Processing-In-Hierarchical-Memory Architecture for Billion-Scale Approximate Nearest Neighbor Search

Zhenhua Zhu*[1], Jun Liu*[1], Guohao Dai[2], Shulin Zeng[1], Bing Li[3], Huazhong Yang[1], Yu Wang[1]
[1]Dept. of EE, BNRist, Tsinghua University, [2]Shanghai Jiao Tong University, [3]Capital Normal University
Corresponding authors: daiguohao@sjtu.edu.cn, yu-wang@tsinghua.edu.cn

***Abstract*—Graph-based approximate nearest neighbor search (ANNS) algorithms achieve the best accuracy for fast high-recall searches on billion-scale datasets. Because of the irregular and large-volume data access, existing CPU-based systems suffer from heavy data movements when dealing with graph-based ANNS algorithms. Near-memory-computing (NMC) architectures have demonstrated great potential in boosting the performance of big-data processing. However, existing NMC architectures face two serious problems when processing graph-based ANNS algorithms: (1) the memory capacity of main memory level NMC (*e.g.*, 64GB) cannot meet the storage requirement of ANNS on billion-scale datasets (*e.g.*, 800GB), resulting in heavy data transfers between main memory and storage; (2) the contradiction between the irregular and fine-grained graph access and the page-level read granularity hinder the throughput of storage level NMC.**

This paper proposes *Pyramid*, the processing-in-hierarchical-memory architecture for graph-based ANNS on billion-scale datasets. *Pyramid* combines the internal bandwidth benefits of main memory level NMC with the capacity benefits of storage level NMC. A hierarchical graph-cluster-based ANNS is also proposed for *Pyramid*. It transforms the irregular data access on large-scale graphs into the irregular access on small-scale graphs at the main memory level and regular sequential in-cluster access at the storage level. Experimental results show that with the same recall of 0.9, *Pyramid* improves the throughput by 21.1∼72.8× and 26.0∼50.7× compared with existing CPU/GPU-based ANNS systems on million-scale and billion-scale datasets, respectively.

I. Introduction

Approximate nearest neighbor search (ANNS) aims to find the nearest (*i.e.*, with the highest similarity) features in the base datasets for a given query vector. ANNS is widely used in many applications, such as recommendation systems [1], images matching [2], semantic document retrieval [3], and other information retrieval applications. In recent years, among various ANNS algorithms, graph-based ANNS algorithms achieve the best accuracy [4]–[6] and are widely used in industry applications, such as Microsoft SPTAG [7].

Figure 1 shows the basic flow of graph-based ANNS algorithms. The algorithms firstly map the features in the dataset to the nodes in high-dimensional space and construct a graph structure using these nodes in the training phase. In the search phase, we traverse the graph from the starting node to find the nearest nodes to the query vector. The main operations of the search include visiting neighbor features, calculating distance between feature and query vector, and sorting distances.

Despite the high accuracy of graph-based ANNS algorithms, they suffer from low throughput on CPU when dealing with billion-scale datasets. The major performance bottleneck of CPU-based ANNS systems is the tremendous amount of data access that accounts for 80% of the total search time. The large data access overhead comes from the irregular, fine-grained (*e.g.*, 128-Byte per feature), and large-volume feature access due to the graph sparsity, which causes the bandwidth utilization and cache hit rate to be less than 39% and 16%, respectively.

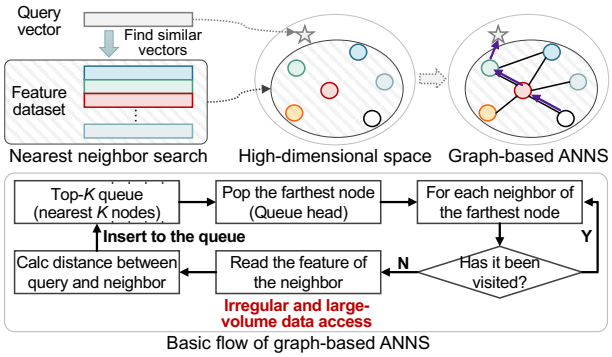

Fig. 1: Graph-based approximate nearest neighbor search.

Recently, near-memory-computing (NMC) architectures have shown great potential in improving the performance of memory-bound algorithms [8]–[13]. The NMC architectures place computation units inside memory (*e.g.*, dual-inline memory module, DIMM) or storage (*e.g.*, solid-state drives, SSD) to exploit the high internal bandwidth and reduce the data transfers among storage, main memory, and CPU.

