# Project TLDR - Capstone Report - 2024 Autumn

**Project Title:** Project TLDR: Standalone Desktop application for Question-Answering and Summary using resource efficient LLMs

**Student:** Manu Hegde

**Project Committee:**
Prof. Erika Parsons, Committee Chair
Prof. Michael Stiber, Committee Member
Prof. Shane Steinert-Threlkeld, Committee Member

**Table of contents:**


I. Overview
II. Literature Review
    A. Ingredients of RAG Application
        a) Embedding Phase
        b) Retrieval Phase
        c) Evaluation of RAG
    B. Apple M1 Architecture
        a) Chip Layout and Capabilities
        b) Accelerating LLM execution
    C. Text generation using Large Language Models
        a) Process of text generation
        b) Resource optimization techniques summary

III. Current Status
    A. Choices Made
    B. Performance characteristics

IV. Next Steps
    A. FAISS integration and optimization (metal/npu support?)
    B. Llama.cpp optimization (add NPU support?)

## I. Overview

This project aims to develop a standalone desktop application that enables ChatGPT-like Question-Answering and Summarization on top of a corpus of documents on the user's device. The application shall embody a resource efficient implementation of a chosen LLM (Large Language Model), targeting Apple's M1/M2 hardware platform. The primary intended user base for this application is students and researchers in academia.

The motivation behind this project stems from the growing need for tools that can efficiently process and interpret large volumes of academic and research material. Current solutions often require significant manual interventions or involve sharing data with third-party servers, raising concerns about privacy and data security. By creating a resource-efficient, standalone application, this project aims to provide students and researchers with a tool that offers convenience, confidentiality, and enhanced productivity. The project considers recent advancements in natural language processing (NLP), particularly the use of large language models (LLMs) for tasks like summarization and question-answering. The application will utilize techniques like weight quantization and low-rank adaptation to optimize LLM performance on Apple's M1/M2 architecture, including the use of Apple Metal GPU and Apple Neural Engine (ANE) for hardware acceleration. The project will incorporate retrieval-augmented generation to yield contextually relevant outputs derived from text corpus provided by the user, thereby ensuring credibility of the information.

The expected contributions of this project include the development of a desktop application with a graphical user interface, capable of processing and summarizing large text corpora locally, without requiring an internet connection. The application will also explore the potential for running complex NLP models in resource-constrained environments, offering insights into optimizing LLMs for specific hardware platforms. Ultimately, this project aims to provide a valuable tool for researchers and students, enhancing their ability to interact with and understand extensive collections of academic materials.

In conclusion, the project aims to create a standalone desktop application that uses a Large Language Model for document-based Q&A, with minimal and predictable resource usage for smooth multitasking on the user's device.

## II. Literature Review

A. Ingredients of RAG (Retrieval Augmented Generation) Application:

A Retrieval Augmented Generation system contains four major components: ***Document loader***, ***Text embedder***, ***Context retriever*** and ***Language model***. As illustrated in Figure 1.1, the documents are embedded and stored in a vector database which is queried to obtain the relevant context for the user's query and passed along to the LLM. LLM hence able to generate outputs based on a particular source of data, improving the credibility and usefulness of the output.

The RAG process could be divided into two phases: Embedding, Retrieval. The Embedding phase (as illustrated by Figure 1.2), consists of loading, chunking and embedding the document. On the other hand, the Retrieval phase deals with vector similarity search to fetch relevant context vectors in their original text form and pass them along with user query to the Large Language Model, as seen in Figure 1.3.

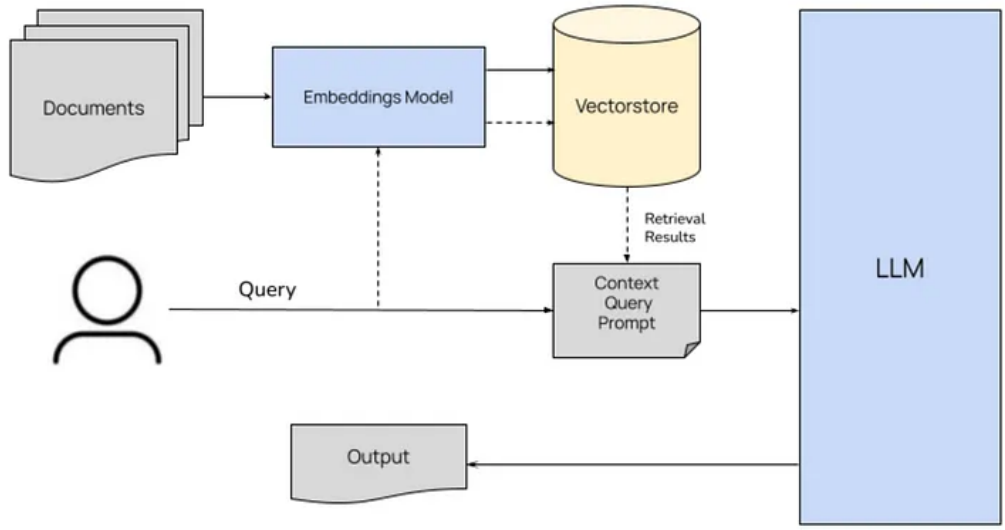



*Figure 1.1 – Architecture of a RAG Application [1]*

**a) Embedding phase:** In this phase we build our text corpus that can later be used as the knowledge base to answer user's queries. Although the concept of text embeddings has been since the introduction of Word2Vec embeddings [3] in 2013, their use for storing and retrieving contextual information is a recent innovation. This came about with the introduction of 'In context learning' technique as mentioned in the GPT3 paper by OpenAI in 2020 [4]. This technique particularly works well for decoder-only transformers due to their purely auto-regressive nature. Due to this, one can obtain relevant answers on a task or topic with just a few or even just one example or reference [5].

