UPVSS: Jointly Managing Vector Similarity Search with Near-Memory Processing Systems

Chun-Chien Liu*, Chun-Feng Wu*, Yunho Jin†
*Department of Computer Science, National Yang Ming Chiao Tung University, Taiwan
†Department of Computer Science, Harvard University, USA
Corresponding Author: Chun-Feng Wu
E-mail: ccliu.cs12@nycu.edu.tw, cfwu417@cs.nycu.edu.tw, yjin@g.harvard.edu

***Abstract*—Vector similarity search plays a pivotal role in modern applications, including recommendation systems, image search, large language models (LLMs), and high-dimensional data retrieval. As data size scales, our research reveals that the search phase imposes substantial demands on DRAM bandwidth, leading to performance limitations in conventional von Neumann architecture with shared memory buses. This data movement bottleneck restricts the efficiency and scalability of vector similarity search due to insufficient memory bandwidth. To mitigate this issue, we leverage UPMEM, an off-the-shelf near-memory processing (NMP) system, to minimize the data movement between memory and compute units. However, UPMEM's computing engine has certain limitations and requires thorough application integration to unleash its high-parallelism capabilities. In this work, we introduce UPMEM-aware Vector Similarity Search (UPVSS), an architecture-aware system that jointly manages vector similarity search and UPMEM's NMP technology. UPVSS prioritizes offloading operations based on their strengths and capabilities, effectively alleviating the data movement bottleneck and improving overall system performance.**

***Index Terms*—Vector Similarity Search, LLM, Inverted File, FAISS, K-NN, PIM, UPMEM, HW/SW Codesign, Parallelism**

I. Introduction

Vector similarity search is a cornerstone of various applications, including large language models (LLMs) [8], [10], [26], recommendation systems [3], [33], image search [15], information retrieval [16], [24], and data retrieval in natural language processing. Instead of processing raw data like images or documents, modern systems represent them as high-dimensional vectors encoding features such as color or edges. Comparing these vectors enables efficient retrieval of similar data points. As systems and data volume scale, the number of vector dimensions has grown significantly, with models like OpenAI's GPT-3 producing embeddings with thousands to tens of thousands of dimensions [23], [34]. Our analysis indicates that running similarity searches on high-dimensional vectors is memory-bound, with cycles per instruction (CPI) values exceeding one due to frequent data movements between memory and CPU cache, which quickly saturate DRAM bandwidth. To address the immense memory bandwidth demand, this work integrates vector similarity search with a real near-memory processing system, UPMEM.

Exact similarity search requires brute-force distance calculations between all vectors, which is inefficient for large-scale, high-dimensional data [32]. To overcome this, modern applications use Approximate Nearest Neighbor (ANN) search, offering a balance between speed and accuracy. State-of-the-art ANN algorithms include cluster-based (e.g., Inverted File (IVF) [11]), graph-based (e.g., HNSW [20], NSG [7]), hash-based [38], and tree-based algorithms [27]. However, tree-based algorithms lack scalability for high-dimensional vectors [36], hash-based algorithms suffer from lower accuracy on large-scale datasets [19], and the memory consumption of graph-based algorithms grows rapidly with dataset size, making them unsuitable for large-scale applications [29]. Industrial vector similarity search libraries, like Meta's FAISS [6], [25], recommend using the IVF for managing large-scale high-dimensional vector datasets. Despite the use of ANN algorithms, the search phase remains limited by the memory bandwidth constraints of the von Neumann architecture, which struggles to meet the high DRAM bandwidth demands of vector similarity search.

Processing-in-memory (PIM) technologies are emerging as efficient solutions to the challenges of excessive data movement by bringing compute and memory units closer together. PIM can be broadly categorized into two types: processing-using-memory (PUM), where memory units perform computation, and near-memory processing (NMP), which integrates compute elements directly into memory modules. Emerging non-volatile memory technologies, such as ReRAM, enable analog-based PUM devices like ReRAM crossbars [14], [28] and TCAM [30], which are particularly effective for matrix multiplication and table look-up tasks. However, these devices rely heavily on Digital-to-Analog Converters (DAC) and Analog-to-Digital Converters (ADC), which together account for more than half of their total energy consumption and area costs [18], [39]. As a result, the significant cost associated with converting signals between analog and digital forms poses a major challenge to the widespread commercialization of analog-based PIM devices.

In contrast, UPMEM [4], [9], a recent NMP system, is the first commercially available off-the-shelf PIM device. UPMEM integrates DRAM Processing Units (DPUs) that combine general-purpose RISC processors with traditional 2D DRAM arrays on a single chip. Current UPMEM NMP systems support up to 2560 DPUs, each offering 1 GB/s of internal DRAM bandwidth, resulting in an impressive aggregate internal bandwidth of up to 2.56 TB/s. However, running applications on UPMEM naively can lead to challenges such

as frequent out-of-memory (OOM) errors due to its limited 64 MB of isolated memory per DPU and significant performance drops caused by high data transmission costs. Consequently, existing research has focused on adapting UPMEM for applications in analytics and databases [1], [13], [17], bioinformatics [2], [5], and data compression and decompression [21]. However, none of these works have explored its potential integration with vector similarity search.

