

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

Wenqi Jiang¹, Marco Zeller¹, Roger Waleffe², Torsten Hoefler¹, and Gustavo Alonso¹

¹Department of Computer Science, ETH Zurich

²Department of Computer Science, University of Wisconsin-Madison

Abstract

A Retrieval-Augmented Language Model (RALM) augments a generative language model by retrieving context-specific knowledge from an external database. This strategy facilitates impressive text generation quality even with smaller models, thus reducing orders of magnitude of computational demands. However, RALMs introduce unique system design challenges due to (a) the diverse workload characteristics between LM inference and retrieval and (b) the various system requirements and bottlenecks for different RALM configurations such as model sizes, database sizes, and retrieval frequencies. We propose *Chameleon*, a heterogeneous accelerator system that integrates both LM and retrieval accelerators in a disaggregated architecture. The heterogeneity ensures efficient acceleration of both LM inference and retrieval, while the accelerator disaggregation enables the system to independently scale both types of accelerators to fulfill diverse RALM requirements. Our Chameleon prototype implements retrieval accelerators on FPGAs and assigns LM inference to GPUs, with a CPU server orchestrating these accelerators over the network. Compared to CPU-based and CPU-GPU vector search systems, Chameleon achieves up to $23.72\times$ speedup and $26.2\times$ energy efficiency. Evaluated on various RALMs, Chameleon exhibits up to $2.16\times$ reduction in latency and $3.18\times$ speedup in throughput compared to the hybrid CPU-GPU architecture. These promising results pave the way for bringing accelerator heterogeneity and disaggregation into future RALM systems.

1 Introduction

The recent advances in language model (LM) quality have been largely attributable to the surging number of parameters within the transformer neural networks, often on the order of the hundreds of billions of parameters [9, 18, 81, 92]. The intuition behind the scaling-up approach is to leverage

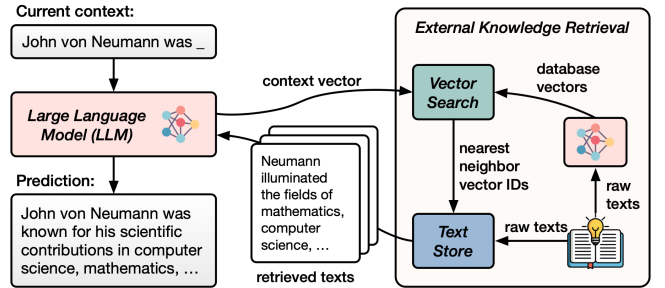


Figure 1: Retrieval-augmented language model (RALM) architecture.

more model parameters to learn and encapsulate textual human knowledge, thus offering more precise and informed responses.

However, improving LM quality by expanding model sizes leads to three major problems. Firstly, the increased computational demands result in higher inference costs [6, 9]. Secondly, updating knowledge in the LM is inflexible: the latest information, like recent news, is not incorporated without additional training, and removing harmful or sensitive data, such as personal information mistakenly included in the training set [10, 97], can be difficult, often requiring the model to be retrained from scratch [8, 11]. Finally, the trustworthiness of such models is still a matter of concern. A large LM can still produce non-factual content [65, 68] because it is challenging to ascertain if the model has acquired the specific knowledge pertinent to the context, and even if it has, there is no guarantee it will employ that knowledge during text generation.

One effective approach to address the aforementioned problems is known as *Retrieval-Augmented Language Models (RALMs)*: the LM focuses on learning linguistic structures, while context-specific knowledge is incorporated during inference. Figure 1 overviews the RALM architecture. The

external knowledge database encodes textual knowledge into vectors using LMs and stores them in a vector database. During inference, 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. Even with LMs of one to two orders of magnitude fewer parameters than conventional LMs, RALMs can achieve superior or comparable performance on various natural language processing (NLP) tasks, including question answering [7, 38, 39, 65, 86], dialogue systems [55, 90], language modeling [31, 54, 101], and machine translation [53, 64], thus significantly lowering the compute costs during inference. Furthermore, the knowledge accessible to the LM is easy to manage because knowledge can be inserted into and deleted from the database, without the need for retraining the LM at all.

Despite its advantages, the efficient serving of RALMs presents two challenges. *First*, the workload characteristics of the LM and the retriever are distinct. While the LM inference primarily relies on rapid matrix multiplication, the retrieval system — often utilizing the Product Quantization (PQ) algorithm [42] — demands both substantial memory capacity for the vector database and fast processing of quantized database vectors during query time. Unfortunately, neither GPUs nor CPUs excel at meeting the retrieval demands. The memory capacity requirement hampers the deployment of retrievers on GPU clusters, as it can be cost-prohibitive given the limited device memory capacity per GPU. On the other hand, transferring the database vectors from the host server to the GPU at query time severely compromises performance, given the low data movement bandwidth relative to the GPU’s device memory. Although CPUs can be coupled with abundant DRAM, they are too slow in evaluating distances between query vectors and quantized database vectors, given the many per-PQ-code cache accesses and instruction dependencies that occur during a search. *Second*, the diverse range of RALM configurations leads to shifting system requirements and bottlenecks. Various model sizes, database sizes, and retrieval frequencies can be used in a RALM, each introducing unique requirements for LM inference computations, memory capacity for the vector database, and retrieval performance.

To address these challenges, we envision that a performant and efficient RALM system should adhere to two fundamental design principles. *First*, given the divergent workload characteristics between the LM and the retriever, the system should employ heterogeneous hardware tailored for each. Specifically, transformer inference should be performed by LM accelerators, while large-scale vector search should be handled by one or multiple retrieval accelerators, each combining abundant device memory for the database and efficient near-memory processing for querying the quantized database vectors. *Second*, to support various RALM configurations efficiently, the accelerators should be disaggregated, in contrast to a monolithic approach where a fixed number of LM

and retrieval accelerators reside on the same server. This is due to the inflexibility of the monolithic architecture: a huge vector database may require more retrieval accelerators than a single server can accommodate, and the varying LM inference and retrieval performance requirements across different RALMs can lead to significant under-utilization of either type of accelerator.

To materialize this vision, we propose *Chameleon*, a heterogeneous and disaggregated accelerator system for efficient, flexible, and high-performance RALM serving. Chameleon consists of three primary components. Firstly, *ChamVS* is a distributed vector search engine consisting of several smart disaggregated memory nodes. Each node, prototyped on an FPGA, contains a shard of quantized database vectors in DRAM, a near-memory retrieval accelerator, and a hardware TCP/IP stack. Secondly, *ChamLM* is a multi-accelerator LM inference engine implemented on GPUs. It produces query vectors and generates texts using the retrieved information. Lastly, a CPU coordinator server orchestrates the network communication between the retrieval and LM accelerators and converts the retrieved vector IDs into their respective textual representations.

We evaluate Chameleon with different LM architectures, LM sizes, database sizes, and retrieval frequencies. For large-scale vector search, ChamVS achieves up to $23.72\times$ latency reduction compared to the state-of-the-art CPU-based vector search system while consuming $5.8\sim 26.2\times$ less energy per query. For end-to-end RALM inference, Chameleon achieves up to $2.16\times$ speedup in latency and $3.18\times$ increase in 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 impressive performance and flexibility of Chameleon open new avenues for future RALM system designs.

Contributions:

- We propose Chameleon, a disaggregated system consisting of heterogeneous hardware for efficient RALM serving.
- We design and implement ChamVS, a distributed system for large-scale vector search, which includes:
 - Smart disaggregated memory nodes, each equipped with a specialized near-memory vector search accelerator.
 - A novel design of fast and resource-efficient approximate top-K selection units integrated into the accelerator.
 - An index scanner for orchestrating the memory nodes.
- We build ChamLM, a multi-GPU engine for producing queries and generating texts using the retrieved contents.

- We evaluate Chameleon using various RALM configurations and showcase its remarkable performance and efficiency.

