Chameleon: a Heterogeneous and Disaggregated Accelerator System for Retrieval-Augmented Language Models

Wenqi Jiang
Systems Group, ETH Zurich
wenqi.jiang@inf.ethz.ch

Marco Zeller
Systems Group, ETH Zurich
mzeller@student.ethz.ch

Roger Waleffe
University of Wisconsin Madison
waleffe@wisc.edu

Torsten Hoefler
SPCL, ETH Zurich
torsten.hoefler@inf.ethz.ch

Gustavo Alonso
Systems Group, ETH Zurich
alonso@inf.ethz.ch

ABSTRACT

A Retrieval-Augmented Language Model (RALM) combines a large language model (LLM) with a vector database to retrieve context-specific knowledge during text generation. This strategy facilitates impressive generation quality even with smaller models, thus reducing computational demands by orders of magnitude. To serve RALMs efficiently and flexibly, we propose *Chameleon*, a heterogeneous accelerator system integrating both LLM and vector search accelerators in a disaggregated architecture. The *heterogeneity* ensures efficient serving for both inference and retrieval, while the *disaggregation* allows independent scaling of LLM and vector search accelerators to fulfill diverse RALM requirements. Our Chameleon prototype implements vector search accelerators on FPGAs and assigns LLM inference to GPUs, with CPUs as cluster coordinators. Evaluated on various RALMs, Chameleon exhibits up to 2.16× reduction in latency and 3.18× speedup in throughput compared to the hybrid CPU-GPU architecture. The promising results pave the way for adopting heterogeneous accelerators for not only LLM inference but also vector search in future RALM systems.

PVLDB Reference Format:
Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. Chameleon: a Heterogeneous and Disaggregated Accelerator System for Retrieval-Augmented Language Models. PVLDB, 18(1): 42 - 52, 2024.
doi:10.14778/3696435.3696439

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fpgasystems/Chameleon-RAG-Acceleration.

1 INTRODUCTION

Vector databases facilitate the development of *Retrieval-Augmented Language Models (RALMs)*, an increasingly popular approach to serve generative large language models (LLMs). The architecture of RALM, as shown in Figure 1, allows the LLM to focus on learning linguistic structures, while incorporating context-specific knowledge during inference. Specifically, the external textual knowledge is encoded as vectors using LLMs and stored in a vector database. Given an inference context (e.g., a prompt), the knowledge retriever identifies relevant knowledge in the database via vector search, which assesses relevance by computing the similarity between the context vector and the database vectors. The retrieved texts are then incorporated into the LLM to facilitate high-quality generation.

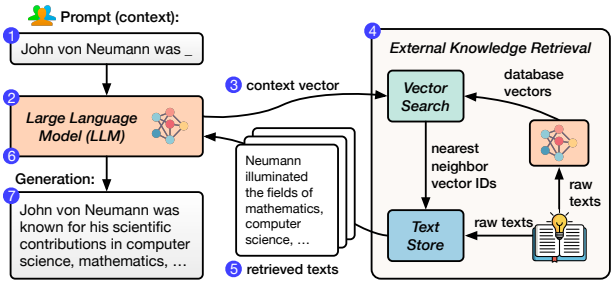

Figure 1: A retrieval-augmented language model (RALM).

RALMs show three major advantages over conventional LLMs. *First of all*, RALMs, even using smaller LLMs with one to two orders of magnitude fewer parameters, can match or surpass the generation quality of conventional LLMs on various tasks [23, 31, 41–43, 48, 49], thus significantly lowering the inference cost. This is because conventional LLMs rely on a vast number of parameters trained on massive datasets to capture and retain textual knowledge [8, 12, 67, 74], while RALMs can integrate retrieved knowledge during inference, not burdening the LLM's parameters. *Moreover*, knowledge editing in RALMs is as straightforward as updating the database, enabling efficient integration of new or private knowledge [3, 4]. In contrast, updating knowledge in conventional LLMs is inflexible, requiring additional training [7, 49]. *Finally*, RALMs enhance the reliability and interpretability of generated content by sourcing knowledge externally, while conventional LLMs are prone to producing non-factual content, known as hallucination [49, 50].

Despite its advantages, efficient RALM inference presents **two challenges**. *First, the workload characteristics of the LLM and the retriever are distinct.* While the LLM inference primarily relies on rapid tensor operations, the vector search system — often utilizing fast and memory-efficient search algorithms like Product Quantization (PQ) [35] — demands both substantial memory capacity to hold the vectors and fast processing of quantized database vectors during query time. *Second, the diverse range of RALM configurations leads to shifting system requirements and bottlenecks.* Regarding retrieval frequency, some models retrieve once per generated token [6, 42, 57], while others retrieve only once per entire sequence [31, 49]. In

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:10.14778/3696435.3696439

terms of scale, database sizes vary from millions [24, 49] to tens of billions of vectors (92 TB) [7, 76], and model sizes range from hundreds of millions [24, 48] to tens of billions of parameters [76].

We envision a high-performance and efficient RALM system to adhere to **two key design principles** to address the two aforementioned challenges. *Firstly, RALMs should incorporate **heterogeneous accelerators**, employing not only inference accelerators such as GPUs but also vector search accelerators*, such that both RALM components are fast and efficient. *Secondly, the heterogeneous accelerators should be **disaggregated** to support diverse RALM demands efficiently*, in contrast to a monolithic approach where a fixed number of LLM and retrieval accelerators reside on the same server. The rationale is twofold: (a) performance bottlenecks shift between various RALMs of different retrieval frequencies, database sizes, and model sizes, thus requiring a case-specific optimal balance between the two types of accelerators; and (b) a huge database (e.g., with tens of TBs of vectors [7, 76]) may necessitate more retrieval accelerators than a single server can accommodate.

To materialize this vision, we propose *Chameleon*, a heterogeneous and disaggregated accelerator system for efficient, flexible, and high-performance RALM inference. Chameleon consists of three primary components. Firstly, *ChamVS* is a distributed and accelerated vector search engine. It consists of several disaggregated memory nodes, each containing a shard of quantized database vectors in DRAM, a near-memory retrieval accelerator prototyped on FPGA, and a hardware TCP/IP stack. Secondly, *ChamLM* is a multi-GPU LLM inference engine. It produces query vectors and generates texts using the retrieved information. Lastly, a CPU coordinator server orchestrates the network communication between the retrieval and LLM accelerators.

We evaluate Chameleon with various LLM architectures, model sizes, database sizes, and retrieval frequencies. For large-scale vector search, ChamVS achieves up to 23.72× latency reduction compared to the optimized CPU baselines while consuming 5.8~26.2× less energy. For RALM inference, Chameleon achieves up to 2.16× and 3.18× speedup in latency and throughput compared to the hybrid CPU-GPU architecture. We further illustrate that the optimal balance between the two types of accelerators varies significantly across different RALMs, making disaggregation essential for achieving both flexibility and high accelerator utilization rates.