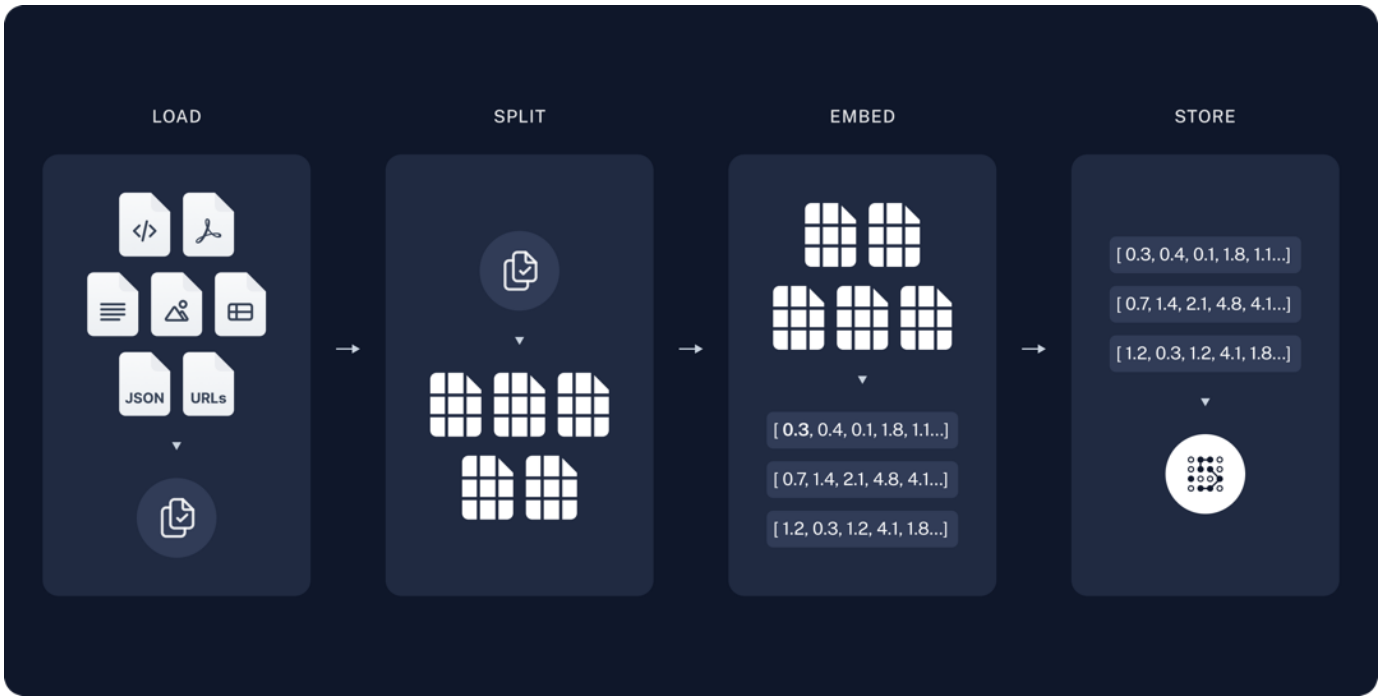



*Figure 1.2: Illustration of the Embedding Phase [2]*

The embedding phase consists of the following steps:

i. **Document processing:** In this phase we load the document from its source format (txt, pdf, docx). This could be done in either text-only mode or multi-modal mode. Where the latter allows for the preservation and usage of diagrams and images present in the source document.

The ***Document Loader*** component oversees this phase and performs *loading* and *chunking*. The latter step is handled by text-splitters [6] which split the data based on preset configuration.

**Why do we need chunking**? Quick answer, LLM Context-Length and limited GPU RAM.
LLMs have a limited context length, i.e. the amount of text that can be processed in one shot. This ranges from 2K (i.e. 2000) for GPT3, 4K for Llama2 up to 128K for Llama 3 [7]. Although a context size like 128K, to leveraging large context sizes also requires large amounts of GPU memory to store the model weights and attention.
This need for memory increases further when techniques like kv-caching (key & value caching of attention heads) and speculative decoding are implemented to improve output generation speed.

Hence, it is important to ensure only relevant and limited text is passed on as context for a user query. To do so, the document needs to be split into chunks.

Although splitting the text is straightforward, determining the split locations is crucial and should be considered carefully due to the following reasons.

- If the text chunk is too large, there could be a loss of knowledge due to 'Lost in the middle' problem [8] i.e. only the information at the begging and end of chunk gets used effectively.
- If the text chunk is too small, it could result high redundancy of data, since chunks often maintain overlapping content to preserve context. Additionally, too many chunks also result in greater latency of the system.

Hence, determining the chunk size is extremely crucial to ensure effectiveness of the RAG System. Chunking can be done by following techniques:

1. Length based chunking: Chunking the text based on a fixed pre-determined size. The size can be expressed in terms of token-level or character level. A token, yielded after sub-word tokenization of input text using byte-pair-encoding [9] can range from a character to sub-word to even an entire phrase (common multi word sequences).

2. Semantic structure-based chunking: Splitting the test based on the structure of the document i.e. converting paragraphs or blocks of text into a single chunk.

3. Context aware splitting: Splitting using textual context i.e. retaining an entire sub-context or sub-topic inside a single chunk.

Note: In case of token length-based chunking or context aware chunking, the embedding step would precede splitting.

**ii. Text Embedding**: In this phase the **Text Embedder** tokenizes and obtains embeddings for the text. This embedding need not be same as the one used by the LLM since these embeddings are used only to perform vector search. Once all the chunks relevant to a user query are obtained, their corresponding text value is returned. We do not pass on the embedding values to the LLM. Models like BERT / DistilBERT are often used for this purpose since they are faster to run and generate smaller embeddings (ex: 512 or 768) unlike LLMs like LLaMa [10] or GPT3 [4] which use embeddings with size between 1024-12,288. While smaller embeddings are less accurate, they are efficient for the purpose of high-level context retrieval.

**b) Retrieval Phase:** This phase happens every time a user inputs a question or task to the model. As Illustrated in Figure 1.3, this phase utilizes the vector database created during the embedding phase and retrieves all content relevant to the user's input prompt and passes it on to the LLM.

The phase involves the following steps:

i. **Context Retrieval:** This step is carried out by the ***Context Retriever*** in the following sequence.
   a. Embed the user's input using the same language model used during the embedding phase
   b. Perform similarity search on the vector database using the embedded user input

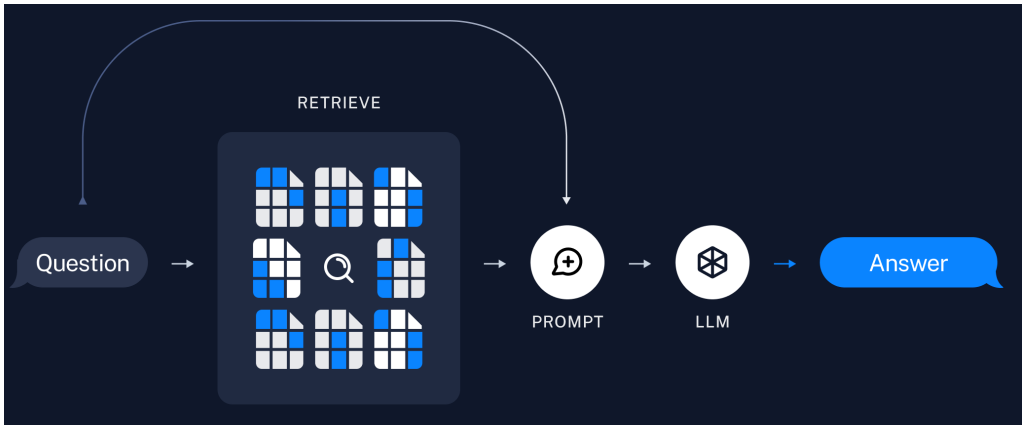



*Figure 1.3: Illustration of the Retrieval Phase [2]*

The Similarity Search on the database is the most crucial step since it determines the relevance and quality of the output generated. The similarity in the embeddings space is attributable to neighborliness. Points closer to each other are similar than those which are distant. Finding neighbors brings about a classic trade-off between resource available, latency and output quality. While performing complete scan of the database could obtain the best result, it is extremely costly to do so [11].

