Deployment and Analysis of TinyLlama 1.1B on Nvidia Jetson Nano: A Study of Query Workloads in Resource-Constrained Environments

Submitted By:
Aayush Kapur

Program:
Master's in Computer Science

Department:
Department of Computer Science, McGill University

Advisor:
Prof. Oana Balmau

August, 2024

Table of Contents

Abstract

Edge devices are characterized by constraints in computational power, memory, and other resources, coupled with their small form factors, limited power supply, and heat dissipation capabilities. With the advancement of large language models (LLMs) and inference technologies, the deployment of AI-powered agents has become increasingly feasible. This raises a critical question: can edge devices effectively support the machine learning workloads associated with LLM-powered agents? This project aims to explore the feasibility and establish a baseline for deploying such agents on edge devices. Specifically, this project involved analyzing the resource usage of query workloads associated with an LLM agent deployed on an Nvidia Jetson Nano board for which a small language model, TinyLlama 1.1B, was deployed using the Llama.cpp inference runtime on Jetson Nano 4GB module. The query workload was categorized into five types—simple, complex, conversational, contextual, and task-oriented—and system usage was measured to analyze the resulting trends.

Introduction

Introduction to the Problem

Edge devices present a compelling use case for LLM agents due to their ability to perform inference locally, which is particularly valuable in applications where data privacy, connectivity limitations, or ecosystem boundaries are significant concerns. The primary question addressed in this project is whether these devices can effectively manage the workload demands of deploying locally hosted small language models, given their inherent memory and hardware constraints.

The analysis focuses on the Nvidia Jetson Nano developer kit board, selected as a representative edge device. This choice establishes specific limitations in terms of memory capacity and software compatibility, as the Jetson Nano is an older model compared to the newer Jetson Orin Nano, which is currently supported by Nvidia.

Nvidia Jetson Nano Developer Kit

The Jetson Nano is a compact computing module designed for AI and machine learning tasks at the edge built for deployment in production environments. It is part of the broader Nvidia Jetson platform, which includes a range of modules and developer kits aimed at integrating AI capabilities into embedded systems. The Developer Kit, which we used, is intended for prototyping and development purposes. It consists of the Jetson Nano module attached to a carrier board, which provides essential I/O ports such as HDMI, USB, Ethernet, GPIO pins, a micro-USB, and a barrel jack connector. *("NVIDIA Jetson Nano")*

Jetson Nano Specifications

For this project, we employed the Jetson Nano Developer Kit 4GB variant, which is equipped with a 128-core NVIDIA Maxwell GPU, a Quad-core ARM Cortex-A57 CPU, and 4GB of 64-Bit LPDDR4 RAM.

Problem Objectives

1. **Deployment Feasibility**:

- Can a small LLM be successfully deployed on the Nvidia Jetson Nano Developer Kit (4GB edition) for text-based tasks without rendering the system unusable for other processes? Additionally, does the system have sufficient capacity to incorporate wrapper files for local agents while executing queries through the LLM?

2. **Workload Analysis**:
    - How do various types of query workloads associated with an LLM agent perform on edge devices? What trends and patterns are observed in these performances, and what underlying factors contribute to these outcomes?
    - Are there specific limitations or weaknesses in deploying LLMs on edge devices for particular workloads?

Background

Small Language Models

Small language models (SLMs) are streamlined versions of large language models, typically containing a few hundred million to a few billion parameters. Although they lack the extensive parameter count of their larger counterparts, SLMs offer a significant advantage in their minimal resource demands. These models are particularly well-suited for edge devices, as they require lower computational power, consume less memory, and provide faster inference times. They will become more prominent in modern AI applications because of their ability to be deployed in resource constrained environments. *(Li et al.)*

TinyLlama

Tinyllama (Zhang et al.) is a decoder-only transformer based on the architecture of the Llama series of models. The TinyLlama project involved the pretraining of a 1.1 billion parameter Llama model on 3 trillion tokens over 90 days, utilizing 16 A100-40G GPUs. (going against the Chinchilla scaling laws) (Hoffmann et al.). It uses Rotary Positional Embedding (RoPE) (Su et al.) for capturing positional information. Further, instead of normalization happening after each transformer layer, following Llama's architecture, the inputs are normalized before each transformer layer using Root Mean Square Layer Normalization (RMSNorm: bzhangGo). RMSNorm shows 10-50% in efficiency without much decrease in performance (Wolfe). Similar to Llama, instead of ReLU activation function, SwiGLU activation function is used. Lastly, Tinyllama also uses Grouped-query attention like the Llama series. This is useful to reduce memory bandwidth and speed up inference as keys and values can be shared between multiple heads requiring less space than multi-head attention in memory. It has 32 heads for query attention and 4 groups for key-value heads.

For this study, we employed the TinyLlama-1.1B-Chat-v1.0-Q5_K_M.gguf model, a 5-bit GGUF format specifically fine-tuned for chat-based applications. This version of TinyLlama was fine-tuned using Hugging Face's Zephyr training recipe, on top of the TinyLlama-1.1B-intermediate-step-1431k-3T model. Initially, it was fine-tuned on a variant of the UltraChat dataset, which includes diverse synthetic dialogues generated by ChatGPT, and was further aligned using Hugging Faces's TRL's DPOTrainer on the openbmb/UltraFeedback dataset, containing 64k prompts and model completions ranked by GPT-4.

Rationale for Choosing TinyLlama

- Efficient Architecture LLaMA based architecture: small memory footprint
- Quantization Support
- Open-Source nature
- Decoder only model

LLM Agents

LLM agents (Weng) extend the capabilities of language models beyond simple text generation. They have the following components.

1. **Planning**: deals with subgoal decomposition and refinement. The LLMs serve as the planning module, determining the next actions based on the current state of the system, which is provided as part of the prompt. An intermediary wrapper captures the system state at each stage, feeding it back into the LLM to generate subsequent actions.

2. **Memory**: The state recorded during interactions, along with external knowledge, is stored in the memory module. Long term memory can be implemented by storing external information in databases. Short term memory means keeping a track of the goals of the current interaction (using vector databases to store embeddings of user utterances is an example)

3. **Tool Use**: LLM agents interact with external APIs to perform tasks.

JetPack SDK Overview

JetPack is a key component of the software stack provided by Nvidia for their edge AI devices ("JetPack SDK"). The JetPack SDK includes:

1. **Jetson Linux**: A reference file system derived from Ubuntu, featuring a Board Support Package (BSP) that includes the bootloader, Linux kernel, Ubuntu desktop environment, and NVIDIA drivers.

2. **AI Stack**:

- The AI stack is CUDA-accelerated and includes tools such as NVIDIA TensorRT and cuDNN for accelerated AI inference, CUDA for general-purpose computing, VPI for computer vision and image processing, Jetson Linux APIs for multimedia acceleration, and libArgus and V4L2 for camera processing. The availability of these packages depends on the JetPack version supported by the Jetson device. Compatibility and support timelines determine the specific packages, OS versions, libraries, and dependencies available for each device. For example, the Jetson Orin Nano is expected to be supported until 2030, minimizing the risk of obsolescence-related issues.

3. **Developer Tools**:
    - **Development and Debugging**: Tools include Nsight Eclipse Edition, CUDA-GDB, and CUDA-MEMCHECK.
    - **Profiling and Optimization**: Tools such as Nsight Systems, nvprof, Visual Profiler, and Nsight Graphics facilitate performance tuning.

Deploying the Model

Several approaches were considered for LLM inference on the Jetson Nano module.

1. **Using Llama.cpp**: This was the preferred approach, as the llama.cpp project aims to "enable LLM inference with minimal setup and state-of-the-art performance on a wide variety of hardware." Our objective was to deploy TinyLlama 1.1B with minimal dependencies and the lowest possible memory overhead, making llama.cpp the optimal choice (ggerganov).
2. **Using TensorRT**: While this approach is expected to provide better performance on Nvidia edge GPUs, it requires exporting the model to ONNX format before running inference, adding complexity to the deployment process.
3. **Using PyTorch Directly**: Running queries directly in PyTorch was considered, but the significant memory overhead due to the unified memory architecture, along with the need to create a wrapper for managing model input, were major drawbacks.

Implementation

Following are steps for implementation of our project categorized by phases of the project. We have also listed the difficulties faced during deployment and warnings based on our failures in the appendix.

Board Setup

1. **Flashing the Board**:
   The Jetson Nano Developer Kit image, including the JetPack SDK (which provides the CUDA toolkit and other essential AI tools), was flashed onto the board. After a "quick format" of the SD card using the SD card formatter tool ("SD Memory Card Formatter"), the image was loaded onto the microSD card using the Etcher tool ("balenaEtcher"). Initial setup was completed with peripherals, including a display, directly attached to the board. Subsequent usage was done in headless mode, with the board connected directly to a PC.

2. **Establishing Communication with the Board**:
   - **SSH and Internet Setup**: Internet access was required for installing and updating packages on the board. This was facilitated by an external Wi-Fi adapter, for which the driver (rtl88x2bu, cilynx) was configured. All development work was conducted using SSH over micro-USB, utilizing the remote SSH extension in VS Code and WinSCP for file transfer.

   - **Connectivity Details**: Over IPv4, only one board can be connected per PC, while IPv6 allows multiple boards to be connected to the same PC. The IP address for SSH access is "192.168.55.1".

Inference Runtime Installation

3. **Building and Installing the Inference Runtime**
   To deploy the TinyLlama model on the Jetson Nano board, we utilized llama.cpp. However, due to the JetPack 4.6 software stack, native support for llama.cpp was not available. This required us to build llama.cpp from source directly on the device. Further, we also needed to build gcc 8.5 on device to compile llama.cpp because of dependency issues. More details can be found in the appendix.