The paper makes the following **contributions:**

- We present Chameleon, an efficient RALM inference system designed around two proposed principles: accelerator heterogeneity and disaggregation.
- We design and implement ChamVS, a distributed engine for large-scale vector search, which includes:
  - Near-memory accelerators for vector search, including a novel resource-efficient top-K selection architecture.
  - A GPU-based index scanner to prune search space.
- We evaluate Chameleon on various RALMs and showcase its remarkable performance and efficiency.

2 BACKGROUND AND MOTIVATION

2.1 Retrieval-Augmented Language Models

A RALM combines an LLM [16, 66, 68] with a vector database. During inference, information relevant to the current context is retrieved from the database and utilized by the LLM to predict subsequent tokens. *We classify RALMs by the content they retrieve:*

The first category of RALMs retrieves *text chunks* containing multiple tokens related to the current context [7, 31, 39, 49, 70]. During inference, the generation context, such as a user's prompt, is encoded as a query vector to retrieve context-related knowledge, i.e., text chunks in the database with similar vector representations [7, 32, 49]. The retrieved text chunks are then integrated by the LLM, leading to the generation of output tokens. When generating long sequences, however, the generated content may gradually diverge from the initially retrieved contents. Thus, instead of initiating retrieval only once at the beginning [31, 49, 72], an effective strategy is to perform multiple retrievals during text generation to improve token generation quality [70], for instance, at a regular interval of every 8~64 generated tokens [7, 39, 70].

The second category of RALMs retrieves only the *next token* of each similar context in the database [6, 42, 57]. At each step of token generation (retrieval interval is one), the last layer's hidden state serves as the query to retrieve similar contexts and the next token of each similar context [42, 57, 82]. The next token of the current context is then predicted by interpolating the next-token probability distribution predicted by the model with that of the retrieved content [41, 42].

2.2 Large-Scale Vector Search

A vector search takes a D-dimensional query vector x as input and retrieves K similar vector(s) from a database Y, populated with many D-dimensional vectors, based on metrics like L2 distances or cosine similarity. Exact K nearest neighbor (KNN) search can be prohibitively expensive for large datasets, requiring a linear scan through all database vectors. Thus, real-world vector search systems adopt approximate nearest neighbor (ANN) search that can achieve much higher system performance. The quality of an ANN search is measured by the recall at K ($R@K$), which denotes the overlap percentage between the exact K nearest neighbors and the K returned by the ANN. In the subsequent sections, we will use the terms *vector search* and *ANN search* interchangeably.

IVF-PQ, combining the inverted-file (IVF) index and product quantization (PQ), is among the most popular vector search algorithms in RALMs [7, 31, 42, 49]. Its more frequent adoption over other ANN algorithms like graph-based vector search [17, 18, 53, 55, 56, 91, 94] is primarily due to memory efficiency: RALMs can involve large databases, with reported sizes reaching up to 30 billion vectors (92 TB) [7, 76], thus the high compression ratio offered by PQ [20, 35, 40] is essential.

Inverted-File (IVF) Index. An IVF index divides a vector dataset Y into many (*nlist*) disjoint subsets, typically using clustering algorithms like K-means. Each of these subsets is termed an IVF list. At query time, the IVF index is scanned, and only a select few (*nprobe*) IVF lists whose cluster centroids are close to the query vector are scanned, such that the search space is effectively pruned.

Product Quantization (PQ). PQ reduces memory usage and computations of vector search by compressing each database vector into m-byte PQ codes. Figure 2 overviews the workflow of PQ.

Training (quantization). All database vectors are partitioned evenly into m sub-vectors ①, which possess a dimensionality of

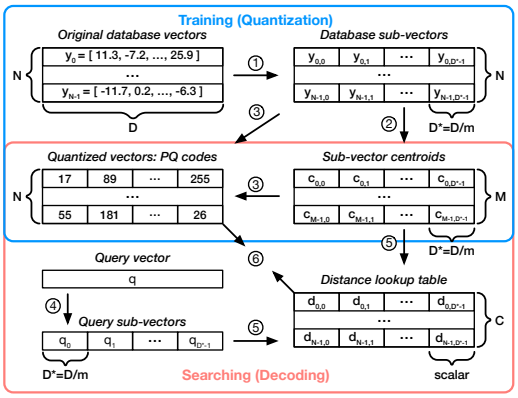

Figure 2: Product quantization (PQ) for vector search.

$D^* = \frac{D}{m}$, typically ranging from 4 to 16 in practice. A clustering algorithm is performed in each sub-space ② to obtain a list of centroids c, allowing each database sub-vector to be approximated by its nearest centroid. Typically, the number of clusters per sub-space is set as $M = 256$, such that a cluster ID can be represented with one byte. Thus, once the cluster centroids are stored, each database vector can be represented by m-byte PQ codes.

Searching (decoding). A query vector is compared against the quantized database vectors. The distance computation can be formulated as $\hat{d}(x, y) = d(x, c(y)) = \sum_{i=1}^{m} d(x_i, c_i(y_i))$, where $\hat{d}(x, y)$ is the approximate distance between a query vector x and a quantized database vector y, and $c(y)$ is the reconstructed database vector using the PQ codes and the cluster centroid vectors per sub-space. To calculate $\hat{d}(x, y)$, the query vector is divided into m sub-vectors (x_i) ④ and compared against the reconstructed quantized sub-database-vectors $c_i(y_i)$. To speed up distance computations given many database vectors, a distance lookup table ⑤ can be constructed and reused within a query, encompassing all combinations between a sub-query-vector and a cluster centroid within the same sub-space. With this table, the value of $d(x_i, c_i(y_i))$ can be swiftly retrieved by looking up the table with the PQ code as the address ⑥, leading to improved computational efficiency.

2.3 Motivation: Efficient RALM Inference

An efficient RALM inference engine should meet the following **system requirements**:

- Both the LLM inference and the large-scale vector search components should be fast and resource-efficient.
- The system should be flexible enough to accommodate diverse RALM configurations, spanning various combinations model sizes, database sizes, and retrieval frequencies.

However, little effort has been devoted to developing efficient RALM systems that meet the above requirements. This is likely because RALM has been an emerging topic within the machine learning community [7, 31, 32, 42, 49], with their prototype implementations exhibiting the following problems:

(P1) Each research RALM system focuses on *being able to run* one or a small number of RALM models, paying little attention to latency, throughput, resource efficiency, and system flexibility.

(P2) While hardware accelerators for LLMs, such as GPUs, are advancing rapidly, less attention has been paid to the vector search aspect, which, as our evaluations will demonstrate, can become the performance bottleneck in RALM inference.

(P2.1) CPUs are slow in scanning PQ codes during query time ⑥. This is due to the frequent cache accesses (for each byte of PQ code, load the code and use it as an address to load a distance) and the instruction dependencies between operations (distance lookups depend on PQ codes and distance accumulations depend on the lookup values). Even with the state-of-the-art SIMD-optimized CPU implementation [1], the throughput peaks at roughly 1 GB/s per core when scanning PQ codes (1.2 GB/s on Intel Xeon Platinum 8259CL @ 2.50GHz). Within a CPU-memory-balanced server, the PQ code scanning process significantly underutilizes the available memory bandwidth, as about 16 cores are required to saturate the bandwidth of a single memory channel (around 20 GB/s).