2 Background and Motivation

2.1 Retrieval-Augmented Language Models

A RALM combines a conventional transformer-based LM [20, 80, 82] with an external knowledge database. During inference, information relevant to the current context is retrieved from the database and utilized by the LM 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. Popular examples of this category employ the encoder-decoder transformer architecture [7, 38, 65], as the decoder-only models are less flexible in integrating text chunks [83]. The encoder-decoder model comprises two primary components: an encoder, responsible for processing input texts, and a decoder, which produces output tokens. During inference, initial states, such as a user’s query, are vectorized to retrieve context-related knowledge, i.e., text chunks in the database with similar vector representations [7, 39, 65]. The retrieved text chunks are then concatenated and processed by the encoder, and their latent knowledge representations are conveyed to the decoder via the cross-attention mechanism [94], leading to the generation of output tokens. When generating long sequences, however, the generated content might gradually diverge from the initial context. Thus, instead of initiating retrieval only once at the genesis of text generation [38, 65, 86], an effective strategy is to perform multiple retrievals during text generation to improve token generation quality [83], for instance, at a regular interval of every 64 generated tokens [7].

The second category retrieves only the *next token* of each similar context in the database. These RALMs employ a decoder-only model [3, 54, 73]. At each step of token generation (retrieval interval equals one), the last layer’s hidden state serves as the query to retrieve some similar contexts from the database, along with the next token of each context [54, 73, 100]. 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 [53, 54].

From a system performance standpoint, the two categories differ in two aspects. Firstly, regarding the retrieval interval, the second category retrieves every step during token generation (higher retrieval cost), while the first category may either employ multiple-token intervals or retrieve just once at the beginning per sequence. Secondly, in terms of computation cost, the first category, as they often adopt the encoder-decoder architecture, introduces the cost of encoder inference per retrieval step and cross-attentions between decoder and encoder

Table 1: Definitions of important symbols in IVF-PQ.

Symbol	Definition
x	A query vector.
y	A database vector.
m	The sub-space number of product quantization.
$nlist$	The number of clusters in the IVF index.
$nprobe$	The number of IVF lists to scan per query.
K	The number of nearest neighbors to return.

every token generation step (higher computational cost), while the second category, with a decoder model, only requires an extra interpolation of the next token’s probability distribution, introducing minimal computational overhead.

2.2 Large-Scale Vector Search with IVF-PQ

A vector search takes a D -dimensional query vector x as input and retrieves K relevant vector(s) from a database Y , populated with many D -dimensional vectors, based on similarity metrics like L2 distances. While the nearest neighbor search retrieves the exact K closest vectors, linearly scanning through a large vector set can be prohibitively expensive. Thus, real-world vector search systems adopt approximate nearest neighbor (ANN) search that trades accuracy for 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, and we will use the terms *vector search* and *ANN search* interchangeably.

In this paper, we focus on the *IVF-PQ* algorithm: thanks to its great performance in large-scale ANN search [29, 42, 47], it has been extensively adopted in various RALMs [7, 38, 54, 65]. In essence, IVF-PQ combines (a) an inverted-file (IVF) index to prune the search space and (b) product quantization (PQ) to compress database vectors and reduce the computational demands during the search process. The most important symbols in IVF-PQ are summarized in Table 1

Inverted-File (IVF) Index. An IVF index divides a vector dataset Y into many ($nlist$) disjoint subsets, typically using clustering methods like K-means. Each of these subsets is termed an IVF list. At query time, only a select few ($nprobe$) IVF lists whose 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 computational requirements of vector search by compressing each database vector into m -byte PQ codes. The training and searching workflow of PQ is shown in Figure 2.

To quantize database vectors, all database vectors are partitioned evenly into m sub-vectors ①. Each sub-vector possesses a dimensionality of $D^* = \frac{D}{m}$, typically ranging from 4 to 16 in practice. A clustering algorithm is performed in each sub-space ②, allowing each database sub-vector to be ap-

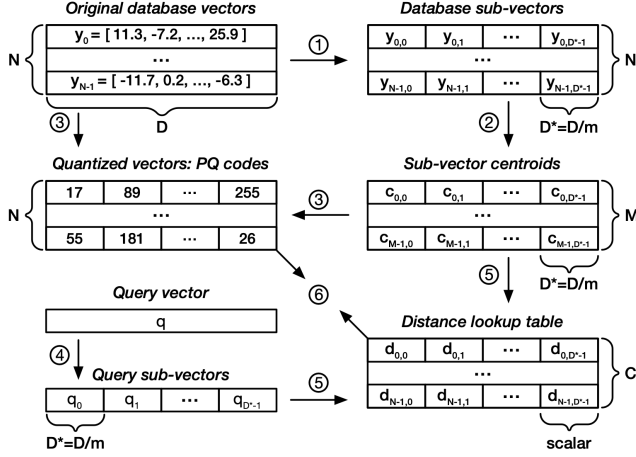


Figure 2: Product quantization (PQ) in training and searching.

proximated by its nearest cluster centroid. Conventionally, the number of clusters per sub-space is set as $M = 256$, enabling the representation of a cluster ID using a single byte. Consequently, once the cluster centroids are stored, each database vector can be represented by m -byte PQ codes.

During the search process, a query vector will be compared against the quantized database vectors. The distance computation can be formulated as $\hat{d}(x, y) = d(x, q(y)) = \sum_{i=1}^m d(x_i, u_i(y_i))$, where $\hat{d}(x, y)$ represents the approximate distance between a query vector x and a quantized database vector y . During the computation, the quantized database vector is approximated by its reconstructed vector $q(y)$ using the PQ codes and the cluster centroid vectors per sub-space. The query vector is also divided into m sub-query-vectors (x_i) ④ and compared against the reconstructed quantized sub-database-vectors ($u_i(y_i)$). To speed up distance computations with many database vectors, it would be beneficial to construct a distance lookup table ⑤, encompassing all combinations between a sub-query-vector and a cluster centroid within the same sub-space. With this table, the value of $d(x_i, u_i(y_i))$ can be swiftly retrieved by a table lookup operation, utilizing the PQ code as the address ⑥. Given many database vectors, the table entries are typically reused multiple times, leading to improved computational efficiency.

2.3 Motivation: the Need for an Efficient RALM System

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

- Both the LM 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 combi-

nations of (a) transformer architectures, (b) model sizes, (c) database sizes, and (d) 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. Specifically, the current RALM systems employed in machine learning research [7, 38, 39, 54, 65] exhibit the following shortcomings:

First, each RALM research focuses on *being able to run* one or a small number of RALM models, paying little attention to performance, resource efficiency, and system adaptability for diverse RALM configurations. In fact, certain system limitations, like the high knowledge retrieval costs, may have already hindered the exploration of alternative model configurations — such as experimenting with shorter retrieval intervals — which could potentially further improve the text generation quality.

Second, while existing research employs commercial processors for vector search, neither CPU nor GPU is the ideal platform for large-scale IVF-PQ.

On the one hand, CPUs are slow in scanning PQ codes during query time ⑥. This inefficiency arises 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 utilizing 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(R) Xeon(R) Platinum 8259CL @ 2.50GHz). Within a core-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).

