CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Profiling and Visualizing CPU-Based LLM Inference in llama.cpp

Bachelor's thesis in Computer Science and Engineering

David Bohman
August Krümmel
Mirveys Mohammadi
Stevan Shehada
Emil Tervo

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Bachelor's thesis 2026

Profiling and Visualizing CPU-Based LLM Inference in llama.cpp

David Bohman
August Krümmel
Mirveys Mohammadi
Stevan Shehada
Emil Tervo

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Profiling and Visualizing CPU-Based LLM Inference in llama.cpp
David Bohman August Krümmel Mirveys Mohammadi Stevan Shehada Emil Tervo

Supervisor (Handledare): Mohammad Ali Maleki, Luigi Altamura, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering
Graded by teacher (Rättande lärare): Aarne Ranta, Department of Computer Science and Engineering

Bachelor's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Profiling and Visualizing CPU-Based LLM Inference in llama.cpp

David Bohman, August Krümmel, Mirveys Mohammadi, Stevan Shehada, Emil Tervo

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Large language models (LLMs) are increasingly deployed on local hardware rather than through cloud services, motivated by concerns around data privacy, network dependence, and operational cost. In many local deployments, the CPU serves as the primary or sole compute resource, since dedicated accelerators such as GPUs are not always available. Running LLMs on CPUs is technically demanding, as inference performance is shaped by complex interactions between the workload and the underlying hardware. Identifying performance bottlenecks is therefore essential for improving local LLM inference, but is difficult without structured measurement and analysis.

This bachelor's thesis presents a profiling and visualization tool for characterizing LLM inference on CPUs using an inference serving framework called `llama.cpp`. The tool collects hardware performance counters, which are low-level CPU metrics such as cache accesses and floating-point operations, at four levels of granularity: the entire inference run, the prefill phase in which the user's input prompt is processed and the decode phase in which the response is generated token by token, individual decoder blocks, and individual tensor operations. Measurements are stored in a structured database and can be explored through a graphical dashboard, allowing users to inspect runtime, memory traffic, cache behavior, and other metrics across different quantization formats and model sizes. By applying roofline analysis, the tool distinguishes between compute-bound and memory-bound behavior and supports the identification of performance bottlenecks during local LLM inference.

Keywords: LLM inference, CPU inference, `llama.cpp`, performance profiling, roofline model, hardware performance counters, transformer

Sammandrag

Stora språkmodeller (LLMs) körs i allt högre grad på lokal hårdvara istället för via molnbaserade tjänster, bland annat på grund av frågor kring integritet, nätverksberoende och kostnader. I många lokala driftsättningar utgör CPU:n den primära eller enda beräkningsresursen, eftersom dedikerade acceleratorer såsom GPU:er inte alltid finns tillgängliga. Att köra LLM-inferens på CPU:er är dock tekniskt krävande eftersom prestandan påverkas av komplexa interaktioner mellan arbetslasten och den underliggande hårdvaran. För att identifiera prestandaflaskhalsar krävs därför strukturerad mätning och analys.

Detta kandidatarbete presenterar ett profilerings- och visualiseringsverktyg för att analysera LLM-inferens på CPU:er med hjälp av ett ramverk för inferenshantering kallat `llama.cpp`. Verktyget samlar in hårdvaruprestanda-data, vilket är lågnivåmått för CPU:n såsom cacheåtkomster och flyttalsoperationer, på fyra granularitetsnivåer: hela inferenskörningen, prefill-fasen där användarens inmatningsprompt bearbetas och decode-fasen där svaret genereras token för token, individuella decoderblock samt individuella tensoroperationer. Mätdata lagras i en strukturerad databas och kan analyseras genom en grafisk dashboard. Genom att tillämpa roofline-analys kan verktyget identifiera om olika delar av inferensen är beräkningsbegränsade eller minnesbegränsade och därmed hjälpa till att lokalisera prestandaflaskhalsar vid lokal LLM-inferens.

Acknowledgements

We would like to express our sincere gratitude to our supervisors, Mohammad Ali Maleki and Luigi Altamura, for their invaluable guidance, insightful feedback, and continuous support throughout the duration of this project. Their expertise and encouragement have been instrumental to our work. We would also like to thank Pedro Trancoso for proposing the thesis topic.

We also wish to extend our appreciation to our examiner Patrik Jansson and co-examiner Arne Ranta, for their time and valuable review of this thesis.

August Krümmel, Emil Tervo, David Bohman,
Mirveys Mohammadi, Stevan Shehada
Gothenburg, June 2026

Contents

List of Figures

List of Tables

1
Introduction

Large language models (LLMs) are a class of AI systems that have shown strong performance across a wide range of language tasks, including text generation, question answering, and code synthesis [1]. Today, many LLMs are deployed through cloud-based inference services hosted in large-scale data centers [2]. However, this deployment model raises concerns related to data privacy, network dependence, and operational cost, which has increased interest in on-device LLM inference on consumer hardware [2], where dedicated accelerators such as GPUs are not always available and the CPU must serve as the primary compute resource.

Running LLM inference on CPUs presents significant challenges. Inference requires large amounts of memory and high memory bandwidth, which places pressure on commodity hardware [3]. Identifying performance bottlenecks therefore requires systematic measurement and analysis [4]. The `llama.cpp` framework [5] enables efficient CPU-based inference for transformer-based language models across a wide range of hardware platforms. In addition, tools such as `PAPI` [6] and `perf` [7] can collect hardware performance data during execution. However, raw measurements alone are insufficient, they must be associated with specific inference phases, transformer layers, and tensor operations, and interpreted using an appropriate performance model such as roofline analysis [8]. This thesis addresses this gap by developing a profiling and visualization tool for CPU-based LLM inference in `llama.cpp`. The source code for the tool is publicly available at https://github.com/mach-research-lab/profiling-llms-llama-cpp.

1.1 Purpose

The purpose of this project is to develop a profiling and visualization tool for analyzing CPU-based LLM inference in `llama.cpp`. The tool is intended to connect low-level hardware measurements, such as cache misses and floating-point operations, to specific parts of the inference process. The project focuses on making performance behavior easier to interpret by collecting measurements at multiple levels of granularity and presenting the results through both structured data storage and a graphical dashboard.

1.2 Research Questions

The project is guided by the following research questions:

- How can hardware performance counters be collected during `llama.cpp` inference at different levels of granularity?
- How can profiling data be structured and visualized to support analysis of transformer-based inference?
- How can roofline analysis be used to identify whether different parts of inference are limited by computation or memory bandwidth?

1.3 Limitations

This project focuses on CPU-based inference using `llama.cpp`. The work does not include training of language models, comparison of model quality, or optimization of the internal implementation of `llama.cpp`. The focus is instead on measurement, data collection, storage, and visualization of performance-related behavior during inference. The project also does not aim to provide a complete performance model for all hardware platforms. Measurements are limited by the hardware performance counters available on the host CPU, their accuracy, and the `PAPI` events supported by the CPU platform.

2
Theory

2.1 Transformer Architecture

The transformer is a neural network architecture for processing sequential data, introduced in *Attention Is All You Need* [9]. It forms the basis of many modern large language models (LLMs). Unlike recurrent neural networks (RNNs), which process tokens step by step, the transformer uses attention mechanisms and feed-forward layers to model relationships between tokens more efficiently [9].

During training, transformer models can process all tokens in a sequence in parallel. During autoregressive inference, however, tokens are generated sequentially because each newly generated token depends on the previously generated tokens. This difference between training and inference is important from a performance perspective, since it affects how well the computation can be parallelized.
The relevance of the transformer architecture in this work is not primarily its modeling capability, but its computational structure. Transformer inference consists largely of repeated matrix multiplications, vector operations, memory accesses to model weights, and accesses to cached attention data. These operations interact directly with the memory hierarchy and vector units of the CPU. Therefore, understanding inference performance requires both a model-level view of token generation and a hardware-level view of computation, memory bandwidth, and cache behavior.

2.1.1 Transformer Layers

A transformer consists of a stack of layers. Each layer contains two main computational components: a multi-head self-attention mechanism and a feed-forward network [9]. These components are typically combined with residual connections and layer normalization, which improve training stability and allow information to flow through deep networks.

At a high level, the self-attention mechanism allows each token to incorporate information from other tokens in the sequence, while the feed-forward network applies additional transformations independently to each token. This combination allows the model to represent both relationships between tokens and token-wise nonlinear transformations.

From a computational perspective, most of the work in a transformer layer comes from dense linear algebra operations. These include the projections used to compute

query, key, and value vectors, the matrix multiplication used to compute attention scores, and the matrix multiplications in the feed-forward network. Although operations within a layer can often be parallelized, the layers themselves are executed sequentially. Consequently, the total computational cost increases approximately linearly with the number of layers.

2.1.2 Self-Attention

Self-attention is the mechanism that allows a transformer to relate different tokens in a sequence to each other. For each input token, the model computes three vectors: a query vector Q, a key vector K, and a value vector V. The query represents what the token is attending from, the key represents what each token offers for comparison, and the value represents the information passed forward. The scaled dot-product attention operation is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimension of the key vectors. The product QK^T computes similarity scores between queries and keys. These scores are scaled by $\sqrt{d_k}$, normalized using the softmax function, and then used to weight the value vectors. The result is a weighted combination of values, where tokens with higher relevance receive greater attention.

Multi-head attention applies this process multiple times in parallel using different learned projections. The model dimension is split into h heads, each operating on a subspace of dimension $d_h = d_{\text{model}}/h$, where d_{model} is the full model dimension. Each attention head can focus on different relationships between tokens. The outputs of the heads are then combined and projected back into the model dimension [9].

In autoregressive language models, attention is causal. This means that a token may only attend to itself and earlier tokens, not future tokens. Causal masking enforces this restriction during both training and inference.

2.1.3 Feed-Forward Networks

After the attention mechanism, each transformer layer applies a feed-forward network independently to each token. In the original transformer architecture, this network consists of two linear transformations with a nonlinear activation function between them [9]:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where W_1, W_2, b_1, and b_2 are learned parameters. The same feed-forward network is applied to every token position, but each token is processed independently.

Modern LLMs may use different activation functions or feed-forward variants, such as GELU or gated linear units. However, the core computational structure remains

similar: the feed-forward network is dominated by dense matrix multiplications. These matrix multiplications are a major contributor to the total computational work during inference.

2.2 Autoregressive Inference

Large language models generate text using autoregressive inference. In this process, the model predicts one token at a time based on the current context. After a token has been selected, it is appended to the sequence and becomes part of the input for the next prediction. This loop continues until an end-of-sequence token is produced or a maximum generation length is reached.

Inference can be divided into four main stages: tokenization, prefill, decode, and sampling. These stages have different computational properties and therefore different performance characteristics.

2.2.1 Tokenisation

Tokenization is the first stage of inference. It converts raw input text into a sequence of integer token IDs that can be processed by the model. Common tokenization schemes such as Byte-Pair Encoding (BPE) split text into subword units, meaning that a single word may map to one or several tokens [10]. The resulting token sequence serves as the numerical input to the model.

Each token ID is mapped to a vector representation through an embedding table, which is a learned lookup matrix where each row corresponds to one token in the models vocabulary, before being processed by the transformer layers. Tokenization is usually not the main computational bottleneck during inference, but it is required before the model can perform the forward pass.