(P2.2) GPUs suffer from two major limitations for large-scale vector search. Firstly, the limited memory capacity of each GPU makes large-scale searches on GPU clusters cost-prohibitive. For instance, accommodating only 1 TB of PQ codes necessitates at least 16 NVIDIA A100 GPUs (cost 300K USD as of March 2024), each with 80 GB of memory, given that a portion of memory should be reserved for intermediate search states. Although an alternative solution is to adopt a hybrid CPU-GPU architecture where the GPU fetches vectors from CPU's memory, the inter-processor bandwidth is way lower than the GPU memory bandwidth. Even for NVIDIA Grace Hopper, with the latest high-performance CPU-GPU interconnect, the single-direction bandwidth of 450 GB/s is only 15% of the GPU's bandwidth. Secondly, the throughput for PQ code scanning on GPUs is considerably lower than the GPU's bandwidth, only around 50% of the bandwidth even with large batch sizes (evaluated on NVIDIA A100), due to the multiple passes of memory accesses to write and read intermediate results at each search step [40].

3 CHAMELEON: SYSTEM OVERVIEW

We design and implement Chameleon, an efficient, flexible, and performant RALM inference system:

- Chameleon employs heterogeneous hardware to accelerate both LLM inference and vector search efficiently.
- Chameleon disaggregates the accelerators, enabling independent scaling for each type of hardware, thus supporting various RALM configurations efficiently.
- The modular design of Chameleon allows flexible hardware upgrades, such as integrating more powerful LLM inference accelerators or ASIC-based ChamVS accelerators in the future.

Figure 3 overviews the Chameleon architecture, which primarily consists of the following components.

Firstly, ChamVS is a distributed accelerator engine for low-latency vector search. On the one hand, ChamVS.idx is a GPU-based IVF index scanner colocated with the ChamLM GPUs (right side of Figure 3). While Chameleon also supports index scan on CPUs, GPUs are generally more favorable for handling this embarrassingly parallel workload due to their superior memory bandwidth and computational capability. Given that GPUs are already integrated into Chameleon, no additional devices are required. The only overhead is a slight increase in GPU memory usage, as the index sizes are small relative to the database vectors. For example,

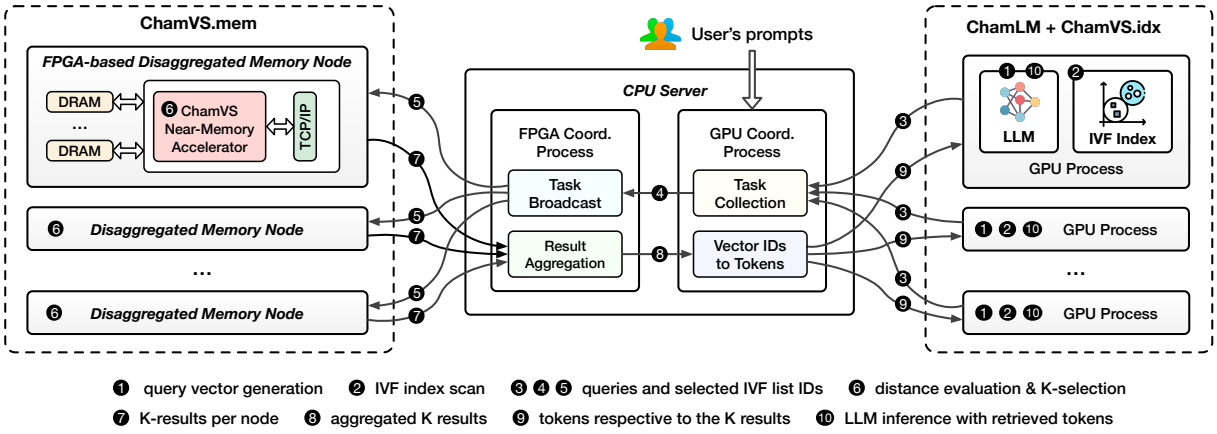

Figure 3: Chameleon is a heterogeneous and disaggregated accelerator system for efficient RALM inference.

assuming 1KB per vector and one thousand vectors per IVF list, a single GB of index can support one million IVF lists, enough for a large database containing one billion vectors. On the other hand, ChamVS.mem is responsible for querying quantized database vectors. ChamVS.mem contains one or multiple disaggregated memory nodes, each with a partition of the database vectors and a near-memory retrieval accelerator prototyped on FPGA for query processing (left side of Figure 3).

Secondly, ChamLM is a multi-GPU LLM inference engine, as shown on the right side of Figure 3. Each GPU, managed by an independent GPU process, can reside on the same or different servers. Currently, ChamLM assigns each GPU a full copy of the LLM, as RALMs can achieve high generation quality even with smaller LLMs [7, 49]. Future larger models could be accommodated by extending ChamLM to support tensor or pipeline parallelism [59, 69, 73] across GPUs. Once a retrieval request is sent, a GPU pauses inference to wait for results. While one potential solution to avoid such GPU idleness is to split the generation into two sub-batches — one executes inference when the other one is waiting for retrieved contents — this approach does not necessarily improve performance. This is because (a) using sub-batches reduces inference throughput, and (b) retrieval latency may not align with inference latency.

Thirdly, the CPU serves as the cluster coordinator, managing the lightweight communication between the GPUs and FPGAs. After receiving search requests from the GPU processes, it dispatches them to the FPGA-based disaggregated memory nodes, aggregates the per-partition results returned by the FPGAs, converts the K nearest neighbor vector IDs into their corresponding texts, and sends the retrieved tokens back to the GPUs. Since each query only requires less than ten KBs of network data transfer, the communication latency is negligible compared to vector search and LLM inference.

Token generation workflow. For each token generation step, the procedure diverges depending on whether the retrieval is invoked. Without retrieval, the GPUs infer the next token as in regular LLMs. With retrieval, the first step is to generate a contextual query vector ❶, either by using the hidden state of the current context [41, 42] or encoding the query tokens through another model [7]. Following this, the IVF index residing on the same GPU is scanned to select the *nprobe* most relevant IVF lists ❷. The query

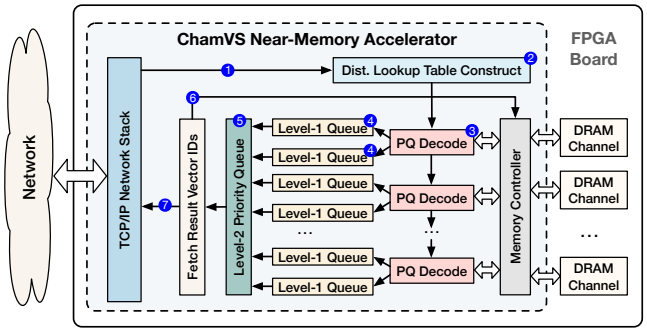

Figure 4: The ChamVS near-memory retrieval accelerator.

vector and the list IDs are then transmitted to the GPU coordinator process running on the CPU node via the network ❸. After recording the association between queries and GPU IDs, the query and list IDs are forwarded to the FPGA coordination process ❹, which broadcasts them to the FPGA-based disaggregated memory nodes ❺. The ChamVS near-memory processor on each node then uses the query vectors to construct distance lookup tables for each IVF list, computes the distances between the query and quantized database vectors, and collects the K nearest neighbors ❻. Subsequently, the result vector IDs and distances from all memory nodes are sent back to the CPU server ❼, which aggregates the results ❽ and returns the tokens of the nearest neighbors to the originating GPU ❾. Finally, the GPU predicts the next token based on both the context and the retrieved tokens ❿.