On the other hand, GPUs present several limitations for large-scale vector search. One significant constraint is the limited memory capacity of individual GPUs, making large-scale searches on GPU clusters cost-prohibitive. For instance, accommodating 1 TB of PQ codes would necessitate at least 16 NVIDIA A100 GPUs, each with 80 GB of memory, given that some memory should be reserved for caching intermediate states during the search. These GPUs cost 300K USD in total ¹ without considering the host servers. 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, the latest high-performance interconnects, the single-direction NVLink bandwidth of 450 GB/s is only 15% of the GPU’s bandwidth (3,000 GB/s). Secondly, the high bandwidth of GPUs is not fully leveraged because (a) the significant amount of shared memory allocated per kernel for the lookup table

¹Price on Amazon as of October 2023.

restricts the number of resident thread blocks per streaming multi-processor, thus limiting the GPU’s capability to hide memory latency by switching to alternate thread blocks [61], and (b) the K-selection process after distance calculations necessitates multiple passes of memory accesses, consuming further bandwidth [47]. Finally, the many-core architecture of GPUs would benefit the most for large batches, while RALM would benefit more from latency on small batches because the batch size of language model inference is often limited due to the key-value cache [59, 102]. Lastly, while the many-core GPU architecture is known for high-throughput large-batch processing, RALMs are more likely to benefit from the reduced vector search latency of smaller query batches, given that transformer inference often operates with limited batch sizes [59, 102].

3 Chameleon: System Overview

We present Chameleon, a heterogeneous and disaggregated accelerator system for efficient, flexible, and high-performance RALM serving. We prototype Chameleon by incorporating the FPGA-based retrieval accelerators, GPU-based LM inference accelerators, and a CPU server for system coordination.

Chameleon addresses the system challenges in RALMs in the following ways:

- Chameleon employs heterogeneous hardware accelerators to handle the distinct workload characteristics of RALM’s LM inference and retrieval subsystems.
- Chameleon disaggregates the accelerators, in contrast to a monolithic design, to support a wide range of RALMs by allowing the independent scaling of each type of hardware accelerator. For example, some RALMs combine frequent retrievals on large databases (high retrieval demands) with small LMs (low computational demands), while others can couple a large model with a low retrieval frequency.
- Chameleon’s modular design enables flexible hardware upgrades or replacements. Future iterations of ChamLM could replace GPUs with TPUs [48] or LM inference ASICs, while the ChamVS near-memory accelerator could be instantiated on more powerful FPGAs or taped out as an ASIC.

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

Firstly, ChamLM is a multi-GPU LM inference engine, as shown on the right side of Figure 3. Each GPU is managed by an independent GPU processes, which can reside on either the same or different servers. Due to the significant reduction in LM sizes by introducing a retriever [7, 65], a single GPU is sufficient to host an entire copy of an LM.

Secondly, ChamVS is a disaggregated vector search engine. On one hand, ChamVS.idx is a GPU-based IVF index scanner

colocated with the ChamLM GPUs (right side of Figure 3). While the index scan can run on CPUs, the parallel nature of the workload makes GPU a more suitable choice. Given that GPUs are already integrated into Chameleon, the only overhead is a minor increment in GPU memory usage, typically under one GB. 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 an FPGA-based near-memory retrieval accelerator (left side of Figure 3).

Thirdly, a CPU server manages communication between the GPUs and FPGAs. It handles incoming search requests from the GPU processes, 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.

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 LMs. With retrieval, the first step is to generate a contextual query vector ①, either by using the hidden state of the current context [53, 54] or encoding the query tokens through another model [7]. Following this, the IVF index residing on the same GPU is probed to select the *nprobe* most relevant IVF lists ②. The query vector and the list IDs are then transmitted to the GPU coordinator process running on the CPU node via the network ③. After recording the association between queries and GPU IDs, the query and list IDs are forwarded to the FPGA coordination process ④, which broadcasts them to the FPGA-based disaggregated memory nodes ⑤. 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 ⑥. Subsequently, the result vector IDs and distances from all memory nodes are sent back to the CPU server ⑦, which aggregates the results ⑧ and returns the tokens of the nearest neighbors to the originating GPU ⑨. Finally, the GPU predicts the next token based on both the context and the retrieved tokens ⑩.

4 Near-Memory Accelerator for Vector Search

Chameleon enables high-performance, large-scale vector search by pairing each disaggregated memory node with a ChamVS near-memory accelerator implemented on an FPGA. The architecture of a ChamVS disaggregated memory node is shown in Figure 4, encompassing an on-chip hardware TCP/IP network stack, a distance lookup table construction unit, several PQ decoding units for evaluating distances between query vectors and quantized database vectors, a group of systolic priority queues for parallel K-selection, and multiple channels of DDR memory. In this section, we mainly

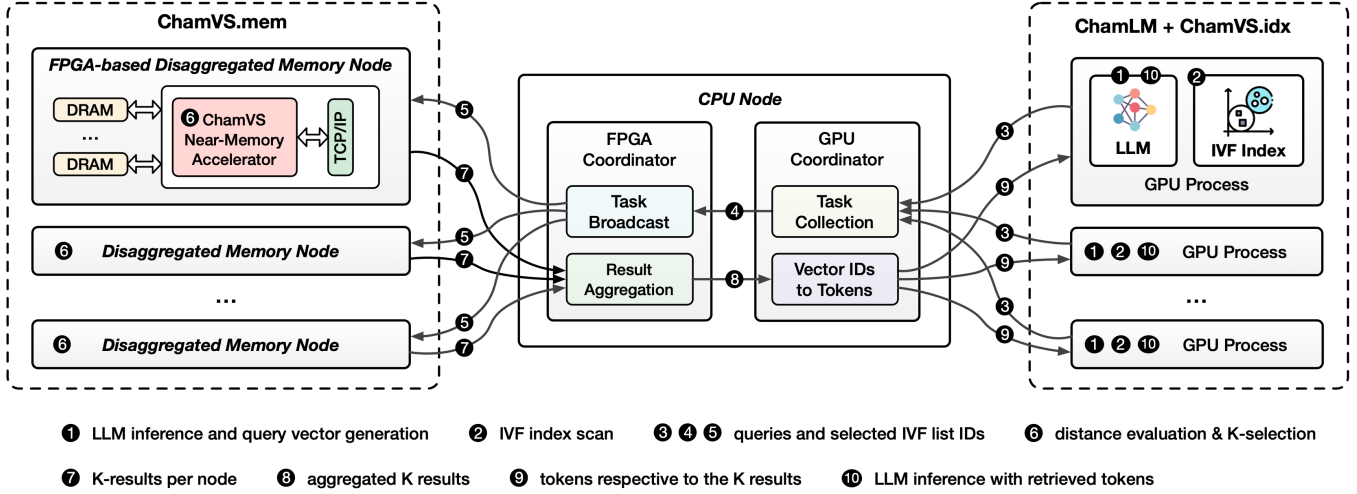


Figure 3: Chameleon is a heterogeneous and disaggregated CPU, GPU, and FPGA system for high-performance and efficient RALM inference.

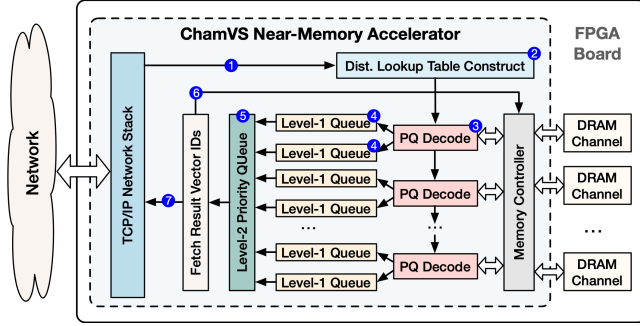


Figure 4: ChamVS couples a near-memory accelerator with each disaggregated memory node to enable high-performance vector search.

elaborate on the PQ decoding units and the K -selection circuit, omitting the distance lookup table construction unit since it simply calculates L2 distances as introduced in § 2.

4.1 PQ Decoding Units

A PQ decoding unit reads quantized database vectors (PQ codes) from DRAM and computes the corresponding L2 distances to query vectors using a distance lookup table.