Resource Usage Measurement

4. Creating the Workload for Inference

The objective was to investigate the suitability of deploying a small language model-powered agent on the Jetson module, focusing on the system's performance rather than the quality of the model's responses. Specifically, we aimed to analyze how the system's performance would vary based on the types of queries handled by the language model. By aligning our queries with those that an agent might typically encounter, we sought to identify trends in resource usage.

To comprehensively evaluate the system, we used a variety of queries that simulate real-world scenarios an agent might face. This approach enabled us to assess the model's performance across different types of tasks.

Categories of Queries:

- **Simple Queries**: Short and straightforward questions or commands.
- **Complex Queries**: Multi-sentence requests that require understanding and generating more detailed responses.
- **Conversational Queries**: Dialogue-based queries that simulate a back-and-forth conversation.
- **Task-Oriented Queries**: Specific tasks or commands, such as calculations or generating lists.
- **Contextual Queries**: Queries that require responses based on the provided context.

5. Curating the Dataset for Experiments

To effectively run our experiments, we curated a dataset comprising 100 queries for each of the five categories. These queries were selected from established NLP datasets commonly used for large language models (LLMs) and aligned with the specific category types.

Simple Queries:

- **Dataset**: WikiQA, a question-answering corpus from Microsoft (*Wiki_qa*).
    - Example Queries:

        - "How does interlibrary loan work?"
        - "Where is money made in the United States?"
        - "How does a dim sum restaurant work?"

Complex Queries:

- **Dataset**: SQuAD v2.0 (Dev set), a benchmark dataset for machine comprehension of text (*The Stanford Question Answering Dataset*).
    - Example Queries:
        - "What does the private education student financial assistance help current high school students who are turned away do?"
        - "Whose infected corpse was one of the ones catapulted over the walls of Kaffa by the Mongol army?"

Conversational Queries:

- **Dataset**: Friends Dataset, containing speech-based dialogue from the Friends TV sitcom, extracted from the SocialNLP EmotionX 2019 challenge (Michelle Li).
    - Example Queries:
        - "Oh, that's right. It's your first day! So, are you psyched to fight fake crime with your robot sidekick?"
        - "Whoa!! Now look, don't be just blurting stuff out. I want you to really think about your answers, okay?"

Task-Oriented Queries:

- **Dataset**: MultiWOZ 2.2, a multi-domain Wizard-of-Oz dataset for task-oriented dialogue systems.We retrieved only the user utterances from the conversation per task and combined them for the model to infer and give the steps in accomplishment of the task (salesforce).
    - Example Query:
        - "This is a bot helping users to find a restaurant and find an attraction. Given the dialog context, please generate a relevant system response for the user: <USER> Hi, could you help me with some information on a particular attraction? <USER> It is called Nusha. Can you tell me a little bit about it? <USER> Thank you. Please get me information on a

particular restaurant called Thanh Binh. <USER> Thanks, please make a reservation there for 6 people at 17:15 on Saturday."

Contextual Queries:

- **Dataset**: Validation set from Question Answering in Context (QuAC), a dataset for modeling, understanding, and participating in information-seeking dialogue (*Question Answering in Context*).
    - Example Query Structure:
        - ::context:: (text provided) ::question:: "Were they ever in any other TV shows or movies?"

6. Measuring System Resource Usage During Inference

Our measuring script loaded all queries in the dataset, appended an instruction based on the category, and invoked the *llama.cpp* command to execute each query (the appended instruction per category can be found in the appendix). The output was recorded in a text file identified by a unique ID, comprising the query category and number. To ensure feasible run durations and prevent overheating the board, we imposed limits on output length for each category.

For each query submission, a new process was spawned, and the associated resource utilization was recorded for that process ID. The results of query execution were captured by connecting to the process's output and error pipes.

Tools Used for Measuring Resource Utilization:

- **Psutil**:
    - cpu_percent: Recorded the CPU utilization of the process as a percentage.
    - memory_percent: Measured the process memory usage as a percentage of the total physical system memory.

- **Jetson Stats**:
    - jetson.stats: Provided a simplified version of tegrastats, allowing for easy logging of the NVIDIA Jetson status, including metrics such as CPU, RAM, fan speed,

temperature, GPU usage, and uptime.

- jetson.processes: Returned a list of all processes running on the GPU, including CPU usage, GPU memory usage, and CPU memory usage ("Jetson-Stats").

Command Format for Query Invocation: Each query was executed using the following command format:

/path/to/compiled/main/bin-for-llama.cpp -m /model/gguf-file-location -p "Query text" -n <tokens-expected>

7. Processing Raw Output and Plotting Results

The frequency of data recording varied depending on the experiment, ranging from 1 centisecond to 1 second. The processing scripts were designed to clean the raw data files, which contained the resulting query outputs, llama.cpp output readings, and system resource usage readings. All relevant data points across the queries were consolidated into a single processed dataframe. Finally, Plotly was used to generate the graphs for analysis, providing a visual representation of the system's performance metrics.

Experiments

1. **Category-Wise Query Submission**:
    - **Process**: We submitted 100 queries from each category sequentially. Every 0.5 seconds, the following metrics were recorded:
        1. **CPU and GPU Memory Usage, CPU Utilization**: These metrics were recorded (across all four cores: CPU utilization) for the query process executed through llama.cpp for TinyLlama.
        2. **System-Wide GPU Utilization**: GPU utilization was monitored while the query execution process was in progress. During data processing, the GPU utilization values used for plots were those that coincided with the same time periods for which per-process readings existed.
        3. **Per Query Metrics**:
            1. **Time to First Token**: The duration before the first token appeared.
            2. **Time Between Tokens**: The time gap observed between the generation of each successive token after the first token had appeared.

2. **Granular Query Analysis**:
    - **Process**: We calculated the median length of prompt tokens for each query category and selected one representative query from each category that closely matched the median token length. For each representative query, the same measurements as above were recorded every centisecond (1 centisecond equals 1/100th of a second, corresponding to 100 ticks per second on Ubuntu systems). Recordings were made after a warm-up phase, as discrepancies were found in the CPU usage of complex queries when measurements were taken without warming up. The initial four readings for other queries in each workload were discarded to ensure consistency.

Following queries were picked:

Category	query_id	prompt_tokens	Category_prompt_tokens_median
complex	complex_13	31	31
contextual	contextual_40	438	438
conversational	conversational_10	43	43
simple	simple_00	29	29
task-oriented	task-oriented_04	190	190

Figure 1. Median prompt lengths across categories

Additional Granular Measurements for Special Scenarios:

For each of the following scenarios, a warm-up phase was included to ensure consistency in measurements.

1. **Task-Oriented Queries**:
   - **Process**: GPU memory usage was recorded every centisecond for four task-oriented queries. These queries were selected based on an increasing number of tokens in the prompt, with the output token count fixed at 50.

2. **Total Time Measurement**:
   - **Process**: The total time taken to complete two queries was recorded—one from the task-oriented workload and one from the contextual workload. These queries were selected to minimize the difference in the number of tokens in the prompt between them.

3. **Conversational Workload Query**:
   - **Process**: A single conversational query with a number of prompt tokens equal to the category median was selected. GPU memory usage was recorded over five runs of the same query, with the expected output length increasing incrementally in each run. The output token counts were set to ["10", "50", "120", "230", "400"].

Results and Analysis

By first tracing the journey of a token in relation to the architecture, we will establish the expected resource usages for encoder-decoder models and decoder-only models. This will be useful in analyzing the plots that follow.

Encoder-decoder models (better suited for information retrieval and question answering systems): The input prompt is tokenized then converted to word embeddings based on vocabulary for that particular model. They are augmented with positional information (order of words in sentences) to obtain positional encodings. This serves as the input on which encoder blocks act (these embeddings can be moved to the CPU memory to free up the accelerators for attention operations). Within each encoder layer, self attention operations are applied to capture the information about relation of tokens amongst each other in the sentence. This is done to capture how much other tokens influence a particular token and scaling the numeric representation of that token by the probability of that influence (Baolin Li et al.).

When we have multi-head self attention heads (MHSA), this means we have multiple sets of weights for keys and values working on the same set of tokens per layer. Eventually, the MHSA block will combine the features from all attention heads to form its output by multiplying it with another weight matrix (which is also trained with the rest of the system). All these matrices need to be in memory for computation. Since we want maximum computation to happen in hardware accelerators, we need to maintain the state of multiple sets of matrices in the accelerator memory to avoid transfer delays and idling (*LLM Inference Unveiled: Survey and Roofline Model Insights*, Alammar).

The combined representation of captured self attention information is normalized and combined back with positional embeddings (residuals to avoid vanishing and exploding gradients) before passing through a feedforward layer where this residual process is repeated with normalization. This is the processing within one block of an encoder. Now, different models can have multiple stacked blocks of encoder (layers). The final one gives the self-attention values for input tokens having captured the semantic context, positional ordering and relationship of tokens amongst each other in the input (Alammar).

The decoding phase in an encoder-decoder model takes advantage of the attention values from the final encoder layer (essentially the higher latent representation of the input data) while generating output at each successive decoder layer. This allows the decoder to keep track of significant words in the input.

Except for the fact above for the decoder blocks in encoder-decoder models, their decoding phase is similar to what is found in typical decoder only models. In decoder based LLMs, the output tokens are

generally generated auto-regressively based on the initial input sequence (prompt). They have a single unit for both encoding the input and generating the output. It uses masked self attention all the time in all its layers. Decoder based LLMs work in two phases: prefill and decode.