4 CHAMVS NEAR-MEMORY ACCELERATOR

ChamVS enables high-performance, large-scale vector search by pairing each disaggregated memory node with a near-memory retrieval accelerator. As shown in Figure 4, the accelerator comprises a distance lookup table construction unit, several PQ decoding units for distance evaluations between query vectors and quantized database vectors, a group of systolic priority queues for parallel K-selection, and multiple memory channels.

4.1 PQ Decoding Units

As shown in Figure 4 ❸, each ChamVS accelerator contains multiple PQ decoding units to fully utilize the memory bandwidth. These

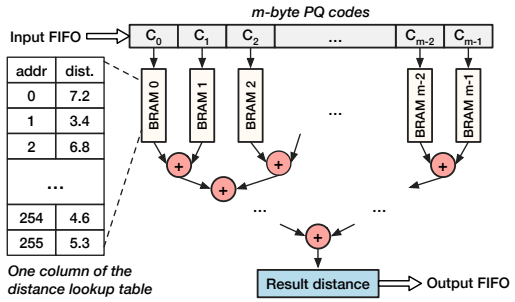

Figure 5: The architecture design of a PQ decoding unit.

units read database vectors (PQ codes) from DRAM and compute their distances to query vectors using a distance lookup table.

The design of a PQ decoding unit involves both operator and pipeline parallelisms, enabling a high throughput of producing one result distance every clock cycle. As shown in Figure 5, the decoding steps — including data ingestion, distance lookups, computation, and output egestion — are fully pipelined, similar to that of [38, 45]. The unit also parallelizes the operators within the distance lookup and computation steps.

Decoding procedure. For each IVF list to scan, the unit first stores the input distance lookup table in BRAM (on-chip SRAM in FPGAs). The shape of the lookup table is $m \times 256$ for the typical 8-bit PQ codes ($2^8 = 256$), where m is the number of bytes per quantized vector. Different table columns are stored in separate BRAM slices, facilitating parallel distance lookups. Subsequently, the PQ codes are loaded from DRAM to the unit via an m-byte-wide FIFO, with each byte serving as an address to retrieve a value from the corresponding column of the table. Finally, an adder tree sums up the retrieved values to produce the approximate distance between the query vector and the quantized database vector.

4.2 Efficient K-Selection Module

The K-Selection module in ChamVS selects the K nearest neighbors from distances computed by the PQ decoding units. Designing an efficient K-selection microarchitecture is challenging, because it has to handle multiple incoming elements per cycle due to the high throughput of PQ decoding units. We propose approximate hierarchical priority queue (AHPQ), a high-throughput, resource-efficient architecture for parallel K-selection in hardware.

4.2.1 Primitive: Systolic Priority Queue. The systolic priority queue facilitates high-throughput input ingestion on hardware accelerators [29, 46], consuming *one input element every two clock cycles.* In short, it is a register array equipped with compare-swap units between the registers, thus *the hardware resource consumption of the queue increases linearly with its length.*

A natural approach to implement K-selection in ChamVS is to instantiate a group of systolic priority queues in a hierarchical structure, as shown in Figure 4 ❹❺. Since a systolic priority queue can only ingest one input every two cycles, two queues, termed as level-one (L1) queues, should be paired with one PQ decoding unit, as it can produce one output per cycle. For each query, each L1 queue collects a subset of the K nearest neighbors, and the level-two (L2) queue subsequently selects the final K results.

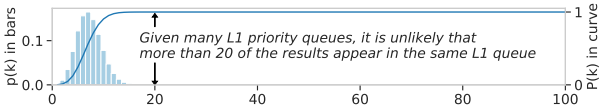

Figure 6: The probability distribution that one out of the 16 L1 priority queues holds k out of the 100 nearest neighbors.

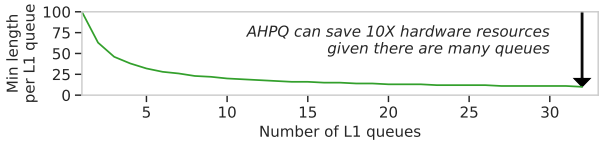

Figure 7: The proposed approximate hierarchical priority queue can save hardware resources by an order of magnitude.

Unfortunately, a straightforward implementation of the hierarchical priority queue can consume excessive hardware resources, making the solution unaffordable even on high-end FPGAs. For example, given 32 instantiated PQ decoding units and $K = 100$, the accelerator would necessitate 64 L1 queues of length 100, an amount that already exceeds the total the total available FPGA resources.

4.2.2 Approximate Hierarchical Priority Queue (AHPQ). **We propose the AHPQ architecture for high-performance and resource-efficient K-selection.** Recognizing that ANN search is inherently approximate, we relax the K-selection objective from selecting the K smallest distances in all queries to collecting precise results in the vast majority of cases, such as in 99% of the queries.

The intuition behind AHPQ is simple: *it is unlikely that all K results are produced by a single PQ decoding unit.* For example, given 16 level-one queues of length K=100, the average number of the results in a queue is only $100/16 = 6.25$. Specifically, the probability that one queue holds k of the K results is $p(k) = C_K^k * (\frac{1}{num_{queue}})^k * (1 - \frac{1}{num_{queue}})^{K-k}$, where C_K^k represents the number of combinations selecting k out of K items. The cumulative probability that a queue contains no more than k of the K results is $P(k) = \sum_{i=0}^{k} p(i)$. Figure 6 shows the probability distribution of p and P given different k in bars and curve: it is almost impossible that a queue holds more than 20 out of the *K=100* results. Thus, the lengths of the L1 queues can be truncated to 20 while producing almost the same results.

Our design aims to reduce the size of the L1 queues while ensuring that the results for 99% of queries remain identical to those obtained with an exact K-selection module. Specifically, for 99% of the queries, none of the L1 queues will omit any result that is supposed to be returned to the user.

Figure 7 shows the resource savings achieved by applying the approximate hierarchical priority queue. As the number of L1 queues increases, the queue sizes can be reduced by an order of magnitude while still retaining 99% of identical results, leading to a corresponding decrease in hardware resource consumption.

4.3 Memory Management and Load Balancing

The memory management mechanism of ChamVS balances workloads across memory nodes and channels. In our current implementation, vectors within each IVF list are evenly partitioned among memory nodes, with these sub-lists further distributed across memory channels to ensure workload balance. For potential scenarios where IVF lists are too small to be partitioned, each list may reside

on different nodes or channels, which could lead to load imbalances, especially with small query batches. Such imbalances can be mitigated by larger batches, as it is less likely that all queries happen to hit the same node or channel. Additionally, for the case of uneven access frequencies across IVF lists, adjusting their placement based on these frequencies can help achieve better load balancing [9].

