. Therefore, a significantly unbalanced mapping may occur. Our proposed method, dynamic partitioning algorithm (DPA), applies a dynamic partitioning scheme according to the various distributions of nonzero elements in the input sparse matrix. Experimental results show that DPA achieves speedups of on average 1.72× over the system where no partitioning is applied and on average 1.31× over the static partitioning.

Acknowledgement

This work was supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-00153, Development of O-RU for in-building based on AI network management using beamforming path)

References

[1] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[2] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, "Slimsell: A vectorizable graph representation for breadth-first search," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 32–41.

[3] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 982–995.

[4] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[5] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[6] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *IEEE Internet computing*, vol. 7, no. 1, pp. 76–80, 2003.

[7] A. Elafrou, G. Goumas, and N. Koziris, "Performance analysis and optimization of sparse matrix-vector multiplication on intel xeon phi," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 1389–1398.

[8] C. Giannoula, I. Fernandez, J. Gómez-Luna, N. Koziris, G. Goumas, and O. Mutlu, "Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory systems," *arXiv preprint arXiv:2201.05072*, 2022.

[9] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on fpgas," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 69–80.

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

[11] C.-Y. Gui, L. Zheng, B. He, C. Liu, X.-Y. Chen, X.-F. Liao, and H. Jin, "A survey on graph processing accelerators: Challenges and opportunities," *Journal of Computer Science and Technology*, vol. 34, pp. 339–371, 2019.

[12] M. Lenjani, A. Ahmed, M. Stan, and K. Skadron, "Gearbox: A case for supporting accumulation dispatching and hybrid partitioning in pim-based accelerators," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 218–230.

[13] Y. Kwon, Y. Lee, and M. Rhu, "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*, 2019, pp. 740–753.

[14] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin *et al.*, "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)*. IEEE, 2021, pp. 43–56.

[15] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 790–803.

[16] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 570–583.

[17] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 556–569.

[18] E. Kayaaslan, B. Uçar, and C. Aykanat, "Semi-two-dimensional partitioning for parallel sparse matrix-vector multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 1125–1134.

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

[20] S. Khoram, Y. Zha, J. Zhang, and J. Li, "Challenges and opportunities: From near-memory computing to in-memory computing," in *Proceedings of the 2017 ACM on International Symposium on Physical Design*, 2017, pp. 43–46.

[21] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[22] JEDEC, "DDR4 SDRAM standard," 2012.