2.2.2 Prefill

Prefill refers to the initial forward pass over the entire input prompt. During this stage, all prompt tokens are processed, and their key and value representations are computed and stored in the KV-cache [11]. The prefill stage often has higher compute-to-memory ratio than decoding, since many tokens are processed together using large matrix multiplications. This makes prefill more suitable for parallel execution. The output of the prefill stage is a probability distribution over the vocabulary, from which the first generated token is selected. The cost of prefill increases with the length of the input prompt. A longer prompt requires more token representations to be processed and more key-value entries to be stored in the KV-cache.

2.2.3 Decode

Decode is the iterative stage in which new tokens are generated one at a time. At each decode step, the most recently generated token is passed through the model. The model computes the query, key, and value vectors for this new token, while the keys and values of previous tokens are retrieved from the KV-cache [11]. Because decode usually processes only one token at a time, it often has lower operational intensity than prefill. The model must repeatedly access model weights and the growing KV-cache for each generated token. For small batch sizes and long contexts, decode is therefore often more sensitive to memory bandwidth than peak computational throughput. However, the exact bottleneck depends on the model size, quantization format, context length, batch size, hardware, and implementation. Decode is central to inference latency because it is repeated for every generated token. Even if a single decode step is relatively short, the total cost can become large when generating long outputs.

2.2.4 Sampling

Sampling determines which token is selected from the model's output probability distribution. The simplest strategy is greedy decoding, where the token with the highest probability is always selected. While deterministic, greedy decoding can produce repetitive or less diverse outputs. Stochastic methods introduce controlled randomness into token selection. Temperature scaling adjusts the sharpness of the probability distribution. A lower temperature makes the model more deterministic, while a higher temperature increases randomness. Top-k sampling restricts selection to the k most likely tokens, while nucleus sampling, also called top-p sampling, selects from the smallest set of tokens whose cumulative probability exceeds a threshold p [12]. The selected token is appended to the sequence and becomes the input to the next decode step. Sampling is usually less computationally expensive than the transformer forward pass, but it is still part of the inference pipeline.

2.2.5 The KV-Cache

During autoregressive inference, each new token attends to all previously processed tokens. Without optimization, the model would need to recompute the key and value representations for all previous tokens at every decode step, leading to significant redundant computation. To avoid this, modern implementations use a key-value cache, or KV-cache. The KV-cache stores the key and value vectors computed for previous tokens, allowing them to be reused directly in later decode steps [13]. This reduces repeated computation and improves inference latency. However, the KV-cache introduces an additional memory requirement. For each token in the context, key and value vectors must be stored for each layer. The total KV-cache size grows linearly with the context length:

$$\text{KV cache size} = 2 \times L \times H_{kv} \times d_h \times T \times b$$

where L is the number of layers, H_{kv} is the number of key-value heads, d_h is the per-head dimension defined in Section 2.1.2, T is the context length, and b is the

number of bytes used per stored element. The factor 2 accounts for storing both keys and values. As context length increases, the KV-cache can become a significant contributor to total memory usage. In addition to capacity constraints, repeated access to cached keys and values places pressure on the memory hierarchy. Therefore, the KV-cache is central to the performance characteristics of transformer inference, especially for long-context generation. One practical dimension of KV-cache optimization is the numerical precision used to store each element. Lower-precision formats, such as 8-bit or 4-bit representations, reduce memory usage and may allow longer contexts or larger batch sizes within the same memory budget [14]. However, lower precision can introduce quantization error, which may affect attention accuracy and output quality. The choice of KV-cache precision therefore involves a trade-off between memory efficiency and model quality.

2.3 Quantization and Model Formats

The numerical representation of model weights and cached data has a large impact on inference performance. Large language models contain many parameters, and these parameters must be stored, loaded, and used during inference. Reducing the precision of these values can reduce memory usage and memory bandwidth requirements, which is especially important for CPU-based inference.

In this project, the model is executed using `llama.cpp` and stored in the GGUF format defined in section 2.4 below. These are important because they determine how the model is represented on disk, how tensors are loaded into memory, and how inference operations are executed.

Quantization is a model compression technique that reduces the memory footprint and computational requirements of neural networks by representing weights or activations with lower-precision data types, such as 8-bit integers, instead of standard 32-bit or 16-bit floating-point numbers [15, 16]. The core mechanism of linear quantization involves mapping a range of continuous floating-point values to a discrete set of integer values. This mapping is typically defined by two parameters: a scale factor S and a zero-point Z. The scale factor determines the step size between quantized values, while the zero-point aligns the integer representation with the real-valued zero. The dequantization process, which approximates the original real value r from the quantized integer q, can be expressed as:

$$r = S(q - Z)$$

Quantization methods are often divided into two broad categories: quantization-aware training (QAT) and post-training quantization (PTQ) [15, 16]. QAT simulates the effects of quantization during training or fine-tuning, allowing the model to adapt to the reduced precision. This can reduce accuracy degradation, but requires additional training resources. PTQ is applied after a model has already been trained. Because it does not require retraining, PTQ is widely used for deploying large language models. PTQ may quantize only weights or both weights and acti-

vations, depending on the method and inference framework. In the context of CPU inference, quantization is a critical optimization. By reducing the physical size of the weight tensors, quantization can reduce memory traffic during autoregressive decoding. This is especially relevant when the bottleneck is moving model weights through the memory hierarchy rather than performing arithmetic operations. Quantization can also make it possible to run larger models within limited system memory. However, quantization introduces approximation error because fewer bits are used to represent numerical values. The impact on output quality depends on the quantization method, model architecture, and task. Therefore, quantization involves a trade-off between memory efficiency, inference speed, and model quality.

2.4 llama.cpp

`llama.cpp` is an inference framework designed for efficient execution of transformer-based language models on consumer hardware, with a strong focus on local and CPU-based inference [5]. It is built on top of `ggml` [17], a lightweight tensor library that provides the low-level primitives for tensor operations, memory management, and quantized computation. It provides optimized implementations of the core operations used during inference, including matrix multiplication, attention, sampling, and KV-cache management.

In this project, `llama.cpp` is relevant because it provides the runtime environment in which inference is executed and measured. The observed performance is therefore influenced not only by the model architecture, but also by how `llama.cpp` implements the inference pipeline. Several implementation details are important for performance. First, `llama.cpp` supports quantized model formats, which reduce memory usage and memory bandwidth requirements. Second, it uses optimized kernels for different hardware backends. Third, it supports multithreaded CPU execution, allowing inference work to be distributed across multiple CPU cores. Finally, it manages the KV-cache used during autoregressive decoding. Because this work profiles CPU inference, `llama.cpp` is also important from a systems perspective. Its use of threads, vector instructions, memory allocation, and quantized kernels affects the hardware performance counters measured during execution. Therefore, the profiling results reflect the interaction between the model, the selected GGUF [18] format, the `llama.cpp` implementation, and the underlying CPU hardware.

GGUF

GGUF is a binary model file format used to store models for inference with `GGML`-based tools, including `llama.cpp` [18]. It stores the tensors required by the model together with metadata needed for inference, such as architecture information, tokenizer data, tensor names, tensor shapes, and tensor types. The GGUF format is important because it defines how model data is represented and loaded by the inference engine. GGUF stores tensor metadata, including the type used to represent each tensor. This enables `llama.cpp` to load and execute models stored in different quantized formats.

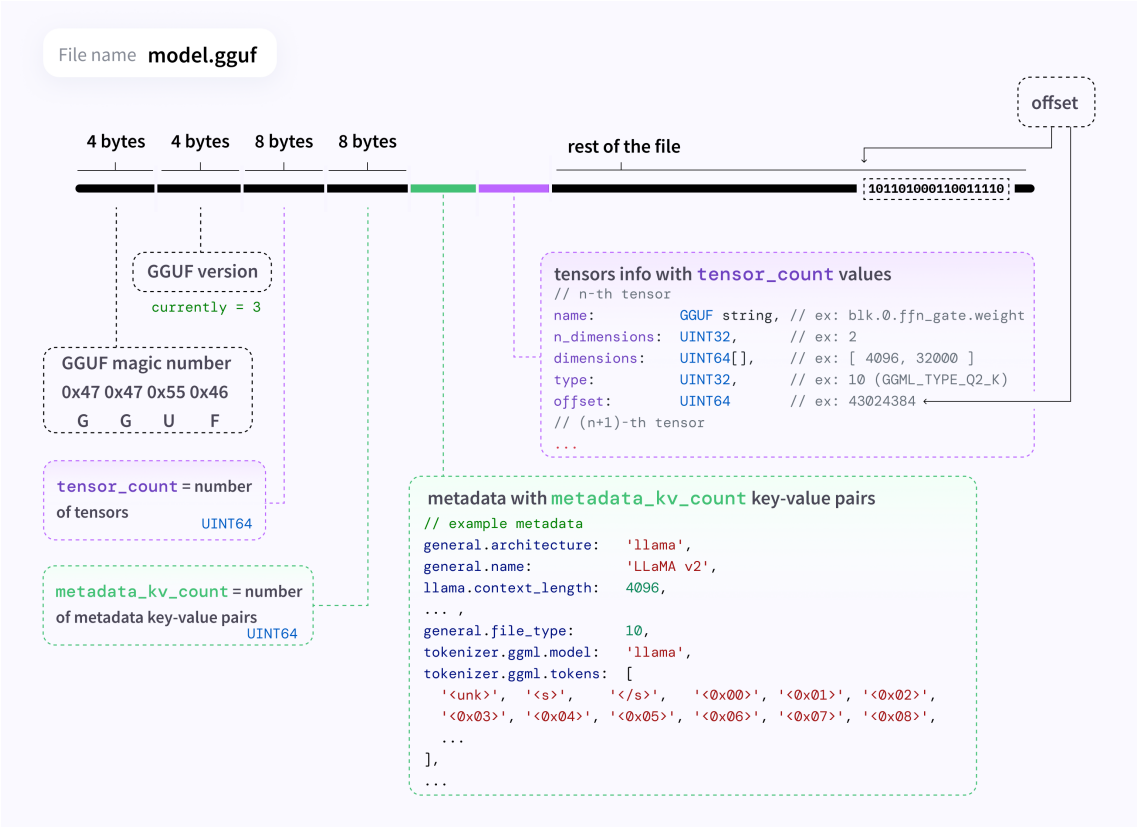

Figure 2.1: Overview of the GGUF file format. Source: Hugging Face Hub documentation [18].

For CPU inference, GGUF quantization is especially relevant. During decode, the model weights are repeatedly accessed for every generated token. If the weights are stored in a smaller quantized format, less data needs to be transferred through the memory hierarchy. This can improve inference speed when memory bandwidth is a limiting factor. However, the quantization format also affects computation. Some quantized formats may require additional operations to dequantize or process values during matrix multiplication. Therefore, the performance impact of a GGUF model depends not only on its size, but also on the efficiency of the kernels used to operate on that quantized format.

In this work, GGUF is relevant because the selected model file determines the stored precision of the weights and therefore influences both memory usage and inference performance.