5 IMPLEMENTATION

Chameleon is implemented in 11K lines of code, including 3K lines of Vitis HLS C/C++ for the ChamVS near-memory accelerator, 1.4K lines of C++ for the CPU coordinator, 3.5K lines of Python for ChamLM, and 3.2K lines of Python for various evaluations. Referring to existing RALM research [41, 42], we build ChamLM on Fairseq [60], a PyTorch-based LLM toolkit. ChamLM extends Fairseq to support multi-GPU inference, initiating retrieval requests, integrating the retrieved tokens into generation processes, and network communication between the retrieval engines and GPU processes. ChamVS.idx uses Faiss [40] for index scanning on GPUs or CPUs. ChamVS.mem integrates an FPGA TCP/IP stack [25]. The CPU coordinator process for query broadcasting and result aggregation is implemented in C++ using the socket library. The simple messages in RALMs allow us to avoid higher-level abstractions like RPCs, minimizing performance overhead.

6 EVALUATION

We evaluate Chameleon to answer the following questions:

- How much performance and energy benefits can ChamVS attain in large-scale vector search? § 6.2
- How does Chameleon perform across different RALMs by introducing heterogeneous accelerators? § 6.3
- Is accelerator disaggregation necessary? § 6.3

6.1 Experimental Setup

LLMs. We evaluate RALM models of similar sizes to those in existing RALM research [7, 31, 51, 72, 86], up to several billions of parameters. We evaluate both smaller (S) and larger (L) decoder-only (Dec) and encoder-decoder (EncDec) models. Table 1 summarizes the four RALMs for evaluation, including input dimensionalities, numbers of layers and attention heads, model sizes, retrieval intervals, and neighbor numbers. For EncDec models, we follow [7] to use a two-layer shallow encoder and a deeper decoder, and set different retrieval intervals. For all the models, we use a vocabulary size of 50K and let them generate 512 tokens per sequence.

Vector datasets. Table 2 summarizes the four evaluated vector datasets. The SIFT and Deep datasets are popular benchmarks for billion-scale ANN. Due to the lack of available datasets for RALM, we create two synthetic datasets by replicating each SIFT vector to the models' dimensionalities (512 and 1024). As a common practice, we set *nlist*, the number of clusters in the IVF index, to approximately the square root of the number of dataset vectors (*nlist=32K*). We set *nprobe* as 32 to scan 0.1% of database vectors per query, for which high recall can be achieved on both real-world datasets (93% on Deep and 94% on SIFT for 100 nearest neighbors). We quantize the SIFT and Deep datasets to 16-byte PQ codes, while the two synthetic datasets adopt 32 and 64-byte PQ codes, respectively.

Table 1: Various RALM configurations in the evaluation.

	Dim.	Layers	Heads	Param.	Interval	K
Dec-S	512	24	8	101M	1	100
Dec-L	1024	96	16	1259M	1	100
EncDec-S	512	2,24	8	158M	8/64/512	10
EncDec-L	1024	2,96	16	1738M	8/64/512	10

Table 2: The vector datasets used in the evaluation.

	Deep	SIFT	SYN-512	SYN-1024
#vec	1E+9	1E+9	1E+9	1E+9
m/D	16/96	16/128	32/512	64/1,024
nprobe/nlist	32/32K	32/32K	32/32K	32/32K
Raw vectors (GB)	384	512	2,048	4,096
PQ and vec IDs (GB)	24	24	40	72

Software. For vector search, we use *Faiss* [1] developed by Meta, known for its optimized PQ implementations for both CPUs and GPUs. Due to its vector-only nature, Faiss's ANN search performance surpasses vector data management systems that support additional relational data functionalities [62]. For LLM inference, we extend Fairseq [60] to support RALMs as introduced in §5.

Hardware. We instantiate the ChamVS near-memory accelerator on AMD Alveo U250 FPGAs (16 nm) equipped with 64 GB of DDR4 memory (4 channels x 16 GB) and set the accelerator frequency to 140 MHz. For a fair comparison, each ChamVS memory node is compared to a CPU-based vector search system with equivalent memory capacity (64 GB) and an 8-core AMD EPYC 7313 processor (7 nm) with a base frequency of 3.0 GHz. We evaluate NVIDIA RTX 3090 GPUs (8nm) with 24 GB GDDR6X memory.

6.2 Large-Scale Vector Search on ChamVS

Search performance. We compare ChamVS with baseline systems using four hardware setups. PQ codes can be processed on CPU/FPGA while the IVF index can be scanned on CPU/GPU, leading to four hardware configurations: CPU, CPU-GPU, FPGA-CPU, and FPGA-GPU. To report the best baseline performance, the CPU and CPU-GPU systems are monolithic, while the FPGA-CPU and FPGA-GPU systems are disaggregated over the network. Figure 8 compares the latency distributions of the four solutions. Each white dot in the violin plots denotes a median latency. The number of CPU cores and the number of accelerators used are listed in the plot legends. We make two primary observations from the experiments:

Firstly, the near-memory accelerator in ChamVS significantly lowers vector search latency. Across different datasets and batch sizes (Figure 8), the FPGA-CPU solution achieves 1.36~6.13× speedup compared to the CPU baseline, and the FPGA-GPU solution shows even higher speedup (2.25~23.72×). This is because the ChamVS near memory accelerator can (a) decode PQ codes in parallel, (b) pipeline the decoding, distance calculation, and K-selection, such that each quantized vector can be processed by the pipeline rapidly.

Secondly, scanning the IVF index on GPU allows further latency improvements compared to the FPGA-CPU solution. As shown in Figure 8, the FPGA-GPU approach achieves 1.04~3.87× speedup compared to the FPGA-CPU solution. This is because the IVF index scan procedure can easily leverage the massively parallelism and the high memory bandwidth of GPUs. In contrast, the hybrid CPU-GPU solution shows little or even negative improvements compared

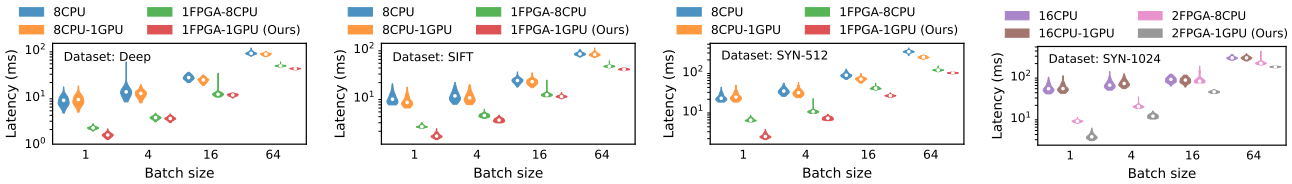

Figure 8: ChamVS achieves significantly lower vector search latency than CPUs and GPUs.

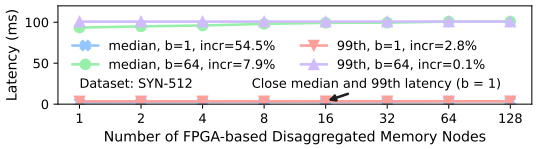

Figure 9: The performance scalability of ChamVS.

