International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 08 Issue: 06 | June - 2024 SJIF Rating: 8.448 ISSN: 2582-3930

Enhancing RAG Systems: A Survey of Optimization Strategies for Performance and Scalability

Nishanth S[1*] and Swetha S[1]

[1*]Department of ISE, RV College of Engineering, Mysore Road, Bengaluru, 560059, Karnataka, India.

*Corresponding author(s). E-mail(s): nishanths.is20@rvce.edu.in; Contributing authors: shwetha.ise@rvce.edu.in;

Abstract

Retrieval Augmented Generation (RAG) systems offer significant advancements in natural language processing by combining large language models (LLMs) with external knowledge sources to improve factual accuracy and contextual relevance. However, the computational complexity of RAG pipelines presents challenges in terms of efficiency and scalability. This research paper conducts a comprehensive survey of optimization techniques across four key areas: tokenizer performance, encoder performance, vector database search strategies, and LLM agent integra- tion. The study explores approaches to accelerate tokenization using Rust-based implementations and investigates hardware optimizations like CUDA and Ten- sor cores to boost encoder efficiency. Additionally, it delves into algorithms and indexing strategies for efficient vector database searches and examines meth- ods for optimising the interaction between retrieved knowledge and LLM agents. By analysing recent research and evaluating various optimization strategies, this paper aims to provide valuable insights into enhancing the performance and practicality of RAG systems for real-world applications.

Keywords: Retrieval-Augmented Generation, Performance optimization, Tokenization, WordPiece, Rust, FastTokenizer,Encoding, GPU, CUDA, Tensor Cores, Parallel Processing, Vector Databases, Indexing, HNSW, FAISS, DiskANN, LLM Agents, Prompt Engineering, Retriever

1 Introduction

RAG systems represent a sophisticated evolution in natural language processing (NLP) paradigms. These systems address the inherent limitations of traditional LLMs by integrating a retrieval component with a generative language model. Unlike LLMs, which rely primarily on knowledge encoded during pre-training, RAG systems dynamically access and incorporate information from external knowledge sources. This augmentation empowers RAG systems to generate more factually grounded, contextually relevant, and up-to-date responses, making them particularly valuable in scenarios requiring real-world knowledge and reasoning.

Furthermore, the grounding in external knowledge sources helps mitigate a com- mon issue plaguing LLMs: the tendency to ”hallucinate”[1] or fabricate information. Since LLMs are trained on massive amounts of text data, they can sometimes generate plausible-sounding but factually incorrect responses, especially when prompted with questions outside their pre-trained knowledge domain. RAG systems, by integrating information retrieval, have a mechanism to verify generated text against retrieved sources, improving the factual accuracy and trustworthiness of their output.

A typical RAG system consists of several key components that work together to generate informative and contextually relevant responses. These components include: **DocumentStore:** The DocumentStore is a

repository of documents that the RAG system can access to extract relevant information. It can be populated with various types of documents, such as web pages, news articles, and research papers.
Tokenizer: The Tokenizer is responsible for breaking down text into individual tokens or words. It plays a crucial role in the preprocessing stage, as it affects the quality of the embeddings generated by the encoder.
Encoder: The Encoder converts the tokenized text into dense vector representations, known as embeddings. These embeddings capture the semantic meaning of the text and are used for efficient retrieval and generation.
Vector DB: The Vector DB is a database that stores the vector embeddings generated by the encoder. It enables fast and efficient retrieval of similar vector embeddings, which are then used to identify relevant documents for the generation process.
LLM: LLM is a powerful neural network model that generates text based on the input it receives. In a RAG system, the LLM is used to generate text that is both informative and coherent, incorporating information retrieved from the Vector DB.
These components work in tandem to enable RAG systems to generate high- quality text that is grounded in real-world knowledge and tailored to the user's query or context. While RAG systems offer significant advantages over traditional LLMs, they also introduce additional computational overhead. Compared to a standard LLM query, RAG systems require additional computations for document retrieval, vector embedding generation, and retrieval of similar embeddings[2].