Figure 5 presents the design of a single PQ decoding unit, which can produce a result distance every clock cycle. The unit operates iteratively in two major phases. The first phase is table initialization, in which the unit takes as input a distance lookup table and stores it into the 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 stands for the number of bytes per quantized vector. Different table columns are stored in separate BRAM slices, facilitating

parallel distance lookups. The second phase is PQ code scanning. The codes are streamed 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. For example, the 5-th byte of the codes is used to fetch a distance from the 5-th BRAM slice in the PQ decoding unit. Upon completion of the m parallel table lookup operations (within a single clock cycle), an adder tree sums up the values to produce the approximate distance between the query vector and the quantized database vector. The lookup and addition processes are pipelined such that the unit can consistently process m bytes of inputs and yield a result distance each clock cycle.

Multiple PQ decoding units operate in parallel to fully utilize the memory bandwidth, as shown in Figure 4 (3). The total number of units is determined by the quantization level m , the number of memory channels, and the width of the memory controller interface. For instance, given $m = 32$ and an FPGA comprising four memory channels, each accessible by a 64-byte-wide AXI interface to the memory controller, there would be $64 * 4 / 32 = 8$ PQ decoding units on the accelerator. When scanning a cluster of vectors, all units share the same distance lookup table, as each cluster is equally distributed across all memory channels. The units are arranged in a one-dimensional array, enabling each unit to forward the table to the subsequent one. This arrangement avoids a broadcasting topology, thus mitigating potential wire routing issues.

4.2 Efficient K -Selection Module

Designing an efficient K -selection microarchitecture within ChamVS.mem presents significant challenges, as each PQ decoding unit generates one distance every clock cycle, requiring the K -selection module to manage multiple incoming elements per cycle. In this section, we begin by examining the

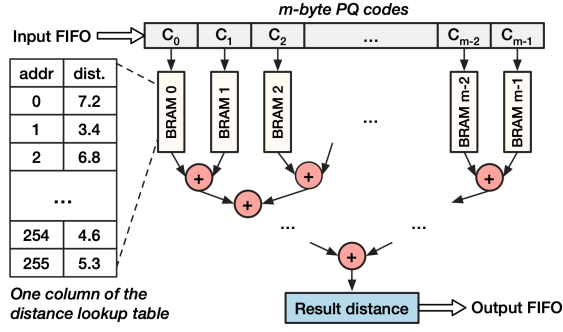


Figure 5: The design of a PQ decoding unit that evaluates distances between queries and quantized database vectors.

register-array-based systolic priority queue, a building block for the K -selection module. However, instantiating many systolic priority queues of length K to satisfy throughput requirements proves too costly due to prohibitively high hardware resource consumption. Consequently, we propose the approximate hierarchical priority queue, a high-throughput, resource-efficient design for parallel K -selection on hardware.

4.2.1 Primitive: Systolic Priority Queue

While software-based priority queues support enqueue, dequeue, and replace operations, the vector search necessitates only the replace operation. Specifically, if the incoming distance is smaller than the largest distance in the queue, the algorithm will dequeue the largest distance and enqueue the incoming distance.

Figure 6 shows the systolic priority queue [36,62], designed to consume one input element every two cycles. The systolic priority queue consists of a register array interconnected by compare-swap units, operating on a repeating two-cycle procedure. During an odd cycle, the leftmost node is replaced with the minimum value between the existing leftmost value and an incoming element, followed by the swapping of all even entries in the array with their corresponding odd neighbors. In the subsequent even cycle, the swapping is reversed, with all the odd entries interchanging with the even ones. Throughout this process, the smallest element is gradually swapped to the rightmost position in the queue. The hardware resource consumption of such a priority queue scales linearly with its length, as both the number of registers and compare-swap units are proportional to the queue size.

A natural approach to implement K -selection in ChamVS is to instantiate a group of systolic priority queues in a hierarchical structure, as illustrated in Figure 4 4 5. Since a systolic priority queue can only ingest one input element every two cycles, yet the module should be capable of consuming all the distances produced by the PQ decode units per cycle, two queues, termed as level-one (L1) queues, must be paired with one PQ decoding unit. Once all the database vectors have

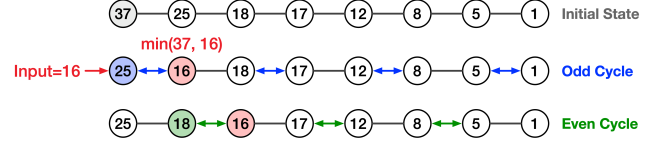


Figure 6: A systolic priority queue takes one input every two cycles.

been scanned within a query, the first level of queues collects $2K$ elements. The level-two (L2) queue subsequently selects the final K from the elements within the L1 queues.

Unfortunately, a straightforward implementation of the hierarchical priority queue can consume an excessive amount of hardware resources, making the solution unaffordable even on high-end FPGAs. For example, a 100-element priority queue would utilize around 2.5% of lookup tables (LUTs) on the Alveo U250, one of AMD’s largest FPGA models. Given 32 instantiated PQ decoding units, the accelerator would necessitate 64 L1 queues to match the throughput of the decoding units, an amount that already exceeds the total LUT resources available on the FPGA. Consequently, it becomes imperative to devise a more resource-efficient approach for K -selection.

4.2.2 Approximate Hierarchical Priority Queue

We propose the approximate hierarchical priority queue architecture for high-performance and resources-efficient K -selection. Recognizing that the approximate nearest neighbor search inherently does not yield exact results, 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 the approximate K -selection design is simple: it is unlikely that all the K results are produced by a single PQ decoding unit. For example, given 16 level-one queues with $K = 100$, the average number of the top 100 results in a queue is $100/16 = 6.25$. More specifically, the probability that one queue holds k of the K nearest neighbors can be formulated as $p(k) = C_K^k * (\frac{1}{\text{num_queue}})^k * (1 - \frac{1}{\text{num_queue}})^{K-k}$, where C_K^k represents the number of combinations selecting of k out of K items. The cumulative probability that a queue contains no more than k of the K results can be calculated by $P(k) = \sum_{i=0}^k p(i)$. The probability distribution of p and P are visualized by the red bars and the blue curve in Figure 7, respectively. The figure demonstrates that it is highly unlikely a queue holds more than 20 out of the $K=100$ results; thus, the length of the L1 priority queue can be truncated to 20 while producing almost the same results.

In our design, we aim 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.

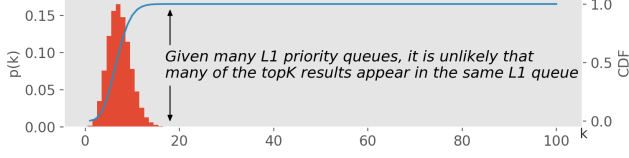


Figure 7: The probability distribution that one out of the 16 level-one priority queues holds k of the top 100 results.

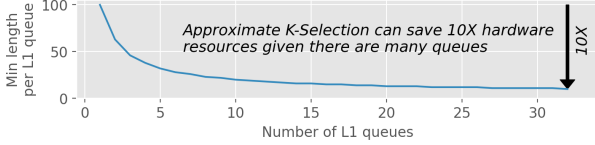


Figure 8: The proposed approximate hierarchical priority queue can save hardware resources by an order of magnitude.

Figure 8 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 the identical results. As the resource consumption of a queue is almost proportional to its length, such a reduction in size leads to a corresponding decrease in hardware resource consumption.

4.3 Memory Management

ChamVS.mem manages several data components, including PQ codes, corresponding vector IDs, and metadata. Specifically, the PQ codes and vector IDs are stored in DRAM, while the metadata is loaded into BRAM during the system initialization phase. Since the entire memory space of a memory node is dedicated to vector search, ChamVS.mem operates directly in the physical address space, avoiding memory virtualization overheads. Because each vector cluster in the IVF index is typically big enough on large datasets, ChamVS.mem evenly distributes the quantized vectors and vector IDs within each cluster among all memory channels, such that the workloads per memory channel are well-balanced. Each memory channel has an AXI interface that bridges the memory controller and the compute logic, facilitating 64-byte data transmission per clock cycle. An address lookup table, as part of the metadata, records the starting physical address of PQ codes and vector IDs of each cluster. Additionally, the centroid vectors of the product quantizer constitute the other part of the metadata.