Table 3: Average energy consumption per query (in mJ) on ChamVS and CPUs using various batch sizes (1~16).

	CPU			ChamVS (FPGA + GPU)		
	b=1	b=4	b=16	b=1	b=4	b=16
SIFT	950.3	434.0	143.3	53.6	28.2	21.5
Deep	929.5	412.9	141.9	52.3	26.9	20.5
SYN-512	1734.9	957.8	372.5	95.6	55.0	41.1
SYN-1024	4459.9	2315.0	918.5	170.1	107.8	85.2

to the CPU-only solution (0.91~1.42×), because the search performance is limited by the slow PQ code scan process on CPU.

Scalability. We extrapolate query latency beyond the limited number of accelerators available in our evaluation. Considering the one-GPU and N-FPGA setup, we estimate the latency distribution by summing up accelerator and network latencies. Each query latency number is the maximum of N randomly sampled latency numbers from the 1-FPGA setup. For network latency, we assume a 100 Gbps bandwidth for the CPU server and apply the LogGP model [5, 13], which assumes a tree topology for broadcast and reduce communications, setting the latency between two endpoints as 10.0 μs (a conservative number compared to 6.0 μs reported in [26, 27]). Figure 9 presents the median and the 99th percentile latencies for different batch sizes on the SYN-512 dataset. The tail latencies remain almost identical to those in the one-node setup due to the negligible network latency compared to the query. As for the median latencies, there is only a 7.9% increase for a batch size of 64, while for the case without batching, the latency increases by 54.5% as the accelerator latency is determined by the slowest one.

Energy consumption. *ChamVS achieves 5.8~26.2× energy efficiency compared to the CPU.* Table 3 summarizes the average energy consumption to serve a single query across different systems. We measure CPU, GPU, and FPGA energy consumption using Running Average Power Limit (RAPL) and NVIDIA System Management Interface, and Vivado, respectively. For ChamVS, we report the energy per query by measuring the power consumption times latency for scanning index on GPU and scanning PQ codes on FPGAs, respectively, and summing the two parts up.

Recall. *ChamVS, with approximate hierarchical priority queues (AHPQ), delivers results nearly identical to those of the software.* Table 4 shows the recall given various AHPQ lengths (8~32) when searching for the $K = 100$ nearest neighbors. Here, R1@100 indicates the percentage of queries where the top nearest neighbor is within the results, while R@100 represents the percentage of

Table 4: Recall of ChamVS using approximate queues.

	CPU (len=100)	AHPQ (len=8)	AHPQ (len=16)	AHPQ (len=32)
R1@100 (Deep)	92.88%	92.85%	92.84%	92.84%
R@100 (Deep)	45.54%	45.49%	45.49%	45.48%
R1@100 (SIFT)	94.21%	94.20%	94.21%	94.21%
R@100 (SIFT)	48.68%	48.66%	48.67%	48.67%

overlap between the true 100 nearest neighbors and the 100 results returned. Compared to software, AHPQ only decreases recall by up to 0.06%. Interestingly, on the Deep dataset, reducing the queue lengths to eight does not necessarily result in lower recall than using a length of 32. This is likely due to the nature of PQ approximation — a higher distance indicated by PQ does not always mean that the original vector is actually farther from the query.

6.3 End-to-end RALM Inference on Chameleon

We evaluate RALM inference performance on Chameleon with different models and retrieval intervals, using the SYN-512 and SYN-1024 datasets for the smaller and larger models, respectively.

RALM performance. We evaluate system performance when generating a 512-token sequence using a single GPU for LLM inference .For the latency evaluation, we disable batching, while the throughput evaluation uses the maximum allowed batch sizes given the GPU's memory capacity (64 for Dec-S and EncDec-S; 8 for Dec-L and EncDec-L). For vector search in RALM, we use the FPGA-GPU solution for ChamVS and the CPU-only solution as the baseline, as CPU-GPU vector search can be even slower using small batches.

Chameleon significantly outperforms the CPU-GPU baseline system in latency for inference steps involving vector search. Figure 10 visualizes the RALM inference latency of Chameleon and the baseline system (CPU-GPU) for the first 128 generated tokens. Inference latency is represented by the grey dots, while retrieval latency accounts for the remaining portion of the end-to-end latency. The time spent on coordinator and index scanning is not marked in the figure, as their latencies of hundreds of microseconds are negligible compared to up to tens of milliseconds for inference and retrieval. Figure 10 shows that ChamVS significantly reduces the latency at the token generation steps requiring retrieval, as the retrieval latency of Chameleon is almost negligible compared to the inference latency executed on GPUs. Specifically, the speedup provided by Chameleon at retrieval-based inference steps (retrieval + inference) ranges from 1.94~4.11×, 1.71~3.02×, 1.76~3.41×, and 1.29~2.13× for Dec-S, EncDec-S, Dec-L, and EncDec-L, respectively.

Chameleon achieves up to 3.18× throughput compared to the CPU-GPU baseline. Figure 11 shows that the lower the retrieval interval, the more throughput advantage Chameleon offers, with the speedup being 3.18× and 2.34× for Dec-S and Dec-L that require retrieval per token generation (interval=1). Chameleon attains greater speedup in batched inference than single-sequence inference (as in latency

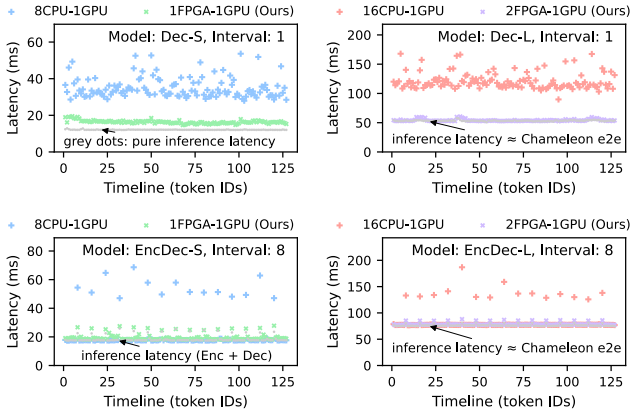

Figure 10: Latency of RALM inference given different LLM configurations and retrieval intervals.

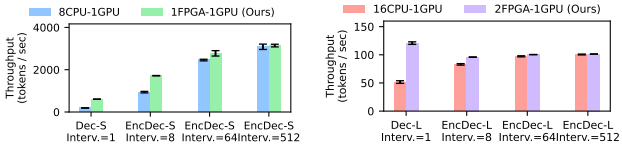

Figure 11: Throughput of RALM inference given different LLM configurations and retrieval intervals.

experiments), because, as the batch size grows, the latency increase for LLM inference is not as significant as that of vector search, due to the many-core parallelism that GPUs offer.