2.5 CPU Performance Concepts

To understand LLM inference performance on a CPU, it is necessary to consider both computation and memory movement. Transformer inference performs many arithmetic operations, but these operations require data to be loaded from memory. If the processor cannot receive data quickly enough, the compute units may remain partially idle even when the theoretical peak performance is high. This section

introduces the CPU memory hierarchy, memory bandwidth, floating-point throughput, and operational intensity. These concepts are later used to interpret inference performance using the roofline model introduced in section 2.6 below.

2.5.1 CPU Memory Hierarchy

Modern CPUs use a hierarchy of storage levels to reduce the cost of memory access. Registers are the fastest and smallest storage locations. Below registers are several levels of cache, commonly referred to as L1, L2, and L3 cache. Main memory has much larger capacity than cache, but also higher latency and lower bandwidth. Caches improve performance by exploiting locality. Temporal locality means that recently accessed data is likely to be accessed again. Spatial locality means that nearby memory addresses are likely to be accessed together. Programs with good locality can reuse data from cache and avoid expensive main-memory accesses.

2.5.2 Memory Bandwidth

Memory bandwidth is the rate at which data can be transferred between memory and the processor. It is usually measured in bytes per second, such as GB/s. A system with higher memory bandwidth can supply data to the CPU more quickly, which is important for workloads that process large amounts of data.

2.5.3 Floating-Point Throughput

Floating-point throughput describes how many floating-point operations a processor can perform per second. It is commonly measured in **FLOP/s** (floating-point operations per second), a measure of *performance*. This should be distinguished from **FLOPs** (floating-point operations), which measure the amount of *work* performed. In short: FLOPs measures work, FLOP/s measures speed.

The theoretical peak FLOP/s of a CPU can be estimated from the number of cores, the clock frequency, and the number of floating-point operations each core can perform per cycle:

$$\text{Peak FLOP/s} = N_{\text{cores}} \times f \times \text{FLOPs per cycle}$$

where N_{cores} is the number of physical cores, f is the operating frequency, and FLOPs per cycle depends on the supported vector instructions and execution units.

For example, assuming 4 physical cores, an average operating frequency of 3.9 GHz, and an AVX-512 instruction set capable of 32 single-precision FLOPs per cycle [19], the theoretical peak performance is:

$$\text{Peak FLOP/s} = 4 \times 3.9 \times 10^9 \times 32 \approx 499 \text{ GFLOP/s}$$

This represents an optimistic upper bound. It assumes that all cores sustain the assumed frequency and that the vector units are fully utilized. In practice, measured

FLOP/s is often lower due to memory stalls, limited vectorization, instruction mix, synchronization overhead, thermal limits, and other microarchitectural effects.

2.5.4 Operational Intensity

Operational intensity (OI) is defined as the ratio between the number of floating-point operations performed and the amount of data moved from memory:

$$\text{OI} = \frac{\text{FLOPs}}{\text{Bytes}}$$

Operational intensity is measured in FLOPs per byte. It describes how much computation is performed for each byte of data moved. A workload with high operational intensity performs many arithmetic operations per byte loaded from memory. A workload with low operational intensity performs relatively little computation per byte and is more likely to be limited by memory bandwidth. The byte count should represent data movement between the processor and the relevant memory level being modeled. In the standard roofline model, this is often interpreted as traffic between main memory and the processor [8]. However, cache-aware variants may consider traffic between different cache levels. Therefore, the interpretation of operational intensity depends on how memory traffic is measured.

2.6 The Roofline Model

The roofline model is a performance model used to analyze the theoretical performance limits of a system based on computational throughput and memory bandwidth [8]. It relates attainable performance to operational intensity (OI) and provides an upper bound on workload performance for a given hardware platform.

The model is typically visualized as a log-log plot, where the x-axis represents operational intensity in FLOPs per byte and the y-axis represents attainable performance in **FLOP/s**. The model contains two main limits: a memory-bandwidth limit and a peak compute limit. The memory limit forms a sloped line where performance scales with operational intensity, while the compute limit forms a horizontal line corresponding to the processor's maximum arithmetic throughput.

In this work, the roofline model is used to interpret whether measured LLM inference behavior is primarily limited by computation or memory movement. An example is shown in Figure 2.2, where the workload operates below the ridge point and is therefore memory-bound.

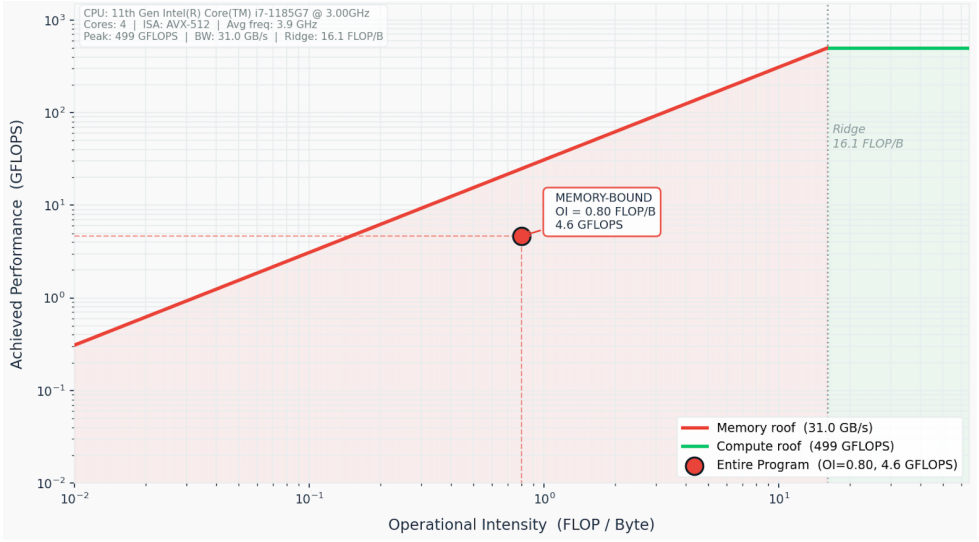

Figure 2.2: Roofline model for a complete inference run using Qwen2.5-1.5B-Instruct (FP8 quantization).

2.6.1 Compute-Bound and Memory-Bound Execution

A workload is memory-bound when performance is limited by memory bandwidth rather than arithmetic throughput. In this region, increasing peak **FLOP/s** does not significantly improve performance because the processor spends time waiting for data movement. The attainable performance is bounded by

$$\text{FLOP/s} \leq \text{Memory Bandwidth} \times \text{OI}$$

A workload becomes compute-bound when it performs enough operations per byte of memory traffic to fully utilize the compute units. Beyond this point, additional memory bandwidth provides limited benefit.

For LLM inference, both regimes can appear depending on the inference phase. Prefill processes many tokens simultaneously using large matrix operations, which typically increases operational intensity. Decode processes tokens sequentially and repeatedly accesses model weights and the KV-cache, making it more sensitive to memory bandwidth.

2.6.2 The Ridge Point

The transition between the memory-bound and compute-bound regions is called the ridge point and is defined as

$$\text{OI}_{\text{ridge}} = \frac{\text{Peak FLOP/s}}{\text{Memory Bandwidth}}$$

Workloads below the ridge point are expected to be memory-bound, while workloads above it are expected to be compute-bound [8]. The ridge point therefore provides a useful threshold for interpreting performance measurements.

2.7 Hardware Performance Counters

Hardware performance counters are special-purpose registers in modern processors that record low-level microarchitectural events during program execution. Examples of such events include executed instructions, CPU cycles, cache references, cache misses, branch mispredictions, and floating-point operations. These counters are useful because they provide insight into how a program interacts with the processor. Instead of measuring only total execution time, performance counters can show whether a program executes many instructions, experiences many cache misses, or fails to use available floating-point throughput efficiently.

In this project, hardware performance counters are used to analyze the behavior of LLM inference. Counters can be used to measure properties such as instruction count, cache behavior, and floating-point activity during different phases of execution. These measurements can then be related to concepts such as operational intensity and the roofline model. However, hardware counters must be interpreted carefully. The available events, event names, and event meanings vary between processor vendors and microarchitectures. Some events correspond directly to architectural behavior, while others approximate lower-level hardware activity. Counter values may also be affected by operating system scheduling, background processes, measurement overhead, and multiplexing.

2.7.1 Counter Multiplexing and Limitations

Modern processors support many different performance events, but only a limited number of hardware counter registers can be active at the same time. A processor may support hundreds of measurable events, while only a small number of physical counters are available per core [20]. To measure more events than there are available counters, profiling tools and operating systems use multiplexing. In multiplexing, different events are measured during different time intervals. The profiling tool then estimates the total count for each event by scaling the measured values according to the fraction of execution time during which the event was active. Multiplexing allows more events to be collected in a single run, but it introduces uncertainty. The scaling process assumes that the measured events are distributed relatively evenly throughout execution. If the program has distinct phases or bursty behavior, this assumption may not hold. For example, prefill and decode have different computational characteristics, so a counter active during only one part of execution may not accurately represent the entire run. This limitation is important for LLM inference profiling because inference consists of multiple phases with different behavior. Measurements based on multiplexed counters should therefore be interpreted with caution, especially when comparing short code regions or phase-specific behavior.

2.7.2 Measurement Overhead

Instrumentation can affect the execution being measured. Calls to profiling libraries such as `PAPI` introduce additional instructions and may disturb cache state, branch

prediction, or scheduling behavior. External tools such as `perf` can also introduce overhead, although they usually require less modification to the measured program. For long-running code regions, measurement overhead is often small relative to the total execution time. For very short regions, however, the overhead can become significant. This is particularly relevant when measuring individual inference steps, where the measured region may be short.

To reduce the impact of measurement overhead, measurements should be repeated and averaged. Very small code regions should be measured with caution, and results should be compared against whole-program measurements when possible.

2.8 Profiling Tools

Profiling tools are used to collect performance information during program execution. In this project, two profiling approaches are relevant: external profiling using Linux `perf`, and code-integrated profiling using `PAPI`. The main difference between these tools is the measurement scope. Linux `perf` can measure whole-program behavior externally, without modifying the source code. `PAPI` is integrated directly into the application and can measure selected regions of code. This makes `perf` useful for validation and broad profiling, while `PAPI` is useful for phase-specific measurements such as prefill and decode.

2.8.1 Linux perf

Linux `perf` is a command-line profiling tool integrated into the Linux kernel through the `perf_events` subsystem [7]. It provides a standardized interface for accessing hardware performance counters, tracepoints, and software performance counters directly from the operating system. One advantage of `perf` is that it operates as an external tool. It can profile an application without requiring modifications, manual instrumentation, or recompilation of the source code. For example, `perf stat` can be used to collect aggregate statistics for an entire program run, such as CPU cycles, instructions, cache misses, and elapsed time.

In this project, `perf` is used as a validation tool. Since it measures the full execution externally, it provides a useful baseline for comparing against measurements collected through code-integrated instrumentation. This helps verify whether the values collected from specific instrumented regions are consistent with the overall behavior of the application. However, `perf` also has limitations. Whole-program measurements may include initialization, model loading, tokenisation, sampling, and other overheads in addition to the inference computation itself. Therefore, `perf` is useful for broad profiling, but may not isolate individual inference phases unless used with more advanced profiling methods.