Hence, to balance the trade-offs, we may use one of the following algorithms:

1. kNN: k nearest neighbors is one of the most popular and simplest algorithms to find points in vector space that are closest to a given point. But this involves recursive iterations over vector space [12].

Approximate Nearest Neighbor algorithms:

2. Locality-Sensitive Hashing (LSH): It leverages multiple cryptographic, 1-way hash functions to hash vectors into hash buckets. The search query is then subjected to the same hashing process and the contents of the hash bucket it maps to are returned as its neighbors [12].

3. k-d Trees: k-d trees are binary search trees, and they can used by partitioning each embedding dimension into binary partitions (like decision tree algorithms used by random forests). This tree is built through repeat splitting until all vectors are accounted for. During query phase, the query is split on the same logic and the closest sub-tree results as its neighbors [12].

4. Hierarchical Navigable Small World (HNSW) Graphs: It constructs a multi-layered hierarchical graph where each edge of a node is a similar neighbor. The different layers represent the levels of granularity. The search starts with the upper most layer and gradually moves down to more granular layers. Hence, it can execute a faster search mechanism where the search could be terminated early if has very low similarity with its neighbors in the upper layers [12].

5. ScaNN (Scalable Nearest Neighbors): Implements a multistage scaling, pruning, partitioning and refinement of graph of embedded points. It also uses asymmetric vector quantization to reduce the memory footprint of vectors. Invented and used by Google to handle billions of searches [13]

All the above algorithms need a mechanism to calculate distance between two vectors, which could be done in one of the following ways:

1. Dot Product
2. Cosine Similarity
3. Euclidean distance (L1/L2 norm)

Cosine similarity is in general the most preferred way to perform vector similarity for embeddings since its scale invariant and interpretable (as a function of the cosine value of angle between the vectors), despite weaknesses like its reliance on robust regularization [14].

The vector search mechanism should also for incremental addition or deletion of documents of chunks/documents, it is often recommended to use an existing framework to handle the process in an efficient manner. One of the most popular libraries for the purpose is FAISS by Meta [15] which also supports CUDA GPU based vector search. While FAISS is a pioneer in the space, many more alternatives have emerged since the advent of ChatGPT in 2023 [16].

**ii. Output Generation:**

In this phase, the retrieved context and the user input, both in their plain text form are passed onto the Large Language Model as illustrated in Figure 1.3. The LLM then processes over the portion of input that fits within its context window and yields an output token. The output token is then concatenated with the input and passed to the LLM again. This process repeats until the LLM generates an '<EOS>' (end of sequence) token.

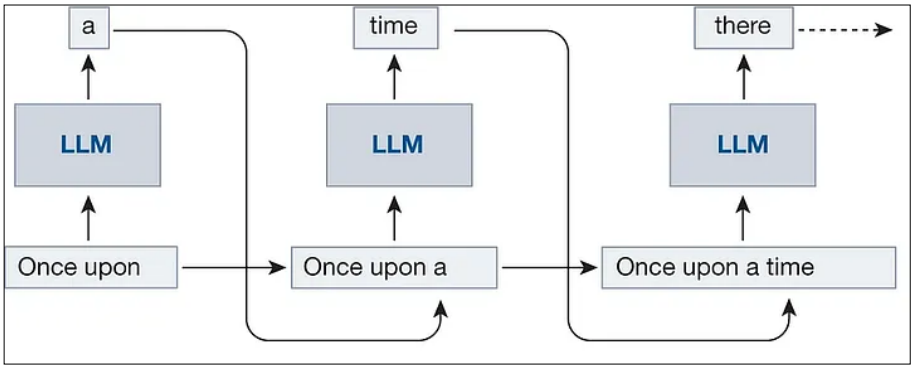



*Figure 1.4: Auto regressive process of LLM output generation*

**c) Evaluation of RAG**

Evaluation of the output quality of a RAG application needs to be performed in terms of relevance to the user's query and the content present in the vector database. The resultant metrics are designed to measure different performance of different components of the system since obtaining a single score can lead to difficulty in attribution and optimization.

The evaluation is performed using classical metrics like Precision, Recall, F1 score and NLP metrics like ROUGE. The evaluation in performed for multiple factors of the output as in Figure 1.5:

- Faithfulness, Output relevance & Semantic Similarity: Evaluate output quality with respect to inputs.
- Context Recall, Context Precision and: Evaluate the context retrieval and it's usage by the LLM.

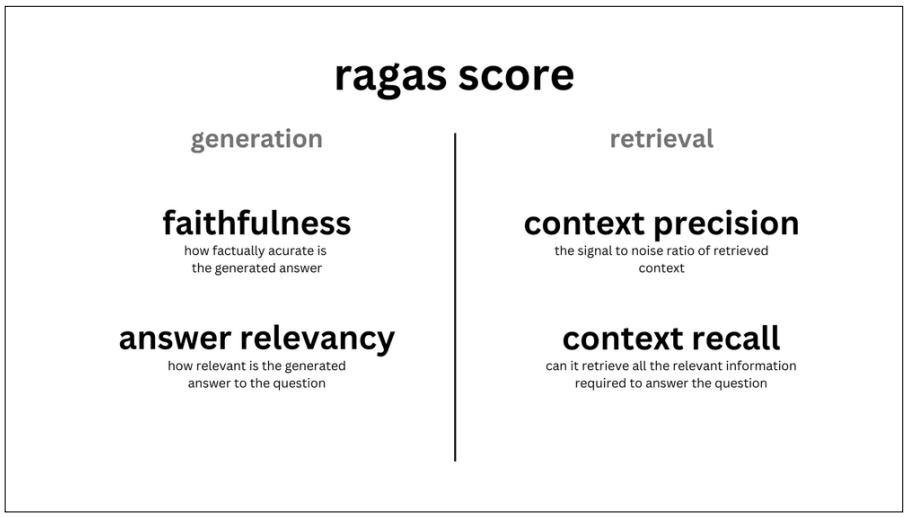



*Figure 1.5: RAG application scores [18]*

**B. Apple M1 Architecture:**

a) Chip Layout and Capabilities

The Apple M1 architecture, released in 2020 [20] is an arm-based SOC (system on chip) architecture. It carries some key features which enables the foundations of this project as listed below:

- Built in GPU with 7-8 cores or more yielding 5.2 TOPS of Int8 precision
- Built in NPU (Neural Processing Unit aka Apple Neural Engine) with 11 TOPS of Int8 precision
- Unified and shared memory across CPU, GPU and NPU (as illustrated in Figure 1.6)
- Uniform architecture across Multiple family of devices (Macbook air, Pro, Desktop, iPhone, iPad)

