A Appendix

A.1 Supplementary of the Cost Model

The execution cost T_{tile} is the time to execute all the tiles used. As tile execution is parallel, we track the runtime of the last tile to finish. Similarly, PEs within a tile are also executed in parallel. Equation 4 estimates T_{tile}:

$$T_{\text{tile}} = (x'_{in} + x'_{out})/\mathsf{B}_1 + T_{\text{PE}} + T'_{\text{accum}} \tag{4}$$

Where T_{PE} is the execution time of the PEs within the tile, considered constant due to its small value; x'_{in} is the total cost of distributing the input vector to all used PEs; x'_{out} is the size of the output vector; B_1 is the tile-level buffer bandwidth (GB/sec); T'_{accum} is the time to accumulate the output results from all used PEs, estimated by

$$T'_{\text{accum}} = \sum_{c=1}^{C'} \sum_{i=1}^{\lceil \log_2(N_{\text{PE}}(c)) \rceil} \frac{N_{\text{col}}(c)}{N'_{\text{Adder}}} \cdot (k_2 + i \cdot k_1) \tag{5}$$

where C' is the number of accumulation groups (we group PEs with the same column indexes of sub-matrices into an accumulation group); N'_{Adder} is the number of adders inside the tile; $N_{\text{PE}}(c)$ is the number of PEs in the c-th accumulation group; and k_2 is the read latency of the tile-level adder.

A.2 The Detailed Measurements Associated with Figure 5

Table 3 serves as a complement to Figure 5, providing more detailed experimental values. Within it, *Latency* signifies the total execution latency of the MVM, *Communication* indicates the percentage of the total latency attributed to the communication cost, *Tile Execution* represents the percentage of the total latency attributed to the tile execution cost, and *Accumulation* denotes the percentage of the total latency attributed to the accumulation cost. The *Size* refers to the in-memory size of the matrix.

Table 3. The detailed measurements associated with Figure 5

ID	Name	Latency (ns)	Communication (%)	Tile Execution (%)	Accumulation (%)	Size
1	qh882	2943	69.15	27.05	3.77	2.97 MB
2	radfr1	2785	64.56	28.51	6.89	4.19 MB
3	illc1850	5106	71.13	26.75	2.06	5.02 MB
4	qh1484	5194	76.16	16.37	7.43	8.40 MB
5	rdist3a	5685	74.25	15.92	9.80	21.94 MB
6	extr1	7335	82.14	10.92	6.93	30.70 MB
7	zenios	3231	83.19	14.67	2.07	31.49 MB
8	bayer09	7959	78.34	11.60	10.05	36.26 MB
9	rdist2	7045	78.00	11.92	10.06	39.01 MB
10	rdist1	9071	79.79	9.79	10.41	65.19 MB
11	crystm01	14132	80.71	11.06	8.22	90.66 MB
12	hydr1	13028	86.18	6.32	7.50	0.10 GB
13	bayer03	16577	86.29	5.71	7.99	0.17 GB
14	Lederberg	22028	87.20	3.55	9.24	0.29 GB
15	circuit_3	33224	83.98	2.78	13.23	0.55 GB
16	bayer10	33049	87.19	2.75	10.06	0.67 GB
17	bayer02	31078	90.44	2.92	6.64	0.72 GB
18	crystm02	47393	89.80	2.73	7.47	0.73 GB
19	bayer04	49794	88.20	1.96	9.84	1.57 GB
20	crystm03	95298	88.67	1.17	10.16	2.27 GB

A.3 Hardness of Storage Optimization

We first discuss how to reorder the matrix under the hypergraph model. The matrix is partitioned into sub-matrices via tiling for storage. The rows (*resp.* columns) of each sub-matrix constitute a partition of the matrix rows (*resp.* columns). Therefore, we can naturally formulate matrix reordering

on CIM architecture as a hypergraph partitioning problem. Since tile size is fixed, we leverage balanced k-way hypergraph partitioning to explore matrix reordering, as formalized in Definition 8. The column order can be derived by partitioning the *row-net* hypergraph, while the row order is obtained from partitioning the *column-net* hypergraph.