2.8.2 PAPI

The Performance Application Programming Interface (PAPI) is a C/C++ library that provides a programming interface for accessing hardware performance counters across different processor architectures [6]. Unlike external profilers such as `perf`, `PAPI` is integrated directly into the application's source code. This makes `PAPI` useful for fine-grained measurements. By placing calls such as `PAPI_start()` and `PAPI_stop()` around selected code regions, it is possible to measure specific phases of execution rather than only the whole program. In the context of LLM inference, this can be used to measure prefill, decode, or other selected parts of the inference loop.

`PAPI` provides two main types of events: preset events and native events. Preset events are standardized event names defined by `PAPI`, such as `PAPI_TOT_INS` for total instructions or `PAPI_L3_TCM` for L3 cache misses. These events are intended to be portable across different systems, although their availability still depends on hardware support. Native events correspond more directly to architecture-specific hardware events provided by the processor. They can expose more detailed information, but are less portable. To use `PAPI`, the application must initialize the library, create an event set, add the desired events, and start and stop counting around the region of interest. The resulting counter values can then be used to analyze execution behavior.

In this project, `PAPI` is relevant because it allows performance counters to be collected from specific parts of the inference process. This makes it possible to compare different phases, such as prefill and decode, and relate the measured behavior to memory bandwidth, cache behavior, and computational throughput.

3
Method

3.1 Executables

To instrument different stages of the inference process, four C++ executables were implemented on top of the `llama.cpp` library, each targeting a different level of granularity. All executables share the same underlying inference loop, built around the `llama.cpp` API, and took inspiration from the `simple-chat.cpp` [21] example found in the `llama.cpp` repository [5]. Figure 3.1 shows a simplified version of the shared conversation loop that all executables are built upon. The four executables are:

- **Top View** measures the entire inference session as a single window, capturing high-level metrics such as total runtime, throughput, energy, memory, and KV-cache size.
- **Phase View** breaks the inference process into its four constituent phases (tokenization, prefill, sampling, and decode) and records metrics independently for each.
- **Decoder Block View** records each individual `llama_decode` call as a separate entry, allowing per-token trends such as growing decode latency to be observed.
- **Tensor Operation View** instruments every individual tensor operation in the `ggml` compute backend via an evaluation callback, providing the finest level of granularity.

The decision to implement four separate executables rather than a single monolithic one was deliberate. By isolating each level of granularity into its own executable, the measurement overhead introduced by each set of probes can be assessed independently, and the results across executables can be cross-validated against one another to build confidence in their accuracy. All four executables accept user-specified `PAPI` events as command-line arguments, allowing hardware performance counters available on the host system to be collected alongside the other metrics at every level of granularity. The source code is available at https://github.com/mach-research-lab/profiling-llms-llama-cpp.

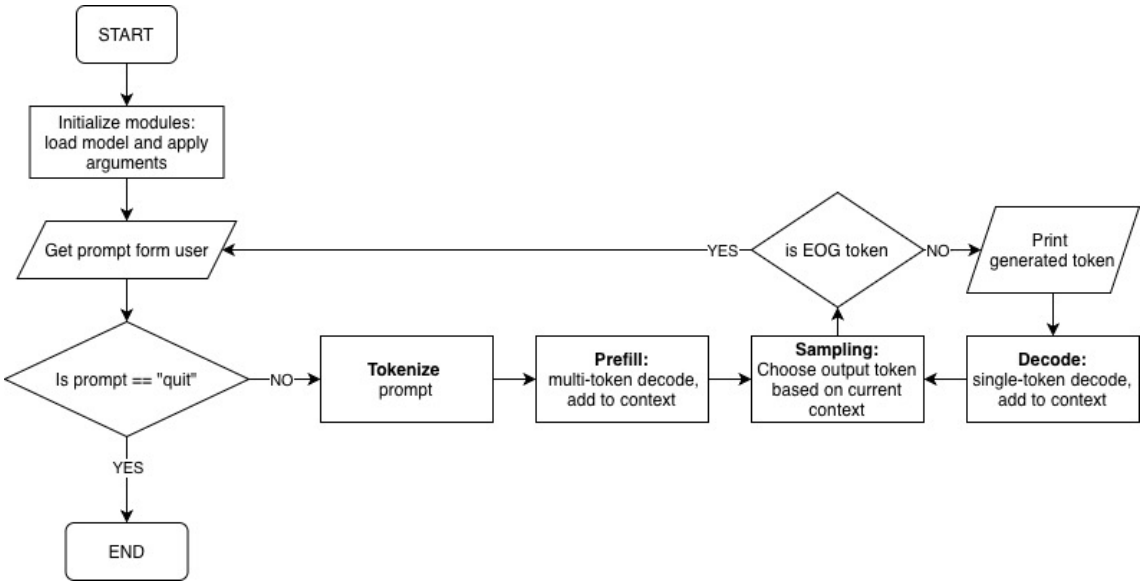

Figure 3.1: Overview of the executable structure. Displays a conversation loop that exits when the user gives the prompt "quit". The generation of new tokens will continue until End of Generation (EOG) token is generated, then the user will be requested to enter a new prompt.

3.1.1 Shared Utilities

Some executables measure the same metrics, such as runtime and energy. A utility header and source file were therefore implemented to keep measurements consistent across executables and to handle other shared functionality.

Time is measured using `clock_gettime` function from the Portable Operating System Interface (POSIX) [22] that uses a consistent clock source, which provides nanosecond resolution and is unaffected by system clock adjustments, making it reliable for measuring durations. This is given by calling the implemented utility function `now_ns()`. Energy consumption is measured through the Linux `perf_event` subsystem, which exposes Intel's Running Average Power Limit (RAPL) [23] hardware counters. Up to three power domains are tracked: the CPU package, the CPU cores, and the full system, though availability depends on the hardware. Raw counter values are read at the start and end of each run and converted to microjoules using a scale factor provided by the kernel, giving a direct measure of the energy consumed during inference.

Beyond measurement, the utility layer also handles argument parsing and output. A shared argument parser processes a common set of command-line flags across all executables, including the selection of `PAPI` events, output file paths, and behavioral flags such as enabling conversation mode, suppressing prints, or providing a list of user prompts to run non-interactively. Unrecognised arguments are forwarded to the underlying `llama.cpp` runtime, preserving its native configuration interface. A custom print wrapper is also provided, allowing terminal output to be selectively disabled via a command-line flag. This is useful when running automated benchmarks

where console output would otherwise interfere with timing or clutter logs.

3.1.2 Top View

The purpose of the **Top View** executable is to measure the whole inference process from start to finish, so measuring starts when the user enters the first prompt and ends when the user gives the prompt "quit". The metrics gathered by the **Top View** executable are:

- Total runtime
- Number of input and output tokens
- Throughput (average token/s)
- Total energy
- Peak RSS memory
- Total model size
- Total KV-cache size
- Average CPU utilization
- User-specified `PAPI` events

All counters are started together before the conversation loop begins and stopped once the conversation ends. Since the executable also supports multi-turn conversations, measurements are paused while waiting for user input and resumed once a new prompt is received, so that idle time is excluded from all results.

Runtime and Throughput

Wall-clock runtime is measured using the shared `now_ns()` utility function, recording timestamps at the start and end of the conversation. When the user is typing, the elapsed waiting time is subtracted from the start timestamp so that only model processing time is reflected in the final duration. Token throughput is then derived by dividing the number of generated output tokens by the total runtime in seconds.

CPU Utilization

CPU utilization is estimated by reading the process CPU time from `/proc/self/stat` at the start and end of the run, which reports the total user and kernel ticks consumed by the process. The difference is divided by the wall-clock runtime and the number of logical CPU cores, giving an average per-core utilization percentage across the entire run.

Energy

Energy is accumulated across all turns using the shared `perf_event` counters. At each point where measurements are paused for user input, the current counter values are read and added to a running accumulator, and the counters are then reset before resuming. This ensures that energy consumed while waiting for input is excluded. The final accumulated raw values are converted to microjoules at the end of the run.

Memory

Peak memory usage is obtained after the conversation ends by reading the `VmPeak` field from `/proc/self/status`, which reports the highest virtual memory footprint reached by the process during its lifetime. Model size is retrieved directly from `llama.cpp`.

KV-Cache

The KV-cache size is estimated from model architecture parameters rather than read directly from an allocator. The number of layers, the number of key-value heads, and the head dimension are queried from the model, and the per-element byte sizes of the key and value cache types are retrieved from the `ggml` type system. Two figures are reported: the estimated size based on tokens actually used during the conversation, and the total capacity allocated for the full context window. The actual serialised size of the KV-cache sequence is also recorded through `llama.cpp`.

Tokens

The total number of input tokens is tracked by accumulating all tokenized prompt tokens across turns into a single vector. Output tokens are counted individually as each one is sampled and accepted during generation.

PAPI Events

`PAPI` counters are started alongside the other measurements and similarly paused and accumulated across turns to exclude user input time. At the end of the run, the accumulated counts for each user-specified event are written to the output alongside the other metrics.

3.1.3 Phase View

While the **Top View** executable treats the entire inference session as a single measurement window, the **Phase View** executable breaks the process down into its four constituent phases: tokenization, prefill, sampling, and decode. Each phase has its own set of measurement probes that are started and stopped around the corresponding `llama.cpp` calls, and the results are accumulated across all turns and tokens before being written to the output. This allows for a more fine-grained picture of where time, energy, and hardware events are spent during inference. The following metrics are gathered independently for each of the four phases:

- Runtime
- Energy consumption
- Per-core CPU utilization
- User-specified `PAPI` events

For each phase, runtime, energy, per-core CPU utilization, and the user-specified `PAPI` events are recorded independently. Runtime is measured using the shared

now_ns() utility, and the elapsed time for each phase invocation is added to a running accumulator in its corresponding measurements struct. Energy is handled the same way: the RAPL counters are reset before each phase, read after it ends, and the raw value is added to the phase's energy accumulator. Per-core CPU utilization is measured by sampling /proc/stat for all logical cores immediately before and after each phase, and the resulting deltas are accumulated across all invocations. This gives a picture of which cores were active during each phase rather than a single process-wide average. PAPI counters follow the same pattern, being started and stopped around each phase and their values accumulated per phase. Since sampling and decode are called once per generated token, their accumulators are updated on every iteration of the token generation loop. The tokenization and prefill accumulators are updated once per conversation turn. At the end of the run, all four phase results are serialised to JSON, with the core utilization broken down by socket, physical core, and logical thread.

3.1.4 Decoder Block View

While the **Phase View** executable accumulates metrics across all invocations of each phase, the **Decoder Block View** executable instead records each individual llama_decode call as a separate entry. This means that every prefill and every single-token decode step is treated as its own decoder block, identified by a block type and an incrementing block ID. This allows for a fine-grained view of how performance and resource usage evolve over the course of a generation, for example how decode latency grows as the KV-cache fills up. The following metrics are gathered for each individual decoder block:

- Runtime
- KV-cache footprint at the time of the block execution
- User-specified PAPI events

Each block is identified by two fields: a block_type field set to either "Prefill" or "Decode", and a block_id that increments after each measurement, starting from zero independently for each type. PAPI counters are reset and started immediately before each llama_decode call and stopped right after, so that each block gets an isolated hardware event reading. Runtime is measured the same way using now_ns(). The KV-cache footprint is read after each decode call by getting the state sequence byte size from llama.cpp, capturing how the cache grows with each new token. Each block is written to a JSON array immediately after it is measured, and if an entry with the same block_type and block_id already exists in the file, for example from a previous batch run with different PAPI events, the new PAPI fields are merged into the existing entry rather than creating a duplicate.