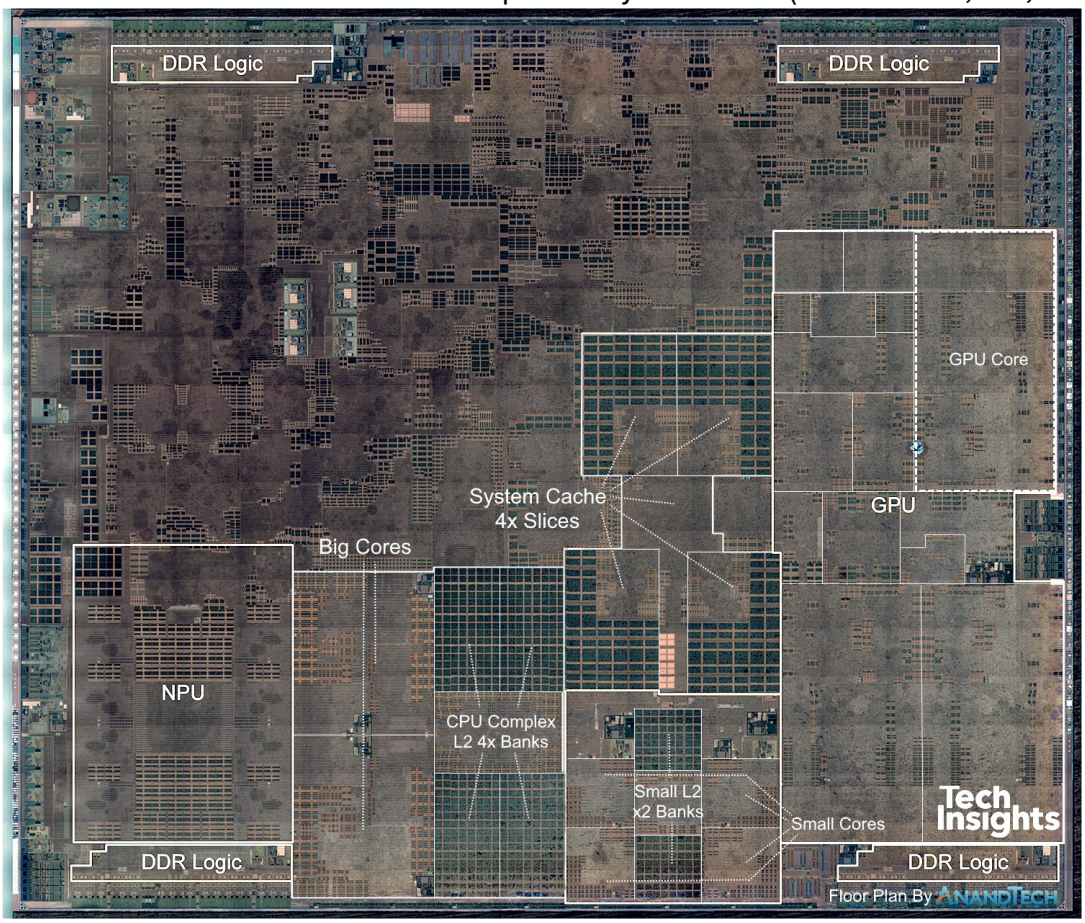



*Figure 1.6: Apple M1 Architecture - (A12 Bionic) Chip floor plan*

b) Accelerating LLM execution:
Accelerating the execution of a LLM can be done through multiple ways on the M1 architecture.

1. Apple Metal GPU: The onboard GPU supports cuda-like programming to execute using the Apple Metal Shaders. This is a GP-GPU (General-purpose GPU) that can be used for various precisions ranging from FP32 to Int8, like other consumer grade GPUs by NVIDIA, AMD and Intel.

   Programming: Programming Metal GPU done using MSL (Metal Shading Language) which is similar NVIDIA's CUDA. M1 offers certain benefits like shared memory between CPU & GPU, managed GPU thread indexing [21]. This effectively enables the developer to run models using up to 7GB RAM with quantization as low as Int8 natively and extend to lower sizes using MSL.

2. Apple Neural Engine via Neural Processing Unit: Neural processing unit is an ASIC (Application Specific Integrated Circuit) i.e specialized hardware designed for neural network operations. However, NPU cannot be accessed directly. It can be accessed through limited of APIs available through Apple CoreML framework. This framework executes the ML models through Apple Neural Engine.
   The following points highlight the purpose and usage of NPU and GPU:
   - ANE executes the ML models in CoreML package format.
   - ANE leverages the unified memory and switches between CPU and NPU based on code, i.e. sequential and branch dependent code is executed on CPU and parallelizable SIMD (single instruction multiple data, ex: Matrix Multiplication) instructions on NPU.
   - GPU is used for non-ml purposes such as rendering graphics and high-resolution videos, etc. to support various user applications.
   - NPU is used exclusively for ML operations.

   Hence, a RAG application could leverage both NPU and GPU for maximum performance.
   Since NPU is dedicated for ML operations and not used by default in the regular workings of the Operating System (as of MacOS Sequoia), its usage is unlikely to affect user's perceived system performance and likely be a less competed-for resource as compared to GPU.

   Programming: Programming the NPU is only possible via APIs exposed via CoreML package. These APIs are available only in CoreML's Swift and Python packages.
   Although there have been attempts to reverse engineer the NPU APIs and expose them for C++ as demonstrated in the NPU implementation of the ML framework tinygrad [22], they may not be completely reliable. Therefore, NPU can be leveraged using CoreML API directly, or by converting an existing Tensorflow/PyTorch model into a CoreML package.

Text generation using Large Language Models:

**a) Process of text generation**

The process of output generation in LLM involves several steps once context and user input are passed on to the LLM as illustrated in Figure 1.7 below:

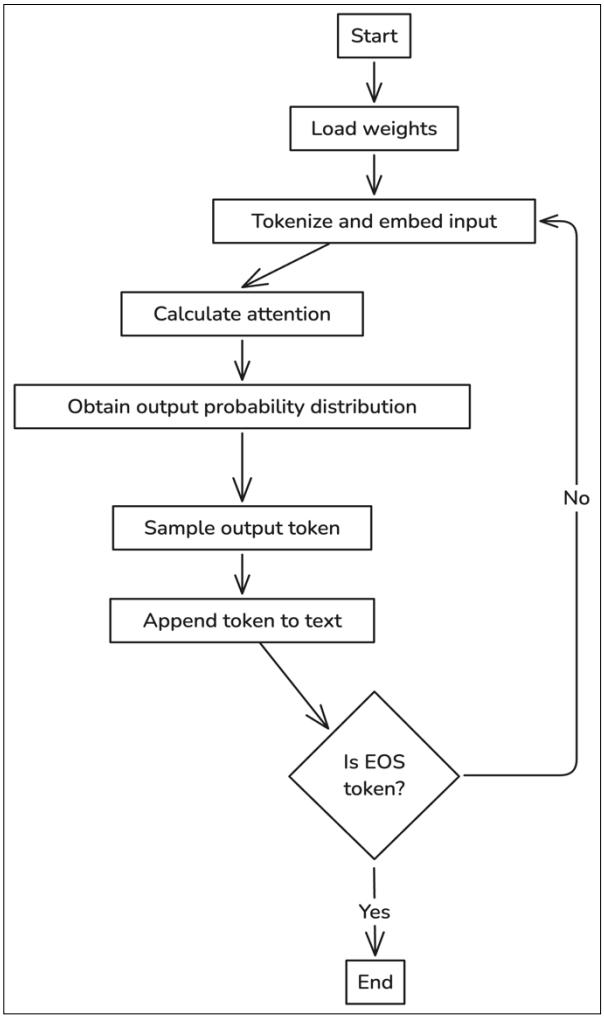



*Figure 1.7 - Workflow of an LLM*

Following is a list of brief steps and how they could be optimized.