The need for resource disaggregation. Accelerator disaggregation allows Chameleon to adjust the ratio between the two types of accelerators across RALM configurations. We model the overall system throughput, measured by generated tokens per second, across various accelerator ratios using a total of 1,000 accelerators, assuming the cost for an inference accelerator and a retrieval accelerator is equivalent. Given retrieval interval i, batch size b, number of inference and retrieval accelerators N_I and N_R, latency per batch for inference and retrieval $L_I(b)$ and $L_R(b)$, the system throughput is determined by the minimum of the inference and retrieval throughput: $Th_{system} = \min(Th_I, Th_R)$, where $Th_I = \frac{i \cdot b \cdot N_I}{i \cdot L_I(b) + L_R(b)}$ and $Th_R = \frac{i \cdot b \cdot N_R}{L_R(b)}$. Figure 12 shows that the optimal ratio of accelerators to achieve the highest throughput varies significantly, ranging from 53.7%~99.0% across RALMs.

The disaggregated design, using the optimal accelerator ratio, consistently outperforms the monolithic ones with fixed ratios, as shown in Figure 13. Given the impracticality of adjusting the ratio for each RALM in a monolithic design, the performance of a monolithic design can only match that of Chameleon on a limited set of RALMs.

7 RELATED WORK

To our knowledge, Chameleon represents the first endeavor to improve RALM inference performance by using heterogeneous accelerator systems. We now introduce related research topics below.

ANN search. Researchers have developed various ANN search algorithms [14, 19, 30, 36, 54, 61, 75, 78, 84, 92, 93] and data management systems [22, 58, 62, 77, 79, 85]. Apart from PQ-based vector search, graph-based searching algorithms [17, 18, 53, 55, 56, 65, 81, 91, 94] are popular as they can achieve high recall and low latency. Locality-sensitive hashing (LSH) [15, 21] offers theoretical guarantees in ANN search but empirically does not perform as well as PQ and graph-based algorithms.

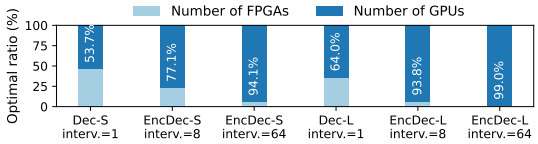

Figure 12: Disaggregation is essential as the optimal accelerator ratio varies significantly across RALM configurations.

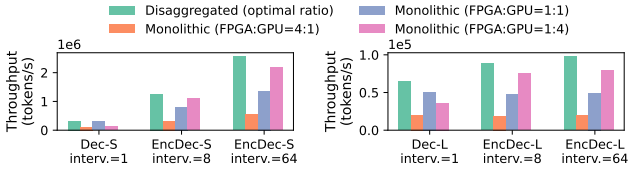

Figure 13: The disaggregated design consistently outperforms the monolithic ones using fixed accelerator ratios.

Vector search on modern hardware. Faiss is the most popular GPU-accelerated ANN search library so far [40], and there are several academic GPU implementations [10, 11, 52, 80]. Lee et al. [45] study ASIC designs for IVF-PQ, and a couple of works [38, 89] implement IVF-PQ on an FPGA, but their designs are constrained by either the limited HBM capacity or the slow CPU-FPGA interconnect. In contrast, Chameleon disaggregates IVF-PQ, with the index on GPUs and PQ codes on FPGA-based memory nodes, and employs the innovative hardware priority queue design to achieve high performance with little hardware resources. While graph-based vector search accelerators can achieve low latency [37, 88], the memory consumption is high, requiring up to one TB of memory for only one billion SIFT vectors, in contrast to 24 GB in our case. Apart from accelerators, researchers also study memory and storage for vector search. One can leverage non-volatile memory [71] and CXL [33] to scale up graph-based ANN, while on-disk ANN has to be more careful with I/O cost [9, 34, 47]. Hu et al. [28] further push down distance evaluation into NAND flash to reduce data movement.

8 CONCLUSION AND OUTLOOK

We present Chameleon, a heterogeneous and disaggregated accelerator system for efficient RALM inference. Given the rapidly evolving algorithms, software, and hardware related to RALMs, Chameleon can be potentially upgraded in the following ways. For LLM inference, ChamLM could be enhanced by supporting low precision [2], continuous batching [87], paged-attention [44], and disaggregated prompt computation and token generation [63]. While currently supporting PQ, ChamVS could potentially be replaced by graph-based ANN accelerators [64, 88]. ChamVS could also be extended to support index updates [83] and relational features [90].

ACKNOWLEDGMENTS

We thank AMD for their generous donation of the Heterogeneous Accelerated Compute Clusters (HACC) at ETH Zurich (https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html), on which the experiments were conducted.

REFERENCES

[1] [n.d.]. Faiss. https://github.com/facebookresearch/faiss/.

[2] [n.d.]. FasterTransformer. https://github.com/NVIDIA/FasterTransformer.

[3] [n.d.]. The Implications of OpenAI's Latest Update on RAG and Vector-Only Databases. https://medium.com/@vishalkalia.er/the-implications-of-openais-latest-update-on-rag-and-vector-only-databases-c3f326cce0a1.

[4] [n.d.]. What does OpenAI's announcement mean for Retrieval Augmented Generation (RAG) and Vector-only Databases? https://medium.com/madhukarkumar/what-does-openais-announcement-mean-for-retrieval-augmented-generation-rag-and-vector-only-54bfc34cba2c.

[5] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. 1995. LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*. 95–105.

[6] Uri Alon, Frank Xu, Junxian He, Sudipta Sengupta, Dan Roth, and Graham Neubig. 2022. Neuro-symbolic language modeling with automaton-augmented retrieval. In *International Conference on Machine Learning*. PMLR, 468–485.

[7] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*. PMLR, 2206–2240.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2111.08566* (2021).

[10] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, Qiang Wang, and Wei Zhao. 2019. Vector and line quantization for billion-scale similarity search on GPUs. *Future Generation Computer Systems* 99 (2019), 295–307.

[11] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, and Wei Zhao. 2019. Robustiq: A robust ann search method for billion-scale similarity search on gpus. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*. 132–140.

[12] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[13] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1–12.

[14] Michele Dallachiesa, Themis Palpanas, and Ihab F Ilyas. 2014. Top-k nearest neighbor search in uncertain data series. *Proceedings of the VLDB Endowment* 8, 1 (2014), 13–24.

[15] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).

[18] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[19] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[20] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 744–755.

[21] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[22] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. 2022. Manu: A Cloud Native Vector Database Management System. *arXiv preprint arXiv:2206.13843* (2022).

[23] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*. PMLR, 3929–3938.

[24] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Realm: Retrieval-augmented language model pre-training. *arXiv preprint arXiv:2002.08909* (2020).

[25] Zhenhao He, Dario Korolija, and Gustavo Alonso. 2021. EasyNet: 100 Gbps Network for HLS. In *2021 31th International Conference on Field Programmable Logic and Applications (FPL)*.

[26] Torsten Hoefler, Andre Lichei, and Wolfgang Rehm. 2007. Low-overhead LogGP parameter assessment for modern interconnection networks. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[27] Torsten Hoefler and Dmitry Moor. 2014. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing frontiers and innovations* 1, 2 (2014), 58–75.

[28] Han-Wen Hu, Wei-Chen Wang, Yuan-Hao Chang, Yung-Chun Lee, Bo-Rong Lin, Huai-Mu Wang, Yen-Po Lin, Yu-Ming Huang, Chong-Ying Lee, Tzu-Hsiang Su, et al. 2022. ICE: An Intelligent Cognition Engine with 3D NAND-based In-Memory Computing for Vector Similarity Search Acceleration. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 763–783.