Definition 8 (Balanced k-way Hypergraph Partitioning). *The k-way hypergraph partitioning problem is to find an ε-balanced k-way partition Π of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N}, c, \omega)$ that minimizes an objective function defined on the nets where each partition $\mathcal{V}_i \in \Pi$ satisfies the balance constraint: $c(\mathcal{V}_i) \leq L_{\max} := (1+\varepsilon)\left\lceil \frac{c(\mathcal{V})}{k} \right\rceil$ for the imbalance ratio $\varepsilon \in [0, 1)$. In this work, we only consider the most commonly used cost function, namely the connectivity metric $\mathcal{F}(\Pi) := \sum_{e \in \mathcal{N}'} (\lambda(e) - 1)\omega(e)$, where $\mathcal{N}'$ is the cut-set (i.e., the set of all cut nets). We denote $\lambda(e)$ as the number of partitions connected by e. The connectivity metric minimizes the weight of all nets and additionally takes into account the actual number of partitions connected by the nets. Optimizing the objective function is known to be NP-hard [49].*

Next, we model the matrix as a *row-net* hypergraph and then prove the problem of finding a communication-optimal column order in the matrix is the special case of the balanced k-way hypergraph partitioning problem to demonstrate its hardness.

Theorem 1. *Finding the communication-optimal column order of the matrix for given multi-MVM queries is NP-hard.*

Proof. Given a matrix $\mathbf{A}$ of size $M \times N$ and a set of multi-MVM queries $\mathcal{Q}$. Each MVM operation in $\mathcal{Q}$ accesses a sub-matrix. We transform each row of all sub-matrices accessed in $\mathcal{Q}$ into a net, from which a *row-net* hypergraph $\mathcal{H}_{\mathcal{R}}$ is formed. The weight of each vertex and each net is 1. H_{tile} and W_{tile} denote the height and width of a tile. We partition all columns into a balanced k-way partition Π, where each partition $\mathcal{V}' \in \Pi$ satisfies the balance constraint: $c(\mathcal{V}') \leq W_{tile}$. In the MVM operation, the row i of the sub-matrix is multiplied by $V(i)$ of V, where V is a dense input vector. We assume that the row i corresponds to the net e_i. If the non-zero elements in row i are stored in s_i tiles, then all vertices in e_i belong to s_i partitions (§3). The variable s_i indicates the number of CIM tiles needed to store the non-zero elements in the i-th row of matrix $\mathbf{A}$. To minimize x_{in} in Equation 2, we should minimize the number of nets involved in all partitions. More precisely, this problem is equivalent to finding a balanced k-way partition Π of $\mathcal{H}_{\mathcal{R}}$ that minimizes the connectivity metric on the nets where $k = \lceil \frac{N}{W_{tile}} \rceil$. It is a special case of the well-known NP-hard problem balanced k-way hypergraph partitioning when $\varepsilon = 0$. Hence, finding the communication-optimal column order of the matrix for given multi-MVM queries is NP-hard. □

To reduce the problem of finding a compression-optimal row order to an NP-hard problem, we use a *net consolidation* strategy as described in Definition 7. Next, we model the matrix as a *column-net* hypergraph and then reduce this row reordering problem to the balanced k-way hypergraph partitioning problem to demonstrate its hardness.

Theorem 2. *Given a column order, finding the compression-optimal row order of the matrix is NP-hard.*

Proof. Given a matrix $\mathbf{A}$ of size $M \times N$, we model $\mathbf{A}$ as a *column-net* hypergraph $\mathcal{H}_C$. Each net in $\mathcal{H}_C$ corresponds to each column in the matrix. We assume that the column i corresponds to the net e_i. The vertices in e_i are the rows connected by the column i, and the weight of each vertex and each net is 1. To formalize the number of tiles used by $\mathbf{A}$ in terms of the hypergraph model, we use the *net consolidation* strategy. Specifically, as the column order is given, we partition the columns into disjoint partitions such that columns $((i-1) \cdot W_{tile} + 1) \sim (i \cdot W_{tile})$ belong to the