However, directly applying graph-based ANNS algorithms to existing NMC architectures faces two challenges. On the one hand, the memory capacity of the **main memory level NMC** in one DIMM (*e.g.*, 64GB in [8], [9]) is hard to meet the storage requirements for billion-scale datasets and their corresponding graph structures (*e.g.*, 800GB). In this case, we still need massive data movements between the main memory level NMC and the storage, causing >10× throughput deterioration of NMC architectures. On the other hand, the **storage level NMC** architectures (also call as in-storage computing) demonstrate the virtue of TB level storage capacity. But the contradiction between the irregular and fine-grained feature access and SSD's page-level read granularity (*e.g.*, 4KB) leads to severe bandwidth under-utilization. What is worse, the read latency of SSD is relative long, *e.g.*, 53μs of SSD vs. 0.4μs of DIMM for 4KB read, which also limits the overall throughput improvement gain of the storage level NMC architectures.

To address the aforementioned challenges, this paper proposes *Pyramid*, the first processing-in-hierarchical-memory architecture for graph-based ANNS on billion-scale datasets. It combines the high internal bandwidth of main memory level

*: Both authors contributed equally to this work.

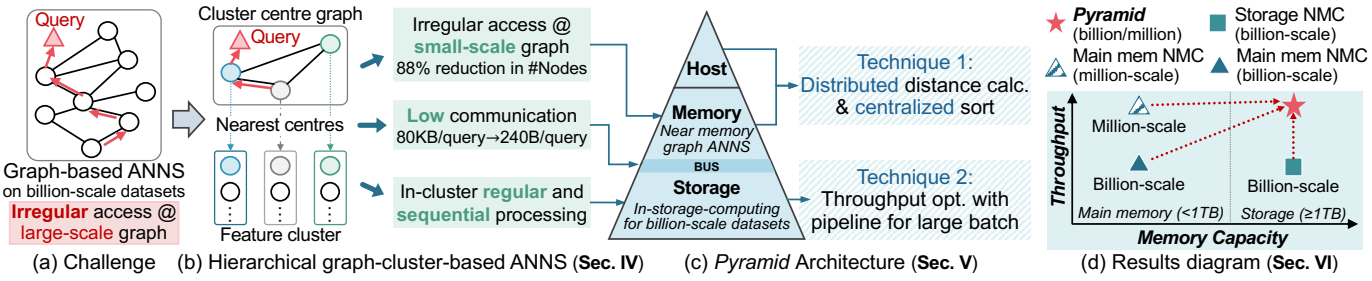

Fig. 2: *Pyramid* overview: (a) Challenges of graph-based ANNS; (b) Hierarchical graph-cluster-based ANNS; (c) *Pyramid* architectures; (d) Key results diagram.

NMC with the capacity benefits of storage level NMC. The contributions of this paper include:

- We propose the hierarchical graph-cluster-based ANNS for *Pyramid*. It transforms the irregular data access on large-scale graphs into two parts, *i.e.*, the irregular data access on small-scale clustering centre graphs and regular sequential data access in feature clusters. The two parts are compliant with the memory access characteristics of DIMM and SSD, respectively.
- We propose *Pyramid* to enable efficient near-memory ANNS. At the main memory level, *Pyramid* leverages the centralized sorting and distributed distance calculation to fully exploit the internal memory bandwidth with little hardware overhead. At the storage level, *Pyramid* reads the feature cluster in page granularity and supports in-storage sequential ANNS processing. We also optimize the pipeline workflow of *Pyramid* to improve the throughput for the large query batch.
- Experimental results show that with the same recall of 0.9, *Pyramid* shows 21.1∼72.8× ANNS throughput improvement on million-scale datasets compared with CPU/GPU. For billion-scale datasets, *Pyramid* demonstrates 26.0×, 50.7×, and 3.4× throughput improvement compared with CPU, single GPU, and 8-GPU system, respectively.

II. Preliminary: Graph-based ANNS

The graph-based ANNS algorithms are widely used to find the nearest features in the base datasets for a given query vector. The basic flow of graph-based ANNS algorithms is shown in Figure 1. During the training phase, a graph structure is constructed using the features in the dataset. The sparsity of the graph (*i.e.*, the ratio of the edge number to the square of the node number) is usually $\ll 1\%$, causing irregular data access. During the search phase, we hold a Top-K priority queue to keep the nearest K nodes to the query vector. In each search step, the node in the queue head (*i.e.*, the farthest node) is read out. Then each neighbor of the farthest node is visited and is determined whether it has been visited before. For the neighbor being visited for the first time, its feature is accessed (*e.g.*, 128-Byte fine-grained access) and used for distance calculation with the query vector. The calculated distance is inserted into the priority queue for subsequent searches. After sufficient search rounds, the nodes stored in the priority queue are the most similar features found by ANNS. The search accuracy is usually assessed using the recall rate, and the hardware search throughput is measured by query per second (QPS).