3.1.5 Tensor Operation View

While the previous executables measure at the granularity of inference phases or decoder blocks, the **Tensor Operation View** executable instruments every individual

tensor operation executed by the ggml compute backend during inference. This is achieved by registering an evaluation callback via llama.cpp's cb_eval hook, which ggml calls twice for every tensor operation: once before the operation begins and once after it completes. The following metrics are gathered for each individual tensor operation:

- Runtime
- Tensor size in bytes
- Number of elements in the tensor
- Operation type and tensor name
- User-specified PAPI events

When the callback is invoked with ask = true, signalling that a tensor operation is about to begin, a timestamp is recorded and the PAPI counters are reset. When the callback is invoked again with ask = false, signalling that the operation has completed, the elapsed time is computed, the PAPI counters are read, and the tensor metadata: name, operation type, size in bytes, and number of elements, is extracted directly from the ggml_tensor struct. Figure 3.2 shows a simplified overview of how the tensor operation measurements are gathered. Tensors without a name are skipped. The phase label ("prefill" or "decode") and a token index are tracked in the callback data struct and updated by the main loop before each llama_decode call, so that each record can be associated with the correct inference phase and token position.

Unlike the other executables, the PAPI counters are started once globally at startup and kept running throughout the entire inference session, with PAPI_reset and PAPI_read used inside the callback to obtain per-operation counts without the overhead of repeatedly starting and stopping the counters. Results are written to a CSV file after each operation, with one row per tensor. Optionally, if the -use-db flag is provided, records are accumulated in memory and bulk-inserted into a SQLite database at the end of the run using prepared statements and a single transaction, which significantly reduces insertion overhead compared to per-row commits.

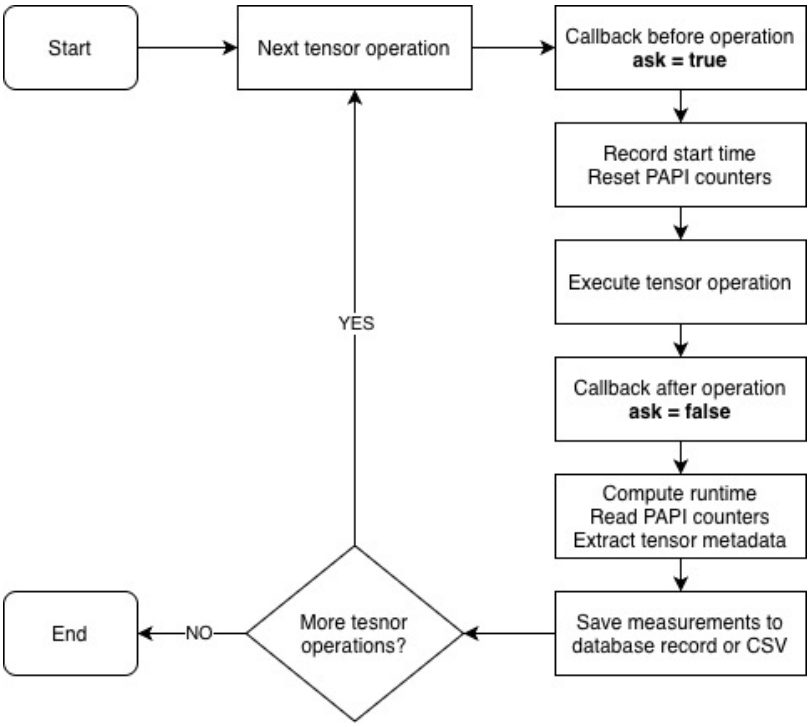

Figure 3.2: Overview of how the **Tensor Operation View** measures tensor operations. Callback function is called with `ask = true` before executing a tensor operation and then called again with `ask = false` once the tensor operation has completed execution.

3.1.6 Repeated Conversations

To support automated and repeatable measurements, all executables accept a list of prompts as a command-line argument. These prompts are loaded into a queue at startup, and the conversation loop consumes them in order as if they were typed interactively by the user. This serves two purposes. First, it allows the same conversation to be replayed across multiple runs, which is necessary because the number of hardware performance counters available on a given processor is limited. Since only a small number of `PAPI` events can be collected simultaneously, the same conversation must be re-executed multiple times with different event selections to gather a full set of hardware metrics. The prompt queue makes this process fully automated. Second, it enables a workflow where the user only interacts with the system once. When running the **Top View** executable interactively, the user's prompts are collected and saved to a JSON file as the conversation progresses. Once the conversation ends, these saved prompts can be passed as arguments to the remaining executables, which then replay the same conversation automatically. This ensures that all executables, across all levels of granularity and all `PAPI` event selections, operate on an identical conversation, making their results directly comparable. To further guarantee that the model produces the same output across all replays, the `--temp` argument is always set to zero, which disables sampling randomness and makes the model's token selection fully deterministic.

3.2 Backend

3.2.1 Database

`SQLite` was used as the database backend for storing tensor-level profiling data. The choice was motivated by the local and experimental nature of the profiling tool. The system does not require a separate database server, concurrent multi-user access, or distributed storage. Instead, profiling runs are executed locally and the resulting data is consumed by the FastAPI server and the React dashboard on the same machine. `SQLite` therefore provides a lightweight storage solution that is simple to deploy, easy to reproduce, and suitable for storing structured profiling results.

The database is primarily used by the **Tensor Operation View** executable, since this profiling mode produces a much larger amount of data than the other views. While top-level, phase-level, and decoder-block measurements can be stored directly as JSON files, tensor-level profiling may generate thousands of records for a single inference run. Storing this data in a relational database makes it possible to query, filter, aggregate, and combine measurements efficiently during post-processing and visualization.

The database schema is organized around two central tables: `event_item` and `event_papi_counter`. The `event_item` table stores one row for each measured tensor operation. Each row contains metadata describing where the operation occurred in the inference process, including the run identifier, inference phase, token index, tensor name, operation type, runtime, tensor size, and number of elements, see Table 3.1. This table represents the structural part of the profiling data. Hardware counter values are stored separately in the `event_papi_counter` table. This table uses a one-to-many relationship with `event_item`, where each tensor operation may have zero or more associated `PAPI` counter values. This design was chosen because different profiling runs may collect different `PAPI` events depending on the selected hardware counters and the limitations of the target processor. Storing `PAPI` events as rows instead of fixed columns avoids changing the database schema when new counters are added, see Table 3.2.

Column	Description
`event_item_id`	Unique identifier for a measured tensor operation
`run_id`	Identifier for the profiling run
`event_item_timestamp`	Timestamp when the event was recorded
`event_phase`	Inference phase, such as `prefill` or `decode`
`event_token_index`	Token index associated with the operation
`event_tensor_name`	Name of the tensor being processed
`event_operation_type`	Type of tensor operation
`event_time_microseconds`	Runtime of the operation in microseconds
`event_size_bytes`	Tensor size in bytes
`event_n_elements`	Number of elements in the tensor

Table 3.1: Columns in the `event_item` table.

Column	Description
`event_item_id`	References a row in `event_item`
`papi_event_name`	Name of the measured `PAPI` event
`papi_value`	Recorded counter value

Table 3.2: Columns in the `event_papi_counter` table.

This schema supports the main analysis operations used by the dashboard. Data can be aggregated by run, phase, token, operation type, tensor name, or transformer layer. By joining `event_item` with `event_papi_counter`, runtime measurements can be combined with hardware counter data such as cache misses, instruction counts, cycle counts, or floating-point operations. This is used for heatmap generation, operation-level bottleneck analysis, and roofline analysis. The database also acts as a persistent intermediate representation between the profiling executables and the visualization layer. Once a profiling run has been stored, the frontend can request different views of the same data without re-running inference. This separation reduces measurement overhead, makes experiments easier to inspect after execution, and allows expensive profiling runs to be reused during later analysis.

3.2.2 Server

The server sits between the instrumented C++ executables and the React frontend, acting as the single entry point for all data flowing into the dashboard. It is implemented in Python using FastAPI and exposes a REST API. Through this API, the frontend can trigger inference runs and multi-run profiling, as well as retrieve the resulting profiling data.

Two responsibilities are separated within the server. The first is orchestration: the server launches the instrumented executables with user-specified `PAPI` events and model parameters, captures their output, and persists the resulting counter data into the `SQLite` database and JSON files. The second is data access: subsequent requests from the frontend are served directly from the database and JSON files, decoupling visualization from execution so that previously recorded runs can be inspected without re-running the executables. This design keeps the C++ profiling layer and the frontend independent of each other. The executables only need to write their output in a format the server understands, and the frontend only needs to consume a stable JSON API, which makes it straightforward to extend either side in isolation.

3.2.3 Utility Scripts

To support the data collection pipeline, three Python utility modules were implemented to handle event grouping, executable orchestration, and post-processing of the collected results. The **event retriever** partitions the requested `PAPI` events into valid groups collectible in a single run using bin packing and hardware validation, yielding a minimal set of executable invocations. The **run handler** orchestrates

these invocations via `subprocess`, ensuring all batches replay an identical conversation and produce deterministic output. Finally, **JSON complementation** enriches the collected data with derived metrics such as IPC, FLOP/s, operational intensity, and per-component breakdowns that require data from multiple sources to compute. Detailed description of each module are given in Appendix F.

3.3 Analysis

3.3.1 Roofline

The roofline model requires two hardware characteristics: the peak floating-point throughput of the processor and the maximum memory bandwidth of the system.

Peak FLOP/s

The theoretical peak throughput is estimated from the processor specifications. Assuming AVX-512 support, capable of executing 32 single-precision floating-point operations per cycle, the peak performance is computed as

$$\text{Peak FLOP/s} = N_{\text{cores}} \times \frac{f_{\text{base}} + f_{\text{max}}}{2} \times 32$$

where the average frequency approximates sustained operating frequency under load.

Memory bandwidth

Maximum memory bandwidth is measured using the `STREAM` benchmark [24], which reports sustained memory bandwidth using sequential access patterns. The measured Copy bandwidth is used as the memory roof in the roofline model.

Operational intensity

Operational intensity (OI) is defined as the ratio of floating-point operations to bytes transferred from memory:

$$\text{OI} = \frac{\text{FLOPs}}{\text{Bytes moved}}$$

Floating-point operations are obtained from the `PAPI_FP_OPS` event. Since `PAPI` does not expose a direct DRAM traffic counter, memory traffic is approximated using the number of L3 cache misses given by the `PAPI_L3_TCM` event:

$$\text{Bytes moved} \approx \texttt{PAPI\_L3\_TCM} \times 64$$

assuming a cache line size of 64 bytes. This approximation does not capture all memory traffic sources, such as write-back traffic and unused prefetched cache lines.

Measurement granularity

The implementation supports roofline analysis at four levels of granularity: the full inference run, the prefill and decode phases, individual decoder blocks, and individual transformer layers. Prefill and decode exhibit different characteristics: prefill processes all prompt tokens in a batch and typically achieves higher operational intensity, while decode generates tokens sequentially and is generally more memory-bound.