Based on our investigations, running IVF-based vector similarity search directly on UPMEM encounters three significant challenges: (1) Each IVF cluster contains numerous high-dimensional vectors, often exceeding a DPU's available memory capacity (64 MB), leading to frequent out-of-memory (OOM) issues. (2) Migrating vectors between DPUs incurs substantial overhead due to costly data movement. (3) To fully utilize the DPU hardware pipeline, it is required to schedule at least 11 hardware threads (also known as tasklets) to execute concurrently. Motivated by the need to address these challenges and enable efficient vector similarity search on UPMEM, this paper introduces UPVSS, a UPMEM-aware Vector Similarity Search framework. UPVSS incorporates two key components. The first is the *DPU-aware Clusters Partition*, which evenly splits the vectors within each IVF cluster while accounting for the DPU's architectural constraints. The second is the *DPU-aware Vector Similarity Search Coordinator*, which strategically offloads vector distance calculations to DPUs and leverages task pipelining to fully exploit the high parallelism offered by the DPUs.

The rest of this paper is organized as follows: Section II presents the background, observation, and motivation. In Section III, the UPVSS is introduced to enhance query processing performance. Section IV provides the analysis and experimental results. Section V concludes this work.

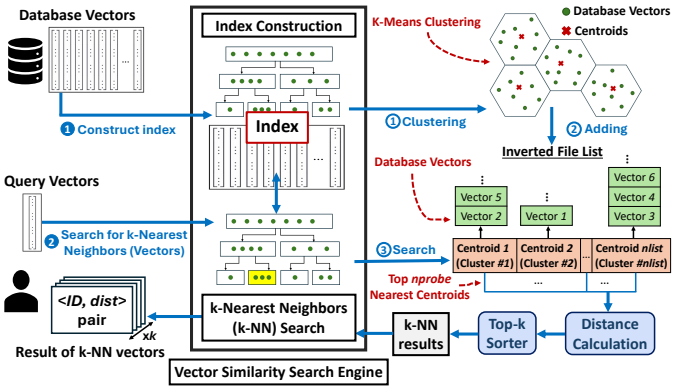


Fig. 1: The workflow of vector similarity search (left), along with the Inverted File (IVF) ANN index algorithm (right).

II. Background, Observation, and Motivation

A. Background

1) Vector Similarity Search

The typical workflow of vector similarity search consists of two steps, as illustrated in Fig. 1 (left). ❶ **Index Construction:** An index is constructed to facilitate efficient searching over the database vectors, enabling effective pruning of the search space during the query phase. This process is conceptually analogous to building traditional data structures such as B-trees or binary search trees, where the data is organized in a manner that allows the search algorithm to quickly eliminate regions unlikely contain the target items. Several indexing techniques have been developed for vector similarity search, such as KD-Tree, R-tree. ❷ **k-Nearest Neighbors (k-NN) Search:** Upon receiving a query vector, the system searches for the top-k similar vectors within the database vectors using a distance function, such as Euclidean distance, cosine similarity. This step involves extensive distance calculations and top-k sorting operations. The result of this step includes the identifiers of the k-NN in the database vectors along with their corresponding distances.

2) Approximate Nearest Neighbor (ANN) Index

Performing exact nearest neighbor search on high-dimensional vectors is computationally demanding and often impractical for large-scale applications [32]. Approximate Nearest Neighbor (ANN) methods address this challenge by reducing search time through a controlled approximation of k-NN results. This trade-off between accuracy and efficiency is suitable for many applications where near-optimal results are sufficient, such as LLMs and recommendation systems. Several ANN algorithms, including tree-based, hash-based, and graph-based methods, perform efficiently for low-dimensional applications but struggle with large-scale, high-dimensional vectors [19], [29], [36]. In contrast, the Inverted File (IVF[1]) index (or cluster-based ANN index) is commonly employed to handle large-scale, high-dimensional vectors effectively [11].

Fig. 1 (right) illustrates the workflow of the Inverted File (IVF) index. ① **Clustering Phase:** The database vectors are partitioned into *nlist* disjoint clusters by applying a clustering algorithm, typically k-means, to generate a list of centroids, where each centroid represents the center of a cluster. ② **Adding Phase:** Once the centroids are established, each database vector is assigned to its nearest centroid based on distance function and stored in the corresponding inverted file list associated with that centroid. This structure enables efficient search by restricting future queries to only the most relevant clusters. ③ **Search Phase:** The system begins by calculating the distances between the query vector and all centroids to identify the *nprobe* closest centroids, which represent the most relevant clusters for the query. Then, only the vectors within the top-*nprobe* closest centroid-associated clusters are scanned. For each vector in these clusters, the system performs distance calculations and employs a top-k sorting mechanism to determine the final k-NN results.