Additionally, the RAG system requires storing and managing the vector embed- dings, which can add to the memory requirements and increase latency. To mitigate these challenges and ensure efficient performance, it is crucial to optimize the RAG

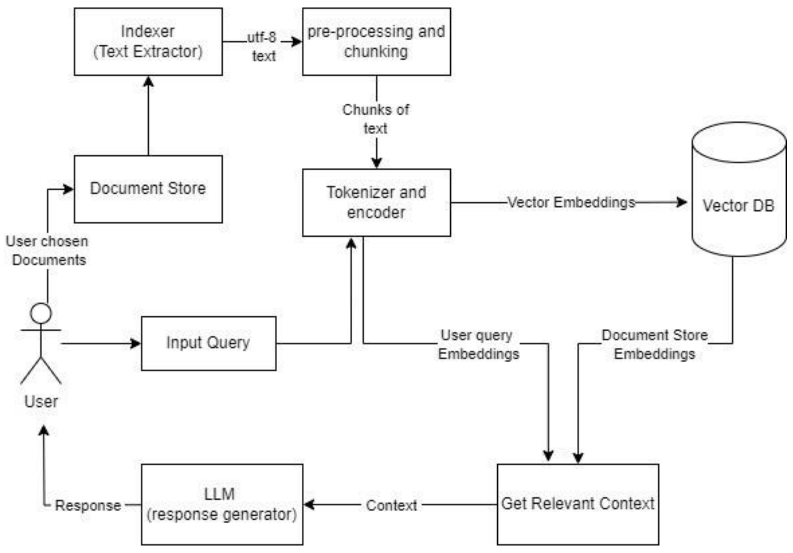

Fig. 1: Basic RAG Architecture

pipeline.

This research paper aims to address these computational challenges hindering the wider adoption of RAG systems. It provides a comprehensive survey of optimiza- tion techniques targeting four crucial areas: tokenizer efficiency, encoder performance, vector database search strategies, and integration of LLM agents. By examining these optimization avenues, the paper strives to offer insights into techniques that can enhance the speed, scalability, and cost-effectiveness of RAG systems, ultimately facilitating their more widespread deployment in real-world applications

2 Literature Review and Discussion

The literature review examines the expanding corpus of studies aimed at enhancing RAG pipeline performance and efficiency. To address the computational complexity inherent in these systems, researchers have explored a variety of approaches. This involves investigating different retrieval models, refining the way the retriever and generating modules interact, and creating new tactics for efficiently using outside knowledge sources.

Pierre et al., (2024) [3] presented a RAG system designed to improve the pre- cision, diversity, and factual grounding of natural language understanding. The research focuses on query optimization using language models like BERT and Orca2 for superior performance in open-domain question answering. It employs a prompt augmenter to dynamically analyse lexical fields and generate refined search queries, leveraging the power of language models for nuanced user intent understanding. This approach significantly improves the accuracy and relevance of document retrieval.

The paper further demonstrates the importance of prompt optimization and genera- tor integration to enhance retrieval efficiency and deliver comprehensive, contextually rich responses.

Gao et al., (2024) [4] provide a comprehensive survey of RAG systems, tracing their evolution from Naive RAG to Advanced and Modular RAG paradigms. The paper meticulously analyses the core techniques employed within RAG frameworks for retrieval, generation, and augmentation. The paper offers a critical evaluation of RAG methods and benchmarks, delving into common challenges faced by these sys- tems. These challenges include hallucinations, outdated knowledge, opaque reasoning, issues with retrieval accuracy, generation irrelevance and toxicity, integration diffi- culties, and redundancies in retrieved information. Additionally, the paper addresses limitations in handling specialised terms, struct.

Jiang et al., (2023) [5] introduce FLARE (Forward-Looking Active Retrieval Augmented Generation), a novel method designed to enhance RAG systems. FLARE dynamically determines when and what information to retrieve during the text gen- eration process. By predicting upcoming sentences and identifying low-confidence tokens, FLARE intelligently triggers the retrieval of relevant documents for sentence regeneration. Comprehensive experiments across four long-form knowledge-intensive generation tasks demonstrate the superiority or competitive performance of FLARE compared to baselines. Evaluation metrics include exact match (EM), token-level F1, precision, recall, RoBERTa-based QA score (DisambigF1), ROUGE, and an overall DR score combining DisambigF1 and ROUGE.

Jin et al., (2024) [6] propose RAGCache, a sophisticated multilevel dynamic caching system specifically to enhance RAG performance. It leverages a knowledge tree to structure intermediate retrieval states, caching them strategically across GPU and host memory. RAGCache employs a prefix-aware Greedy Dual-Size Frequency (PGDSF) replacement policy to prioritise vital key-value tensors. Additionally, it implements cache-aware reordering for improved hit rates and thrashing prevention, and dynamic speculative pipelining to mask latency by overlapping retrieval and LLM inference. RAGCache demonstrates significant performance gains, reducing Time to First Token (TTFT) by up to 4x and boosting throughput by up to 2.1x in comparison to vLLM with Faiss integration.