3.3.2 Heatmap

The heatmap visualizations are generated from profiling data collected by the **Tensor Operation View** executable. Six heatmap variants were implemented, divided into layer-level heatmaps and phase comparison heatmaps, where the displayed metrics are execution time, memory pressure, and instructions per cycle (IPC).

The layer-level heatmaps show how a chosen metric is distributed across operation types and transformer layers. Each row corresponds to an operation type such as `MUL_MAT` or `FLASH_ATTN_EXT`, while each column represents a transformer layer inferred from the tensor name. Tensors that cannot be matched to numbered layers are mapped to synthetic labels such as `embd` or `final`.Memory pressure is represented using L3 cache misses measured through the `PAPI_L3_TCM` event.

The phase comparison heatmaps aggregate the same metrics separately for the prefill and decode phases, together with a delta column showing the signed difference between phases. These heatmaps are constructed from the JSON produced by the **Phase View** when available, otherwise derived directly from the `SQLite` database.

Each heatmap matrix is constructed by grouping measurements by operation type and layer or phase, followed by aggregation of the chosen metric. IPC is computed after aggregation as:

$$IPC = \frac{\sum \text{instructions}}{\sum \text{cycles}}$$

Instructions and cycles are obtained from the `PAPI_TOT_INS` and `PAPI_TOT_CYC` events, respectively. This computation ensures that IPC reflects the aggregated execution behavior of each operation group rather than individual short-running events.

Runtime and memory heatmaps use logarithmic scaling due to the large variation in operation cost, while phase comparison heatmaps use symmetric logarithmic scaling to visualize both positive and negative delta values. IPC heatmaps use linear or diverging scales because IPC values have a smaller dynamic range. To improve readability, the visualizations are limited to the operation types with the highest total contribution to the selected metric.

3.4 Frontend

3.4.1 Terminal Interface

The terminal interface is implemented as a Python script and serves as the primary command-line entry point for configuring and launching profiling runs. Rather than exposing raw command-line flags directly to the user, it provides a guided, menu-driven experience that walks through each configuration step in sequence: selecting a run type, choosing a storage backend, picking a model, specifying `PAPI` events, and setting a prompt and token budget. Each choice is confirmed interactively before proceeding, reducing the risk of misconfigured runs.

3.4.2 Graphical Interface

The user interface was developed in three stages, combining iterative design with AI-assisted scaffolding and manual refinement. The **key** design-problem to solve is presenting a large swath of statistics and data to the user in an intuitive manner, without overloading the user. An important aspect to tackle the design process was the definition of a navigation structure that reduces the intensity of the statistics that are presented to the user.

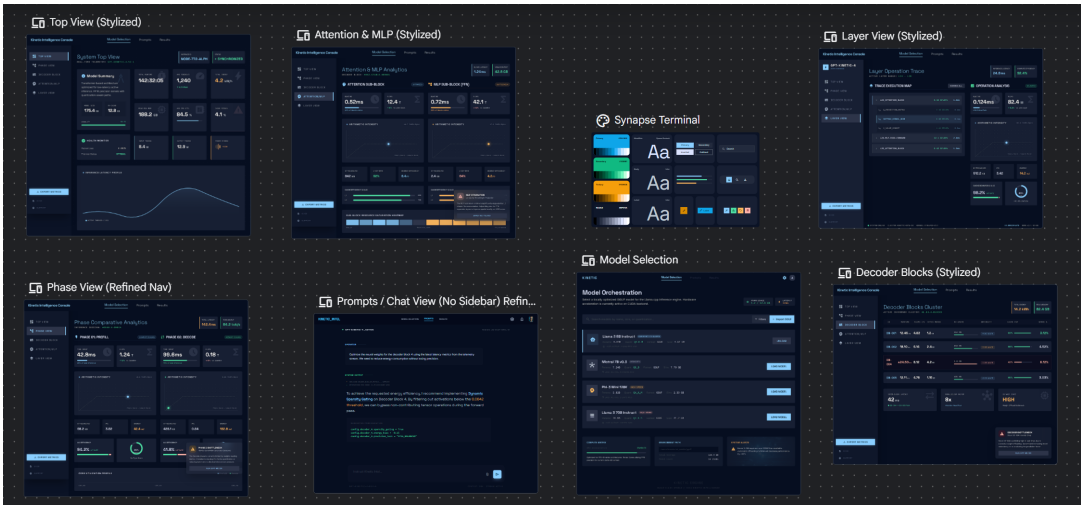

Figure 3.3: AI Assisted prototyping through Stitch [25]

In the first stage, low-fidelity wireframes were produced using Stitch [25] to explore the layout and information hierarchy of the tool. The iterations focused on translating the hierarchical structure of performance metrics into a coherent dashboard interface spanning multiple levels of abstraction, from a global model overview to fine-grained layer-level analysis. Multiple variants were explored, including linear drill-down flows and multi-panel dashboards.

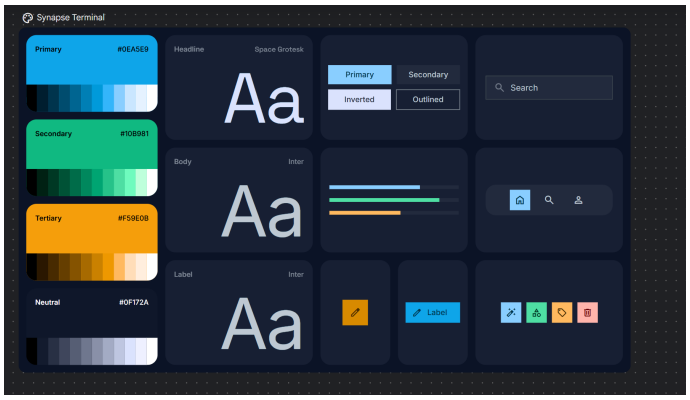

Figure 3.4: Color Scheme - Stitch [25]

Figure 3.5: Metric Prototypes [25]

In parallel, the arrangement of information within each view was iteratively refined. Early wireframes explored different grid layouts and card-based groupings to present key metrics such as runtime, FLOPs, memory usage, and energy consumption. Particular attention was given to the Top View dashboard, where multiple configurations were evaluated to balance information density with readability. These included variations in card size, grouping, and the visual emphasis of critical values. Additional iterations addressed the consistent representation of metric categories across all views. Various visual encoding strategies, including grouping, labeling, and color accents, were explored to maintain consistency and detail.

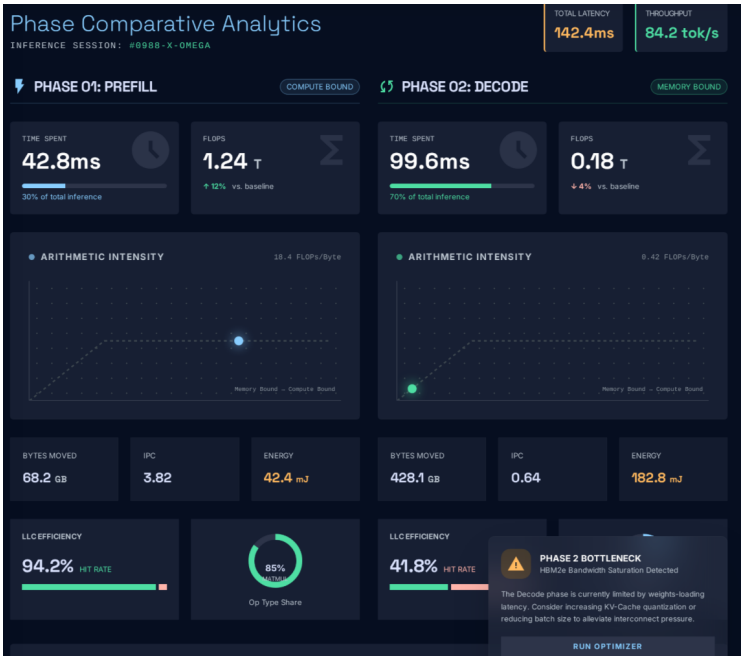

Figure 3.6: Phase View - Stitch [25]

For the Phase View, iterations focused on comparative layouts enabling direct juxtaposition of prefill and decode phases. Different designs were evaluated for presenting phase-specific metrics side-by-side and for integrating analytical visualizations such

as operational intensity plots and bottleneck indicators. These design choices support key analytical tasks, including identifying whether execution is compute- or memory-bound and comparing the relative cost of each phase.

In the second stage, an initial code skeleton for the web application was generated using Google Gemini. The model was prompted with the selected wireframe and a description of the intended functionality, producing a static React scaffold. This output provided a structural starting point but lacked routing, state management, and backend integration, and was therefore treated as a foundation rather than a complete implementation.

In the third stage, the scaffold was extended and refined. During this stage, several components were implemented or substantially restructured, including:

- A controller layer responsible for mediating communication between the user interface and the backend;
- A large set of dynamic UI components to improve modularity and encapsulation of reusable UI elements;
- A database adapter abstracting queries to the data store;
- A coherent user flow that provides clear navigation by enforcing a certain linear traversal while providing clear pliancy;
- Visual styling was altered to ensure consistency with the different main result sections;
- A dynamic unit conversion module, that presents the data from the database in the most appropriate unit visually.
- Real-time decimal precision control to the user.
- A heatmap presentation view, that presents the heatmap data that was constructed earlier,
- Modular visualization components to introduce stronger encapsulation and increase extendability.

This third stage of development revolved around intensive iteration, the developed UI and user journey were then discussed, and further refined. This to compound the accumulated improvements from each iteration.

3.5 Testing

All tests and example runs were performed on the machine specified in Table 3.3. The machine runs Ubuntu 24.04.4 LTS on an Intel Core i7-1185G7 processor with 4 cores, 8 threads, and 31 GiB of RAM.

Computer	
Model	Dell Latitude 5520
OS	Ubuntu 24.04.4 LTS
Kernel	Linux 6.17.0-14-generic (x86-64)
Processor	
Model	Intel Core i7-1185G7 (11th Gen)
Base / Boost	3.00 GHz / 4.80 GHz
Cores / Threads	4 cores / 8 threads
L1 Cache	320 KiB
L2 Cache	5 MiB
L3 Cache	12 MiB
Memory	
RAM	31 GiB
Swap	8.0 GiB
Storage	
Drive	Toshiba XG6 NVMe SSD
Total Capacity	953.9 GiB
Root (/)	937 GiB
EFI Partition	1.1 GiB

Table 3.3: Test machine specifications

3.5.1 Integration Test Suite

The profiling tool was tested using an automated `pytest` test suite. Since the system depends on compiled C++ executables, `PAPI` hardware counters, energy measurements, and generated output files, the tests are primarily integration tests rather than isolated unit tests. Each test runs the actual profiling binaries and verifies that the produced measurements are structurally valid and numerically reasonable. A shared multibatch baseline is used as the main reference point. The baseline is generated using the same model, prompt, token count, cache configuration, and deterministic sampling settings as the individual tests. The prompt is fixed to `Hello`, the number of generated tokens is fixed to five, and the temperature is set to zero. This makes repeated profiling runs comparable by ensuring that the same workload is executed.