For distributed vector search with multiple disaggregated memory nodes, there are two approaches for data partitioning. The first way, which we apply in this paper, is to assign a portion of every single Voronoi cell to each memory node. For example, if there are four memory nodes and a thousand Voronoi cells, each memory node will hold all the one thou-

sand cells but only one-fourth of the vectors per cell. In this case, the workload between memory nodes is always balanced: the coordinator sends the same scan request (cell IDs) to all the memory nodes, and the memory nodes scan the same amount of database vectors. This is practical in large-scale vector search, because each Voronoi cell for a large dataset contains many vectors, and dividing a cell into multiple shards will not turn sequential scans into random memory accesses. The second way is to assign a subset of Voronoi cells to a memory node such that different nodes hold distinct cells. This is suitable when each Voronoi cell contains few vectors while there are many memory nodes. The scan workloads, in this case, are asymmetric between memory nodes, as it is possible that all the cells to scan happen to be on the same memory node.

5 Implementation

Chameleon is implemented in 11K lines of code in total, including 3K lines of Vitis HLS C/C++ for the near-memory accelerator, 1.4K lines of C++ for the network coordination program for multi-FPGA vector search, 3.5K lines of Python for the language model and the retrieval interfaces, and 3.2K lines of Python for various experiments and tests.

ChamLM. Referring to existing RALM research project [53, 54], we build ChamLM on top of Fairseq [76], a language model toolkit based on PyTorch [77] that allows flexible instantiation of different LM architectures and parameters. ChamLM extends Fairseq to support functionalities such as multi-GPU inference, initiating retrieval request using the model’s hidden states, integrating the retrieved tokens or documents to the models, with TCP/IP network communication between the vector search engines and different GPU processes.

ChamVS. For ChamVS.idx, we use the Faiss [47] to support such efficient index centroid vector scan on both CPUs and GPUs. For ChamVS.mem, we develop the entire near-memory accelerator using Vitis HLS 2021.2 in C/C++ and use an open-source FPGA TCP/IP network stack [34] which connects to the accelerator kernel. The coordinator process between ChamVS.idx and ChamVS.mem for query broadcasting and result aggregation is implemented using C/C++ and the socket library. For ChamTS, we implement it as a dummy content generator given input vector IDs.

6 Evaluation

We evaluate Chameleon to answer the following questions:

1. How much performance benefit can ChamVS attain in large-scale vector search? (§ 6.2)
2. What are the energy and hardware resource consumptions of the ChamVS near-memory accelerator? (§ 6.2)

Table 2: Decoder-only and encoder-decoder RALMs for 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

3. How does Chameleon perform across different RALM configurations? (§ 6.3)
4. Is accelerator disaggregation necessary? (§ 6.3)

6.1 Experimental Setup

LM models. We evaluate RALM models of similar sizes (from 100 million to more than 1 billion parameters), aligning with existing RALM research [7, 38, 69, 86, 101]. For each decoder-only (Dec) and encoder-decoder (EncDec) RALM, we experiment with a smaller model (S) and a larger model (L). Table 2 summarizes the four RALMs for evaluation, including the hidden state dimensionality, the number of layers, attention heads, the model parameters, the retrieval interval, and the number of results to return. For encoder-decoder 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. For the large-scale vector search microbenchmark, we use two real-world datasets and two synthetic datasets, all containing at least one billion vectors. Specifically, the SIFT and Deep datasets are the most popular benchmarks for billion-scale ANN. Due to the lack of available trained RALM vector datasets online, we create two synthetic datasets by replicating each SIFT vector to the transformer’s dimensionalities (512 and 1024). As in common practice, we set $nlist$, the number of clusters in the vector index, to approximately the square root of the number of vectors in the dataset ($nlist=32K$). We set $nprobe$, the percentage of lists to scan per query, as 32 (scan 0.1% of database vectors), for which high recall can be achieved on both real-world datasets ($R@100=93\%$ on Deep and $R@100=94\%$ on SIFT). We quantize the SIFT and Deep datasets to 16-byte PQ codes and the two synthetic datasets of higher dimensionalities using 32-byte and 64-byte PQ codes, respectively.

Software. For vector search, we use *Faiss* [1] (v1.7.2) developed by Meta as the baseline software. *Faiss* is currently the most popular product-quantization-based ANN library, and it is known for its highly optimized implementations for both CPUs and GPUs. The GPU version of *Faiss* supports multi-GPU search within a server, which suffices our

Table 3: The vector datasets in the evaluation.

	Deep	SIFT	SYN-512	SYN-1024
#vec	1E+9	1E+9	1E+9	1E+9
D	96	128	512	1,024
m	16	16	32	64
$nlist$	32,768	32,768	32,768	32,768
Raw vectors (GB)	384	512	4,096	8,192
PQ and vec ID (GB)	24	24	40	72

evaluation. For transformer inference, we extend *fairseq* [], a multi-GPU LM inference and training framework, to support the integration of retrieved content during text generation.

Hardware. We instantiate the ChamVS near-memory accelerator on an AMD Alveo U250 FPGA (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 and a max turbo frequency of 3.7 GHz. We evaluate NVIDIA RTX 3090 GPUs (8nm) with 24 GB GDDR6X memory. As we will show later, the ChamVS can achieve better performance and energy efficiency even if instantiated on FPGAs manufactured in older technology.

6.2 Vector Search on ChamVS

Performance. We compare ChamVS with baseline systems on four large-scale vector datasets, each using four different configurations: searching solely on CPU (CPU), scanning the IVF index on GPU and the PQ codes on CPU (CPU-GPU), scanning the index on CPU and the PQ codes on FPGA (FPGA-CPU), and scanning the index on GPU and the PQ codes on FPGA (FPGA-GPU). To report the best baseline performance, the CPU and CPU-GPU systems are run on the same server, while the CPU-FPGA and GPU-FPGA system is disaggregated over the network. Figure 9 shows the vector search latency distributions of different batch sizes using 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. There are two primary observations from the experiments:

Firstly, the near-memory accelerator in ChamVS results in significant lower vector search latency compared to the CPU baseline. Across different datasets and batch sizes (Figure 9), the FPGA-CPU solution achieves $1.36\sim 6.13\times$ speedup compared to the CPU baseline, and the GPU-FPGA solution shows even higher speedup ($2.25\sim 23.72\times$). 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 with an initiation interval of a

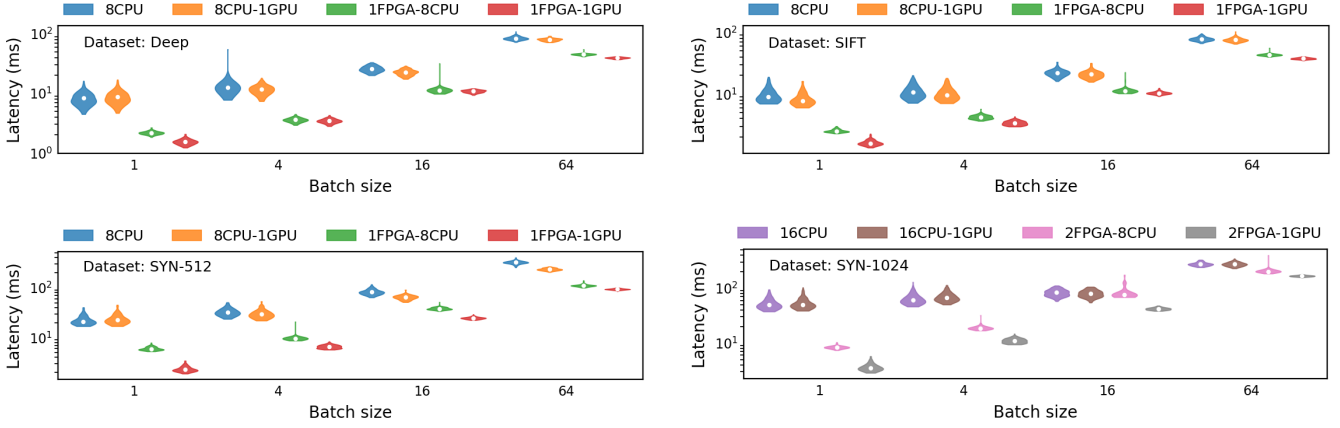


Figure 9: ChamVS achieves significantly lower vector search latency than CPUs and GPUs.