Mashagba et al., (2011) [7]. explored the application of Genetic Algorithms (GA) for query optimization within the Vector Space Model (VSM) of information retrieval. Their focus is on the Arabic language, and they investigate various similarity mea- sures (Dice, Inner Product), fitness functions, mutations, and crossover strategies within the GA framework. Their findings indicate that a GA approach employing a one-point crossover operator, point mutation, and the Inner Product similarity measure as its fitness function yields the most significant improvement in IR system performance within the VSM context. Specifically, this GA configuration (GA1) demonstrates an 11.94% performance gain compared to traditional VSM approaches.

Yan, Wang & Chu(2020) [8] conducted a thorough investigation into the inner workings of NVIDIA Turing architecture Tensor Cores, specifically as they pertain to half-precision matrix multiplication. They unveil details about instructions, register usage, data layouts, along with the throughput and latency of Tensor Core operations. Furthermore, they benchmark Turing GPU memory systems and provide performance analysis. Interestingly, their analysis uncovers that memory bandwidth (DRAM, L2 cache, shared memory) has become the limiting factor in HGEMM performance, contrasting with the previous assumption of computation-bound limitations. Building upon their findings, they introduce optimizations to Tensor Core-based HGEMM, focusing on blocking size, data layouts, prefetching, and instruction scheduling. Comprehensive evaluations demonstrate that their optimised HGEMM implementation achieves an average speedup of 1.73x on NVIDIA Turing RTX2070 GPUs and 1.46x on T4 GPUs compared to the native cuBLAS 10.1 implementation. The use of tensor core will help in optimization of inference speed in the encoding process in the RAG pipeline.

In this upcoming section, the approach taken to survey optimization techniques for RAG systems is outlined.

Table 1: Literature survey summary table

Study	Focus	Approach	Key Findings
Pierre et al.	Query optimization in RAG	BERT, Orca2, prompt augmenter	Improved accuracy, rel- evance, and contextual richness in document retrieval
Gao et al.	Evolution and chal- lenges in RAG systems	Survey of RAG tech- niques	Critical evaluation of RAG challenges and optimization avenues
Jiang et al.	Enhancing RAG with dynamic retrieval	FLARE method	Improved perfor - mance in long-form knowledge-intensive tasks
Jin et al.	Performance enhance- ment in RAG	RAG Cache system	Reduced TTFT by up to 4x, increased throughput by up to 2.1x

Mashagba et al.	Query optimization in VSM	Genetic Algorithms	11.94% performance gain in IR system for Arabic language
Yan, Wang, Chu	Optimization of Tensor Cores	Half-precision matrix multiplication	1.73x speedup on RTX2070, 1.46x speedup on T4 GPUs

2.1 Tokenizer Performance

A tokenizer [9] is a crucial component in NLP systems responsible for breaking down raw text into smaller, meaningful units called tokens. These tokens can be words, subwords, or even characters, depending on the tokenizer's configuration and the requirements of the task at hand. The primary goal of tokenization is to prepare the text for further processing, such as parsing, semantic analysis, or machine learning tasks. Tokenization typically involves following phases:

Normalization: In this phase, the text undergoes preprocessing steps to standardise its format and remove any inconsistencies or noise. This may include tasks such as converting text to lowercase, removing punctuation, handling contractions and applying Unicode normalisation.

Pre-tokenization: The pre-tokenization phase involves splitting the text into initial units, such as words or subwords. This step is crucial for languages with complex morphology or agglutinative characteristics, where words may consist of multiple morphemes or have variable forms.

Running the input through the model: Once pre-tokenized, the text is pro- cessed through the model, which may involve converting the tokens into numerical representations suitable for machine learning algorithms or other downstream tasks. This phase typically relies on the model architecture and specifications to determine how tokens are processed and encoded.

Post-processing: Finally, in the post-processing phase, additional operations may be applied to the tokenized output to prepare it for specific tasks or applications. This includes adding special tokens, generating attention masks or assigning token type IDs, depending on the requirements of the model or task.