The first phase called prefill stage for input processing is highly parallel where inputs are embedded and encoded with multi head attention (following the steps - tokenisation - embeddings - positional embeddings - self attention values - normalization with residuals) which computes the keys and values with matrix multiplication utilizing the hardware accelerators. The multi-head attention allows the decoder to consider different parts of the sequence in different representational spaces. (Baolin Li et al.)

For getting the attention values, masked self-attention is used whereby the decoder only considers the previous and current token while calculating self attention This is because the decoder does not know the ungenerated sequence. It is repeated over a number of decoder layers till we get the final outputs. To prevent recomputing the key and values again for the input token while the output is being generated, we can store the KV values for the previous input tokens in the accelerator RAM through KV cache. This is a big sink of memory space during execution but necessary for faster inference speed.

Second phase is decode which involves the autoregressive generation of new tokens by the model with updates to KV cache per new token (which means KV cache needs to be in the accelerator memory during this phase). It continues generating output till it encounters <end of string> token or reaches max output length. This is the core part of the inference process.

To get texts out of values from the decoder layer, a linear layer is used to convert it to the dimension of model vocabulary which is given as input to softmax function to get probability distribution over the vocab tokens in turn making our output.

Resource usage contributors:

Model size: The number of parameters of an LLM is the approximate total number of weights and biases across all layers that are being used in that model. Thus, the number of weights and the precision in which they are stored will directly influence the space the model takes when loaded in the memory (Zhang et al.).

Attention Operation: As the input length becomes longer, the memory and computational costs associated with attention operation increase quickly because it exhibits quadratic computational complexity in the input length (Zhang et al.).

Decoding Approach: Decoding requires loading KV cache in memory on the accelerator. Though, it happens step by step per token but this cache needs to be in memory for each step and KV cache grows in size as well. This may lead to fragmented memory and irregular memory access patterns (Zhang et al.).

Analysis of plots across workloads

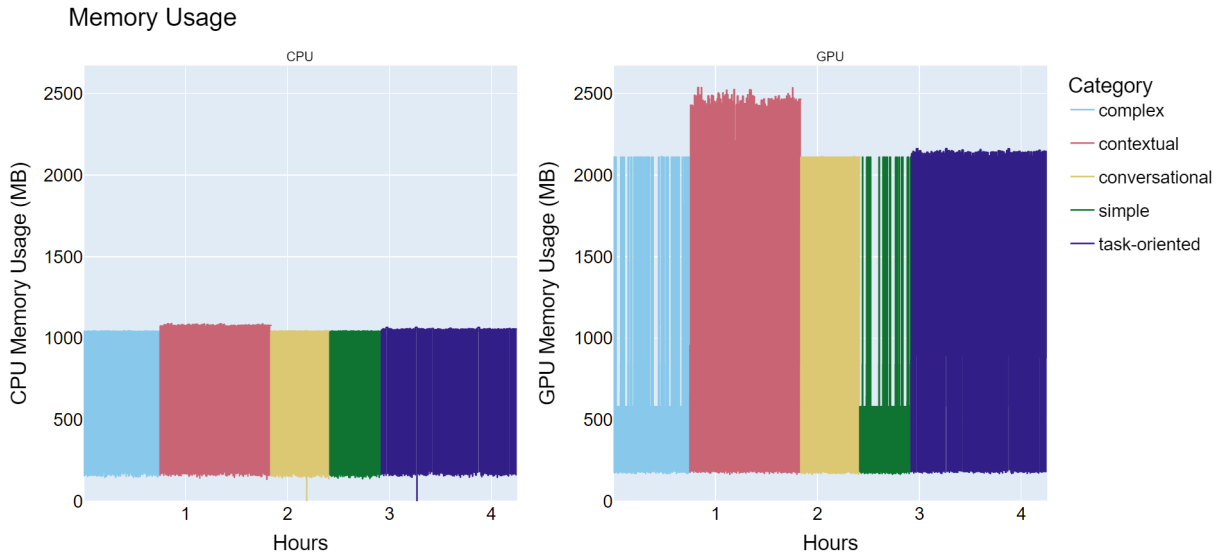
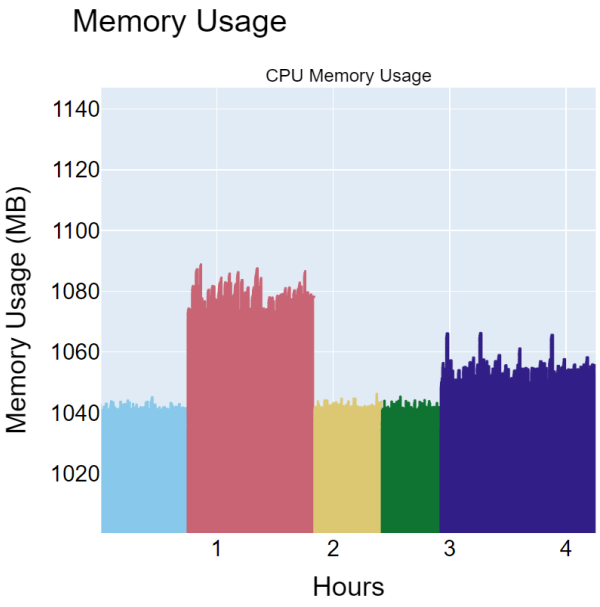


Figure 2. Analysis of CPU and GPU Memory Usage across workloads

Analysis of Memory usage

We observed that CPU memory usage remains relatively consistent across different workloads, even when the length of queries varies. This consistency is likely due to the storage of embeddings in CPU memory during inference, as these embeddings are required either at the start of the computation for each query or at the end, to convert token IDs back to text from individual probabilities. Since the size of these embeddings does not significantly vary between workloads, the CPU memory usage remains fairly constant. The primary computational load, particularly for attention operations, occurs on the GPU. When the GPU memory becomes overwhelmed, we observe some data being offloaded to CPU memory. This is



evident in workloads such as contextual and task-oriented queries, which exhibit higher than average GPU memory usage, accompanied by a slight increase in CPU memory usage, as shown in the zoomed-in version of the CPU memory usage plot.

Figure 3. Zoomed in CPU memory usage across workloads

Regarding GPU memory usage, the contextual workload stands out with significantly higher memory consumption, while the task-oriented workload shows slight spikes compared to others. This difference can be attributed to the storage of weights, biases, and the KV cache in GPU memory during inference. The KV cache is constructed during the prefill stage and updated with each new token generated during the decoding phase, allowing subsequent tokens to utilize previous KV entries for computing attention. Consequently, the size of the KV cache is directly proportional to the number of tokens, leading to increased memory usage for longer prompts. The average prompt length for queries in the contextual workload is notably longer than in other workloads, with the task-oriented workload following closely behind.

Figure 4: Mean prompt length across categories

Category	Category_prompt_tokens_mean
complex	32
contextual	436
conversational	43
simple	29
task-oriented	189

Analysis of Compute

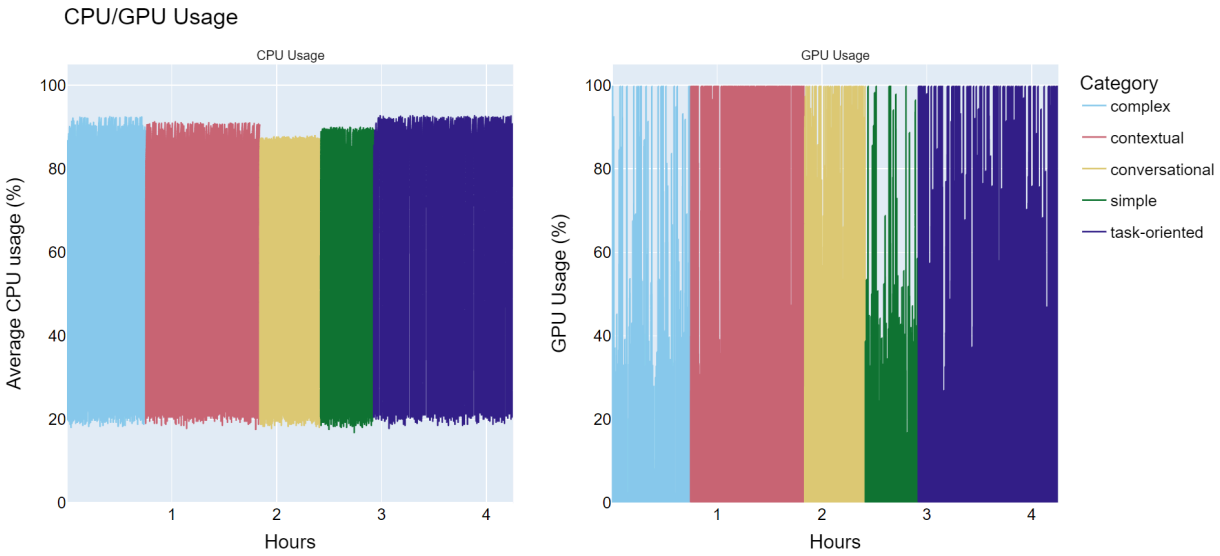


Figure 5. Average CPU and GPU usage across workloads.

We observed that average CPU usage is lower for conversational workloads compared to other types. This may be attributed to the TinyLlama model being specifically fine-tuned for chat-related use cases. If conversational patterns were better represented during the model's training stages, the model might be more efficient in handling these workloads, leading to quicker generation times. The near-full GPU utilization observed for conversational workloads could indicate that the GPU is managing these tasks more efficiently, with less need for the CPU to handle intermediate states. Similarly, the simple workload also shows relatively low average CPU usage, likely because these queries are straightforward and recall-based, requiring minimal CPU intervention to manage intermediate states on the GPU.