same partition, where $i \in [1, \lceil \frac{N}{W_{tile}} \rceil]$. For any columns i and j, we assume that e_i and e_j belong to the same partition only if i and j belong to the same partition. And then, we consolidate all nets belonging to the same partition into a new net and name the consolidated *column-net* hypergraph as $\mathcal{H}'_C$. We partition all vertices in $\mathcal{H}'_C$ into a balanced k-way partition Π, where each partition $\mathcal{V}' \in \Pi$ satisfies the balance constraint: $c(\mathcal{V}') \leq H_{tile}$. The number of used tiles can be formulated as $N_{tile} := \sum_{e \in \mathcal{N}'} \lambda(e)$, where $\mathcal{N}'$ is the cut-set (i.e., the set of all nets of $\mathcal{H}'_C$). Hence, this problem is equivalent to finding a balanced k-way partition Π of $\mathcal{H}'_C$ that minimizes the connectivity metric on the nets where $k = \lceil \frac{M}{H_{tile}} \rceil$. Given a column order, the problem of finding a compression-optimal row order of a matrix is reduced to a balanced k-way hypergraph partitioning problem by the *net consolidation* strategy, thus proving that this problem is NP-hard. □

Table 4. Performance Breakdown on "zenios" matrix.

Method	Used Tiles	Compression Rate	Execution Time (ns)	Communication (ns)	Max Tile Execution (ns)	Accumulation (ns)
HyperMR	6	1.66667	2948	2596	348	0
GSMR	3	3.33333	3072	2596	430	41
ReSpar	1	10	3159	2596	559	0
kMeans	4	2.5	3201	2596	543	58
GMR	1	10	3186	2596	585	0
METIS	6	1.66667	2989	2596	389	0
TraNNsformer	10	1	3190	2687	432	67

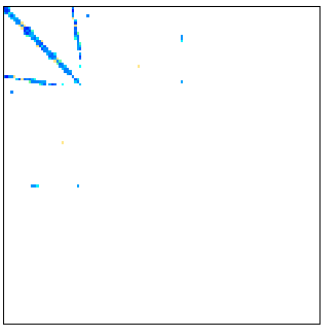

Fig. 13. The visualization of "zenios".

A.4 Performance Breakdown Analysis on "zenios" matrix

We have compiled the performance breakdown of all CIM-based methods on "zenios", as shown in Table 4. GMR and ReSpar achieve a high CR ('10.00') on a special case, "zenios". Our analysis indicates that their heuristic rules consolidate all non-zero entries into a single tile, which caters to the distinctive matrix pattern of "zenios", i.e., non-zero entries densely concentrated in the upper-left corner of the matrix (see the visualization of "zenios", Figure 13). Specifically, GMR arranges rows with a higher number of non-zero entries together, and since each row in "zenios" has its non-zero entries clustered on the leftmost side, this arrangement facilitates the aggregation of all non-zero entries within a single CIM tile. ReSpar, on the other hand, arranges columns with fewer leading zeros together, and the columns on the left side of "zenios" indeed have fewer leading zeros, thus facilitating their aggregation within a single CIM tile.

However, these rules also result in longer tile execution times, and thus HyperMR is faster than GMR and ReSpar for this special case. In comparison, HyperMR has a maximum tile execution time of 348 ns, versus 585 ns for GMR and 559 ns for ReSpar. Moreover, GMR only provides

Table 5. Denser Matrices.

ID	Name	Height	Width	Sparsity	Symmetry	Used Tiles	Size (MB)	Application
21	lp_fit1d	24	1,049	0.46667	0	3	0.10	Linear Programming Problem
22	lp_fit2d	25	10,524	0.50953	0	21	1.00	Linear Programming Problem
23	cari	400	1,200	0.68167	0	3	1.83	Linear Programming Problem
24	bibd_13_6	78	1,716	0.80769	0	4	0.51	Combinatorial Problem
25	heart2*	2,339	2,339	0.87564	1	25	20.87	2D/3D Problem
26	heart1*	3,557	3557	0.89051	1	47	48.26	2D/3D Problem
27	Trec12	551	2,726	0.89932	0	12	5.73	Combinatorial Problem
28	Trec13	1,301	6,561	0.92332	0	36	32.56	Combinatorial Problem
29	us04	163	28,016	0.93484	0	55	17.42	Linear Programming Problem
30	mycielskian11*	1,535	1,535	0.94283	1	9	8.99	Undirected Graph
31	psmigr_1	3,140	3,140	0.94491	0.479	49	37.61	Economic Problem
32	CAG_mat1916	1,916	1,916	0.94661	0.3	16	14.00	Combinatorial Problem
33	mycielskian12*	3,071	3,071	0.95682	1	32	35.98	Undirected Graph
34	cegb2919*	2,919	2,919	0.96226	1	28	32.50	Structural Problem
35	f855_mat9	2,456	2,511	0.97224	0	19	23.53	Combinatorial Problem
36	FX_March2010	1,319	9,498	0.9759	0	57	47.79	Term/Document Graph
37	gen2	1,121	3,264	0.97763	0	17	13.96	Linear Programming Problem
38	fp	7,548	7,548	0.98536	0.758	211	217.33	Electromagnetics Problem
39	rail516	516	47,827	0.98722	0	172	94.14	Linear Programming Problem
40	rail582	582	56,097	0.98768	0	219	124.54	Linear Programming Problem