III. *Pyramid* Overview

To tackle the problem of heavy irregular data access in graph-based ANNS, we propose *Pyramid* to improve the throughput by fully exploiting the memory internal bandwidth and reducing data transfer. Figure 2 shows the overview of *Pyramid*.

Firstly, we propose the hierarchical graph-cluster-based ANNS at the algorithm level, as shown in Figure 2(b). The graph-cluster-based ANNS performs clustering onto the feature dataset and build the graph only use the cluster centres. Therefore, the irregular data access on the large-scale graph is transformed into the reduced irregular data access on small-scale graph and the in-cluster regular and sequential data access. Then the architecture of *Pyramid* is meticulously designed to execute computations of the hierarchical graph-cluster-based ANNS. *Pyramid* mainly consists of two parts: *Pyramid-M* at the main memory level and *Pyramid-S* at the storage level. As shown in Figure 2(c), *Pyramid-M* performs graph-based ANNS on the clustering centre graphs in DIMM. It leverages the centralized sorting and distributed distance calculation to reduce near memory hardware overhead. *Pyramid-S* reads the feature cluster in page granularity and performs sequential in-storage computing. The pipeline workflow between *Pyramid-M* and *Pyramid-S* is also optimized to further improve the throughput for large query batch.

Based on the high internal bandwidth of *Pyramid-M* and the capacity advantage of *Pyramid-S*, *Pyramid* achieves a better Pareto frontier in terms of throughput and memory capacity, as shown in Figure 2(d).

IV. Hierarchical Graph-Cluster-Based ANNS

In order to reconcile the fine-grained feature data access with the page-level read granularity of SSD, a straight forward idea is to store all the features to be calculated in one page and sequentially process all the features within this page. For achieving this goal, existing work presents a memory-storage hybrid searching system, SPANN [14]. SPANN stores the centroid points of the posting lists (*i.e.*, features) in the memory and the large posting lists in the storage and constructs space partition trees for nearest centroids search on CPUs.

Inspired by SPANN, we propose the hierarchical graph-cluster-based ANNS, which contains two layers, as shown in Figure 3. The bottom layer contains all the nodes of the feature

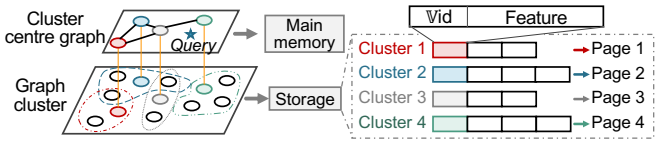

Fig. 3: Diagram of the hierarchical graph-cluster-based ANNS.

dataset. We apply the clustering algorithm used in SPANN to the nodes in the bottom layer. For each graph cluster, we assign multiple SSD pages to store its information that contains node index and feature. The centres of the graph clusters are extracted to form the top layer. We build a graph structure using these centres and store the graph and centre features in the main memory. The fine-grained graph data access of the top layer and the page-level cluster data access of the bottom layer are compliant with the memory access characteristics of DIMM and SSD, respectively. During the search, we perform the graph-based ANNS at the top layer to find the nearest centres, as introduced in Section II. The nearest centres determine the clusters need to be searched in the bottom layer. For the select cluster, all its features are read out and calculated one-by-one. Finally, all features of the total selected clusters are sorted by distance, and the nearest features are returned.

The virtues of the hierarchical graph-cluster-based ANNS are threefold. Firstly, the number of cluster centres is 1.2×10^8 for a billion-scale dataset (*i.e.*, feature number is 10^9). It demonstrates that 88% of the graph data are reduced, making it possible to store the entire centres-based graph in the main memory and leverage the NMC capability of *Pyramid-M*. Secondly, the nodes in the nearest cluster are processed sequentially, avoiding the irregular graph data access. The total feature size of one cluster is usually 4∼12KB, which also matches with the page-level read granularity (4∼16KB) of existing SSD. Thirdly, for each query, the number of the selected nearest nodes is ∼60, each of which is 4-Byte. Thus, we only need to transfer 240-Byte data between main memory and storage (80KB per query in CPU-based systems), effectively reducing the data traffic.