We also noted significant variation in GPU usage across complex and simple workloads, with noticeable drops in GPU memory usage for these categories, while conversational workloads consistently occupy GPU memory. Fluctuations in GPU usage may occur when the model needs to reference different parts of its knowledge base. Complex queries often demand multiple layers of reasoning and context processing, whereas simple queries typically require straightforward recall, triggering fewer deep transformer layers. This difference is reflected in the GPU usage plot, where the simple workload shows fewer spikes in GPU utilization compared to the complex workload. The same behavior is observed in GPU memory usage. An interesting insight is that some queries in the simple workload may be similar in sentence length and

information requirements to those in the complex workload, as suggested by the observed patterns in GPU usage.

The GPU is heavily utilized when numerous parallel operations are required, particularly during token generation (e.g., attention computation, normalization, and feedforward layers—essentially matrix multiplication). The consistently high GPU usage for contextual and conversational workloads indicates that these tasks are more generation-intensive than others. This is intuitive, as the GPU performs significant work in establishing attention with existing tokens in contextual workloads, where queries are generally longer. For conversational workloads, the model's fine-tuning for chat-related queries might result in greater GPU efficiency or utilization, explaining why task-oriented workloads do not exhibit consistently high GPU usage, despite involving step-by-step generation based on user utterances.

Finally, the plots for both memory usage and compute illustrate that different workload types require varying amounts of time to execute, largely due to the expected number of output tokens. Task-oriented workloads, which have an expected output of 100 tokens—the highest among all categories—naturally take longer to generate their output.

Limits on number of output tokens:

{"complex": "70", "contextual": "50", "conversational": "40", "simple": "30", "task-oriented": "100"}

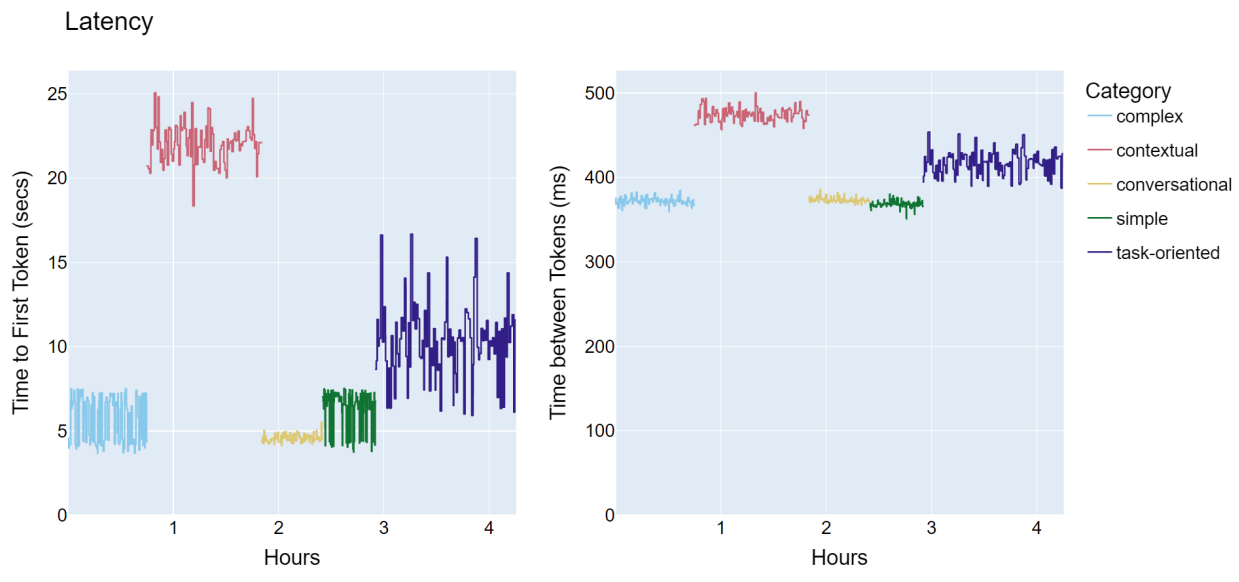


Figure 6. Time to First token (TTFT) and Time between Tokens (TBT) across workloads.

Latency Analysis

We measured latency in two forms: the time to generate the first output token for a query, referred to as "Time to First Token" (TTFT), and the latency between generating each subsequent token during the decode stage, referred to as "Time Between Tokens" (TBT).

Time to First Token (TTFT):
For workloads with longer prompts, such as contextual and task-oriented queries, we expected TTFT to be greater than for other workloads, which is consistent with our observations. We noticed a larger variation in TTFT for task-oriented workloads compared to others, likely due to the wider range of token counts in the prompts of task-oriented queries. TTFT is heavily influenced by the prefill stage of LLM inference, where the length of the prompt and the time required to build the KV cache directly affect TTFT. An interesting observation is that conversational queries exhibit the least variation in TTFT and have a shorter TTFT compared to simple and complex workloads, despite not being shorter in length. This could be attributed to TinyLlama being specifically fine-tuned for chat instructions, which may enhance its efficiency in handling conversational prompts.

Time Between Tokens (TBT):
TBT essentially represents the time it takes to execute one step of the decoding cycle given the existing sequence. During this stage, the KV cache, which was built and loaded into memory during the prefill stage, must be updated for each new token. Additionally, other operations, such as matrix calculations for normalization and the feedforward layer, are performed. A longer TBT suggests that the GPU is taking more time to complete these operations, potentially indicating that the GPU is compute-bound or that GPU memory is insufficient, leading to data being swapped between CPU and GPU memory, causing delays.

In our analysis, we observed a slight increase in CPU memory usage for contextual and task-oriented workloads, which could suggest some level of data transfer between CPU and GPU memory. However, given that these spikes are minimal, the delays are likely due to the GPU being compute-bound rather than memory-constrained. This indicates that the contextual and task-oriented workloads place a higher strain on the GPU compared to other workloads, as expected.

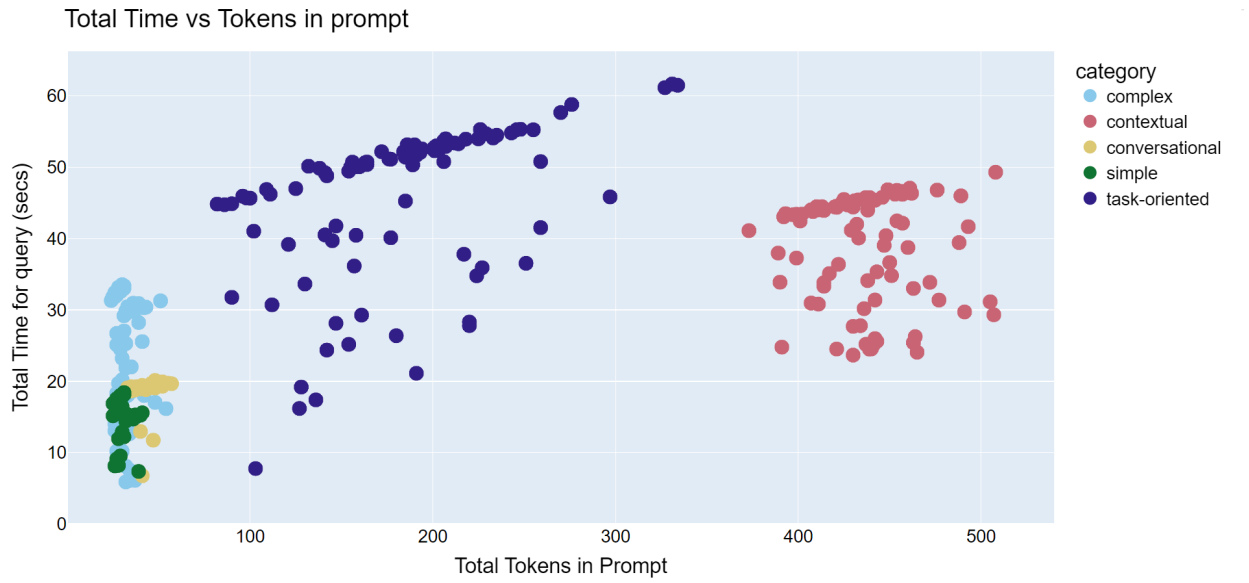


Figure 7. Total time per query vs Tokens in Prompt across workloads.

Analysis of Total Execution Time for Queries

Task-oriented queries exhibit the largest variation in prompt size compared to other workloads. We anticipated that an increase in the number of tokens in the prompt would result in a longer total execution time for a query, primarily due to the expected increase in TTFT. Additionally, a higher number of expected output tokens would naturally extend the total execution time, as more tokens need to be generated.

In the plot, an increasing trend in "total time for query" is evident within each workload as the number of tokens in the prompt increases. Task-oriented workloads show elevated values for "total time taken for query," which can be attributed to the higher expected output token count in these queries compared to other workloads. However, we also observe variations in the total time taken for queries within the same workload, even when the number of prompt tokens is similar.

These variations may be influenced by how each individual query interacts with the model, particularly in terms of recalling information from deeper layers, reasoning, understanding context, and the GPU's role in generating tokens. The complexity and nature of each query likely contribute to differences in execution time, even within the same workload category.