3) Off-The-Shelf NMP system: UPMEM

Near-memory processing (NMP) technology, which places computing units closer to memory, offers a promising solution to alleviate the data movement bottleneck across the shared bus between memory and compute units. The UPMEM NMP system is an off-the-shelf commercial solution, consisting of host CPUs, DRAM, and NMP memory modules (UPMEM DIMM modules), as shown in Fig. 2. The current system supports up to 20 UPMEM DIMMs, with each DIMM containing 128 DRAM processing units (DPUs), which are specialized

[1]IVF is included in industrial vector similarity search libraries, such as Meta FAISS.

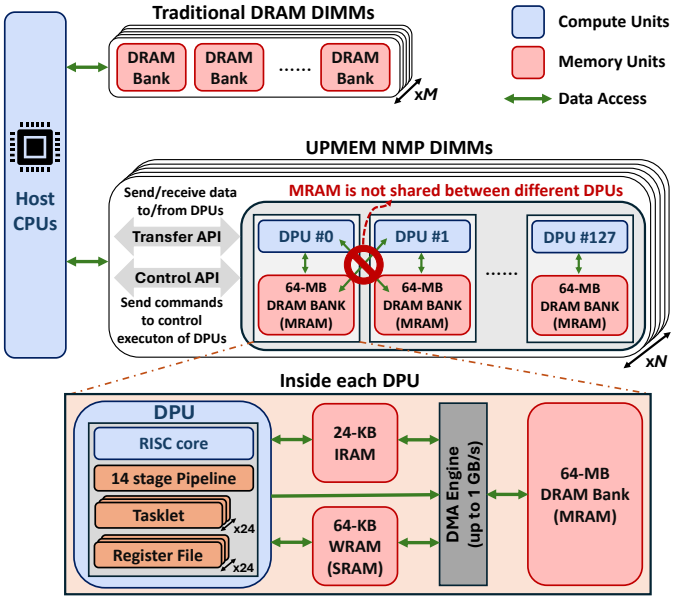


Fig. 2: The high-level architecture of UPMEM NMP system.

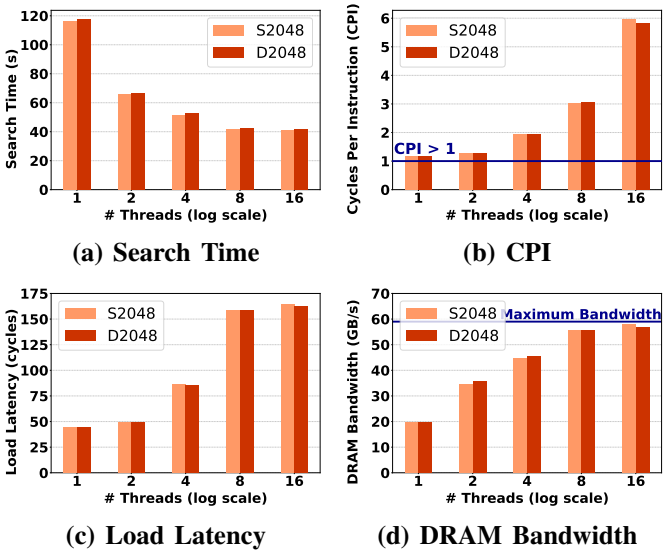


Fig. 3: Experimental evaluation of FAISS-IVF. The datasets and parameter settings are provided in Table I and Table II, respectively.

RISC processors embedded directly within standard DDR4 DRAM chips. Each DPU is connected solely to a dedicated 64 MB local DRAM bank, called MRAM, along with a 24 KB instruction memory (IRAM) and a 64 KB fast working memory (WRAM) serving as cache buffer. DPUs can only access their own MRAM, with no support for direct inter-DPU communication, thereby preventing data sharing between DPUs. Developers can implement customized WRAM memory management for their applications, optimizing resource utilization and improving overall performance. A DPU has 24 hardware threads, also known as tasklets, with each tasklet containing 24 32-bit registers. The DPU has a pipeline depth of 14 stages; however, due to hardware constraints, at least 11 tasklets must run concurrently to fully utilize the pipeline.

The UPMEM SDK [31] provides both control and transfer APIs, enabling the host CPUs to send and receive data to and from the MRAM of each DPU. This allows indirect communication between DPUs through host CPU intervention by copying data via the intermediate host DRAM. In addition, the host CPUs issue commands to the DPUs to control and manage their execution, enabling coordinated operation across multiple DPUs. Each DPU's internal DRAM bandwidth can reach up to 1 GB/s [4]. As a result, when all DPUs are active, the UPMEM NMP system achieves an aggregate internal DRAM bandwidth of up to 2.56 TB/s.