[29] Muhuan Huang, Kevin Lim, and Jason Cong. 2014. A scalable, high-performance customized priority queue. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–4.

[30] Qiang Huang, Yifan Lei, and Anthony KH Tung. 2021. Point-to-Hyperplane Nearest Neighbor Search Beyond the Unit Hypersphere. In *Proceedings of the 2021 International Conference on Management of Data*. 777–789.

[31] Gautier Izacard and Edouard Grave. 2020. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282* (2020).

[32] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299* (2022).

[33] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. {CXL-ANNS}:{Software-Hardware} Collaborative Memory Disaggregation and Computation for {Billion-Scale} Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 585–600.

[34] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[35] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[36] Minhao Jiang, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. 2015. Exact top-k nearest keyword search in large networks. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 393–404.

[37] Wenqi Jiang, Hang Hu, Torsten Hoefler, and Gustavo Alonso. 2024. Accelerating Graph-based Vector Search via Delayed-Synchronization Traversal. *arXiv preprint arXiv:2406.12385* (2024).

[38] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, et al. 2023. Co-design Hardware and Algorithm for Vector Search. *arXiv preprint arXiv:2306.11182* (2023).

[39] Wenqi Jiang, Shuai Zhang, Boran Han, Jie Wang, Bernie Wang, and Tim Kraska. 2024. Piperag: Fast retrieval-augmented generation via algorithm-system co-design. *arXiv preprint arXiv:2403.05676* (2024).

[40] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* (2019).

[41] Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Nearest neighbor machine translation. *arXiv preprint arXiv:2010.00710* (2020).

[42] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2019. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172* (2019).

[43] Mojtaba Komeili, Kurt Shuster, and Jason Weston. 2021. Internet-augmented dialogue generation. *arXiv preprint arXiv:2107.07566* (2021).

[44] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[45] Yejin Lee, Hyunji Choi, Sunhong Min, Hyunseung Lee, Sangwon Beak, Dawoon Jeong, Jae W Lee, and Tae Jun Ham. 2022. ANNA: Specialized Architecture for Approximate Nearest Neighbor Search. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 169–183.

[46] Charles E Leiserson. 1979. *Systolic Priority Queues*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[47] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. 2008. NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis*

and Machine Intelligence 31, 5 (2008), 869–883.

[48] Mike Lewis, Marjan Ghazvininejad, Gargi Ghosh, Armen Aghajanyan, Sida Wang, and Luke Zettlemoyer. 2020. Pre-training via paraphrasing. *Advances in Neural Information Processing Systems* 33 (2020), 18470–18481.

[49] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[50] Zihao Li. 2023. The dark side of chatgpt: Legal and ethical challenges from stochastic parrots and hallucination. *arXiv preprint arXiv:2304.14347* (2023).

[51] Zonglin Li, Ruiqi Guo, and Sanjiv Kumar. 2022. Decoupled context processing for context augmented language modeling. *Advances in Neural Information Processing Systems* 35 (2022), 21698–21710.

[52] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2023. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. *arXiv preprint arXiv:2312.01712* (2023).

[53] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 15, 2 (2021), 246–258.

[54] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. 2012. Efficient processing of k nearest neighbor joins using mapreduce. *arXiv preprint arXiv:1207.0141* (2012).

[55] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.

[56] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[57] Yuxian Meng, Xiaoya Li, Xiayu Zheng, Fei Wu, Xiaofei Sun, Tianwei Zhang, and Jiwei Li. 2021. Fast nearest neighbor machine translation. *arXiv preprint arXiv:2105.14528* (2021).

[58] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-Throughput Vector Similarity Search in Knowledge Graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[59] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.

[60] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038* (2019).

[61] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Progressive top-k nearest neighbors search in large road networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1781–1795.

[62] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021* (2023).

[63] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677* (2023).

[64] Hongwu Peng, Shiyang Chen, Zhepeng Wang, Junhuan Yang, Scott A Weitze, Tong Geng, Ang Li, Jinbo Bi, Minghu Song, Weiwen Jiang, et al. 2021. Optimizing fpga-based accelerator design for large-scale molecular similarity search (special session paper). In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–7.

[65] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[66] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[67] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. 2021. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446* (2021).

[68] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[69] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[70] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-context retrieval-augmented language models. *arXiv preprint arXiv:2302.00083* (2023).

[71] Jie Ren, Minjia Zhang, and Dong Li. 2020. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. *Advances in Neural Information Processing Systems* 33 (2020), 10672–10684.

[72] Devendra Singh Sachan, Mostofa Patwary, Mohammad Shoeybi, Neel Kant, Wei Ping, William L Hamilton, and Bryan Catanzaro. 2021. End-to-end training of neural retrievers for open-domain question answering. *arXiv preprint arXiv:2101.00408* (2021).

[73] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[74] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. 2022. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990* (2022).

[75] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment* (2014).

[76] Boxin Wang, Wei Ping, Lawrence McAfee, Peng Xu, Bo Li, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Instructretro: Instruction tuning post retrieval-augmented pretraining. *arXiv preprint arXiv:2310.07713* (2023).

[77] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[78] Xiaoyang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Muhammad Aamir Cheema. 2015. Optimal spatial dominance: an effective search of nearest neighbor candidates. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 923–938.

[79] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[80] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik Lensch. 2016. Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2027–2035.

[81] Yubao Wu, Ruoming Jin, and Xiang Zhang. 2014. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*. 1139–1150.

[82] Frank F Xu, Uri Alon, and Graham Neubig. 2023. Why do Nearest Neighbor Language Models Work? *arXiv preprint arXiv:2301.02828* (2023).

[83] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561.

[84] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Wei Wang. 2015. Reverse k nearest neighbors query processing: experiments and analysis. *Proceedings of the VLDB Endowment* 8, 5 (2015), 605–616.

[85] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2241–2253.

[86] Dani Yogatama, Cyprien de Masson d'Autume, and Lingpeng Kong. 2021. Adaptive semiparametric language models. *Transactions of the Association for Computational Linguistics* 9 (2021), 362–373.

[87] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[88] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, et al. 2023. DF-GAS: a Distributed FPGA-as-a-Service Architecture towards Billion-Scale Graph-based Approximate Nearest Neighbor Search. (2023).

[89] Jialiang Zhang, Soroosh Khoram, and Jing Li. 2018. Efficient large-scale approximate nearest neighbor search on OpenCL FPGA. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4924–4932.

[90] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. 2023. {VBASE}: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 377–395.

[91] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1979–1991.

[92] Yuxin Zheng, Qi Guo, Anthony KH Tung, and Sai Wu. 2016. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single

index. In *Proceedings of the 2016 International Conference on Management of Data*. 2023–2037.

[93] Huaijie Zhu, Xiaochun Yang, Bin Wang, and Wang-Chien Lee. 2016. Range-based obstructed nearest neighbor queries. In *Proceedings of the 2016 International Conference on Management of Data*. 2053–2068.

[94] Chaoji Zuo and Dong Deng. 2023. ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2645–2658.