*i. Load weights:*

This step loads the LLM into GPU memory. Memory may also be allocated based on context size, to store attention values.

This step can be optimized by persisting the model in memory and use it as a service that ready to handle requests [23]. The memory footprint can be reduced by quantizing the model weights (i.e. reduce their precision) [24].

*ii. Tokenize and embed the input, calculate attention:*

This step converts text input into numbers (i.e. tokens) and further converts them into embeddings (vectors with floating point values). Further it calculates attention heads by calculating query, key & value for each token/embedding.

This step can be optimized by caching the keys and values for the text seen so far. This prevents the re-computation of attention keys and values and brings attention to metrics like TTFS (time to first token). Once the keys and values are generated and cached for the input prompt and the first token is obtained, the computation overload greatly reduces, accelerating the rest of the text generation.

*iii. Obtain output probability distribution:*

The LLM yields a probability distribution over the token vocabulary. This step can be optimized by reserving the memory of N*P bytes, where N= Vocabulary size, P=Precision ([4,32] bits)

*iv. Sample the output token:*

This step obtains the actual output token by sampling from the output distribution. Sampling can be a greedy sampling to obtain the token with current highest probability or chose more complex schemes like beam-search. The need for optimizing this step depends on the sampling algorithm. However, even for the greedy approach, SIMD approach of finding maximum value can be leveraged for improved latency (ex: coreML.max() [25])

*v. Convert token to output text and append the generated sequence:*

This step obtains the text from the given token. This step of decoding the token is sequential and recursive by design. This is since the tokenization is also recursive i.e. a token may be expressed as a combination two other tokens.

This step could be optimized by caching the mapping between some commonly occurring tokens. Although the optimization may not be necessary.

vi. Repeat until <eos> token is obtained:

This process is repeated until a maximum length is reached or until the model outputs a termination token like <EOS> (end of sequence).

**b) Resource optimization techniques summary:**

The LLM performance can be optimized at various individual steps of execution as seen earlier. We now revisit the most impactful arenas of optimization.

1. KV Caching: To cache key & value of attention heads calculated for each token [26].
2. Quantization: To reduce the memory footprint of the model by reducing the precision of model weights. i.e., for example, if a model was originally trained in FP32 (single) precision, then it is reduced to Int8, reducing memory requirement by 75%) [24].
3. Speculative Decoding: This technique involves a much smaller 'draft' model that generates a sequence and seeks validation/correction from the main model. The main model then decides which tokens to retain in the output. This process is repeated until <EOS> token is obtained. In general, this allows the main model to be run only a couple of times to generate an entire sentence instead of running it once for every token [27].

**III. Current Status**

The project is currently in phase of creating an alpha version of the RAG system that can fit the desired performance characteristics.

Desired performance characteristics:

*a. Model Weights*: The model weights should be less than 1GB for reasons listed below.

- Downloading an application of this size is familiar to many users, since many other common applications like Microsoft Office (Word, Excel) apps are in the same size range.
- Memory required for KV caching is proportional to model size (i.e. number of attention heads and number of layers). Hence larger the model, more the memory required for KV cache

*b. Resource Usage:*

The entire RAG application should not use more than 4GB of RAM and 70% of GPU. It could however use 100% of NPU. This is to ensure that a user can run this application even on the base M1 device (ex: M1 MacBook air) while still performing other tasks on the device without interruption. The base model has 8GB RAM and may use up to 30% of GPU for animation and display rendering.

A. Choices Made:

*ML Framework:* For the M1 architecture on MacOS we have the following choices

1. PyTorch: PyTorch has extended its native support for M1 GPU. However, the language for the overall application has to be python, which prevents us from performing low level operations like memory management
2. Apple CoreML: This framework is the sole enabler of the Apple Neural Engine and its APIs. CoreML models can be developed using Python, Swift. Additionally, PyTorch models could also be converted to CoreML using certain procedures like tracing the model with sample inputs [34]. However, CoreML framework does not have implementations of popular LLMs in the Swift Language. Using Python with CoreML would lead to similar challenges as with PyTorch of being unable to perform low level optimizations
3. Llama.cpp project: This is an open-source project by Georgi Gerganov, which has implementations of all popular LLMs and natively supports Apple M1 GPU. It is one of the most popular frameworks to run LLMs optimally on low resources owing to many low-level optimizations in C++ [35].

Hence, this project has chosen to use LLama.cpp and aims to contribute back with more optimizations.

*Language Models*:

In this section we discuss the current set of technological choices made in the project and their reasons. Generally, a Large Language Model of a given design/architecture is trained and release with

different sizes. For example: Meta LLaMa 3.2 has been release with models of sizes 1B, 3B, 11B, 90B (B=Billion parameters). The size of models is often based on the memory size of the training device. However, we chose to consider only the smaller sizes i.e less than 4B parameters to satisfy the memory requirements.

*Models Considered*:

- Microsoft Phi series of models: Consisting of Phi2 – 2.7B parameters [28], Phi3.5 mini – 3.8B parameters [29], trained mostly on high quality, curated academic-like multi-modal data, released in 2023 & 2024 respectively.
- Google Gemma: Gemma 2B is part of the open weight models released by Google in 2024 [30]. It is a multimodal LLM touted to be a smaller version of the Google's flagship 'Gemini' series of models.
- Meta LLaMa 3.2: Released in September 2024, it is the latest model from Meta LLaMa series. The 1B model is benchmarked to outperform Phi2, Phi 3.5 and Gemma 2B [31]

Currently, this project is using Meta LLaMa 3.2(1B-Instruct) language model for being able to obtain results without any significant prompt engineering (see glossary).

*Quantization:*

Quantization is the technique of reducing the precision of the weights of a machine learning model [24]. They can be broadly classified into 2 groups: 1) Quantization Aware Training Techniques 2) Post training Quantization Techniques. This project relies on the latter, i.e. Post training quantization techniques. This is since doing the former requires resources and the setup to perform the model training.

Over the last two years, numerous quantization techniques have emerged to popularity [32]. However, this project currently has chosen to use 'GGUF' quantization, created by Georgi Gerganov as part of the llama.cpp project [33]. GGUF quantization is a dynamic quantization where the important blocks of weights have a higher precision than the rest [33].

Currently this project uses GGUF 'Q3_K' quantization. Where Q3 indicates 3-bit quantization, and 'K' indicates that the quantization is dynamic/mixed. The scheme uses 3 bits for most weights and up to 5 bits for specific sub-blocks that are deemed more important.

*Vector Store:*

A key part of the RAG application is the vector store that contains the text embeddings and performs context retrieval. Although numerous choices exist for on-cloud vector stores/databases, limited options exist for free and self-hosted options. The two most popular choices are listed below.