In this sub-section, the focus lies on examining the substantial performance gaps between Rust-based fast tokenizers and Python-based slow tokenizers provided by the Hugging Face library. Rust, a systems programming language renowned for its emphasis on performance, memory safety, and concurrency, serves as the backbone for the fast tokenizers implemented in the Hugging Face Tokenizers library.

Leveraging Rust's inherent advantages in speed and efficiency as discussed in [10], these fast tokenizers demonstrate remarkable processing speeds, particularly when dealing with large batches of text simultaneously. This speed advantage is particu- larly pronounced during batch encoding, where parallelization capabilities inherent to Rust-based implementations lead to drastic reductions in processing times. While the speed advantage of fast tokenizers may not be immediately apparent when tok- enizing individual sentences, their superior performance becomes unmistakable when processing extensive datasets. The following tokenization findings were obtained from the UC Irvine Machine Learning Repository's Drug Review dataset, as reported on the hugging face website.

Table 2: Fast vs Slow Tokenizer Results

	Fast tokenizer	Slow tokenizer
batched=True	10.8s	4min 41s
batched=False	59.2s	5min 3s

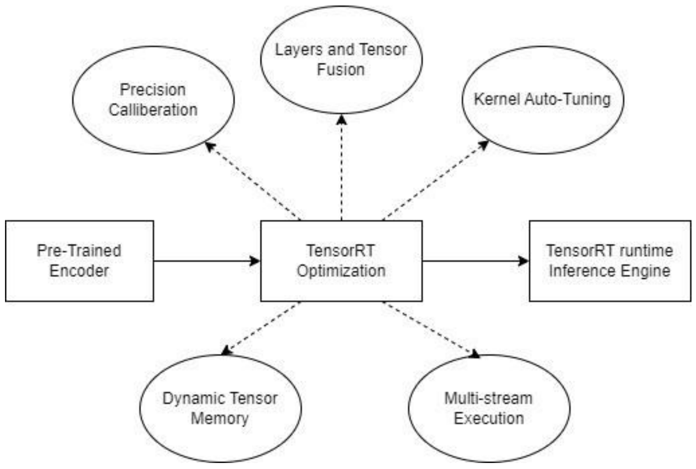

Fig. 2: Modules in TensorRT Optimization [27]

2.2 Encoder Performance

The encoder model [11] plays a crucial role in transforming tokenized textual input into vector embeddings. The model, often built upon deep learning architectures such as transformers, receive tokenized sequences as input and produce dense vector representations, or embeddings, for each token. The encoder's task is to capture the semantic and contextual information of the input tokens, encoding them into high- dimensional vector spaces. By leveraging self-attention mechanisms and multi-layer architectures, encoder models can effectively capture complex relationships within the input text, enabling downstream tasks such as text generation, sentiment analysis, and language understanding.

The advancements in hardware acceleration, such as CUDA and Tensor Cores provided by Nvidia, significantly augment the performance of encoder models. These specialised computing units are designed to handle parallel processing tasks effi- ciently. By harnessing the power of parallelization, CUDA and Tensor Cores expedite the computations involved in encoding tokenized sequences into vector embeddings. This parallel processing capability leads to substantial performance boosts, enabling encoder models to process large volumes of data swiftly and efficiently. As a result, tasks reliant on encoder models, such as text classification, sentiment analysis, and language understanding, benefit from accelerated processing times and enhanced performance.
CUDA cores [12], integral components of Nvidia GPUs, play a pivotal role in accelerating parallel processing tasks. The number and speed of CUDA cores directly affect data processing efficiency. Analogous to checkout lanes in a supermarket, CUDA cores operate in parallel, with each core capable of processing one task at a time. Just as opening multiple lanes expedites checkout for numerous customers, increasing the number of CUDA cores enables parallel processing of data points, significantly enhancing throughput. As Nvidia GPUs evolve, the proliferation of CUDA cores has surged. Tensor cores [13], specifically engineered for accelerating deep learning computations, excel in performing complex matrix operations

crucial for neural network processes at remarkable speeds.