valid optimization for 10% of the matrices in Table1, and ReSpar for 25%. In contrast, HyperMR provides valid optimizations for all matrices. This is aligned with our observation in **L1 (Inadequate Optimization Objectives)** (see § 1) that focusing solely on CRs does not always lead to the efficient acceleration of MVM operations.

A.5 HyperMR vs. CIM-based Methods on Denser Matrices

In this section, we evaluate all CIM-based methods on 20 denser real-world matrices, as listed in Table 5, serving as a supplement to Section 6. These matrices exhibit sparsity ranging from 0.466 to 0.988. The rationale for selecting this sparsity range is that we found matrices with a sparsity less than 0.466 in the SuiteSparse matrix repository [17] can be accommodated by a single CIM tile, thus obviating the need for optimization. Notably, 12 of these matrices have a pattern symmetry index of 0, indicating highly complex and irregular matrix patterns. Table 6 presents a detailed comparison of CIM-based baselines in terms of execution time on typical MVM workloads and the CR of the matrices. The experimental results indicate that HyperMR provides valid optimization for all denser matrices, achieving the best performance in 90% of the matrices. In contrast, GSMR can only provide valid optimization for 70% of the matrices, TraNNsformer for 20%, kMeans for 30%, METIS for 15%, GMR for 35%, and ReSpar for 25%.

Compared to the "None" method before storage optimization, HyperMR can provide valid optimization for all matrices. HyperMR achieves an average execution time acceleration of 13.28% and an average CR improvement of 20.81%. The efficiency of HyperMR is achieved via optimizations of **O1** and **O2**, which not only speed up execution but also enhance compression by concentrating zero elements in the optimized matrix to maximize idle tiles.

GSMR is a bandwidth reduction-based scheme that assumes a diagonal block pattern exists in matrices. Consequently, GSMR is challenging to optimize all matrices and fails to provide valid optimization for 30% of matrices. In contrast, HyperMR offers valid optimization for all matrices without being limited to specific matrix structures. HyperMR outperforms GSMR with an average execution time acceleration of 11.33% and an average CR enhancement of 0.3%. Moreover, HyperMR achieves the fastest execution times in 90% of matrices, while GSMR does so in only 5% of matrices.

TraNNsformer is a graph clustering-based scheme limited by its inability to eliminate a few connections between clusters, leading to the allocation of tiles for a few non-zero elements and thus causing hardware resource waste. Furthermore, it does not consider performance optimization.

Table 6. Comparisons with CIM-based storage schemes on denser matrices. Method "None" means that the matrix is not optimized. The best execution time and CR on each matrix are bolded. *Valid Optimization* is highlighted with a grey background, indicating an optimized CR is greater than or equal to 1 and has faster execution than the unoptimized version.