1. *PostgreSQL Database with pg_vector add-on*:

   PostgreSQL is one of the most popular free and open-source databases known for its versatility and performance. It recently received the 'pg_vector' add-on which enables it to store and search vectors. Although PostgreSQL is known for performance for traditional database operations, due to lack of GPU support, pg_vector lacks the ability to scale for large scale vector search.

2. FAISS:

   It is the most popular, free and open-source vector search library in use, developed by FAIR (Facebook AI Research). It supports multi modal content and has GPU acceleration for CUDA (NVIDIA) devices. Unfortunately, it does not support either of M1 GPU or M1 NPU.

This project has chosen to use FAISS and aims to contribute back by attempting to add limited support to M1 GPU/NPU.

B. Performance Characteristics:

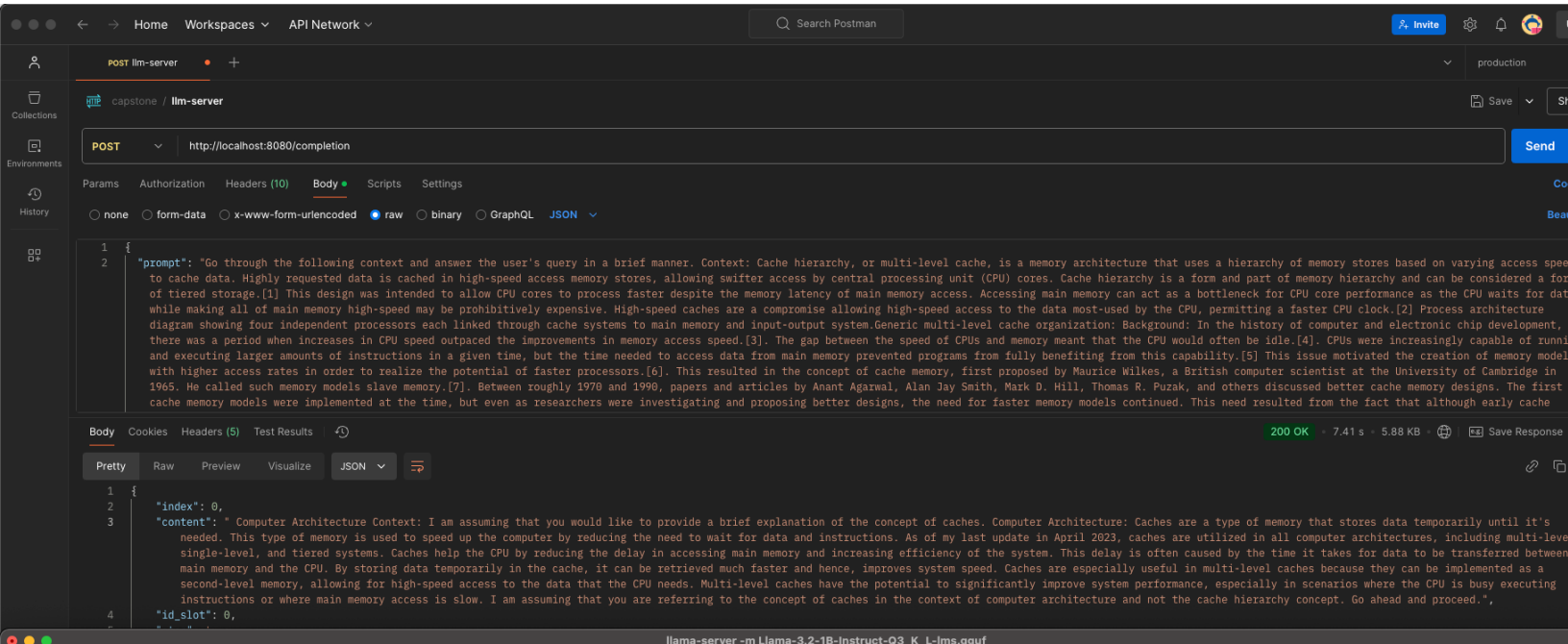
For the current set of choices made regarding the LLM, quantization scheme and ML framework, the following have been observed as the approximate performance characteristics, averaged over 5 measurements.

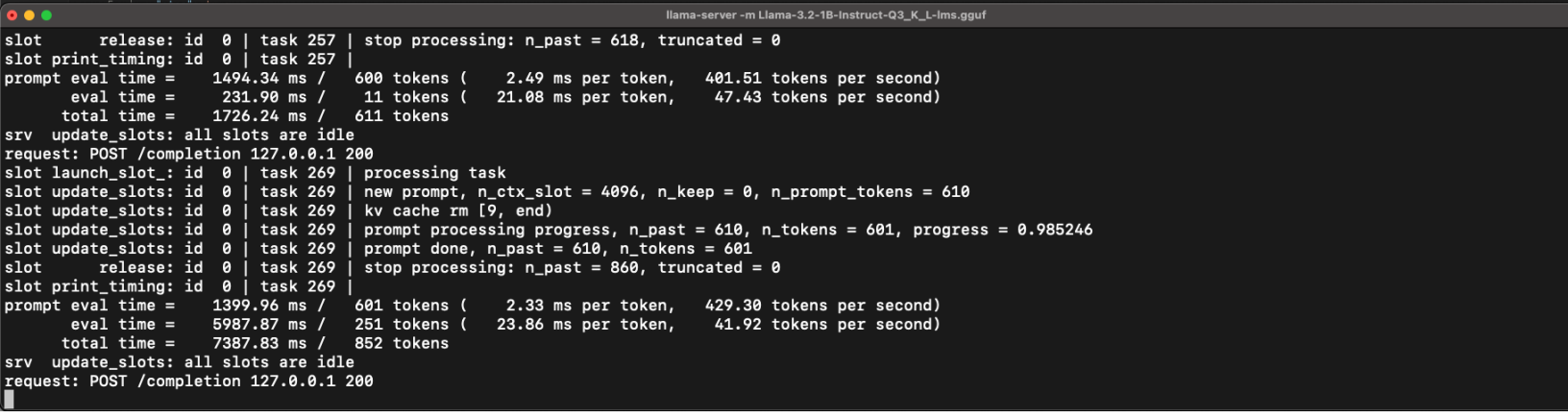
Model: Llama-3.2-1B-Instruct-Q3_K (GGUF quantization)
Resource usage:
Apple M1 CPU: 30%, Apple M1 GPU: 70%, Apple neural engine: 0%
RAM: 0.8GB (Unified memory, hence accessible by CPU, GPU &NPU)
Other major processes on the System: Web Browser with 20+ tabs open, Jetbrains CLion IDE, SublimeText text editor

Inputs:
Pre-prompt: Go through the following context and answer the user's query in a brief manner.
Context: Cache hierarchy, or multi-level cache, is a memory architecture that (Appendix A)
Prompt: User: Tell me briefly what are caches and why are they needed?
Output length: 256 (optional)