Table 4: The retrieval accelerator consumes little FPGA resources.

Dataset	LUT	FF	BRAM	URAM	DSP
SIFT	25.3%	16.2%	13.7%	4.4%	12.2%
Deep	23.7%	15.4%	13.0%	4.4%	10.4%
SYN-512	23.2%	15.5%	23.2%	4.4%	8.4%
SYN-1024	28.0%	19.0%	35.7%	4.4%	11.9%

Table 5: Average energy consumption per query (in mJ) on CPU and ChamVS using various batch sizes (1~16).

	CPU			ChamVS (FPGA + GPU)		
	1	4	16	1	4	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

single clock cycle. Such specialization leads to significantly better performance compared to CPU in scanning PQ code and selecting the K nearest neighbors.

Secondly, scanning the IVF index on GPU allows further latency improvements compared to the FPGA-CPU solution, given that GPUs are inherently available in RALM systems for LM inference. As shown in Figure 9, the FPGA-GPU approach achieves $1.04\sim 3.87\times$ speedup compared to the FPGA-CPU solution. This is because the IVF index scan procedure can easily harvesting the massively parallelism and the high memory bandwidth of GPUs: the query vector are compared against all centroid vectors in the index, and the $nprobe$ closest centroids are selected. In contrast, the hybrid CPU-GPU solution show little or even negative improvements compared to the CPU-only solution ($0.91\sim 1.42\times$), because the search performance is limited by the slow PQ code scan process on CPU.

Resource and energy consumption. For CPUs and GPUs, we measure their energy consumption using *Intel RAPL* and *NVIDIA System Management Interface*, respectively. For the FPGA-based near-memory accelerator, we report the resource and energy consumption using Vivado.

The ChamVS near-memory accelerator consumes few FPGA resources. The AMD Alveo U250 FPGA contains 1.4M LUTs, 2.9M FFs, 2.1K BRAM, 1.3K URAM, and 12K

DSPs, and four memory channels. The accelerator only consumes around 20% of the hardware resources. To further improve the cost-efficiency of ChamVS, one can choose an FPGA device with a higher ratio of memory channels to FPGA resources. For example, given the same FPGA chip as the U250 device, increasing the number of memory channels to, e.g., 12, would lead to around $3\times$ PQ-code scan performance. This is feasible as high-end FPGAs have enough I/O pins even for HBM that has much higher bandwidth [96]. Another improvement would be increasing the memory capacity per FPGA. Instead of using the U250 FPGA with only 64 GB DRAM, one can increase the memory capacity per FPGA board such that fewer FPGAs are needed in Chameleon to serve a large dataset. For example, Enzian [19] is an academic FPGA platform equipped with up to 1 TB of DRAM per device.

The ChamVS achieves $5.8\sim 26.2\times$ energy efficiency compared to the CPU. Table 5 summarizes the average energy consumption (in mJ) of different systems to serve a single query, given different batch sizes. For ChamVS, we report the energy per query by measuring the power consumption and latency for index scan on GPU and PQ-code scan on FPGAs, and summing the two parts up.

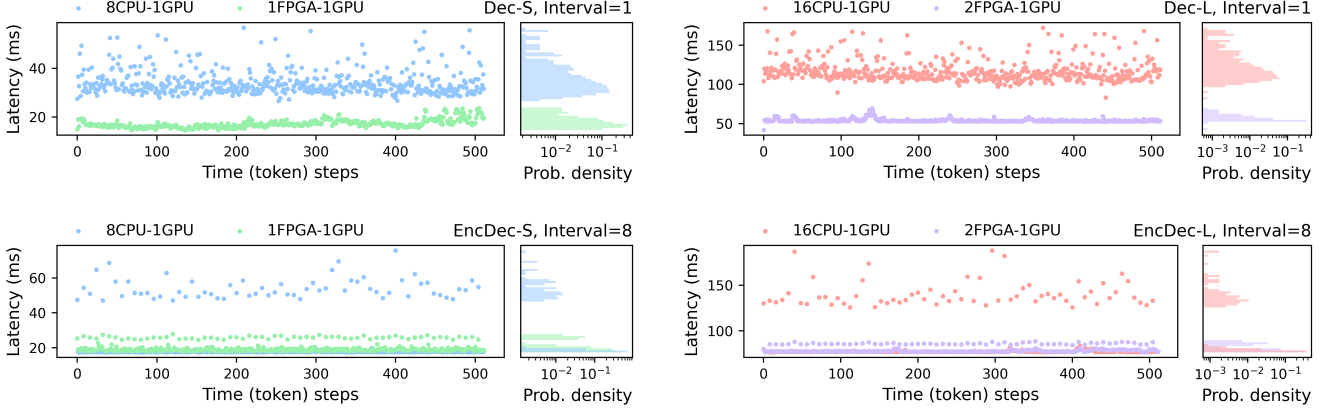


Figure 10: RALM serving latency given different LM configurations and retrieval intervals.

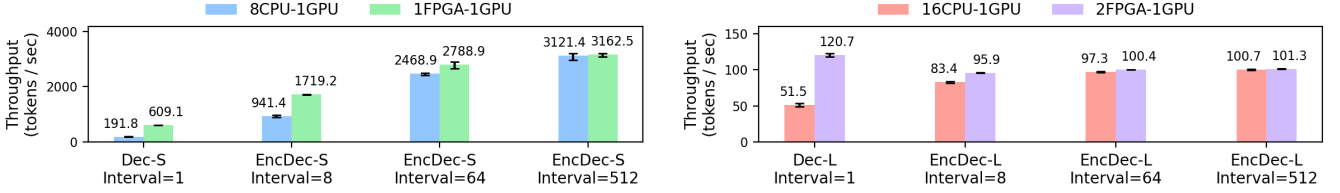


Figure 11: RALM serving throughput given different LM configurations and retrieval intervals.

6.3 End-to-end RALM on Chameleon

In this section, we evaluate the end-to-end RALM performance on Chameleon given different model configurations and retrieval intervals.

Performance. For both latency and throughput evaluation, we evaluate system performance when generating a 512-token sequence using a single GPU for LM inference, and each experiment is executed three times. For the latency evaluation, we disable batching, while, for the throughput evaluation, we set the batch size as the maximum allowed given the RTX 3090 GPU’s memory capacity (batch size = 64 for Dec-S and EncDec-S; 8 for Dec-L and EncDec-L) and assume that all sequences in the batch will generate 512 consecutive tokens, given that early termination for a subset of sequences can be easily addressed via preemptive scheduling [59]. For vector search in RALM, we use the FPGA-GPU solution for ChamVS and the CPU-only solution as the baseline.

Figure 10 compares the RALM serving latency between Chameleon (FPGA-GPU) and the baseline system (CPU-GPU). The left column shows the small models (Dec-S and EncDec-S), while the right column shows the large models. Each row uses the same retrieval interval (1, 8, 64, and 512, respectively). For each plot, the left subplot shows the latency over token generation steps, while the right one depicts the latency distribution.