Granular analysis of Queries

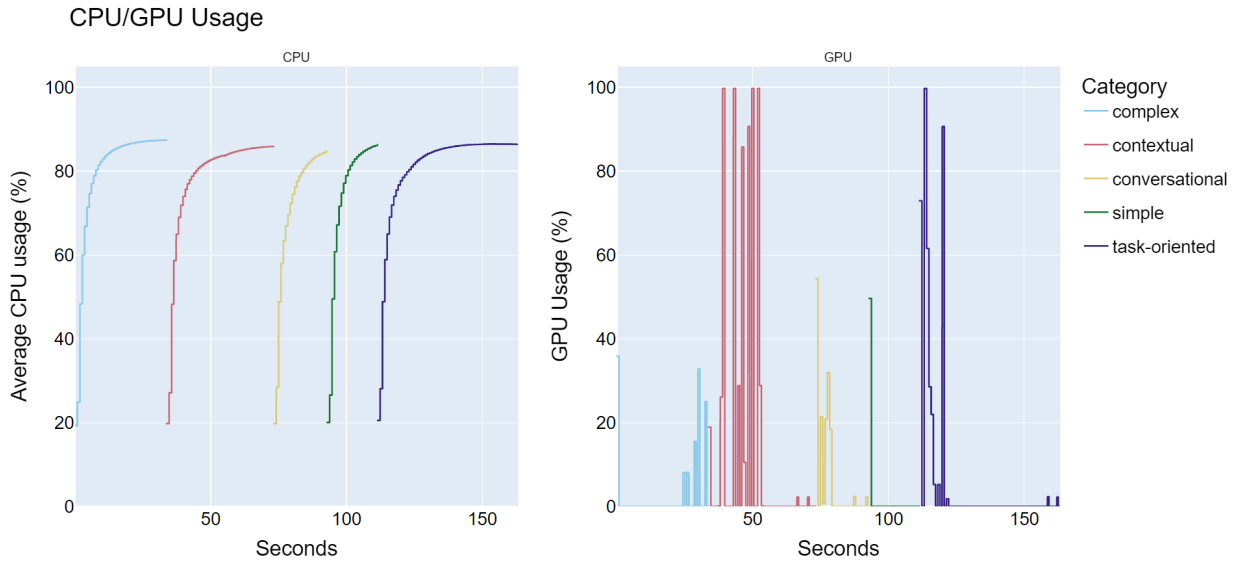


Figure 8. Average CPU and GPU use for a single query.

Analysis of CPU and GPU Usage Patterns

Our analysis shows that the average CPU usage across different queries remains fairly consistent (excluding measurements from the warm-up phase). However, we observed that task-oriented and contextual queries exhibit two or more GPU usage spikes—typically at the beginning and the end of the query—whereas other query types generally show a spike only at the beginning.

This behavior can be attributed to the nature of these workloads. In contextual queries, the prompt usually includes a paragraph providing context, followed by a question related to that context. For task-oriented queries, the model is asked to function as a bot, assisting the user with specific tasks such as booking movies, hotels, or flights. These queries often involve a sequence of <USER> utterances, requiring the model to understand the task and generate a step-by-step plan for its completion.

As a result, queries in these workloads are generally longer than those in other categories and require the model to deliver output after processing and understanding the context. The multiple GPU usage spikes likely indicate heavy computation occurring in two or more phases. In the first phase, the model might be establishing the context by filling up the KV cache, which is relatively larger than in other workloads. In the second or subsequent phases, the model performs iterative processing to generate responses based on the previously established context.

This pattern is particularly interesting because it suggests that even other workloads (except for simple queries) show smaller subsequent GPU usage spikes, indicating that the model requires initial GPU computation to establish context and additional GPU resources later to generate responses. Although this is a decoder-only model, which typically does not exhibit phase differentiation in GPU usage as encoder-decoder models do, the observed behavior suggests that similar phase-dependent GPU usage patterns may still occur during inference.

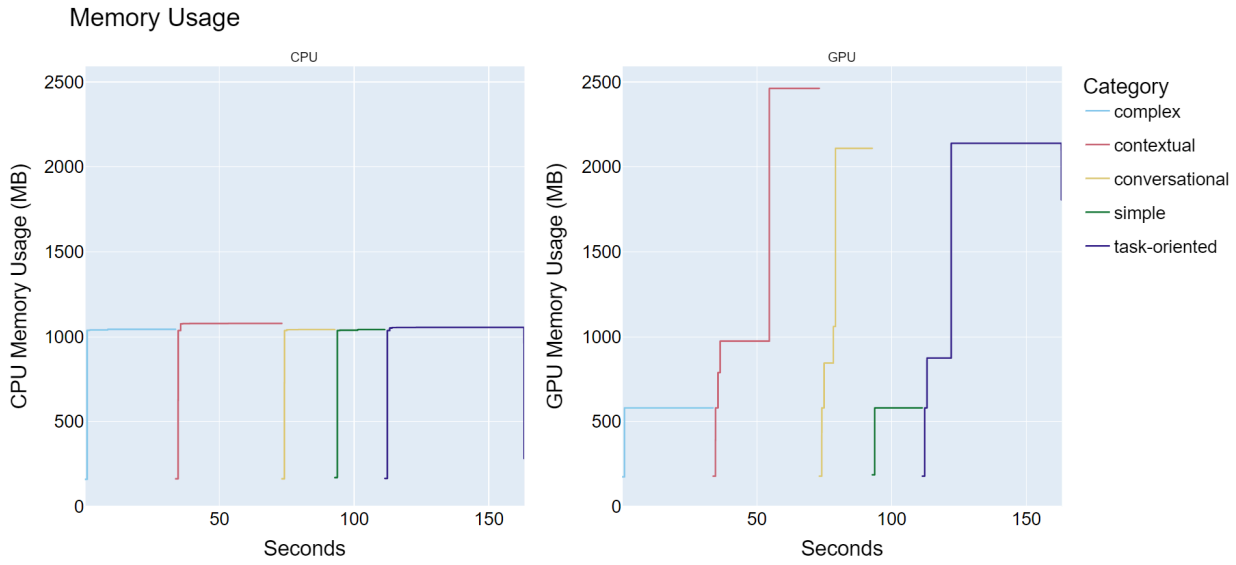


Figure 9. CPU and GPU memory usage for individual queries.

Granular memory usage analysis

Components Contributing to CPU Memory Usage

The following components are stored in CPU memory during inference, based on observations from the query logs generated by llama.cpp:

1. **Model Weights**: The model weights are stored in CPU memory. Since these weights are consistent across all query types, this factor remains constant.

2. **Embeddings for Input Tokens**: The embeddings, which map each input token to a high-dimensional vector space, are computed and stored in CPU memory. The embedding layer is the same for all query types, making this a consistent factor across workloads.

3. **KV Cache Allocation**: A fixed portion of the KV cache is allocated in CPU memory, determined by the maximum context length, the number of layers, and the number of heads. This allocation does not vary with the query type.

4. **Pre-Allocated Input and Compute Buffers**: These buffers store intermediate data before it is transferred to the GPU. They remain the same regardless of the query type.

Components Contributing to GPU Memory Usage

More complex or lengthy queries, which involve long contexts, more reasoning, and less recall-based output, tend to occupy more GPU memory. This is especially true for contextual, conversational, and task-oriented workloads, as not all queries from complex and simple workloads exhibit these characteristics. The key components taking up GPU memory include:

1. **Temporary Tensor Cores**: GPU memory is required to store temporary tensor cores for operations such as matrix multiplications and attention calculations. Although these tensors are often deallocated after each operation, they can occupy significant space during execution. More complex queries, such as those in contextual or task-oriented workloads, generate more intermediate results, requiring more temporary tensors and thus higher GPU memory usage.

2. **KV Cache**: The KV cache primarily resides in the GPU for faster memory access, supporting attention calculations during the token generation process. While the size of this cache is influenced by the actual context length, it remains fairly consistent across queries, given the absence of an order of magnitude difference in context length.

    - Estimating KV cache usage (Verma) as:

      Total size of KV cache in bytes = 2 (two matrices - K,V) * (context length) * (number of heads) * (embedding dimension per head) * (number of layers) * 2 Bytes (sizeof fp16)

      Consider a context length of 512 (on the higher side) across all workloads, we get KV cache size as 2 * (512) * (32) * (64) * (22) * 2 = 92MB

- We consider fp 16 even with quantised model because this Jetson module supports quantisation till FP16 only (not even bf16): ("Quantization in TensorRt")

3. **Activation Outputs**: The outputs of each transformer layer (activations) are temporarily stored in GPU memory as the model processes the input tokens. Longer or more complex queries might result in larger or more numerous activation outputs, leading to higher GPU memory usage.

4. **Attention Mechanism**: The attention mechanism requires storing the results of attention scores, weighted sums, and other operations in GPU memory during inference. For longer and more complex queries, the size of the attention buffers will be larger.

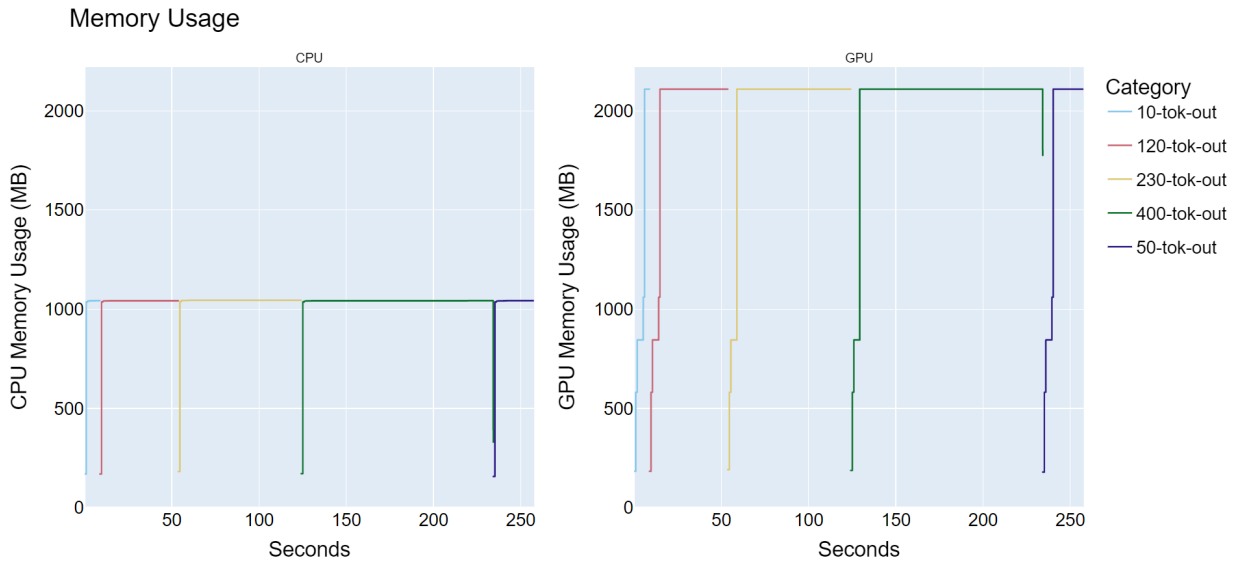


Figure 10. Same conversational query executed with increasing number of expected output tokens.