B. Observation & Motivation

Vector similarity search engines, such as Meta FAISS [6], [25], have been extensively optimized for traditional CPU architectures [22], [37]. However, these systems are fundamentally constrained by the von Neumann architecture, where CPU cores share memory and a common bus. When multiple threads perform concurrent k-Nearest Neighbors (kNN) searches, data movement between memory and CPU cache quickly saturates memory bandwidth, preventing linear performance scaling with added threads. Our analysis reveals that vector similarity search engines are memory-bound due to a low CPU cache reuse rate and low computational intensity. Most database vectors in the CPU cache are used for a single distance calculation and then flushed, limiting reuse. Additionally, each distance calculation involves few instructions yet requires moving large vectors, which fails to amortize the cost of data transfer. This issue is particularly evident in modern implementations like FAISS, which employs Intel Advanced Vector Extensions 2.0 (AVX2[2]) to process multiple data simultaneously per instruction.

We conduct a series of observational experiments, as shown in Fig. 3, to demonstrate that FAISS-IVF performance saturates as additional threads are added and to analyze the underlying causes of this saturation. We run Meta's FAISS-IVF on a system featuring an Intel i7-12700 processor and 128 GB of DDR5 DRAM, providing a theoretical peak DRAM bandwidth of 59 GB/s. We use Intel Vtune to profile the hardware usage while running FAISS-IVF. As shown in Fig. 3(a), the total search times for running FAISS-IVF over S2048 and D2048 datasets scale with the number of CPU threads, with the x-axis representing thread count and the y-axis indicating total vector search time. The results reveal that FAISS-IVF achieves limited performance gains and poor scalability as thread count increases, with overall search time plateauing beyond 8 threads.

Fig. 3(b) illustrates the cycles per instruction (CPI), with the x-axis representing the number of CPU threads and the y-axis denoting the CPI. The results indicate that CPI increases with the number of CPU threads, exceeding 1 for both datasets, which suggests the system is likely memory-bound. This is notable given that modern superscalar processors can issue up to four instructions per cycle. To investigate the high CPI,

[2]Intel AVX2 is an x86 instruction set extension enabling 256-bit wide processing for integer operations.

Fig. 3(c) shows that average load latency also rises with more CPU threads, likely due to congestion in the shared DRAM bus. This congestion causes load instructions to queue, increasing latency. Additionally, Fig. 3(d) reveals that DRAM bandwidth utilization saturates as thread count grows. By the time thread count reaches 8, DRAM utilization approaches its theoretical peak, confirming that limited performance gains stem from the shared memory bus failing to meet increased memory bandwidth demands. This challenge intensifies with high-dimensional vectors, as limited cache space can store fewer vectors. These observations highlight the need for a near-memory processing (NMP) system to address the data movement bottlenecks encountered in vector similarity search.

III. UPVSS: UPMEM-aware Vector Similarity Search

A. UPVSS System Architecture Design Overview

The primary constraints of UPMEM DPUs are *(1) each DPU's MRAM is limited to 64 MB, making efficient data placement a non-trivial task; (2) MRAM is not shared across DPUs, resulting in costly inter-DPU communication; and (3) to fully utilize the DPU hardware pipeline, at least 11 hardware threads (tasklets) must run concurrently.* This section introduces our UPMEM-aware Vector Similarity Search (UPVSS) system, designed to tightly integrate vector similarity search with UPMEM DPUs to address the data movement bottleneck encountered in von Neumann architecture. In addition, it aims to overcome the constraints of the DPU architecture and fully exploit its capabilities, particularly its high parallelism and substantial aggregate internal DRAM bandwidth, which can reach up to 2.56 TB/s.

Our UPVSS comprises two key components. First, the *DPU-aware Clusters Partition* evenly splits the vectors belonging to each IVF cluster, with considering the DPU architecture. This enables future k-NN search operations without requiring synchronization across DPUs and ensures that all DPUs are actively utilized, leveraging both the high parallelism and substantial aggregate internal DRAM bandwidth. Second, the *DPU-aware Vector Similarity Search Coordinator* significantly reduces data movement across the shared bus between host CPU and host memory during the search phase by offloading distance calculations to DPUs. Additionally, it facilitates collaboration between vector similarity search and DPUs and incorporates customized WRAM memory management to further optimize performance, considering running multiple tasklets to fully utilize the DPU hardware pipeline.

B. DPU-aware Clusters Partition

The key to implementing UPMEM-aware Vector Similarity Search lies in effectively distributing the vectors of each cluster across the DPUs. Upon receiving a query vector, it is then broadcast to each DPU that holds a portion of database vectors, allowing the DPUs to collaboratively work with the host CPUs to determine the final k-nearest neighbors (k-NN) results. To realize this, we propose *DPU-aware Clusters Partition*, which ensures a balanced distribution of clusters across all DPUs to maximize parallelism and resource utilization.