V. Architecture Design of *Pyramid*

A. Architecture Overview

Figure 4(a) depicts the architecture overview of *Pyramid*. *Pyramid* supports near memory ANNS in DIMM and SSD on the premise of retaining the original storage functions. *Pyramid-M*, *Pyramid-S*, and CPU communicate with each other through the memory controller hub (MCH). The MCH is also responsible for controlling the entire data flow. **At the main memory level**, *Pyramid-M* divides DIMMs into the neighbor DIMM and the feature DIMM for storing graph structures and feature datasets, respectively. For million-scale datasets, *Pyramid-M* stores the entire datasets and their corresponding graphs. While for billion-scale datasets, *Pyramid-M* only stores the clustering centre graphs and the features of cluster centres. The distance calculations are distributed in multiple NMC modules of feature DIMMs, and the distance results are sorted by a centralized priority queue module in the MCH. **At the storage level**, *Pyramid-S* stores the entire feature datasets based on the results of graph clustering. The computing functions of *Pyramid-S* are enabled when dealing with billion-scale datasets. *Pyramid-S* performs the in-cluster processing of the bottom layer in the hierarchical graph-cluster-based ANNS.

Figure 4(a) also illustrates the data flow of *Pyramid*. At the beginning of the search phase, the host CPU sends the query vector to the MCH ①. And the MCH flushes the priority queue. In each search step, the queue head is read out, and the MCH sends its node index ($\mathbb{V}_{\mathrm{id}}$) to the neighbor DIMMs ②. The neighbor DIMMs query the neighbor indices ($\mathbb{N}_{\mathrm{id}}$s) of $\mathbb{V}_{\mathrm{id}}$ ③ and return them to the MCH ④. Then, multiple $\mathbb{N}_{\mathrm{id}}$s are sent to different ranks of the feature DIMMs for distributed distance calculation ⑤. Before the distance calculation, the feature DIMMs leverage a Content-Addressable-Memory (CAM) [15] to filter the visited $\mathbb{N}_{\mathrm{id}}$ whose distance to the query vector has been calculated before. After that, they execute distance calculations for the unvisited $\mathbb{N}_{\mathrm{id}}$ ⑥ and return the distances to the MCH ⑦. The priority queue module in the MCH inserts these distance results to the queue and performs sorting ⑧. The main memory level graph-based search is completed after a fixed number of search steps (②∼⑧).

For medium-sized datasets (*e.g.*, million-scale), the priority queue contains the node indices of the nearest features to the query vector, which are the final output of ANNS ⑫. While for the large-sized datasets (*e.g.*, billion-scale), the nodes stored in the priority queue are the nearest cluster centres to the query vector. The corresponding cluster indices ($\mathbb{C}_{\mathrm{id}}$s) of these centres are sent to *Pyramid-S* ⑨. *Pyramid-S* executes in-storage distance calculations and sorting ⑩ and returns the nearest node list to the MCH ⑪.

B. Workload Analysis of Graph-Based ANNS

The graph-based ANNS contains three major operations: the neighbor indices access, the distance calculation, and the distance sorting. According the the operational intensity (operations per Byte), these three operations are memory and communication-intensive, memory-intensive, and computation-intensive, respectively.

Here we use one search step (Figure 4(a) ②∼⑧) of *Pyramid-M* on the SIFT10M dataset [16] as the case study for workload analysis. In SIFT10M, the feature dimension is 128 and each feature element is 1-Byte. The calculated distance is truncated to 4-Byte and Top-100 minimum distances are required. After constructing the graph structure using the dataset, each node has 40 neighbors on average and one neighbor index is 4-Byte.

In each search step, the **neighbor indices access** involves 160-Byte (40×4-Byte) sequential DRAM read and data transfer. The main computation is to generate the head address of neighbors. The address generation is negligibly time consuming compared to memory read and data transfer. For the **distance calculation**, we need to read 5,120-Byte (40×128×1-Byte) feature data from DRAM, perform 15,320 (40×(128 SUB+128 MUL+127 ADD)) operations, and transfer 160-Byte (40×4-Byte) distance results. The adjacent feature reads (two 128-Byte) are usually non-contiguous and irregular; and the