GPU Memory Usage and Output Length Analysis

To investigate how GPU memory usage changes with increasing output length, we executed a conversational query with a prompt length equal to the median prompt length for the conversational workload category. We selected a conversational query type because TinyLlama is fine-tuned for chat-related tasks. After excluding the warm-up phase, we repeatedly executed the same query, instructing the model to generate progressively longer responses with each run.

Our expectation was that GPU memory usage would increase proportionally with the length of the output. However, an interesting observation emerged: the GPU memory usage remained constant across all five runs, regardless of the output length. Instead, the time spent by the GPU executing the query increased linearly with the increasing output length.

This unexpected result led us to conduct a reverse experiment to further explore the relationship between input and output tokens. We aimed to observe how GPU memory usage would change when varying the number of input tokens while keeping the number of output tokens constant.

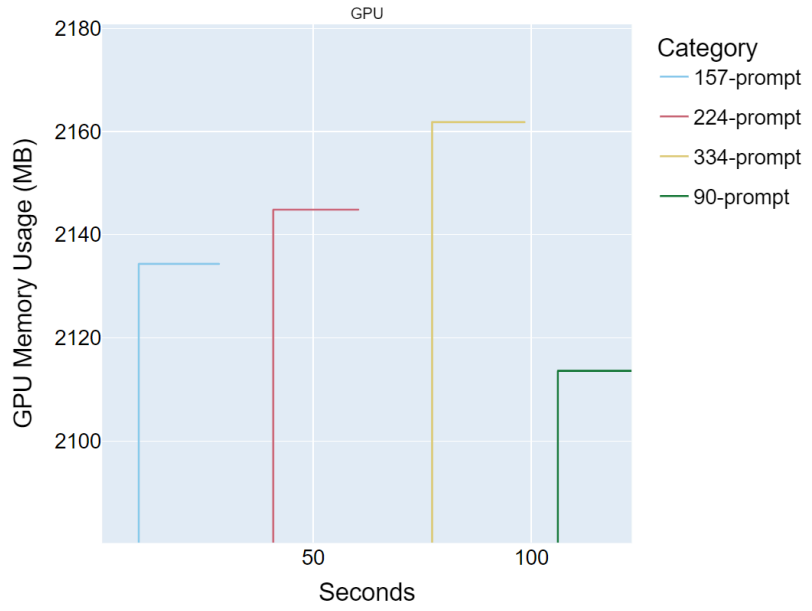


Figure 11. (Zoomed version) Four task-oriented queries with different prompt lengths executed with the same number of expected output tokens (50).

GPU Memory Usage with Varying Input Tokens

For this experiment, we selected four different queries from the task-oriented workload. Task-oriented queries were chosen over conversational queries due to the greater variation in the number of input tokens available in this workload. The output length was set to 50 tokens for all four queries, and any data from warm-up runs was discarded.

The zoomed-in plot reveals that GPU memory usage does indeed increase as the number of tokens in the prompt increases. While the change in the number of tokens is not orders of magnitude different, there is still a noticeable difference in GPU memory usage, measured in megabytes (MB).

However, the reasoning behind the lack of observed difference in GPU memory usage for varying output tokens in the previous experiment remains unclear. It is possible that the nature of the query being "conversational" may have influenced this outcome.

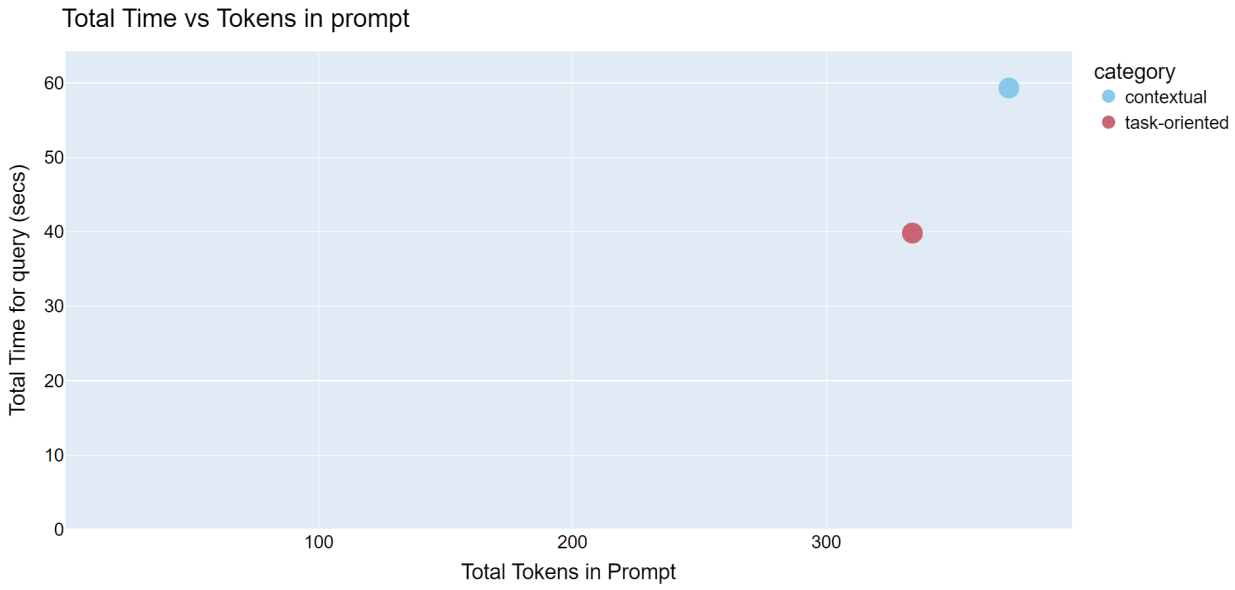


Figure 12. Total query time comparison for similarly sized contextual and task-oriented query.

Validation of Query Execution Time Relative to Token Count and Query Complexity

To validate our hypothesis that both the number of tokens in the prompt and the complexity of the query influence the total time required for execution, we compared contextual queries with task-oriented queries. Specifically, we anticipated that contextual queries, which generally involve longer prompts and greater complexity, would take longer to execute than task-oriented queries.

In our workload plot (total time per query vs. total tokens in prompt), we observed that task-oriented queries generally exhibited higher execution times compared to contextual queries, despite the latter typically having longer prompts. This discrepancy was attributed to the difference in the number of output tokens to be generated: 100 tokens for task-oriented queries versus 50 tokens for contextual queries.

To test this further, we selected one query from each workload with similar prompt lengths and set the number of expected output tokens to 50 for both. After discarding warm-up runs, we found that the contextual query indeed took longer to execute than the task-oriented query when the number of expected

output tokens was controlled. This finding confirms that, when output token length is held constant, the complexity and context length of the query significantly impact the total execution time, with contextual queries requiring more time than task-oriented ones.

Conclusion

In this project, we deployed the TinyLlama 1.1B chat fine-tuned model on an Nvidia Jetson Nano 4GB module using the llama.cpp inference runtime. We executed and analyzed five categories of query workloads: simple, complex, contextual, conversational, and task-oriented. Through detailed observation of CPU and GPU memory usage, as well as overall system utilization during workload execution, we found that the Jetson Nano module possesses sufficient capacity to handle additional computational tasks. However, it is important to note the issue of the device heating up relatively quickly under load.

Our analysis revealed distinct trends among the different categories of query workloads, which we explained in the context of the TinyLlama architecture. We highlighted interesting patterns in system performance, both during bulk query execution and individual executions for more granular measurements.

The query workloads tested in this project represent the common types that an LLM agent might encounter. Through this work, we established a baseline for evaluating the feasibility of deploying small language models on edge devices for agent-based tasks.

Future work on this topic could explore the following avenues:

1. **System Performance Analysis of Encoder-Decoder Models**: Investigating the performance of encoder-decoder based models on edge devices, particularly under different workload types.

2. **Custom Wrapper Development for LLM Agents**: Building a custom wrapper for LLM agents, independent of frameworks like LangChain, that includes in-memory database and prompt formatting functionalities using the llama.cpp inference runtime. This could be especially valuable for older devices that do not meet the dependency requirements of more recent frameworks.

3. **Multimodal Deployment on Jetson Orin Series**: Utilizing the Jetson Orin series with larger memory capacities (16GB/64GB) for multimodal deployments, followed by tracking and analyzing resource usage.