Chameleon significantly outperforms the CPU-GPU base-

line system in latency for inference steps involving vector search. While the LM inference is still executed on the GPU, the FPGA-GPU retrieval engine significantly reduces the latency at the token generation steps requiring retrieval. Specifically, the speedup provided by Chameleon at retrieval-based inference steps (retrieval + inference) ranges from $1.94 \sim 4.11\times$, $1.71 \sim 3.02\times$, $1.76 \sim 3.41\times$, and $1.29 \sim 2.13\times$ for Dec-S, EncDec-S, Dec-L, and EncDec-L, respectively.

Chameleon achieves up to $3.18\times$ throughput compared to the CPU-GPU baseline system. As shown in Figure 11, the lower the retrieval interval, the higher the throughput advantage Chameleon can provide, with the speedup being $3.18\times$ and $2.34\times$ for Dec-S and Dec-L that requires retrieval per token generation (interval = 1). Chameleon attains greater speed-up in batched serving compared to single-sequence serving. This is because the latency increase for LM inference is less significant than that of vector search as the batch size grows due to the many-core parallelism offered by the GPU during LM inference. Thus, the speed-up observed in batched serving is closer to that of vector search than in unbatched serving.

The need for resource disaggregation. Given the broad range of configurations in RALMs — such as different model sizes, retrieval intervals, and database sizes — it is likely that either the LM inference or vector search engine would be underutilized if the hardware resource ratio is not carefully configured. Based on the evaluated batched inference through-

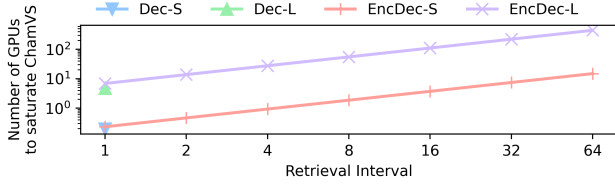


Figure 12: The accelerator ratio between accelerators in different RALM configurations.

put, Figure 12 illustrates that the number of GPUs required to saturate the ChamVS vector search engine in a batched inference setting can vary dramatically, ranging from 0.2 to 442. This makes a monolithic design approach, which entails installing a fixed number of accelerators on a single server, both inflexible and sometimes impractical. Chameleon’s disaggregated architecture addresses this issue by allowing for flexible combinations of hardware resources over the network, thereby enhancing overall accelerator utilization.

In future cloud deployments of such a disaggregated accelerator system, resource sharing across users is also an option to enhance resource utilization. For example, ChamVS can be wrapped as a service accessible by multiple users, each operating an RALM on one or more GPUs. This approach allows the vector search accelerators to be well utilized even for models of low retrieval frequencies.

7 Related Work

To the best of our knowledge, Chameleon represents the first endeavor to analyze retrieval-augmented language models from a systems perspective, addressing associated challenges via a heterogeneous and disaggregated accelerator architecture. We proceed to introduce research on related topics below.

Resource disaggregation. Resource disaggregation has become increasingly popular in data centers. CPUs, memory, storage, and accelerators are connected through a high-speed network [22], such that these resources can be combined according to system needs without overprovisioning any type of the resources [24, 52, 70, 71]. The flexible introduction of extra resources, such as remote memory, helps in system performance compared to servers with monolithic designs [4, 105].

Near data processing. Data movements from memory or storage devices to processors are expensive, thus various literature has proposed to offload some computation workloads to a processor near or within memory/storage [21, 56, 75, 85, 87, 91, 93]. This approach is especially effective for data-intensive applications because the offloaded compute logic is simple, and a large portion of data is filtered out near memory/storage before being sent to CPUs. Typical use

cases include database systems [21, 46], big data processing [30, 49, 50], recommender systems [43, 44, 99], time series processing [23], and genome sequence analysis [72]. The processors near the data can be regular CPUs [2], vector processors [78], reconfigurable hardware [27, 57], or ASICs [72].

AI accelerators. GPUs are commonly used for deep learning acceleration nowadays [17, 37, 95]. Alternative specialized architectures have also been implemented on FPGAs [5, 25, 74, 79, 89, 107] and ASICs [28, 32, 51, 58, 88, 106]. These accelerators are often co-designed with algorithm relaxations such as neural network pruning and quantization [26, 33, 66, 67]. Compiler-based solutions are also necessary to reduce the programming effort to map tensor programs to the various hardware backends [13, 60].

Vector search on modern hardware. Google proposes to accelerate exact nearest neighbor search on TPUs and show great performance on small datasets [16]. Similarly, the exact search can be implemented on FPGAs [103]. For ANN search, the most popular GPU-accelerated library so far is Faiss developed by Meta [47]. Researchers have also built several GPU-based ANN systems [14, 15, 98]. Lee et al. [61] study ASIC designs for IVF-PQ, and the simulation-based evaluation shows significant speedup over GPUs. A couple of works [45, 104] 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, even for large neighbor numbers. Apart from accelerator-based solutions, researchers also study modern storage for vector search. Hu et al. [35] propose to push down vector distance evaluation to NAND flash to reduce data movement. Ren et al. [84] suggest storing vectors in non-volatile memory to scale up graph-based ANN, while on-disk ANN has to be careful with I/O cost [12, 41, 63]. The emerging CXL technology has introduced another level of memory hierarchy as an option for ANN search [40].

8 Conclusion

Retrieval-augmented language models, although advantageous from a machine learning standpoint, introduce unique challenges for system designs. Leveraging accelerator heterogeneity and disaggregation, Chameleon enhances RALM inference efficiency for two primary reasons: firstly, it enables efficient execution of both LM inference and retrieval, and secondly, it allows each system component to be scaled independently, satisfying various system requirements of different RALM configurations. Our Chameleon prototype on a heterogeneous CPU, GPU, and FPGA cluster achieves up to $2.16\times$ speedup over the state-of-the-art systems. The notable outcomes underscore the potential of leveraging heterogeneous and disaggregated accelerator systems in future RALM

system designs.

References

- [1] Faiss. <https://github.com/facebookresearch/faiss/>.
- [2] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Kai Wang, Thomas Roewer, Adam McPadden, Oliver O'Halloran, Deming Chen, Jinjun Xiong, et al. Application-transparent near-memory processing architecture with memory channel network. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 802–814. IEEE, 2018.
- [3] Uri Alon, Frank Xu, Junxian He, Sudipta Sengupta, Dan Roth, and Graham Neubig. Neuro-symbolic language modeling with automaton-augmented retrieval. In *International Conference on Machine Learning*, pages 468–485. PMLR, 2022.
- [4] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [5] Aman Arora, Samidh Mehta, Vaughn Betz, and Lizy K John. Tensor slices to the rescue: Supercharging ml acceleration on fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 23–33, 2021.
- [6] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 610–623, 2021.
- [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. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR, 2022.
- [8] Lucas Bourtole, Varun Chandrasekaran, Christopher A Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. Machine unlearning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 141–159. IEEE, 2021.
- [9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [10] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650, 2021.
- [11] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. When machine unlearning jeopardizes privacy. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 896–911, 2021.
- [12] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighbor search. *arXiv preprint arXiv:2111.08566*, 2021.
- [13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.
- [14] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, Qiang Wang, and Wei Zhao. Vector and line quantization for billion-scale similarity search on gpus. *Future Generation Computer Systems*, 99:295–307, 2019.
- [15] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, and Wei Zhao. Robustiq: A robust ann search method for billion-scale similarity search on gpus. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 132–140, 2019.
- [16] Felix Chern, Blake Hechtman, Andy Davis, Ruiqi Guo, David Majnemer, and Sanjiv Kumar. Tpu-knn: K nearest neighbor search at peak flop/s. *arXiv preprint arXiv:2206.14286*, 2022.
- [17] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022.