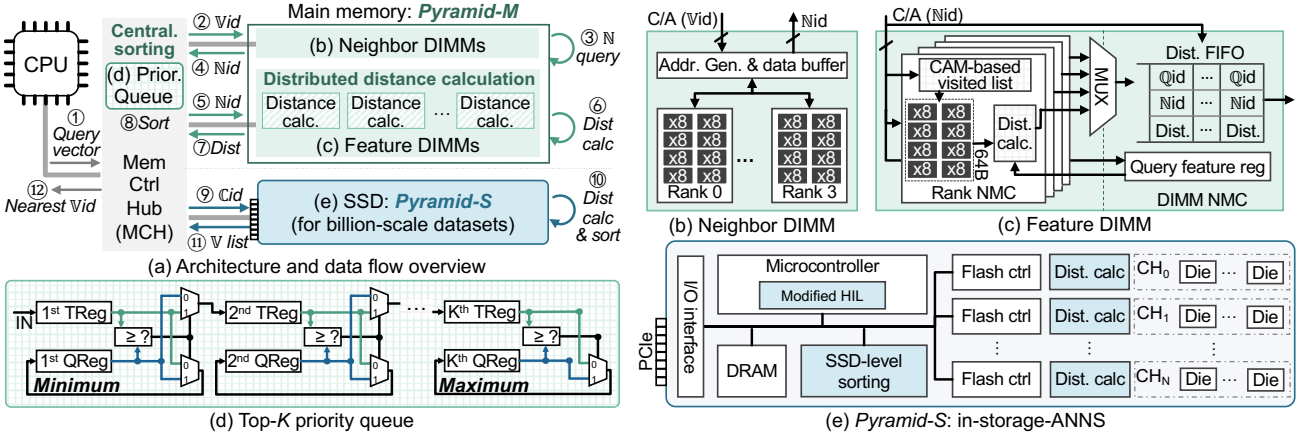

Fig. 4: (a) Overview and data flow of *Pyramid*; (b) neighbor DIMM; (c) feature DIMM; (d) Top-K priority queue module; (e) *Pyramid-S* architecture.

operational intensity is 2.99 operations per Byte. The **distance sorting** operation receives and inserts the 40 distances into the Top-100 priority queue. Each inserting incurs at most 100 4-Byte comparisons with the operational intensity of 25 operations per Byte. According to the above analysis, the **distance calculation** occupies more than 97% of the total memory access and data transfer, which definitely needs to be addressed using NMC. In *Pyramid-M*, different ranks and their NMC modules work independently, and the distance calculations for different features can also be paralleled. So we split and store the entire datasets into multiple ranks and design the rank-level NMC modules for distributed and parallel distance calculations. Further, to minimize the additional hardware costs, we reduce the number of multipliers and adders as much as possible. The size of sub-features processed at one time per NMC module is:

$$sub\text{-}feature_{size} = \frac{f_{DRAM} \times 8 \times 2}{f_{digital} \times B_f}, \quad (1)$$

where f_{DRAM} and $f_{digital}$ are the frequency of DRAM and NMC modules, respectively. B_f is the Byte size of each feature element. "8" and "2" mean the DRAM rank outputs 8-Byte data at a time with double data rate. Equation 1 ensures that the DRAM read and distance calculation can be pipelined.

While for the **distance sorting**, its performance bottleneck exists in the sorting operations rather than the data transfer. Therefore, placing the priority queue inside the NMC modules is hard to obtain the expected throughput gain. In addition, when handling the queries in batches (*i.e.*, process multiple queries concurrently), multiple priority queues must be placed, each of which is assigned to one query. However, the Top-K priority queue needs register resources of $O(K)$, the duplication of priority queue is expensive for NMC. Considering these limitations, we design the centralized priority queue module in the MCH, which has more relaxed hardware constraints and can efficiently interact with memory and storage.

C. Pyramid-M Architecture Details

Based on the conclusions of the previous workload analysis, we design *Pyramid-M* for graph-based ANNS. Figure 4(b)∼(d) show the architecture details of *Pyramid-M*.

Neighbor DIMM. The neighbor DIMM shown in Figure 4(b) is responsible for looking up the neighbors of the given node index. Compared with the original memory-oriented DIMMs, we add an address generation module at the DIMM level. The address generation module receives the node index ($\mathbb{V}_{\text{id}}$) and calculate the head address of its neighbors.