Matrix	None	GSMR		TraNNsformer		kMeans		METIS		GMR		ReSpar		HyperMR	
Name	Time (ns)	Time (ns)	CR	Time (ns)	CR	Time (ns)	CR	Time (ns)	CR	Time (ns)	CR	Time (ns)	CR	Time (ns)	CR
lp_fit1d	1,389	1,361	1	1,386	1	1,389	1	\	\	1,389	1	\	\	**1,359**	**1**
lp_fit2d	9,525	9,485	1	9,519	1	9,523	1	\	\	9,524	1	\	\	**9,464**	**1**
cari	3,094	2,890	1	3,178	1	3,119	1	\	\	3,119	1	\	\	2,958	**1**
bibd_13_6	2,230	2,226	1	2,266	1	2,250	1	\	\	2,251	1	\	\	**2,221**	**1**
heart2*	10,368	10,672	1.32	11,774	1	11,482	1	9,116	1	13,232	1	10,301	1.19	**8,471**	**1.47**
heart1*	16,766	16,083	1.34	21,250	0.96	19,137	1	16,849	0.96	24,331	0.96	17,202	1.09	**13,675**	**1.38**
Trec12	6,169	6,041	1.50	6,691	1	6,895	1	\	\	6,907	1	\	\	**5,685**	1.09
Trec13	18,854	18,963	1.12	21,357	0.92	21,799	0.92	\	\	22,170	0.92	\	\	**18,043**	1
us04	28,822	28,189	1	\	\	28,616	1	\	\	29,894	1	\	\	**26,274**	**1**
mycielskian11*	7,170	5,923	1.12	6,815	1	6,196	1	6,907	1	6,099	1.12	5,862	1.12	6,458	1
psmigr_1	22,368	22,718	1.02	22,556	1	21,162	1	\	\	21,217	1	21,452	1	**19,853**	**1.02**
CAG_mat1916	9,594	7,748	1.07	9,865	1	8,515	1	\	\	8,431	1	7,002	1.23	**6,931**	1.07
mycielskian12*	17,940	14,462	1.07	18,058	0.89	15,335	0.89	20,190	0.89	14,720	1.23	13,454	1.28	**13,095**	1.03
cegb2919*	9,064	9,662	1.40	11,045	0.78	13,658	0.78	8,752	1	14,998	0.78	9,222	1.12	**7,747**	**1.47**
f855_mat9	9,906	12,441	0.76	13,152	0.76	10,474	0.76	\	\	10,608	0.76	10,773	0.86	**9,356**	**1**
FX_March2010	23,342	21,891	1.02	23,236	1	21,234	1.04	\	\	21,571	1.04	\	\	**19,382**	**1.08**
gen2	8,124	9,037	0.85	8,722	0.81	9,447	0.81	\	\	9,036	0.81	\	\	**7,134**	**1**
fp	52,839	33,684	2.24	45,399	0.95	49,780	0.97	\	\	68,214	0.94	33,106	1.12	**27,330**	**2.78**
rail516	57,557	54,204	1.81	\	\	61,626	0.99	\	\	57,102	1.77	\	\	**49,637**	1.54
rail582	68,731	64,614	1.63	\	\	72,715	0.99	\	\	76,378	1.17	\	\	**59,814**	1.24
Valid Optimization	0	14		4		6		3		7		5		**20**	
Best	0	1		0		0		0		0		1		**18**	

TraNNsformer provides valid optimization for 20% of the matrices. HyperMR outperforms TraNNsformer with an average execution time acceleration of 28.18% and an average CR enhancement of 25.49%.

METIS is a graph partitioning-based scheme that treats the matrix as an adjacency matrix of a graph. Due to its design for special symmetric matrices (i.e., adjacency matrices), METIS cannot handle asymmetric matrices. Moreover, METIS shares a similar limitation with TraNNsformer, leading to hardware resource waste. Although METIS is designed for symmetric matrices, it does not provide valid optimization for two symmetric matrices. Thus, METIS only provides valid optimization for 15% of matrices. HyperMR outperforms METIS with an average execution time acceleration of 30.93% and an average CR enhancement of 20.99%.

kMeans is the K-means clustering-based column exchanging scheme with the Hamming distance. kMeans reorders the columns of a matrix based on the similarity between columns using clustering methods. It is hard to prove that clustering similar columns together would lead to performance improvement, thus lacking effective performance optimization objectives. Therefore, kMeans only provides valid optimization for 30% of matrices. HyperMR outperforms kMeans with an average execution time acceleration of 27.12% and an average CR enhancement of 21.83%.

GMR utilizes the density of non-zero entries to swap rows and columns. Its heuristic rule is designed to aggregate non-zero entries in one matrix corner but lacks a performance optimization objective. Thus, GMR only provides valid optimization for 35% of matrices. HyperMR outperforms GMR with an average execution time acceleration of 28.73% and an average CR enhancement of 21.34%.