Bibliography

1. *Alammar, Jay. The Illustrated Transformer. https://jalammar.github.io/illustrated-transformer/. Accessed 16 Aug. 2024.*
2. *"balenaEtcher." Flash OS Images to SD Cards & USB Drives, https://etcher.balena.io/. Accessed 16 Aug. 2024.*
3. *bzhangGo. "GitHub - bzhangGo/Rmsnorm: Root Mean Square Layer Normalization." GitHub, https://github.com/bzhangGo/rmsnorm. Accessed 16 Aug. 2024.*
4. *cilynx. "GitHub - Cilynx/rtl88x2BU_WiFi_linux_v5.3.1_27678.20180430_COEX20180427-5959: Rtl88x2bu Driver Updated for Current Kernels." GitHub, https://github.com/cilynx/rtl88x2BU_WiFi_linux_v5.3.1_27678.20180430_COEX20180427-5959. Accessed 16 Aug. 2024.*
5. *ggerganov. "GitHub - Ggerganov/Llama.Cpp: LLM Inference in C/C++." GitHub, https://github.com/ggerganov/llama.cpp. Accessed 16 Aug. 2024.*
6. *Hoffmann, Jordan, et al. "Training Compute-Optimal Large Language Models." arXiv.Org, 29 Mar. 2022, https://arxiv.org/abs/2203.15556.*
7. *"JetPack SDK." NVIDIA Developer, https://developer.nvidia.com/embedded/jetpack. Accessed 16 Aug. 2024.*
8. *"Jetson-Stats." Jetson-Stats, https://rnext.it/jetson_stats/. Accessed 16 Aug. 2024.*
9. *Li, Baolin, et al. "LLM Inference Serving: Survey of Recent Advances and Opportunities." arXiv.Org, 17 July 2024, https://arxiv.org/abs/2407.12391.*
10. *Li, Beibin, et al. "Small Language Models for Application Interactions: A Case Study." arXiv.Org, 23 May 2024, https://arxiv.org/abs/2405.20347.*
11. *LLM Inference Unveiled: Survey and Roofline Model Insights. https://arxiv.org/html/2402.16363v1. Accessed 16 Aug. 2024.*
12. *Michelle Li. Friends_dataset. https://huggingface.co/datasets/michellejieli/friends_dataset. Accessed 16 Aug. 2024.*
13. *"NVIDIA Jetson Nano." NVIDIA,*

https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/. Accessed 16 Aug. 2024.

14. Psutil Documentation — Psutil 6.0.1 Documentation. https://psutil.readthedocs.io/en/latest/. Accessed 16 Aug. 2024.
15. "Quantization in TensorRt." NVIDIA Developer Forums, 21 Jan. 2022, https://forums.developer.nvidia.com/t/quantization-in-tensorrt/201227.
16. Question Answering in Context. https://quac.ai/. Accessed 16 Aug. 2024.
17. salesforce. "DialogStudio/Task-Oriented-Dialogues/MULTIWOZ2_2 at Main · Salesforce/DialogStudio." GitHub, https://github.com/salesforce/DialogStudio/tree/main/task-oriented-dialogues/MULTIWOZ2_2. Accessed 16 Aug. 2024.
18. "SD Memory Card Formatter." SD Association | The SD Association, 11 Dec. 2020, https://www.sdcard.org/downloads/formatter/sd-memory-card-formatter-for-windows-download/.
19. Su, Jianlin, et al. "RoFormer: Enhanced Transformer with Rotary Position Embedding." arXiv.Org, 20 Apr. 2021, https://arxiv.org/abs/2104.09864.
20. The Stanford Question Answering Dataset. https://rajpurkar.github.io/SQuAD-explorer/. Accessed 16 Aug. 2024.
21. Verma, Shashank. "Mastering LLM Techniques: Inference Optimization." NVIDIA Technical Blog, 17 Nov. 2023, https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/.
22. Weng, Lilian. "LLM Powered Autonomous Agents." Lil'Log, 23 June 2023, https://lilianweng.github.io/posts/2023-06-23-agent/.
23. Wiki_qa. https://huggingface.co/datasets/microsoft/wiki_qa. Accessed 16 Aug. 2024.
24. Wolfe, Ph. D. Cameron R. "LLaMA-2 from the Ground Up." Deep (Learning) Focus, 14 Aug. 2023, https://cameronrwolfe.substack.com/p/llama-2-from-the-ground-up.
25. Zhang, Peiyuan, et al. "TinyLlama: An Open-Source Small Language Model." arXiv.Org, 4 Jan. 2024, https://arxiv.org/abs/2401.02385.
26. Zhou, Zixuan, et al. "A Survey on Efficient Inference for Large Language Models." arXiv.Org, 22 Apr. 2024, https://arxiv.org/abs/2404.14294.

Appendices

1. The Importance of LLM Inference on Edge

Performing inference directly on edge devices, such as the Jetson Nano, offers several significant advantages. First, it eliminates the need for data to travel to a central server or cloud for processing, thereby drastically reducing response time—a critical factor for applications requiring quick decision-making or real-time feedback, such as autonomous vehicles, real-time language translation, and interactive customer service systems.

Furthermore, processing data locally ensures that sensitive information remains on the device, which is crucial for maintaining confidentiality in sectors like healthcare and finance, where data privacy is paramount. Edge devices also operate independently of the cloud, enabling continued functionality even during network outages. This reliability is essential for remote monitoring systems in agriculture, healthcare in isolated areas, and industrial IoT applications where uninterrupted operation is necessary. Local data processing also aids in compliance with regulations such as the GDPR in Europe, which imposes strict rules on data transfer and storage, particularly concerning the handling of personal data across borders.

2. Significance of Deploying SLM Agents on Edge Devices

The local processing capability of edge devices also supports a range of use cases (for robots or drones) where mobility is critical. These can be integrated into surveillance systems to interpret and respond to audio cues or alerts. In healthcare, on-device, real-time interaction can provide support through medication reminders, basic diagnostics, and health advice. In industrial IoT, edge LLMs can enable predictive maintenance and real-time monitoring of machinery in environments where internet access is limited or unreliable. Additionally, on-device analysis of sensitive documents can ensure compliance with data protection regulations, minimizing the risk of data leakage.

3. Nvidia Jetson Nano Unified memory

The Jetson Nano board features a unified memory management system, where the CPU and GPU share a common pool of DRAM. This allows memory allocated by the CPU to be directly accessed by the GPU and vice versa. Special care is required when working with PyTorch GPU code, as it typically does not account for unified memory architecture, potentially leading to data being copied twice in memory—once by the CPU and once by the GPU.

4. Additional steps for compiling gcc 8.5 and llama.cpp on device

We referenced the steps by Flor Sanders described here and added some modifications to suit our environment.

a. GCC Compiler Setup:

The board ships with GCC 7, which is incompatible with llama.cpp, necessitating the installation of GCC 8.5. Since JetPack 4.6 does not natively include GCC 8.5, we built it from source. We downloaded the GCC 8.5 source files and pre-requisites, configured them with the following options: `-enable-checking=release -enable-languages=c,c++`, and then built and installed GCC 8.5 on the device. We set up the GCC and G++ environment variables to ensure that the llama.cpp makefile would utilize these.

b. Cloning and Configuring Llama.cpp:

We cloned the llama.cpp repository and checked out a known working commit compatible with the JetPack 4.6 software stack (`git checkout a33e6a0`). The makefile for llama.cpp required specific modifications:

- **MK_NVCCFLAGS**: Set `maxrregcount=80`.
- **MK_CXXFLAGS**: Removed the `mcpu=native` flag.

We also added the CUDA toolkit to the system path to enable `nvcc` for compiling llama.cpp. The compute capability of the Jetson Nano module was determined to be 5.3, which was used during the build process with the command:

```
make LLAMA_CUBLAS=1 CUDA_DOCKER_ARCH=sm_53 -j 6
```

c. Running Inference:

After storing the TinyLlama model on the device, we executed inference by invoking the main file in llama.cpp with the following command:

```
/nano-llama/llama.cpp/main -m
/nano-llama/llama.cpp/TinyLlama-1.1B-Chat-v1.0-Q5_K_M.gguf -p "You are a
very helpful AI assistant who gives to the point responses to the user:
Hi, how are you today?"
```

5. Challenges and Solutions related to deployment

During the deployment of the TinyLlama model on the Jetson Nano, several challenges were encountered, which required specific workarounds and solutions.

Difficulties Faced During Deployment

1. **Flashing the SD Card**:
    - **Issue**: The SD card flash process can fail if the lock switch on the SD card adapter is engaged, even though formatting may still work.
    - **Solution**: Ensure that the lock switch is in the correct position before flashing. The locked status will be reflected in the Etcher tool, so verify it there as well.
2. **Power Management**:
    - **Issue**: Running the Jetson Nano in 5W mode (default when using micro-USB) can lead to throttling of system resources.
    - **Solution**: Use a power adapter to run the board in 10W mode. Connect the J48 Power Select Header pins using jumpers to disable power supply via micro-USB.
3. **JetPack SDK Limitations**:
    - **Issue**: The Jetson Nano 4GB is limited to JetPack 4.6.5, which includes Jetson Linux 32.7.5 based on Ubuntu 18.04, CUDA 10.2, cuDNN 8.2, TensorRT 8.0, and GCC 7. This restricts the versions of software and libraries that can be deployed.
    - **Solution**: Ensure that all dependencies for workloads are compatible with this environment to take advantage of the GPU.
4. **Building GCC 8 Compiler**:
    - **Issue**: Building the GCC 8.5 compiler on the Jetson Nano board is a time-consuming process, taking approximately 4-5 hours.
    - **Solution**: Use a larger SD card (128GB) to accommodate the base OS image, which occupies 15-20GB, leaving sufficient space for models and experiments. After installing the GCC compiler, clear the build files to free up space.
5. **Cross-Compilation Challenges**:
    - **Issue**: Cross-compiling llama.cpp on a different Unix machine (Intel x64) introduced complexities, such as setting up the ARM cross-compiler, copying necessary libraries and headers, and configuring GCC with sysroot support. Additionally, the llama.cpp makefile required numerous environment variables, which were challenging to configure.
    - **Solution**: After several errors, it was concluded that building both the GCC compiler and llama.cpp directly on the Jetson Nano, using Flor Sanders's steps, was more efficient. Although transferring the built GCC compiler to the Jetson Nano was theoretically possible, the complexity of environment setup led to the decision to build directly on the board.