Feature DIMM. Figure 4(c) illustrates the design of the feature DIMM that contains the rank-level CAM-based visited list, the rank-level distance calculation module, and the DIMM-level distance FIFO (first-in-first-out). At the rank level, we store the indices of the visited nodes in the CAM-based visited list, each CAM row stores one visited node index. When a new neighbor index arrives ($\mathbb{N}_{\text{id}}$), the CAM is used for comparing the $\mathbb{N}_{\text{id}}$ with all visited node indices in parallel. If the $\mathbb{N}_{\text{id}}$ is found in the CAM, *i.e.*, the distance of the $\mathbb{N}_{\text{id}}$ to the query has been calculated, the rank-level NMC module processes the next neighbor. Otherwise, the $\mathbb{N}_{\text{id}}$ is written into the CAM and its feature is read out from DRAM. The readout feature is then split into multiple sub-features and sent to the distance calculation module one-by-one. We also design a distance FIFO at the DIMM level to buffer the calculated distances from the ranks in this DIMM. Each entry of the distance FIFO contains three parts: the query ID ($\mathbb{Q}_{\text{id}}$), the neighbor index ($\mathbb{N}_{\text{id}}$), and its distance result.

Centralized Top-K priority queue module. To improve the throughput of sorting, we design a priority queue module with one input per cycle, which is shown in Figure 4(d). The priority queue module uses $2K$ registers for high-throughput Top-K sorting, which consist of K queue registers (QRegs) and K temporary registers (TRegs). The queue head represents the minimum distance. In each cycle, the input distance is written into the first TReg. For other registers, if the distance stored in the i^{th} TReg is larger than that in the i^{th} QReg, the data of the i^{th} TReg is transferred to the $(i+1)^{th}$ TReg. Otherwise, the data of the i^{th} TReg is written into the i^{th} QReg, and the data of the i^{th} QReg is written into the $(i+1)^{th}$ TReg.

D. Pyramid-S Architecture and Workflow Optimization

Figure 4(e) depicts the architecture overview of *Pyramid-S*. The architecture design is also based on the idea of centralized

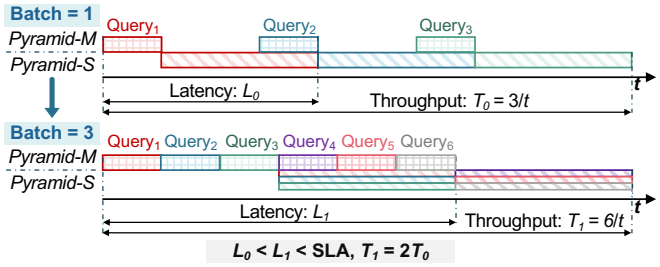

Fig. 5: *Pyramid* workflow optimization.

TABLE I: Memory configurations used in *Pyramid*.

DIMM configurations	
Memory capacity	64GB
Channels/Rank per channel	2/8
NMC per rank	2
DDR4 DRAM parameters	
DDR4 configurations	4Gb x8 3200
Clock frequency	1600MHz
tRCD-tCAS-tRP	22-22-22
SSD configurations [13]	
Storage/Page capacity	1TB/16KB
Channels/Chips per channel	32/4
Flash array access latency	53μs
Flash channel bandwidth	800MHz

sorting and distributed distance calculation. *Pyramid-S* contains three modifications compared with original SSDs. Firstly, at the micro-controller level, the flash firmware of host interface layer (HIL) is modified to enable the translation between the centre index and the logical block address (LBA). Secondly, at the flash channel level, since different channels can work simultaneously, we attach one distance calculation module to each flash controller to explore the channel-level parallelism. Thirdly, at the SSD level, multiple priority queues are placed to sort the distances from all channels, and the queue number determines the largest batch size that *Pyramid-S* supports.

To improve the throughput and making *Pyramid-M* and *Pyramid-S* work better together, we optimize the workflow of *Pyramid* as shown in Figure 5. Considering the metric of throughput under latency constraint (*i.e.*, service-level agreement, SLA) in server applications, the key idea is to improve the throughput with acceptable latency overhead. Since the effective bandwidth of SSD increases with more requests [17], we increase the number of queries (*i.e.*, batch size) processed by *Pyramid-S* at a time. Besides, the search latency of *Pyramid-M* is lower than *Pyramid-S*. Therefore, in order to be pipelined with *Pyramid-S*, *Pyramid-M* processes the queries one-by-one, which can also avoid the hardware duplication for large batch size. In the example shown in Figure 5(c), by setting the batch size to three, the throughput is improved by 2× while the latency still satisfies the SLA.

VI. Experimental Results

A. Experimental Setup

1) Pyramid setup: We build a trace-based ANNS simulator and use the memory configurations in Table I to simulate *Pyramid*. We use Ramulator [18] and MQSim [19] to evaluate

TABLE II: Throughput results (QPS) on million-scale datasets.