As shown in Fig. 4, our system begins with IVF index construction on the host side, where the host organizes the

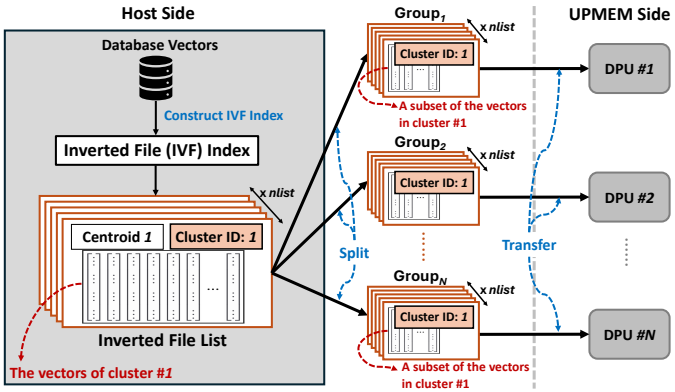


Fig. 4: DPU-aware Clusters Partition for UPVSS system.

database vectors into multiple clusters, each with an associated centroid. These database vectors are then inserted into the corresponding inverted file list of that centroid within the IVF index structure. To ensure all DPUs are activated during k-NN searches, we then evenly partition the vectors of each cluster into N groups. More precisely, the vectors within each centroid-associated cluster in the IVF index are divided into N equal segments, ensuring that each group receives a subset of every cluster. After partitioning, each of these N groups is then transferred to N distinct DPUs within the UPMEM NMP system, where N represents the maximum operational DPUs. With this partition strategy, all DPUs are actively utilized during the search phase, as each DPU contains a subset of every cluster.

C. DPU-aware Vector Similarity Search Coordinator

Fig. 5 provides a comprehensive overview of the DPU-aware Vector Similarity Search Coordinator, which encompasses the joint management and coordination between the host CPUs and DPUs. Given that the search phase primarily involves extensive distance calculations with low computational intensity, it cannot amortize the cost of data movement. Therefore, the core concept of DPU-aware Vector Similarity Search Coordinator is to offload these distance calculations between the query vector and database vectors to the DPUs. The host then collects $\langle ID, dist \rangle$ key-value pairs returned by all DPUs and merges them to obtain the final k-NN results. This approach significantly reduces data movement across the shared bus, thereby enhancing the CPU cache utilization, as each $\langle ID, dist \rangle$ pair is considerably smaller than each database vector. Furthermore, the distance calculations for vectors on each DPU are independent, eliminating the need for inter-DPU synchronization and thus effectively leveraging the high parallelism of the UPMEM system.

The workflow of the DPU-aware Vector Similarity Search Coordinator on the host side is as follows: ❶ The host receives the query vector and identifies the *nprobe* closest centroids, determining the specific *nprobe* clusters for the DPUs to perform distance calculations. The IDs of these clusters are then combined with the query vector to form the Query Metadata. ❷ The Query Metadata is broadcast to all DPUs. ❸ The host sends commands to launch each DPU, prompting them to execute distance calculations for the specified clusters

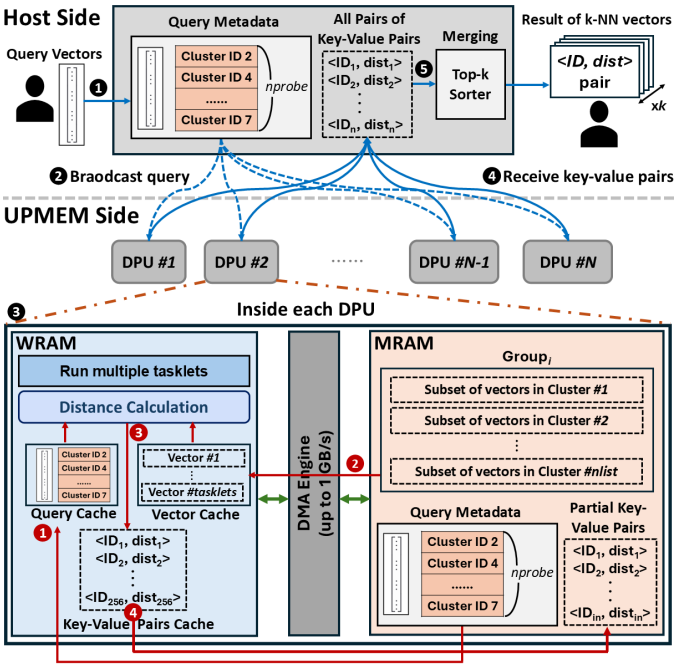


Fig. 5: The workflow of DPU-aware Vector Similarity Search Coordinator.

based on the cluster IDs in the Query Metadata. ❹ Upon completing the distance calculations, each DPU generates a set of $\langle ID, dist \rangle$ pairs as partial results, which are then sent back to the host CPU for further processing. ❺ Finally, the host merges all $\langle ID, dist \rangle$ from each DPU to determine the final k-NN results. Note that steps ❶ – ❺ are executed iteratively to continuously process incoming queries.