Buber et al., (2018) [14] did a comparative study on the performance of CPU and GPUs and concluded that GPUs were around 4-5 times faster than CPUs. Nishanth et al., (2022) [15] also reinforced the theoretical superiority of GPUs over CPUs when it comes to deep learning model training and inference. It was shown that GPUs were faster by 8.8 times when trained on Convolutional Neural Networks (CNNs) and 4.90 times faster when trained with Recurrent Neural Networks (RNNs). Ghadani et al.,(2020) [16] reported that the frames per second (FPS) was 4.5 times faster than the non-optimized inference speeds when trained with CUDA and TensorRT execution optimization. The table 2 provides inference optimization provided by TensorRT based on [27]

Table 3: Inference Speed of TensorRT against PyTorch

Model name	Throughput (PyTorch)	Throughput (TensorRT)	Latency (PyTorch)	Latency (TensorRT)
VGG16	196	327	5.27	3.11
Alexnet	997	1443	1.05	0.789
Resnet 152	69.8	160	14.9	6.3
Densenet 161	47.7	135	22.2	7.48
Mobilenet v2	219	1163	4.51	0.92

2.3 LLM agents and Prompt Engineering

LLM agents [17] are AI systems that combine LLMs with other tools and components to perform complex tasks and interact with various environments. They leverage the language understanding and generation capabilities of LLMs to plan actions, execute them using external tools, observe the outcomes, and adapt their strategies based on feedback. This allows LLM agents to solve problems that require multi-step reasoning, decision-making and interaction with the real world or digital systems.

Prompt engineering is the process of designing and refining the input (prompts) given to an AI model to elicit desired responses. It involves carefully crafting instructions, questions or contexts to guide the model's behaviour and output. By understanding how language models work and experimenting with different phrasing, formatting and examples, prompt engineers can optimise the quality, relevance, and creativity of the generated text [18].

Liu et al., (2023) [19] propose P-Tuning, a novel method for optimising the performance and stability of prompt-based natural language understanding (NLU). The paper identifies that manual discrete prompts, while effective, can lead to unpredictable fluctuations in performance. P-Tuning addresses this by introducing trainable continuous prompt embeddings in addition to discrete prompts, aiming to stabilise training and boost performance across various NLU tasks. Houlsby et al. (2019) [20] talk about adaptor modules and transfer learning. This method adds only a few trainable parameters per job, resulting in a compact and flexible model. The method adds just 3.6% of the parameters per job and achieves within 0.4% of the performance of complete fine-tuning on GLUE. Fine-tuning, on the other hand, trains all of the parameters for each job.

Guo et al., (2024) [21] explored the integration of LLMs within multi-agent systems (MAS). It delves into how LLMs, with their advanced natural language processing abil- ities can enhance MAS by facilitating communication, cooperation and coordination among agents. The survey discusses various approaches for incorporating LLMs into MAS architectures, including centralised and decentralised frameworks. It also exam- ines the potential benefits of LLM-based agents, such as improved decision-making, adaptable behaviours and the ability to handle complex tasks that require language understanding and generation.

2.4 Vector Embedding Indexing and Search Algorithms

Vector embeddings are numerical representations of complicated data, such as text, graphics, or audio [22]. Vector databases are specialised databases made to store and handle vector embeddings effectively. Vector databases store data as points in a high-dimensional space as opposed to standard databases, which store data in rows and columns. This enables similarity search and retrieval based on distance metrics.

Johnson, Douze and J´egou (2017) [23] discuss Facebook AI Similarity Search (FAISS) algorithm for indexing of vector embedding in vector databases. Developed by Facebook AI Research, FAISS focuses on optimised implementations of fun- damental techniques like multi-threading and BLAS libraries for efficient distance computations. It also utilises SIMD vectorization and offers GPU support for accel- erated performance. Notably, FAISS employs approximate nearest neighbour search algorithms like Product Quantization and Inverted File Index to handle massive datasets effectively. The FAISS algorithm can construct a k-nearest neighbour graph (k=10) on 95 million images in 35 minutes using four GPUs. Furthermore, FAISS can efficiently cluster 67.1 million vectors into 262,144 centroids in 43.8 minutes on eight GPUs. These benchmarks demonstrate the substantial performance gains provided by FAISS, particularly when dealing with large-scale datasets.