	SIFT		*SPACEV*		*GeoMean*
	10M	*100M*	*10M*	*100M*	
CPU	7116	3443	5738	3362	4662
GPU	39039	14846	20427	5648	16108
Pyramid	403637	301974	336813	323056	339357
Speedup to CPU	56.7×	87.7×	58.7×	96.1×	**72.8**×
Speedup to GPU	10.3×	20.3×	16.5×	57.2×	**21.1**×

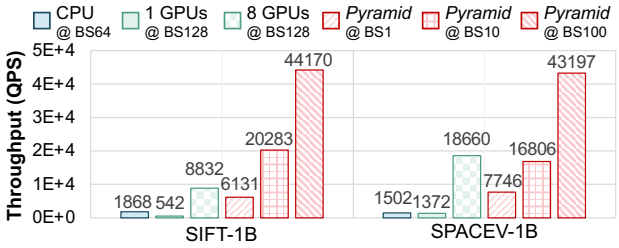

Fig. 6: Throughput results (QPS) of CPU, GPU, and *Pyramid* on billion-scale datasets, "BS" represents batch size.

DRAM and SSD, respectively. The NMC modules are modeled using Synopsys Design Compiler with TSMC 65nm technology library at the frequency of 500MHz. We scale the performance results to 32nm technology node according to [20].

2) Baseline: For the CPU baseline, we run two SOTA graph-based ANNS algorithms HNSW [5] and SPANN [14] on the Intel Xeon 8-core CPU running at 2.1GHz with 64GB DIMM for million-scale and billion-scale datasets, respectively. We run the SOTA GPU-based design GGNN [21] on Nvidia RTX 3090 with 24GB memory as the GPU baseline. The storage-based NMC baseline we used refers to [12].

3) Datasets: We use two representative ANNS datasets with different scales, SIFT [16] and SPACEV [22]. The throughput is evaluated with the recall of 0.9.

B. Performance on Million-scale Datasets

Table II shows the throughput of *Pyramid*, CPU, and GPU on million-scale datasets (*10M* for ten million and *100M* for one hundred million). Since the main memory can store the entire million-scale datasets, *Pyramid-S* is disabled in *Pyramid*. *Pyramid* can achieve 72.8× and 21.1× throughput improvement compared with CPU and GPU, respectively. The relative high throughput of GPU is achieved at the cost of high latency, causing GPU cannot satisfy strict SLA constraints (*e.g.*, 1ms). The average search latency of CPU, GPU, and *Pyramid* is 0.2ms, 5.1ms, and 3.0μs, respectively.

C. Performance on Billion-scale Datasets

1) Comparisons to CPU and GPU: Figure 6 provides the throughput results of CPU, GPU, and *Pyramid* with different batch sizes. Note that the memory of single GPU cannot meet the storage requirement of billion-scale datasets, which needs frequent data interaction with SSD. So we also use an 8-GPU system with 192GB memory as the baseline. *Pyramid* with the batch size of 100 achieves 26.0×, 50.7×, and 3.4×

TABLE III: Throughput results (QPS) of different NMC architectures on billion-scale datasets.

	SmartSSD [12]	*Pyramid-M* only	*Pyramid-S* only	*Pyramid*
SIFT	76	2003	94	44170
SPACEV	Not provided	1685	63	43197

TABLE IV: *Pyramid* hardware overhead.

	Pyramid-M dist. calc.	*Pyramid-M* sorting	*Pyramid-S*
Area (μm^2)	170589	996	10041658
Power (mW)	68.39	0.62	6244

throughput improvements compared with CPU, single GPU and 8-GPU, respectively. As shown in Figure 7, when improving the batch size from one to 100 with the optimized workflow, the throughput of *Pyramid* is increased by 5.6∼7.2×. Further, the throughput gain from increasing the batch size becomes small as the memory bandwidth bound is approached. Therefore, we use 100 as the optimal batch size.

2) Comparisons with existing NMC architectures: Table III demonstrates the comparison among *Pyramid* and other NMC designs/configurations on billion-scale datasets. For the in-storage processing architectures, *i.e.*, the SmartSSD-based design [12] and *Pyramid-S* only, *Pyramid* achieves 469.9∼685.6× throughput improvements. The underutilized memory bandwidth and frequent data movements between SSD internal DRAM and flash arrays cause the serious performance deterioration. In the main memory level NMC baseline, we use CPU and original SSD to process the bottom level graph clusters. Compared with this *Pyramid-M* only design, *Pyramid* shows 22.1∼25.6× throughput improvement.