6. Data fields recorded using jetson stats

(https://rnext.it/jetson_stats/reference/jtop.html#jtop.jtop.stats)

Name	Type	Reference	Description
time	datetime.datetime		local time in your board
uptime	datetime.timedelta	uptime	up time on your board
cpu X	float	cpu	The status for each cpu in your board, if disabled *OFF*
RAM	float	memory	RAM used / total
SWAP	float	memory	SWAP used / total
EMC	float	memory	*(Optional)* EMC Percentage of bandwidth
IRAM	float	memory	*(Optional)* IRAM used / total
GPU	float	gpu	*(Optional)* Status of your GPU
engine X	float	engine	*(Optional)* Frequency for each engine, if disabled *OFF*
fan	float	fan	*(Optional)* Fan speed
Temp X	float	power	*(Optional)* Current power from rail X
Temp TOT	float	power	*(Optional)* Total power
jetson_clocks	str	jetson_clocks	*(Optional)* Status of jetson_clocks, human readable
nvpmodel	str	nvpmodel	*(Optional)* NV Power Model name active

7. Instruction appended to each query

The length of prompt referred to in the experiments includes the tokens instructing the model what is to be done for each query category.

```
query_category_prompts = {

"complex": "You are a helpful assistant. Give a short answer. Query:
$$**$$ Answer: ",
```

```
 "contextual": "You are a very helpful assistant. Answer the question:
$$**$$ Answer: ",

 "conversational": "You are a helpful assistant.> User: $$**$$? Assistant:
",

"simple": "You are a helpful assistant who knows everything. Give a short
answer. Query: $$**$$ Answer: ",

"task-oriented": "$$**$$ <BOT>: Great! Based on the provided information,
following are the steps: "}
```

8. Query samples from experiments

Queries picked for granular experiments (median length)

Simple
what is i2c i/f

Complex
Who do states and governments often work in lockstep with?

Contextual
::context:: kinison acquired much of his material from his difficult first two marriages, to patricia adkins (1975-1980) and terry marze (1981-1989). he began a relationship with dancer malika souiri toward the end of his marriage with marze. in 1990, souiri alleged she was raped by a man kinison had hired as a bodyguard that same day, while kinison was asleep in the house. the bodyguard stated that the sex was consensual; the jury deadlocked in the subsequent trial and the charges were later dropped. on april 4, 1992, six days before his death, kinison married souiri at the candlelight chapel in las vegas. they honeymooned in hawaii for five days before returning home to los angeles on april 10 to prepare for a show that night at the riverside resort hotel and casino in laughlin, nevada. souiri sued kinison's brother bill in 1995 for allegedly defaming her in his book brother sam: the short spectacular life of sam kinison, and then again in 2009 for allegedly forging sam's will. in february 2011, the toronto sun reported that kinison had fathered a child with the wife of his best friend and opening act, carl labove, who had been paying child support for the girl for nearly 13 years. labove filed legal papers claiming the girl was kinison's, and dna tests taken from kinison's brother

bill show a 99.8% likelihood that kinison was the father of the unnamed woman, who was 21 at the time of the toronto sun story, and excluded labove as her father. ::question:: who did he marry?

Conversational

I know, I know I really like you too. But we-we cant date. Its against the rules. Its forbidden.

Task Oriented

This is a bot helping users to find a restaurant and book a train ticket. Given the dialog context, please generate a relevant system response for the user. : <USER> Are there trains arriving in Cambridge before 18:45 on Sunday? <USER> I'm leaving Bishops stortford. <USER> Can I get the price, travel time, and departure time of the one arriving at 18:07? <USER> Not at this time, but I am also looking for a european food restaurant in the expensive price range. Can you help with that? <USER> In the centre of town please. <USER> Yes, please. It will be just me and I'd like to eat at 21:00 on the same day as my train.

Queries used for experiment: Validation of Query Execution Time Relative to Token Count and Query Complexity

Task-oriented

This is a bot helping users to complete multiple tasks, e.g. find a restaurant, find a taxi, and find a hotel. Given the dialog context, please generate a relevant system response for the user. : <USER> Find me a Mediterranean restaurant in the centre of Cambridge. <USER> Yes, please book that for me. <USER> I need the table booked for monday at 12:15 for 3 people. Thanks. Also could i get the ref #? <USER> Great I also am looking for a hotel to stay at that includes free parking. What's available? <USER> Actually, I'm looking for a hotel and in the same price range as the restaurant. <USER> Yes I need a hotel in the same price range as the restaurant with free parking. <USER> Can you give me the first one and the star rating for it please? It has to have free parking. <USER> Does the Express by Holiday Inn have free parking? <USER> Yes also I will need a taxi to get me between the hotel and the restaurant. I will need a contact number and car type with the booking. <USER> I'd just like to book a taxi between the two. I'd like to get to the restaurant by my reservation time and I'll need the contact number and car type please. <USER> What time will the taxi arrive? <USER> That will be fine.

Contextual

```
::context:: by the beginning of 2004 the band announced their first
greatest hits compilation was going to be released, a cover of nick
kamen's "i promised myself" became the last single from the band. the
band shot the video for the song in march 2004, and was premiered on
ztv in early april, the song went to international radios on late
april/may becoming the last hit of the band. the album was a
compilation of thirteen singles, which each one of them made the top
twenty in at least one country and three new tracks, two of them were
written by the band members. promotion for the album was slow, the
band did a few shows in sweden and international interviews to
magazines. many were reporting the band was splitting up after six
years in the pop world. the band quickly denied the rumours on their
official website. the single became another top-two hit for the band
in sweden, it became one of the band's biggest hits in south america
(especially in argentina) and the album became the band's first to
not make the top ten in their homeland while it brought back the
attention to them in other countries in latin america and eastern
europe. the album was just released in selected european countries,
asia and latin america. it did not receive a u.s. release, although,
when mca records went bankrupt and was absorbed by both geffen and
interscope, the former had plans to release it in november, ready for
the christmas sales, but plans were scrapped when the band finished
their tour and dhani released his first solo single. ::question:: why
did they release this
```

Query used for experiment: GPU Memory Usage and Output Length Analysis

Same conversational query executed with increasing number of expected output tokens.

```
I know, I know I really like you too. But we-we cant date. Its
against the rules. Its forbidden.
```

Queries used for experiment: GPU Memory Usage with Varying Input Tokens

Four task-oriented queries with different prompt lengths executed with the same number of expected output tokens (50).

(90 tokens[1])

[1] The prompt token count here also includes the number of instruction tokens.

```
This is a bot helping users to find a restaurant. Given the dialog
context, please generate a relevant system response for the user. :
<USER> I am looking for a cheap restaurant in the centre. <USER> Yes,
may I have the address, postcode, and phone number for Golden House?
I'll book it myself.
```

(157 tokens[2])

```
This is a bot helping users to find a restaurant and find an
attraction. Given the dialog context, please generate a relevant
system response for the user. : <USER> Hi I am looking for a college
to go to in town. <USER> Yes, the centre area. <USER> Yes, and can
you find me an Italian restaurant in the same area? <USER> Yes, I'm
looking for something expensive. <USER> I think I'll try the Caffe
Uno. Can I get a table for 5 at 12:15 on Sunday? <USER> Yes could I
have my booking reference number please?
```

(224 tokens)[3]

```
This is a bot helping users to complete multiple tasks, e.g. find a
restaurant, book a train ticket, and find a hotel. Given the dialog
context, please generate a relevant system response for the user. :
<USER> Can you help me find a train departing from cambridge going to
kings lynn? <USER> I want to leave on Thursday and arrive by 14:00.
<USER> I don't have a specific time to leave but I do want to arrive
before 14:00. How much will this cost? <USER> No thanks. Not at this
time. Can you help me find a room to stay that is moderately priced?
<USER> No, I don't care where it is. I like 3 stars and I absolutely
need free wifi. <USER> Yes, make the reservation please. <USER>
Actually I don't need a reservation at this time.
```

(334 tokens)[4]

```
This is a bot helping users to complete multiple tasks, e.g. find a
restaurant, find a taxi, and find a hotel. Given the dialog context,
please generate a relevant system response for the user. : <USER>
Find me a Mediterranean restaurant in the centre of Cambridge. <USER>
Yes, please book that for me. <USER> I need the table booked for
monday at 12:15 for 3 people. Thanks. Also could i get the ref #?
<USER> Great I also am looking for a hotel to stay at that includes
free parking. What's available? <USER> Actually, I'm looking for a
hotel and in the same price range as the restaurant. <USER> Yes I
need a hotel in the same price range as the restaurant with free
parking. <USER> Can you give me the first one and the star rating for
it please? It has to have free parking. <USER> Does the Express by
```

[2] The prompt token count here also includes the number of instruction tokens.
[3] The prompt token count here also includes the number of instruction tokens.
[4] The prompt token count here also includes the number of instruction tokens.

Holiday Inn have free parking? <USER> Yes also I will need a taxi to get me between the hotel and the restaurant. I will need a contact number and car type with the booking. <USER> I'd just like to book a taxi between the two. I'd like to get to the restaurant by my reservation time and I'll need the contact number and car type please. <USER> What time will the taxi arrive? <USER> That will be fine.