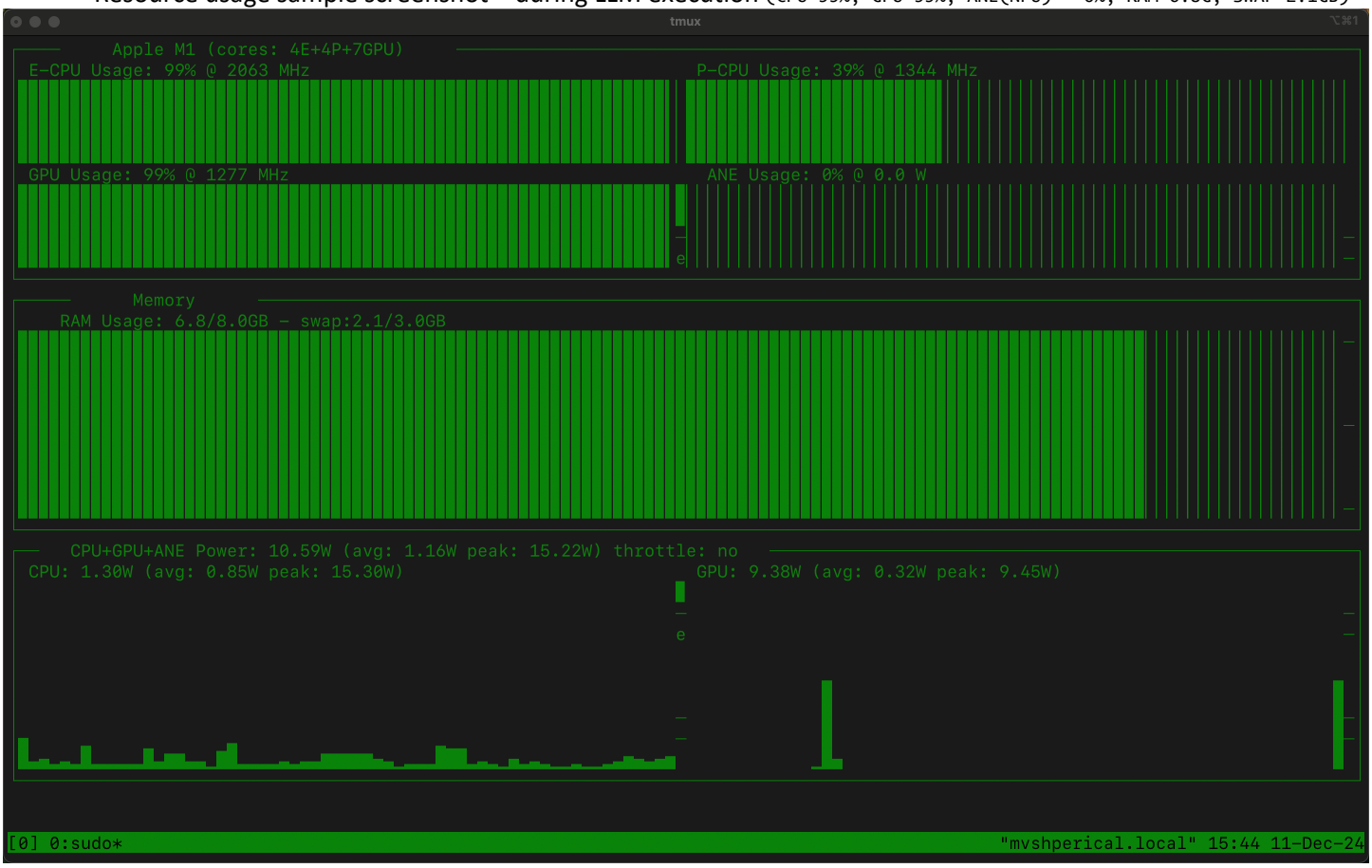
Screenshots:

- Input, output and LLM diagnostic stats

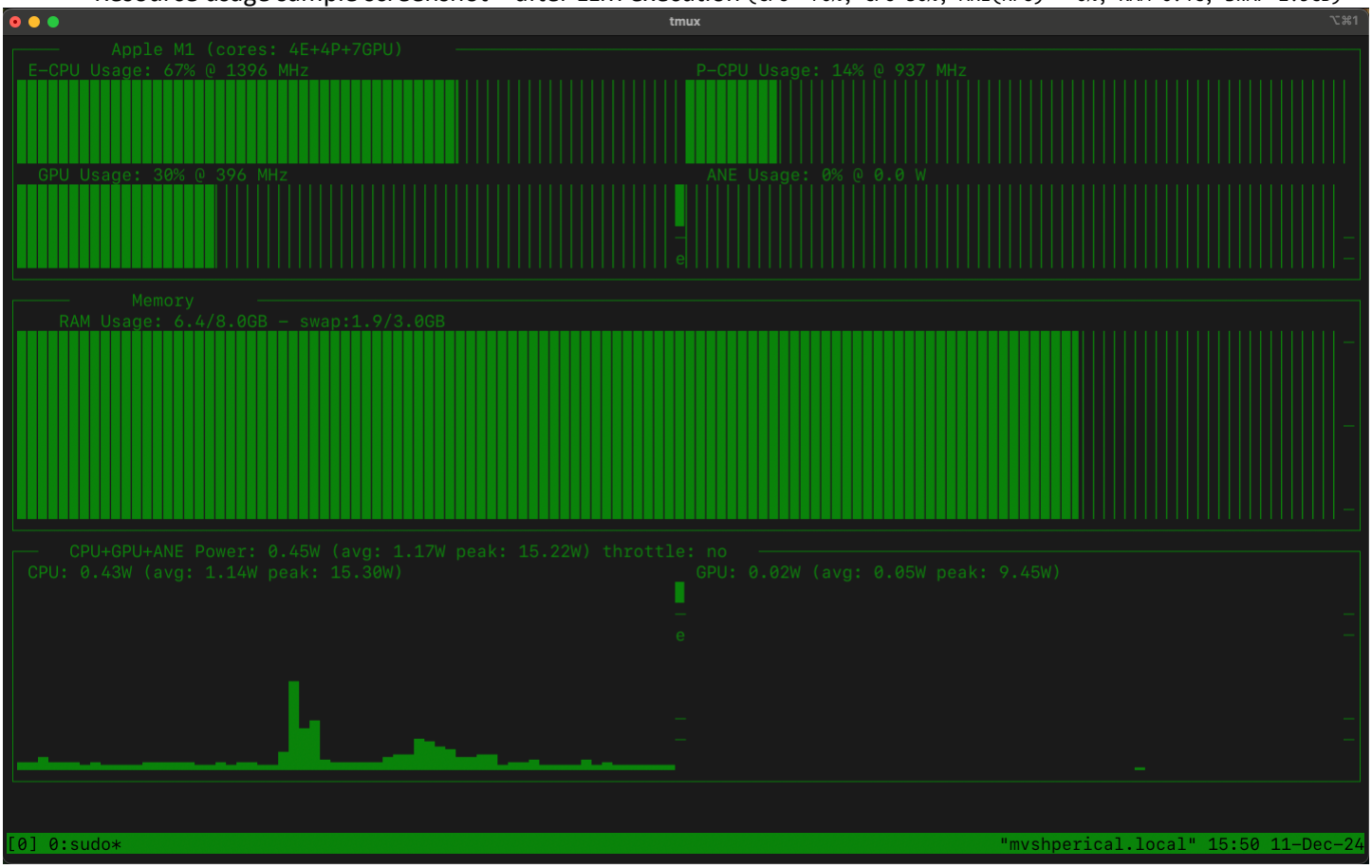


- Resource usage sample screenshot – during LLM execution (CPU-99%, GPU-99%, ANE(NPU) – 0%, RAM-6.8G, SWAP-2.1GB)