To further optimize performance and facilitate the efficient and concurrent execution of multiple tasklets, the workflow of the DPU-aware Vector Similarity Search Coordinator on the DPU side is organized as follows: ❶ We design a dedicated Query Cache in WRAM to store the Query Metadata, allowing it to be shared among all tasklets within the same DPU and ensuring rapid access. ❷ The subset of vectors assigned to each DPU is then evenly distributed across all active tasklets. Each tasklet has its own 2 KB Vector Cache in WRAM, allowing it to independently hold and process its assigned portion of the vectors. ❸ Subsequently, each tasklet performs distance calculations on its designated vectors, with the resulting $\langle ID, dist \rangle$ pairs stored in a shared Key-Value Pairs Cache accessible to all tasklets. ❹ The Key-Value Pairs Cache is configured with a fixed size of 2 KB, in accordance with the DPU architecture's limitation of a maximum 2 KB per WRAM-to-MRAM transfer [31]. Moreover, prior research indicates that this transfer size minimizes data movement overhead [9]. With this configuration, the cache can store up to 256 $\langle ID, dist \rangle$ pairs in WRAM. Once the Key-Value Pairs Cache reaches capacity, a designated tasklet is responsible for managing the transfer and writing the cached result back to persistent key-value storage in MRAM. The workflow and WRAM memory management strategy effectively optimize WRAM cache buffer utilization and enable concurrent multi-tasklet execution with minimal synchronization overhead, enhancing parallelism at the software level and fully utilizing the DPU pipeline at the hardware level.

IV. Experiment

A. Experiment Setup

This section evaluates the effectiveness of the proposed UPVSS system for improving the query processing performance. We compare the performance of the UPVSS with the state-of-the-art FAISS library, which is optimized using multi-threading with OpenMP, AVX2 SIMD vector instructions, and the BLAS (Basic Linear Algebra Subprograms) library to accelerate linear algebra computations. Both UPVSS and FAISS are evaluated using the IVF ANN indexing algorithm on the same system.

The experiments were conducted on a commercially available UPMEM near-memory processing (NMP) server running a Linux-based environment. The server is powered by two Intel Xeon Silver 4216 CPUs and equipped with 256 GB of DRAM, offering a theoretical peak bandwidth of 32 GB/s. Additionally, the system is equipped with 64 GB of UPMEM NMP memory distributed across 8 UPMEM DIMM modules, incorporating a total of 1024 DRAM Processing Units (DPUs). For baseline comparisons, the FAISS library was configured to run with 64 CPU threads.

The datasets used to evaluate the performance of the proposed UPVSS against FAISS are summarized in Table I. These datasets are synthesized based on the distributions of real-world SIFT [11], [12] and DEEP [35] datasets to generate high-dimensional vectors with dimensions of 2048, 3072, and 4096, named S2048, S3072, S4096, D2048, D3072, and D4096. Each dataset comprises 1 million (1M) database vectors and 1000 query vectors, with all vectors quantized to 8-bit integer format (uint8_t). Table II provides the default parameter settings used in our experiments along with a description of each parameter. Unless otherwise specified, all experiments were conducted using these default settings.

Dataset	# Dims	# Vectors	Dataset	# Dims	# Vectors
S2048	2,048	1,000,000	D2048	2,048	1,000,000
S3072	3,072	1,000,000	D3072	3,072	1,000,000
S4096	4,096	1,000,000	D4096	4,096	1,000,000

TABLE I: Datasets.

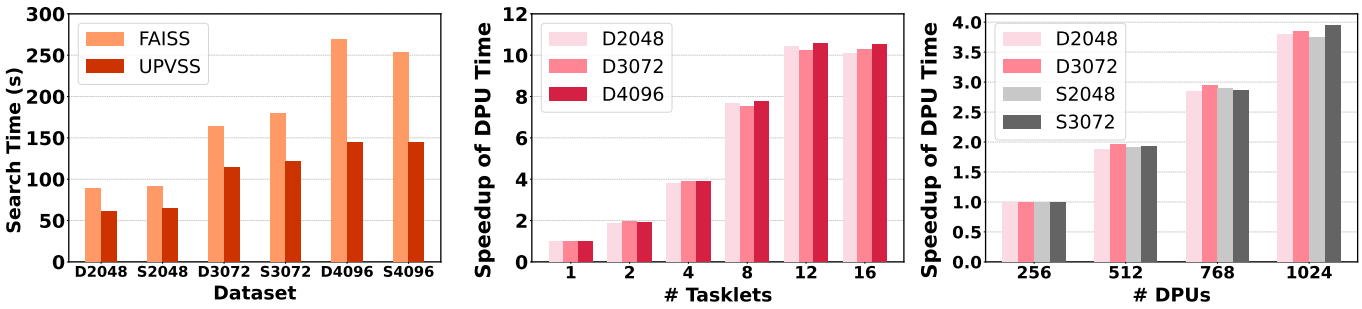
Parameters	Default value & Explanation
k	Top-k closest vectors when performing vector similarity search. **Default value: 100**
nlist	The number of clusters used to partition the database vectors. **Default value: 250**
nprobe	The number of nearest clusters explored during the k-NN search phase. **Default value: 20**

TABLE II: Parameters setting.