- [18] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [19] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, et al. Enzian: an open, general, cpu/fpga platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 434–451, 2022.
- [20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [21] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230, 2013.
- [22] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.
- [23] Ivan Fernandez, Ricardo Quisilant, Eladio Gutiérrez, Oscar Plata, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. Natsa: a near-data processing accelerator for time series analysis. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 120–129. IEEE, 2020.
- [24] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J Rossbach. Dgsf: Disaggregated gpus for serverless functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–750. IEEE, 2022.
- [25] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.
- [26] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- [27] Mingyu Gao and Christos Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–137. Ieee, 2016.
- [28] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764, 2017.
- [29] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence*, 36(4):744–755, 2013.
- [30] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moon-sang Kwon, Chanhoo Yoon, Sangyeun Cho, et al. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News*, 44(3):153–165, 2016.
- [31] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR, 2020.
- [32] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. A³: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020.
- [33] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [34] Zhenhao He, Dario Korolija, and Gustavo Alonso. Easynet: 100 gbps network for hls. In *2021 31th International Conference on Field Programmable Logic and Applications (FPL)*, 2021.
- [35] 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. 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)*, pages 763–783. IEEE, 2022.
- [36] Muhuan Huang, Kevin Lim, and Jason Cong. A scalable, high-performance customized priority queue. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.
- [37] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.
- [38] Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*, 2020.
- [39] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299*, 2022.
- [40] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. {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)*, pages 585–600, 2023.
- [41] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.
- [42] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.
- [43] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. Microrec: efficient recommendation inference by hardware and data structure solutions. *Proceedings of Machine Learning and Systems*, 3:845–859, 2021.
- [44] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, et al. Fleetrec: Large-scale recommendation inference on hybrid gpu-fpga clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 3097–3105, 2021.
- [45] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, et al. Co-design hardware and algorithm for vector search. *arXiv preprint arXiv:2306.11182*, 2023.
- [46] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaehoon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.
- [47] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.
- [48] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.
- [49] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2015.
- [50] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*, pages 1–12. IEEE, 2013.
- [51] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. *Proceedings of Machine Learning and Systems*, 3:387–400, 2021.
- [52] Liu Ke, Xuan Zhang, Benjamin Lee, G Edward Suh, and Hsien-Hsin S Lee. Disaggrec: Architecting disaggregated systems for large-scale personalized recommendation. *arXiv preprint arXiv:2212.00939*, 2022.
- [53] Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Nearest neighbor machine translation. *arXiv preprint arXiv:2010.00710*, 2020.

- [54] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172*, 2019.
- [55] Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. *arXiv preprint arXiv:2107.07566*, 2021.
- [56] Gunjae Koo, Kiran Kumar Matam, Te I, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–231, 2017.
- [57] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milošević, and Gustavo Alonso. Farview: Disaggregated memory with operator off-loading for database engines. *arXiv preprint arXiv:2106.07102*, 2021.
- [58] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83. IEEE, 2021.
- [59] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- [60] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.
- [61] Yejin Lee, Hyunji Choi, Sunhong Min, Hyunseung Lee, Sangwon Beak, Dawoon Jeong, Jae W Lee, and Tae Jun Ham. Anna: Specialized architecture for approximate nearest neighbor search. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 169–183. IEEE, 2022.
- [62] Charles E Leiserson. Systolic priority queues. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1979.
- [63] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. 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):869–883, 2008.
- [64] Mike Lewis, Marjan Ghazvininejad, Gargi Ghosh, Armen Aghajanyan, Sida Wang, and Luke Zettlemoyer. Pre-training via paraphrasing. *Advances in Neural Information Processing Systems*, 33:18470–18481, 2020.
- [65] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.
- [66] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.
- [67] Zhengyi Li, Cong Guo, Zhanda Zhu, Yangjie Zhou, Yuxian Qiu, Xiaotian Gao, Jingwen Leng, and Minyi Guo. Efficient activation quantization via adaptive rounding border for post-training quantization. *arXiv preprint arXiv:2208.11945*, 2022.
- [68] Zihao Li. The dark side of chatgpt: Legal and ethical challenges from stochastic parrots and hallucination. *arXiv preprint arXiv:2304.14347*, 2023.
- [69] Zonglin Li, Ruiqi Guo, and Sanjiv Kumar. Decoupled context processing for context augmented language modeling. *Advances in Neural Information Processing Systems*, 35:21698–21710, 2022.
- [70] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news*, 37(3):267–278, 2009.
- [71] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.
- [72] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla

- Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserri, et al. Genstore: a high-performance in-storage processing system for genome sequence analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 635–654, 2022.
- [73] Yuxian Meng, Xiaoya Li, Xiayu Zheng, Fei Wu, Xiaofei Sun, Tianwei Zhang, and Jiwei Li. Fast nearest neighbor machine translation. *arXiv preprint arXiv:2105.14528*, 2021.
- [74] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 804–817, 2021.
- [75] Mark Oskin, Frederic T Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 192–203. IEEE, 1998.
- [76] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019.
- [77] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- [78] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE micro*, 17(2):34–44, 1997.
- [79] Shvetank Prakash, Tim Callahan, Joseph Bushagour, Colby Banbury, Alan V Green, Pete Warden, Tim Ansell, and Vijay Janapa Reddi. Cfu playground: Full-stack open-source framework for tiny machine learning (tinyml) acceleration on fpgas. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 157–167. IEEE, 2023.
- [80] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- [81] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- [82] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.
- [83] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhl-gay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *arXiv preprint arXiv:2302.00083*, 2023.
- [84] Jie Ren, Minjia Zhang, and Dong Li. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. *Advances in Neural Information Processing Systems*, 33:10672–10684, 2020.
- [85] Erik Riedel, Christos Faloutsos, Garth A Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.
- [86] Devendra Singh Sachan, Mostofa Patwary, Mohammad Shoeybi, Neel Kant, Wei Ping, William L Hamilton, and Bryan Catanzaro. End-to-end training of neural retrievers for open-domain question answering. *arXiv preprint arXiv:2101.00408*, 2021.
- [87] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A {User-Programmable}{SSD}. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, 2014.
- [88] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–27, 2019.
- [89] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

- [90] Kurt Shuster, Mojtaba Komeili, Leonard Adolphs, Stephen Roller, Arthur Szlam, and Jason Weston. Language models that seek for knowledge: Modular search & generation for dialogue and prompt completion. *arXiv preprint arXiv:2203.13224*, 2022.
- [91] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. Near-memory computing: Past, present, and future. *Microprocessors and Microsystems*, 71:102868, 2019.
- [92] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- [93] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: Towards {Energy-Efficient},{In-Situ} data analytics on {Extreme-Scale} machines. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, 2013.
- [94] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [95] Xiaohui Wang, Yang Wei, Ying Xiong, Guyue Huang, Xian Qian, Yufei Ding, Mingxuan Wang, and Lei Li. Lightseq2: Accelerated training for transformer-based models on gpus. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2022.
- [96] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119. IEEE, 2020.
- [97] Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, et al. Ethical and social risks of harm from language models. *arXiv preprint arXiv:2112.04359*, 2021.
- [98] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik Lensch. Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2027–2035, 2016.
- [99] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 717–729, 2021.
- [100] Frank F Xu, Uri Alon, and Graham Neubig. Why do nearest neighbor language models work? *arXiv preprint arXiv:2301.02828*, 2023.
- [101] Dani Yogatama, Cyprien de Masson d’Autume, and Lingpeng Kong. Adaptive semiparametric language models. *Transactions of the Association for Computational Linguistics*, 9:362–373, 2021.
- [102] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.
- [103] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. {FAERY}: An {FPGA-accelerated} embedding-based retrieval system. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 841–856, 2022.
- [104] Jialiang Zhang, Soroosh Khoram, and Jing Li. Efficient large-scale approximate nearest neighbor search on opencl fpga. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4924–4932, 2018.
- [105] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the effect of data center resource disaggregation on production dbms. *Proceedings of the VLDB Endowment*, 13(9), 2020.
- [106] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.
- [107] Xinyi Zhang, Yawen Wu, Peipei Zhou, Xulong Tang, and Jingtong Hu. Algorithm-hardware co-design of attention mechanism on fpga devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.