The **Top View** tests verify whole-inference measurements. The tests check that the output contains required fields such as runtime, generated tokens, token throughput, peak memory usage, KV-cache size, energy, and the selected `PAPI` counter. Basic sanity checks ensure that runtime, throughput, and memory usage are positive, and that the number of generated tokens matches the configured value.

The **Phase View** tests verify that measurements are separated correctly between the prefill and decode phases. Each phase must be present in the output and contain

runtime and PAPI data. The total energy and cache-miss counts across phases are compared against the multibatch baseline within predefined bounds.

The **Decoder Block View** tests verify measurements at decoder-block granularity. The output must contain both prefill and decode block entries. Each block is checked for required fields, positive runtime, non-negative KV-cache footprint, and contiguous block identifiers. The summed runtime and cache-miss values are then compared against the baseline.

Additional PAPI interference tests evaluate whether enabling more hardware counters changes the measured results. The same inference workload is executed with increasing numbers of active PAPI events. The tests compare PAPI_L1_DCM, runtime, tensor size, and number of tensor elements across runs. Tensor size and element count act as control values, since they should remain identical if the same tensor operations are measured. Exact equality is not expected, because low-level profiling is affected by system noise, CPU scheduling, cache state, thermal behaviour, and measurement overhead. The tests therefore use tolerance-based validation. This allows normal variation between runs while still detecting missing fields, invalid output formats, incorrect token counts, excessive counter divergence, and large regressions in the profiling pipeline.

3.5.2 Cross-view measurement validation

To validate the consistency of the instrumentation strategy across different profiling granularities, measurements collected from the **Phase View**, **Decoder Block View**, and **Tensor Operation View** were compared against the coarser **Top View**, which was used as the reference baseline. For each finer-grained view, the recorded metric values were aggregated by summing all measured regions across the full inference execution. The resulting totals were then compared to the corresponding **Top View** measurement using a ratio defined as the **Top View** value divided by the summed value from the other view, where a ratio of 1.0 indicates perfect agreement. Validation was performed using both runtime measurements and PAPI hardware counters under single-event and multi-event configurations in order to evaluate the impact of instrumentation overhead and potential counter multiplexing effects. All experiments were executed using identical inference settings and model configurations across all instrumentation views. Detailed validation results are presented in Appendix B.

4
Results

4.1 Overview of Tool Output

The tool produces output at four levels of granularity, each corresponding to one of the instrumented executables: **Top View**, **Phase View**, **Decoder Block View** and the **Tensor Operation View**. To ensure all views are directly comparable, the tool records the user's prompts during the first run and replays them automatically with deterministic sampling across all subsequent invocations, guaranteeing that every view operates on an identical conversation. To present results from each view, a test run with the tool was performed to produce the output data expressed in Sections 4.1.1, 4.1.2, 4.1.3 and 4.1.4. The machine specifications used for the test run are shown in Table 3.3 with the profiling configuration shown in Table 4.1.

Configuration	
Model	qwen2.5-1.5b-instruct-q4_0.gguf
Prompts	"Explain how LLMs work!","Tell me more","quit"
Generated token limit / answer	100
K-cache datatype	fp16
V-cache datatype	fp16
Events	Default
Event group size	Unlimited

Table 4.1: Run configuration, for default events see Appendix D. *Generated token limit / answer* is the limit of tokens generated for each answer produced by the model given a single prompt.

4.1.1 Top View

The **Top View** captures a single aggregate measurement over the entire inference session, reporting total runtime, token throughput, peak memory, energy consumption, KV-cache size, CPU utilization, and user-specified `PAPI` counters. Measured metrics are saved to a JSON file. See Table 4.2 for **Top View** output with specified setup shown in Table 3.3 and 4.1.

Top View Result	
L1 cache misses	912918873
L2 cache misses	793021765
L3 cache misses	628317812
Average CPU utilization	49.97 %
Core energy	183.52 (J)
Package energy	201.41 (J)
System energy	286.54 (J)
Generation	
Generated tokens	200
Token throughput	38.40 (tokens/s)
Total tokens in context	227
Runtime	5207.74 (ms)
Memory	
Model size	1011.16 (MB)
Peak RSS	3184.93 (MB)
KV-cache capacity	896.00 (MB)
KV-cache used	6.21 (MB)
Estimated KV-cache size	6.21 (MB)
KV-cache token capacity	32768 (tokens)
KV-cache tokens stored	227

Table 4.2: Output from **Top View**.

4.1.2 Phase View

The **Phase View** reports runtime, energy, per-core CPU utilization, `PAPI` counters and operation data independently for each of the four inference phases: tokenization, prefill, sampling, and decode. Measured metrics are saved to a JSON file which is later enhanced with operation data collected from the **Tensor Operation View**. See Table 4.3 for **Phase View** output that presents the result for the prefill phase with the specified setup shown in Table 3.3 and 4.1.

Phase View Result (Prefill phase)	
Runtime	176.39 (ms)
Last level cache hits	5112042
Last level cache misses	3769444
Last level cache miss rate	42.44 %
FLOPs	943292182
Total cycles	677830430
Total instructions	1486827289
IPC	2.19 (instructions / cycle)
Throughput	5.35 (GFLOPs/s)
Bytes moved	241.24 (MB)
Operational intensity	3.91 (FLOPs / byte)
Average core utilization	49.22 %
Average package power	47.58 (W)
Core energy	7.96 (J)
Package energy	8.39 (J)
System energy	10.90 (J)
Core utilization socket 0: Core 0	
Thread 0	47.06 %
Thread 4	52.94 %
Core utilization socket 0: Core 1	
Thread 1	43.75 %
Thread 5	33.33 %
Core utilization socket 0: Core 2	
Thread 2	0.00 %
Thread 6	100.00 %
Core utilization socket 0: Core 3	
Thread 3	16.67 %
Thread 7	100.00 %
Operation data: MUL_MAT	
Operation count	394
Share of total operations	20.54 %
Total runtime	334646 (μs)
Share of total runtime	92.57 %
Average runtime	849.36 (μs)
Operational intensity ratio	0.25
Total bytes transferred	108.99 (MB)
Total elements processed	27246336
Total instructions	2015616058
Total cycles	932008194
IPC	2.16 (instructions / cycle)
Bytes moved	219.56 (MB)
Operation data: (Remaining operations)	

Table 4.3: Output from **Phase View**. This table only showcases the results given for one of the four phases: prefill. The operation data contains all tensor operations performed and named, summarized to present metrics during each phase.

4.1.3 Decoder Block View

The **Decoder Block View** records every individual `llama_decode` call as a separate entry, capturing runtime, KV-cache footprint, and `PAPI` counters for each block. Measured metrics are saved to a JSON file which is later enhanced with sub compo-

nent data collected from the **Tensor Operation View**. See Table 4.4 for **Decoder Block View** output that presents the result of one specific decoder block with the specified setup shown in Table 3.3 and 4.1.

Decoder Block View Result (Decode, Block 0)	
Runtime	25.79 (ms)
Runtime share	0.367 %
FLOPs	44225001
FLOPs/s	1.715e9
IPC	1.9941
Operational intensity	0.239 (FLOPs/byte)
Bytes moved	185254656 (bytes)
KV cache footprint	488316 (bytes)
Cache Behavior	
L1 misses	4300554
L2 misses	3835883
L3 accesses	3835883
L3 hits	941279
L3 misses	2894604
L3 hit rate	24.54 %
L3 miss rate	75.46 %
Subcomponent: Attention	
Runtime	60097 (μs)
FLOPs	95836955
Bytes moved	82697856 (bytes)
Operational intensity	1.159 (FLOPs/byte)
IPC	1.5087
L3 accesses	2928239
L3 misses	1292154
L3 miss rate	44.13 %
Subcomponent: MLP	
Runtime	158027 (μs)
FLOPs	238193280
Bytes moved	711026176 (bytes)
Operational intensity	0.335 (FLOPs/byte)
IPC	2.3842
L3 accesses	18683338
L3 misses	11109784
L3 miss rate	59.46 %

Table 4.4: Output from **Decoder Block View**. Decoder Block 0 is shown, exact same metric types are given for each prefill block and other decode blocks.

4.1.4 Tensor Operation View

Tensor Operation View instruments every tensor operation executed by the `ggml` backend, recording runtime, tensor size, element count, operation type, and `PAPI` counter values per operation. The **Tensor Operation View** stores its measurements in a `SQLite` database given the volume of per-tensor data produced. See Table 4.5 for **Tensor Operation View** output that presents the result of one specific tensor operation with the specified setup shown in Table 3.3 and 4.1.

Tensor Operation View Result (prefill, embd, GET_ROWS)	
Phase	prefill
Token index	1
Tensor name	embd
Operation type	GET_ROWS
Time	92 (μs)
Size	196608 (bytes)
Elements	49152
FLOPs	50688
Cache Counters	
L1 data cache misses	1454
L1 instruction cache misses	1428
L1 total cache misses	2882
L2 data cache accesses	2358
L2 data cache misses	1751
L2 data cache reads	762
L2 instruction cache misses	675
L2 load misses	297
L2 total cache accesses	3682
L2 total cache misses	2209
L3 data cache accesses	1751
L3 data cache reads	309
L3 load misses	109
L3 total cache accesses	2328
Cycle & Instruction Counters	
Total cycles	159594
Total instructions	152645

Table 4.5: Output from **Tensor operation view**. This showcases the metrics gathered for the tensor operation `embd / GET_ROWS` during prefill phase for the first token.

4.2 Data Structuring and Storage

Separating summary-level analysis from low-level event data, all outputs from the full measurement workflow are written to the `run_every_view_results` directory. The phase-level view writes to `phase_view.json`, the top-level view writes to `top_view.json`, the decoder-block view writes to `decoder_block_view.json`, and the tensor-operation view writes to `tensor_op_view.db`. Using fixed filenames makes the later analysis scripts deterministic, since they can load the expected files directly from the results directory. The JSON files contain aggregated measurements enriched during post-processing with derived fields such as IPC, FLOPs per second, LLC miss rate, estimated bytes moved, operational intensity, and average power. The JSON summaries can additionally be enriched with operation-level statistics queried from the `SQLite` database, allowing high-level summaries to include selected tensor-operation details while the complete raw data remains in the database for deeper analysis and reproducibility.

Database Schema

The database consists of two main tables: `event_item` and `event_papi_counter`. The `event_item` table stores one row for each measured tensor operation during inference. Each row contains metadata such as the run identifier, inference phase, token index, tensor name, operation type, execution time, tensor size, and number of tensor elements.

Column	Description
`event_item_id`	Unique identifier for a measured tensor operation
`run_id`	Identifier for the profiling run
`event_item_timestamp`	Timestamp when the event was recorded
`event_phase`	Inference phase, either `prefill` or `decode`
`event_token_index`	Token index associated with the operation
`event_tensor_name`	Name of the tensor being processed
`event_operation_type`	Type of tensor operation
`event_time_microseconds`	Runtime of the operation in microseconds
`event_size_bytes`	Tensor size in bytes
`event_n_elements`	Number of elements in the tensor

Table 4.6: Columns in the `event_item` table.

The `event_papi_counter` table stores hardware performance counter values associated with tensor operations. Since different profiling runs may collect different `PAPI` events, the counters are stored in a separate table instead of being fixed columns in `event_item`.