ReSpar uses the XOR operation to measure between rows. It performs row reordering by computing the common feature and then reorders columns with fewer leading zeros. Its heuristic rule is designed to aggregate non-zero entries, yet it lacks a performance optimization objective. As shown in Table 6, ReSpar failed to output the reordered indices for 11 matrices correctly. The experimental results show that ReSpar only outputs partial indices of a matrix. This issue can be attributed to the complex patterns of these matrices, which result in the formation of some disconnected subgraphs. The intricate matrix patterns exacerbated its performance since ReSpar relies on graph traversal for reordering. Thus, ReSpar only provides valid optimization for 25% of

matrices. HyperMR outperforms ReSpar with an average execution time acceleration of 11.7% and an average CR enhancement of 22.7%.

In summary, HyperMR provides valid optimization for all evaluated matrices with various matrix structures, a feat that existing research has been unable to achieve. Also, Table 6 demonstrates HyperMR as the best storage scheme for 18 matrices and competitive with the rest.

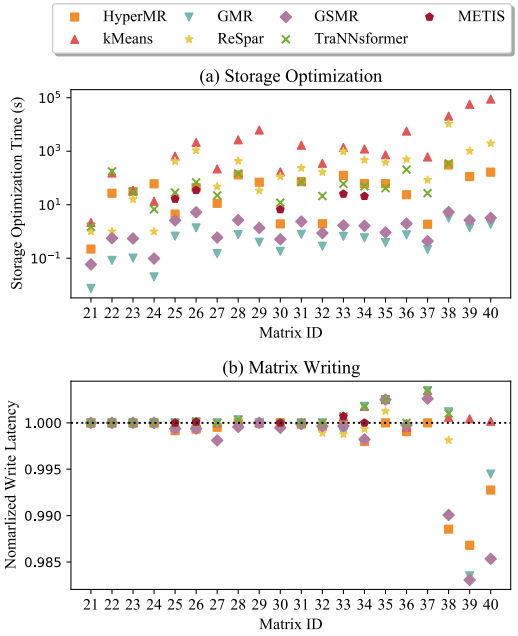

Fig. 14. Time overheads of all schemes on denser matrices.

Time Overhead. The time overhead is divided into two phases: storage optimization and matrix writing for all schemes. Figure 14(a) shows the storage optimization times. On average, HyperMR is 21.14 × faster than ReSpar, 117.44 × faster than kMeans, 4.12 × faster than TraNNsformer, and 1.69 × faster than METIS; however, HyperMR is 8.79 × and 28.75 × slower than GSMR and GMR, respectively. Although HyperMR exhibits a lower speed than GSMR and GMR, as described in Sections 6.2 and 6.4, HyperMR surpasses both methods regarding query execution acceleration, compression ratio, and adaptability to diverse matrix patterns and access patterns.

Figure 14 (b) shows matrix write times normalized to the unoptimized matrix write time. We found that all baselines incur no more than 1% additional matrix write overhead when unable to achieve a matrix storage layout with CR $\geq$ 1. In contrast, HyperMR, on average, improves matrix write time by 2% and does not introduce additional overhead for all cases. Despite HyperMR primarily focusing on optimizing the performance of multi-MVM queries and enhancing matrix compression ratios (§2.2), it can also slightly optimize matrix write times across all cases in cold start scenarios.

Summary. The key finding is that even on denser matrices, HyperMR still provides valid optimization for all matrices, whereas other CIM-based methods are limited to optimizing only certain matrices. Evaluating all CIM-based methods on denser matrices reveals that the disparities in compression ratios and execution time among these methods diminish because denser matrices imply reduced optimization space. However, we observe that other CIM-based methods exhibit irregular performance, making it difficult to predict which matrix patterns they can effectively

optimize. Therefore, HyperMR effectively compensates for this by providing universally valid optimization.

A.6 Cost Model Validation

Table 7. Cost model validation.

Accessed Columns [1, 20000]	Accessed Rows	[1, 2500]	[1, 5000]	[1, 7500]
	Accuracy	99.98%	99.99%	99.99%
	Accessed Rows	[1, 10000]	[1, 12500]	[1, 15000]
	Accuracy	99.99%	99.99%	99.99%
Accessed Rows [1, 20000]	Accessed Columns	[1, 2500]	[1, 5000]	[1, 7500]
	Accuracy	99.96%	100.00%	99.98%
	Accessed Columns	[1, 10000]	[1, 12500]	[1, 15000]
	Accuracy	99.99%	99.99%	99.99%

Received July 2024; revised September 2024; accepted November 2024