D. Area and Power Analysis

Table IV provides the area and power overhead of *Pyramid*. For *Pyramid-M*, the distance calculation module causes $0.17mm^2$ and 68.39mW additional overhead per DIMM, which is much smaller than the area ($\sim 3500mm^2$) and power of entire DIMM (∼10W). For *Pyramid-S*, we place 100 SSD-level sorting modules for batch size of 100. These sorting modules together with channel-level distance calculation modules occupy $10.04mm^2$ and 6.24W, which meet the area budget of $30mm^2$ and the power budget of 55W for in-storage computing [13].

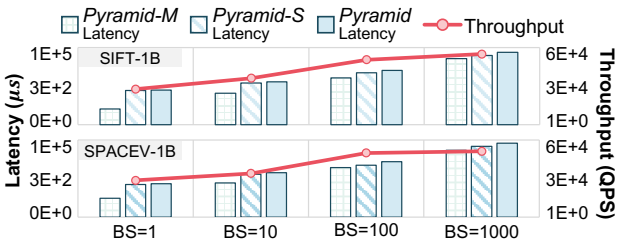

Fig. 7: Latency and throughput of *Pyramid* with different batch sizes (BS).

VII. Conclusion

This paper proposes *Pyramid*, the first Processing-In-Hierarchical-Memory architecture for graph-based ANNS on billion-scale datasets. By combining high internal bandwidth of main memory level NMC with the capacity benefits of storage level NMC, *Pyramid* can achieve 21.1∼72.8× and 26.0∼50.7× throughput improvements compared with CPU/GPU-based ANNS on million-scale and billion-scale datasets, respectively.

VIII. Acknowledgement

This work was supported by the National Natural Science Foundation of China (No. 62104128, U19B2019, 61832007, U21B2031, 62204164), Tsinghua EE Xilinx AI Research Fund, Tsinghua-Meituan Joint Institute for Digital Life, and Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] J. Johnson *et al.*, "Billion-scale similarity search with gpus," *IEEE TBD*, vol. 7, no. 3, pp. 535–547, 2019.
[2] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *IJCV*, vol. 60, no. 2, pp. 91–110, 2004.
[3] S. Deerwester *et al.*, "Indexing by latent semantic analysis," *JASIST*, vol. 41, no. 6, pp. 391–407, 1990.
[4] W. Li *et al.*, "Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement," *IEEE TKDE*, vol. 32, no. 8, pp. 1475–1488, 2019.
[5] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE TPAMI*, vol. 42, no. 4, pp. 824–836, 2018.
[6] S. Jayaram Subramanya *et al.*, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *NeurIPS*, vol. 32, 2019.
[7] Microsoft, "Sptag: A library for fast approximate nearest neighbor search," [Online], 2022, https://github.com/microsoft/SPTAG.
[8] G. Dai *et al.*, "Dimmining: pruning-efficient and parallel graph mining on near-memory-computing," in *ISCA*, 2022, pp. 130–145.
[9] L. Ke *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *ISCA*. IEEE, 2020, pp. 790–803.
[10] W. Huangfu *et al.*, "Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm," in *Micro*, 2019, pp. 587–599.
[11] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Micro*, 2019, pp. 740–753.
[12] J.-H. Kim *et al.*, "Accelerating large-scale graph-based nearest neighbor search on a computational storage platform," *IEEE TC*, no. 01, pp. 1–1, 2022.
[13] V. S. Mailthody *et al.*, "Deepstore: In-storage acceleration for intelligent queries," in *Micro*, 2019, pp. 224–238.
[14] Q. Chen *et al.*, "Spann: Highly-efficient billion-scale approximate nearest neighborhood search," *NeurIPS*, vol. 34, pp. 5199–5212, 2021.
[15] S. Jeloka *et al.*, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE JSSC*, vol. 51, no. 4, pp. 1009–1021, 2016.
[16] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *ICASSP*. IEEE, 2011, pp. 861–864.
[17] M. Jung *et al.*, "Simplessd: Modeling solid state drives for holistic system simulation," *IEEE CAL*, vol. 17, no. 1, pp. 37–41, 2017.
[18] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE CAL*, vol. 15, no. 1, pp. 45–49, 2015.
[19] A. Tavakkol *et al.*, "Mqsim: a framework for enabling realistic studies of modern multi-queue ssd devices," in *FAST*, 2018, pp. 49–65.
[20] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.
[21] F. Groh, L. Ruppert, P. Wieschollek, and H. Lensch, "Ggnn: Graph-based gpu nearest neighbor search," *IEEE TBD*, 2022.
[22] "Spacev datasets," [Online], 2022, https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B.