If I/O efficiency is the priority during the indexing process, DISK-ANN is an algorithm to look out for. DiskANN, and its improved version LM-DiskANN [25], are graph-based Approximate Nearest Neighbour (ANN) search algorithms designed to handle extremely large datasets that cannot fit entirely in memory. Unlike traditional graph-based indexes that reside solely in memory, DiskANN leverages disk storage for storing the index structure, loading portions into memory on-demand during search. LM-DiskANN further enhances this approach by storing complete routing information within each node, minimising memory consumption and disk I/O operations. This allows for efficient search on large-scale datasets while maintaining a low memory footprint

Table 4: Benchmarking of different Vector Databases

Engine	**Dataset**	**Upload + Index Time(m)**	**P95(ms)**	**Latency(ms)**
qdrant	dbpedia-openai-1M-1536-angular	24.43	4.95	3.54
elasticsearch	dbpedia-openai-1M-1536-angular	83.72	72.53	22.10

redis	dbpedia-openai-1M-1536-angular	92.49	160.85	140.65
weaviate	dbpedia-openai-1M-1536-angular	25.98	465.42	279.44
milvus	dbpedia-openai-1M-1536-angular	1.16	441.32	393.31

3 Methodology

The initial phase of this literature survey involved a comprehensive search for schol- arly articles and publications using major online research databases such as Google Scholar, Semantic Scholar, and arXiv. These platforms were chosen for their extensive coverage of academic literature in computer science, artificial intelligence and natural language processing.

Beyond academic databases, the literature survey also explored various online resources to identify cutting-edge research and implementation practices. These resources included blogs, forums and GitHub repositories of prominent RAG projects. Additionally, relevant conference proceedings and workshops were examined to stay up-to-date on the latest advancements in the field. The survey primarily focused on peer-reviewed journal articles and conference papers published within the past few years to ensure the inclusion of the most recent and relevant research findings. How- ever, medium articles by recognized researchers in the field of NLP were also included to capture insights and practical applications that might not be readily available in traditional academic publications.

4 Conclusion

This comprehensive survey of optimization techniques for RAG systems underscores their immense potential while highlighting the critical need for continuous improve- ment in their efficiency and scalability.

While this research has highlighted substantial progress in optimising RAG sys- tems, significant challenges remain. Current systems can struggle with issues such as retrieving missing content, failing to rank relevant documents highly enough [26], or not extracting answers correctly even when the relevant context is present. Fur- ther challenges arise in handling response format, ensuring appropriate specificity, integrating knowledge sources effectively, and mitigating the phenomenon of hallu- cination. Although methods like the Agentic RAG approach from LlamaIndex offer potential solutions to some of these issues, a robust and comprehensive solution to these challenges remains an active area of research.

5 Discussion

The existing research on RAG reveals a multifaceted approach to improving its performance and addressing its inherent challenges. Studies have demonstrated sig- nificant progress in optimising various stages of the RAG pipeline. This includes enhancing query formulation and document retrieval accuracy, as well as refining response generation techniques. One area of focus is query optimization. Researchers have investigated leveraging large language models to dynamically refine user queries, leading to more precise and relevant document retrieval. This focus on understanding user intent has been pivotal in improving the quality of retrieved information. Addi- tionally, the application of genetic algorithms has showcased potential in optimising query formulation for specific languages and information retrieval models.

On the other hand, research has explored both software and hardware-based optimizations to tackle the computational demands of RAG systems. Innovative caching systems have been proposed to expedite document retrieval and reduce latency. Additionally, the utilisation of specialised hardware architectures like GPUs and their associated programming models like CUDA has demonstrated substantial speedups in encoder performance. However, memory bandwidth constraints remain a bottleneck even with hardware acceleration, emphasising the need for continued research into efficient data management and transfer within RAG pipelines.

Furthermore, the literature underscores the importance of prompt engineering and LLM agent integration in enhancing the effectiveness of RAG systems. While prompt engineering techniques have proven successful in improving response quality, the emer- gence of LLM agents introduces new possibilities for multi-step reasoning and complex task execution within RAG frameworks.