B. Experiment Results

1) Performance Results over State-Of-The-Art

Fig. 6(a) presents the total search time of the proposed UPVSS compared with the state-of-the-art FAISS, both running the IVF indexing algorithm across six datasets. The x-axis represents the six different datasets, while the y-axis indicates



(a) Search time comparison with FAISS. (b) Impact of tasklets on DPU time. (c) Scalability with increasing DPUs.

Fig. 6: Performance and scalability evaluation of UPVSS.

the total search time for UPVSS and FAISS on each dataset. The results show that UPVSS can reduce the overall search time by 30% to 46% and achieves a speedup of 1.42x to 1.86x compared to FAISS across all datasets.

To validate the efficiency of our WRAM memory management in DPU-aware Vector Similarity Search Coordinator under multi-tasklet execution, Fig. 6(b) presents the speedup of DPU execution time with respect to the number of tasklets per DPU across three datasets: D2048, D3072, and D4096. The x-axis represents the number of tasklets, while the y-axis denotes the speedup of DPU execution time relative to a single tasklet. The results show that the speedup of UPVSS scales linearly with the number of tasklets and saturates at 12 tasklets, consistent with the hardware constraints of the DPU. This indicates that UPVSS effectively maximizes the utilization of the DPU hardware pipeline.

2) Scalability of UPVSS

Fig. 6(c) shows the scalabiltiy of UPVSS with respect to the number of active DPUs. The x-axis represents the number of DPUs activated in the system, while the y-axis denotes the speedup of DPU execution time relative to 256 DPUs. The results show that the speedup of UPVSS scales linearly with the number of active DPUs. This scalability can be attributed to the DPU-aware Clusters Partition, which balances each cluster across DPUs, and the DPU-aware Vector Similarity Search, which offloads distance calculations to DPUs, significantly reducing data movement across the shared bus. Unlike conventional CPU architectures, UPVSS can further enhance performance by increasing the number of DPUs.

3) The impact of nlist

Fig. 7 shows the performance evaluations across six datasets for varying *nlist* values. The x-axis represents different *nlist* values, while the y-axis denotes the normalized speedup in performance relative to FAISS. The results indicate that UPVSS achieves a speedup of 1.42x to 2.62x compared to FAISS across all datasets, with an average speedup of 1.95x. Notably, as *nlist* value decreases, the number of vectors assigned to each cluster increases. This leads to more distance calculations per query, futher amplifying the memory-bound nature of FAISS. Consequently, UPVSS achieves higher speedups compared to FAISS.

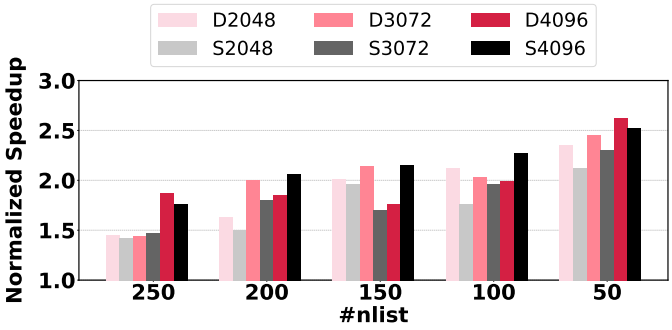


Fig. 7: Normalized speedup of UPVSS compared to FAISS for different *nlist* values.

V. Conclusion

Aiming to address the data movement bottleneck encountered in vector similarity search due to low computational intensity of distance calculations and substantial data movement across the shared bus, this paper proposes the UPMEM-aware Vector Similarity Search (UPVSS), an architecture-aware system that jointly manages vector similarity search and UPMEM's near-memory processing (NMP) technology to tackle this challenges. UPVSS comprises two key components: (1) *DPU-aware Clusters Partition*, which evenly splits vectors while accounting for the DPU's architectural constraints, unlocking its high parallelism and substantial aggregate internal DRAM bandwidth; and (2) *DPU-aware Vector Similarity Search Coordinator*, which significantly reduces data movement across the shared bus by offloading distance calculations to DPUs and incorporates customized WRAM memory management within the DPUs to fully utilize the DPU hardware resources. Evaluation results show that UPVSS achieves an average speedup of 1.95x compared to the state-of-the-art FAISS.

VI. Acknowledgement

We would like to thank Prof. David Brooks and Prof. Gu-Yeon Wei from Harvard University for their thoughtful comments and insightful suggestions. This work was supported in part by the National Science and Technology Council under grant nos. 113-2628-E-A49-021, 113-2640-E-A49-012 and Ministry of Education under Yushan Young Fellow Program.

REFERENCES

[1] A. Baumstark, M. A. Jibril, and K.-U. Sattler. Adaptive query compilation with processing-in-memory. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, pages 191–197, 2023.

[2] L.-C. Chen, C.-C. Ho, and Y.-H. Chang. Uppipe: A novel pipeline management on in-memory processors for rna-seq quantification. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[3] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 271–280, New York, NY, USA, 2007. Association for Computing Machinery.