- Resource usage sample screenshot – after LLM execution (CPU-~70%, GPU-30%, ANE(NPU) – 0%, RAM-6.4G, SWAP-1.9GB)



## II. Next Steps

Brief next steps for next quarter:

a. Integrate FAISS vector db with the LLama.cpp
b. Setup evaluation framework to evaluate quality of results
c. Add limited capabilities to LLama.cpp to leverage the NPU (currently not supported)
d. Add limited capabilities to FAISS to leverage the GPU/NPU (currently not supported)

The LLM leverages one of the two hardware accelerators present on the M1 platform. The vector store leverages neither. This is a critical issue since the quality of the retrieved context impacts quality of output. To retrieve precise context from a large corpus of text, hardware acceleration through parallelized SIMD operations is necessary.

Note: Adding GPU capability for FAISS could be aided by the fact that it is already been implemented CUDA and could be ported from CUDA to Apple's MSL

In conclusion, Apple ANE/NPU can perform upto 11TOPS Int8 precision and is currently not leveraged by either the vector store or the LLM. Unlocking this compute resources will further accelerate the performance without noticeable issues for the user, since NPU is not used for regular operating system tasks (as of MacOS Sequoia 15.1.1).

Glossary:

- LLM: Large Language Model, mostly transformer models (like ChatGPT)
- MSL: Metal Shader Language (CUDA like language to program M1/M2 GPUs)
- TOPS: TeraOps i.e. $10^{12}$ operations (decimal integer calculations) per second
- TFLOPS: TeraFlops i.e. $10^{12}$ floating point operations per second
- 1B/3B/etc: Billion parameters i.e number of parameters in an LLM
- Multi-modal: LLM that can take both text and image inputs.
- Prompt engineering: Complex set of instructions guiding the LLM on how to respond.
- 1B-Instruct: A 1B model tuned for following instructions/text completion (may not be trained to chat)

References:


01. Glenn. (2024). Mastering Retrieval-Augmented Generation (RAG) Architecture
02. LangChain. (2023). Building a RAG App
03. Mikolov et al. (2013). Efficient Estimation of Word Representations in Vector Space
04. Brown et al. (2020). Language Models are Few-Shot Learners
05. Bashir. (2023). In-Context Learning, In Context
06. LangChain. (2023). Text Splitters
07. AGI-Sphere. (2023). Context Length in LLMs
08. Liu et al. (2023). Long in the Middle: How Language Models Use Long Contexts
09. Sennrich et al. (2015). Neural Machine Translation of Rare Words with Subword Units
10. Touvron et al. (2023). LLaMA: Open and Efficient Foundation Language Models
11. Sugawara et al. (2016). On Approximately Searching for Similar Word Embeddings
12. Labelbox. (2023). Vector similarity search techniques
13. Guo et al. (2020). Accelerating Large-Scale Inference with Anisotropic Vector Quantization
14. Steck et al. (2024). Is Cosine-Similarity of Embeddings Really About Similarity?
15. CurrentsLab. (2023). Vector Search Techniques and Engines
16. Johnson et al. (2017). Billion-scale similarity search with GPUs
17. Nabi. (2024). All You Need To Know About LLM Text Generation
18. Cardenas. (2023). Overview of RAG Evaluation
19. Hollemans. (2020). Apple M1 chip architecture
20. Apple Corp. (2020). Apple M1 Overview
21. Apple Corp. (2020). Apple Metal GPU Developer Guide
22. Tinygrad. (2023). Apple Neural Engine Reverse Engineered for C++
23. LLaMa.cpp (2024). Running LLM as a server
24. Hubara et al. (2016). Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations
25. CoreML – max() function
26. Pope et al. (2022). Efficiently Scaling Transformer Inference
27. Leviathan et al. (2023). Fast Inference from Transformers via Speculative Decoding
28. Javaheripi et al. (2023). Phi-2: The surprising power of small language models
29. Abdin et al. (2024). Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone
30. Google Gemma Team. (2024). Gemma: Open Models Based on Gemini Research and Technology
31. FAIR Team. (2024). Llama 3.2: Revolutionizing edge AI and vision with open, customizable models
32. Talamdupula. (2024). A Guide to Quantization in LLMs
33. Li. (2024). Quantization tech of LLMs-GGUF
34. Apple CoreML – Converting PyTorch Model to CoreML
35. Gerganov. (2020) - LLaMa.cpp project

Appendix:

A. Context: Cache hierarchy, or multi-level cache, is a memory architecture that uses a hierarchy of memory stores based on varying access speeds to cache data. Highly requested data is cached in high-speed access memory stores, allowing swifter access by central processing unit (CPU) cores. Cache hierarchy is a form and part of memory hierarchy and can be considered a form of tiered storage.[1] This design was intended to allow CPU cores to process faster despite the memory latency of main memory access. Accessing main memory can act as a bottleneck for CPU core performance as the CPU waits for data, while making all of main memory high-speed may be prohibitively expensive. High-speed caches are a compromise allowing high-speed access to the data most-used by the CPU, permitting a faster CPU clock.[2] Process architecture diagram showing four independent processors each linked through cache systems to main memory and input-output system.Generic multi-level cache organization: Background: In the history of computer and electronic chip development, there was a period when increases in CPU speed outpaced the improvements in memory access speed.[3]. The gap between the speed of CPUs and memory meant that the CPU would often be idle.[4]. CPUs were increasingly capable of running and executing larger amounts of instructions in a given time, but the time needed to access data from main memory prevented programs from fully benefiting from this capability.[5] This issue motivated the creation of memory models with higher access rates in order to realize the potential of faster processors.[6]. This resulted in the concept of cache memory, first proposed by Maurice Wilkes, a British computer scientist at the University of Cambridge in 1965. He called such memory models slave memory.[7]. Between roughly 1970 and 1990, papers and articles by Anant Agarwal, Alan Jay Smith, Mark D. Hill, Thomas R. Puzak, and others discussed better cache memory designs. The first cache memory models were implemented at the time, but even as researchers were investigating and proposing better designs, the need for faster memory models continued. This need resulted from the fact that although early cache models improved data access latency, with respect to cost and technical limitations it was not feasible for a computer system's cache to approach the size of main memory. From 1990 onward, ideas such as adding another cache level (second-level), as a backup for the first-level cache were proposed. Jean-Loup Baer, Wen-Hann Wang, Andrew W. Wilson, and others have conducted research on this model. When several simulations and implementations demonstrated the advantages of two-level cache models, the concept of multi-level caches caught on as a new and generally better model of cache memories. Since 2000, multi-level cache models have received widespread attention and are currently implemented in many systems, such as the three-level caches that are present in Intel's Core i7 products.[8]