6 References

1. Ziwei Xu, Sanjay Jain, & Mohan Kankanhalli. (2024). Hallucination is Inevitable: An Innate Limitation of Large Language Models.
2. Yuren Mao, Xuemei Dong, Wenyi Xu, Yunjun Gao, Bin Wei, & Ying Zhang. (2024). FIT-RAG: Black-Box RAG with Factual Information and Token Reduction.
3. Julien Pierre Edmond Ghali, Kosuke Shima, Koichi Moriyama, Atsuko Mutoh, & Nobuhiro Inuzuka. (2024). Enhancing Retrieval Processes for Language Generation with Augmented Queries.
4. Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, & Haofen Wang. (2024). Retrieval-Augmented Generation for Large Language Models: A Survey.
5. Zhengbao Jiang, Frank F. Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi- Yu, Yiming Yang, Jamie Callan, & Graham Neubig. (2023). Active Retrieval Augmented Generation.
6. Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, & Xin Jin. (2024). RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation.
7. Mashagba, Eman & Fares, Feras & Nassar, Mohammad. (2011). Query Optimiza- tion Using Genetic Algorithms in the Vector Space Model. International Journal of Computer Science Issues. 8.
8. D. Yan, W. Wang and X. Chu, ”Demystifying Tensor Cores to Optimise Half- Precision Matrix Multiply,” 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, 2020, pp. 634-643, doi: 10.1109/IPDPS47924.2020.00071.
9. Webster, Jonathan & Kit, Chunyu. (1992). Tokenization as the initial phase in NLP. 1106-1110. 10.3115/992424.992434.
10. Matsakis, N., & Klock, F. (2014). The rust language. In Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (pp. 103–104). Association for Computing Machinery.
11. Zhao, Penghao & Zhang, Hailin & Yu, Qinhan & Wang, Zhengren & Geng, Yunteng & Fu, Fangcheng & Yang, Ling & Zhang, Wentao & Cui, Bin. (2024). Retrieval- Augmented Generation for AI-Generated Content: A Survey.
12. Ho, Khoa & Zhao, Hui & Jog, Adwait & Mohanty, Saraju. (2022). Improving GPU Throughput through Parallel Execution Using Tensor Cores and CUDA Cores.
13. Markidis, S., Chien, S., Laure, E., Peng, I., & Vetter, J. (2018). NVIDIA Ten- sor Core Programmability, Performance & Precision. In 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE.
14. Buber, Ebubekir & Diri, Banu. (2018). Performance Analysis and CPU vs GPU Comparison for

Deep Learning. 1-6. 10.1109/CEIT.2018.8751930.
15. N. S, M. S. Rao, S. B M, P. T and C. N K, ”Performance of CPUs and GPUs on Deep Learning Models For Heterogeneous Datasets,” 2022 6th International Con- ference on Electronics, Communication and Aerospace Technology, Coimbatore, India, 2022, pp. 978-985, doi: 10.1109/ICECA55336.2022.10009148.
16. Al Ghadani AKA, Mateen W, Ramaswamy RG. Tensor-Based CUDA Opti- mization for ANN Inferencing Using Parallel Acceleration on Embedded GPU. Artificial Intelligence Applications and Innovations. 2020 May 6;583:291–302. doi: 10.1007/978-3-030-49161-1 25. PMCID: PMC7256376.
17. Zhiheng Xi, Wenxiang Chen, X̄in Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou,

Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, & Tao Gui. (2023). The Rise and Potential of Large Language Model Based Agents: A Survey.
18. Banghao Chen, Zhaofeng Zhang, Nicolas Langren´e, & Shengxin Zhu. (2023). Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review.
19. Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, & Jie Tang. (2023). GPT Understands, Too.
20. Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, & Sylvain Gelly. (2019). Parameter-Efficient Transfer Learning for NLP.
21. Guo, Taicheng & Chen, Xiuying & Wang, Yaqi & Chang, Ruidi & Pei, Shichao & Chawla, Nitesh & Wiest, Olaf & Zhang, Xiangliang. (2024). Large Language Model based Multi-Agents: A Survey of Progress and Challenges. 10.13140/RG.2.2.36311.85928.
22. Yikun Han, Chunjiang Liu, & Pengfei Wang. (2023). A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge.
23. Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazar´e, Maria Lomeli, Lucas Hosseini, & Herv´e J´egou. (2024). The Faiss library.
24. Jeff Johnson, Matthijs Douze, & Herv´e J´egou. (2017). Billion-scale similarity search with GPUs.
25. Y. Pan, J. Sun and H. Yu, ”LM-DiskANN: Low Memory Footprint in Disk-Native Dynamic Graph-Based ANN Indexing,” 2023 IEEE International Conference on Big Data (BigData), Sorrento, Italy, 2023, pp. 5987-5996, doi: 10.1109/Big- Data59044.2023.10386517.
26. Greyling, C. (2024, April 1). Challenges in adopting retrieval-augmented generation solutions. Medium. https://cobusgreyling.medium.com/challenges-in- adopting-retrieval-augmented-generation-solutions-eb30c07db398
27. Ranvir, H. (2020, August 21). Speed up deep learning inference using TEN- SORRT on gpus. Medium. https://hemantranvir.medium.com/speed-up-deep- learning-inference-using-tensorrt-on-gpus-29c709e6a9db