[4] F. Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, 2019.

[5] S. Diab, A. Nassereldine, M. Alser, J. Gómez Luna, O. Mutlu, and I. El Hajj. A framework for high-throughput sequence alignment using real processing-in-memory systems. *Bioinformatics*, 39(5):btad155, 03 2023.

[6] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

[7] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, Jan. 2019.

[8] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang. Realm: retrieval-augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[9] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access*, 10:52565–52608, 2022.

[10] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei. s^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

[11] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[12] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864, 2011.

[13] H. Kang, Y. Zhao, G. E. Blelloch, L. Dhulipala, Y. Gu, C. McGuffey, and P. B. Gibbons. Pim-tree: A skew-resistant index for processing-in-memory. *Proc. VLDB Endow.*, 16(4):946–958, Dec. 2022.

[14] Y.-W. Kang, C.-F. Wu, Y.-H. Chang, T.-W. Kuo, and S.-Y. Ho. On minimizing analog variation errors to resolve the scalability issue of reram-based crossbar accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11), 2020.

[15] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *2009 IEEE 12th International Conference on Computer Vision*, pages 2130–2137, 2009.

[16] P. P.-H. Kung, Z. Fan, T. Zhao, Y. Liu, Z. Lai, J. Shi, Y. Wu, J. Yu, N. Shah, and G. Venkataraman. Improving embedding-based retrieval in friend recommendation with ann query expansion. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 2930–2934, New York, NY, USA, 2024. Association for Computing Machinery.

[17] C. Lim, S. Lee, J. Choi, J. Lee, S. Park, H. Kim, J. Lee, and Y. Kim. Design and analysis of a processing-in-dimm join algorithm: A case study with upmem dimms. *Proc. ACM Manag. Data*, 1(2), June 2023.

[18] T.-S. Lo, C.-F. Wu, Y.-H. Chang, T.-W. Kuo, and W.-C. Wang. Space-efficient graph data placement to save energy of reram crossbar. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2021.

[19] K. Lu, H. Wang, W. Wang, and M. Kudo. Vhp: approximate nearest neighbor search via virtual hypersphere partitioning. *Proc. VLDB Endow.*, 13(9):1443–1455, May 2020.

[20] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[21] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi, and A. Fedorova. A case study of Processing-in-Memory in off-the-Shelf systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 117–130. USENIX Association, July 2021.

[22] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang. Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4236–4247, 2024.

[23] OpenAI. GPT-3 Embedding. https://platform.openai.com/docs/guides/embeddings, 2024.

[24] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[25] M. A. Research. FAISS: A library for efficient similarity search and clustering of dense vectors. https://github.com/facebookresearch/faiss, 2024.

[26] T. Shen, G. Long, X. Geng, C. Tao, Y. Lei, T. Zhou, M. Blumenstein, and D. Jiang. Retrieval-augmented retrieval: Large language models are strong zero-shot retriever. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 15933–15946, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics.

[27] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[28] L. Song, X. Qian, H. Li, and Y. Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*, pages 541–552. IEEE, 2017.

[29] S. J. Subramanya, Devvrit, R. Kadekodi, R. Krishaswamy, and H. V. Simhadri. *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[30] C.-L. Tsai, C.-F. Wu, Y.-H. Chang, H.-W. Hu, Y.-C. Lee, H.-P. Li, and T.-W. Kuo. A digital 3d tcam accelerator for the inference phase of random forest. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[31] UPMEM. UPMEM SDK 2024.2.0 Documentation. https://sdk.upmem.com/2024.2.0/, 2024.

[32] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[33] C.-F. Wu, C.-J. Wu, G.-Y. Wei, and D. Brooks. A joint management middleware to improve training performance of deep recommendation systems with ssds. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 157–162, 2022.

[34] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang, P. Cheng, and M. Yang. Spfresh: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 545–561, New York, NY, USA, 2023. Association for Computing Machinery.

[35] A. B. Yandex and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2055–2063, 2016.

[36] W. Yang, T. Li, G. Fang, and H. Wei. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2241–2253, New York, NY, USA, 2020. Association for Computing Machinery.

[37] B. Zheng, Z. Yue, Q. Hu, X. Yi, X. Luan, C. Xie, X. Zhou, and C. S. Jensen. Learned probing cardinality estimation for high-dimensional approximate nn search. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 3209–3221, 2023.

[38] B. Zheng, X. Zhao, L. Weng, N. Q. V. Hung, H. Liu, and C. S. Jensen. Pm-lsh: A fast and accurate lsh framework for high-dimensional approximate nn search. *Proc. VLDB Endow.*, 13(5):643–655, Jan. 2020.

[39] Q. Zheng, Z. Wang, Z. Feng, B. Yan, Y. Cai, R. Huang, Y. Chen, C.-L. Yang, and H. H. Li. Lattice: An adc/dac-less reram-based processing-in-memory architecture for accelerating deep convolution neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.