Column	Description
`event_item_id`	References a row in `event_item`
`papi_event_name`	Name of the measured `PAPI` event
`papi_value`	Recorded counter value

Table 4.7: Columns in the `event_papi_counter` table.

The relationship between the two tables is one-to-many: one tensor operation in `event_item` may have several associated `PAPI` counter values in `event_papi_counter`. For example, a single operation may have values for counters such as `PAPI_L1_DCM`, `PAPI_L2_DCM`, `PAPI_L3_TCM`, or `PAPI_FP_OPS`, depending on which counters were enabled.

This schema makes it possible to aggregate profiling data by run, phase, token, tensor name, or operation type. By joining the two tables, runtime measurements can also be analysed together with hardware counter data. This is used to study cache behaviour, operation-level bottlenecks, and values needed for later roofline analysis, such as floating-point operations and estimated memory traffic.

4.3 Visualization and Dashboard

All functionality is accessible through two interfaces. The terminal interface provides a guided, menu-driven experience for configuring and launching profiling runs from the command line, providing the user with measurment results through JSON and CSV files. The graphical interface is a React-based dashboard that presents the collected results interactively through various methods of visualization, allowing the user to navigate and explore the profiling data through various perspectives to maximize the users internalization of the data, without re-running inference.

4.3.1 General Attributes of the Graphical Interface

The nature of the analysis tool, creates a need to minimize the numerical intensity of the presentation of statistics to the user. And the associated cognitive load needed from the user to effectively process their results.

To reduce the required cognitive load by the user the following strategies were used:

1. Guide the user through the model selection and inference settings
2. Create a user journey where the user is slowly introduced to the depth of data
3. Rely on intuitive visualizations for as much data presentation as possible.

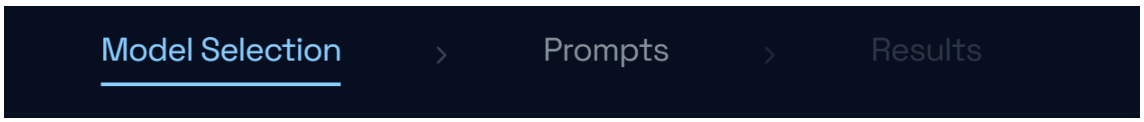

Figure 4.1: A figure showing the application's breadcrumb navigation path, illustrating the user's current location and next steps for inference set-up

Figure 4.1 shows the "breadcrumbs" of the tool. This is a method to show the path that the user is intended to follow during inference set-up. By using breadcrumbs in the user-interface, design strategy 1 is partially implemented.

Figure 4.2: A figure showing vertical navigation, with menu items

Figure 4.2 shows the second navigational method that the application uses. The "Sidebar" is used to navigate through the presented statistics after inference. This is to implement design strategy 2.

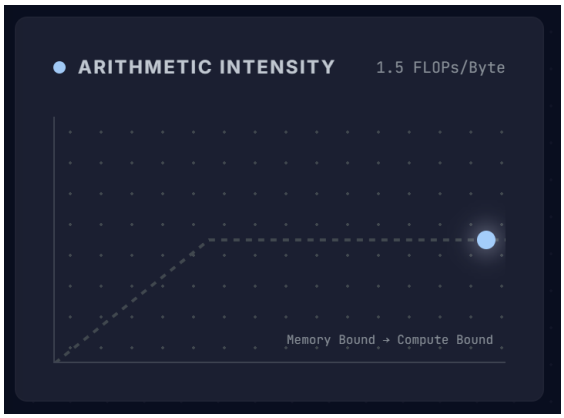

Figure 4.3: Figure showing the GUI presentation of the roofline-mode

Figure 4.4: Figure showing example of how the heatmaps look on the GUI

The "Roofline" is one of the major visualization methods that are used in the interface, it along with the "Heatmap" make up a significant portion of the vizualization methods. These components are used in multiple views on the dashboard and contribute to a less intimidating experience in combination with rawer numerical data, while intuitively conveying data to the user, thus implementing design strategy 3.

4.3.2 User Journey

This section describes the users path through the graphical interface, and the limitations the interface imposes

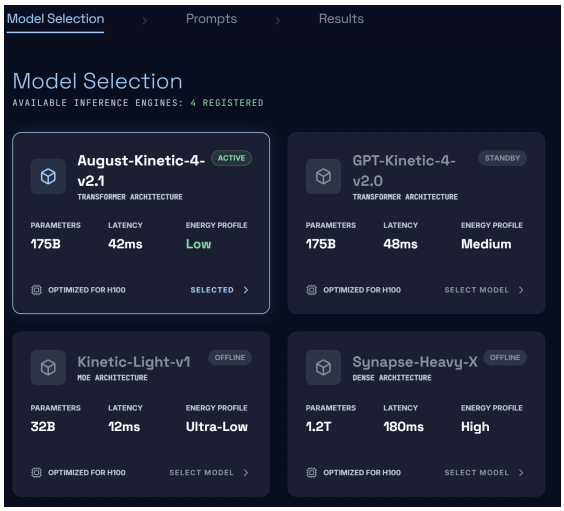

Figure 4.5: Figure showing the model selection page

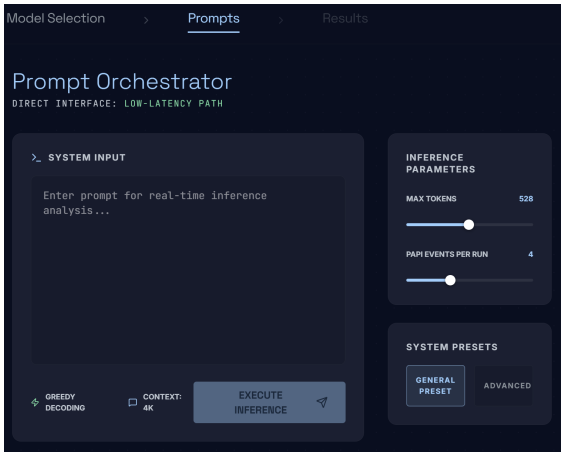

Figure 4.6: Figure showing the prompt page where the user alters inference settings and send prompts

Figure 4.5 shows the initial homepage for the tool. Here the user is presented with their first choice: The model to run the analysis on. This page directly communicates the core functionality of the entire tool through the offered alternatives. When a model is selected, the user is prompted to proceed to the next page, shown in Figure 4.6.

The Prompt page offers more analysis alternatives to the user. Here the user is slowly introduce to more choices. In the bottom right container there are alternatives for system presets. "General Preset" is initially selected to hide further alternatives to the user in terms of specific "papi events" to run during the analysis. This is to reduce cognitive load for users.

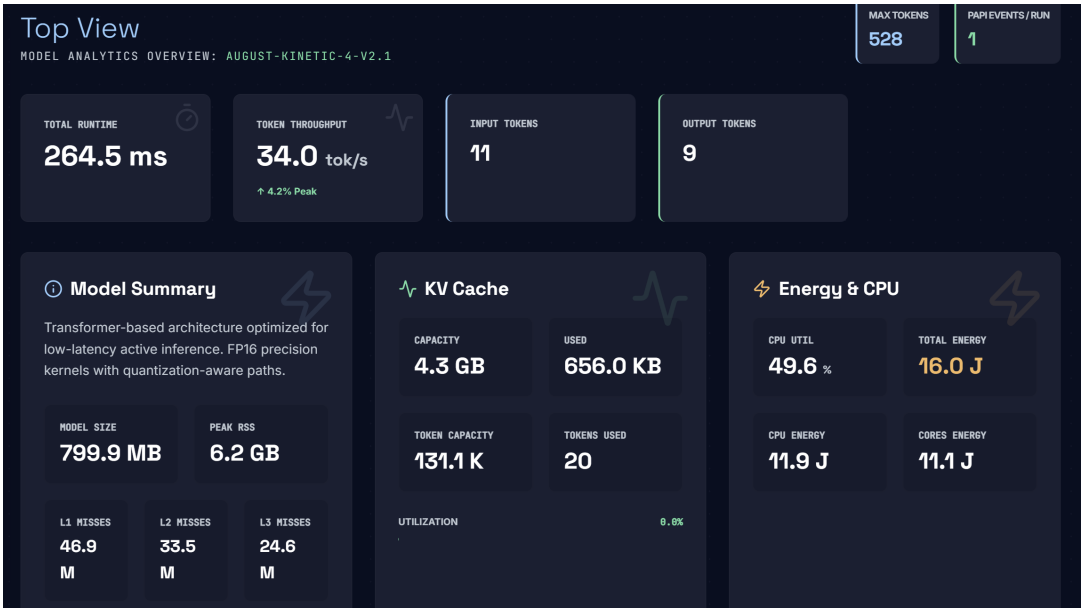

Figure 4.7: Figure showing Top View where general metrics about the inference

After the inference has been completed and all data has been gathered from the users queries, the user is then prompted to continue to the Results page. At this point the user has completed the inference set-up and is ready to view the data. The user is slowly introduced to the depth of data by initially showing the overview of the data, the "Top View" shown in Figure 4.7. The placement of the tabs in the sidebar are indicative of the order of depth into the analysis.

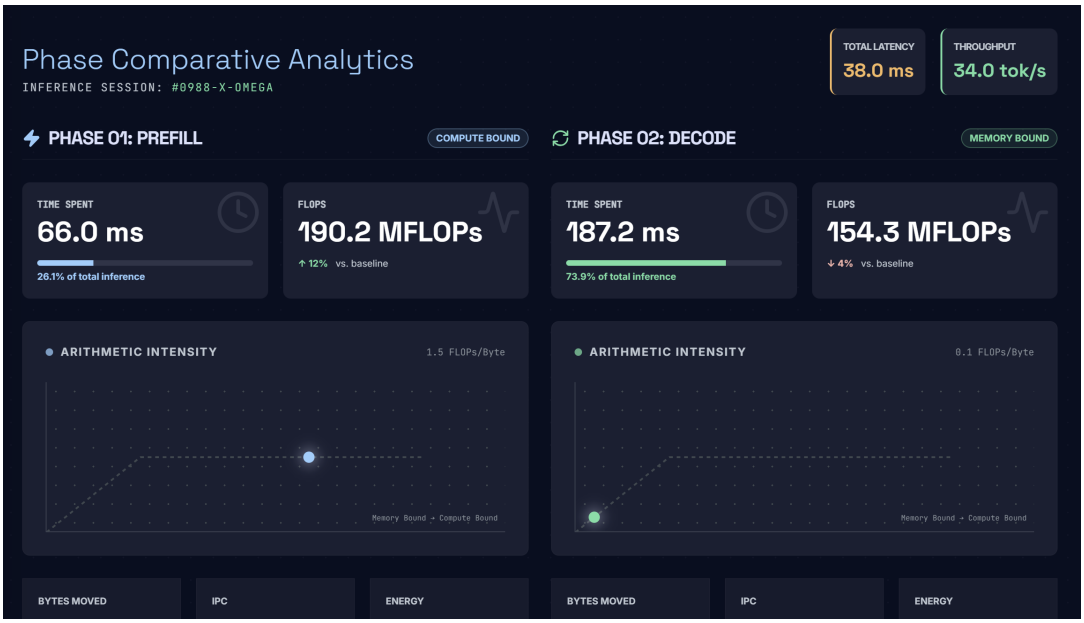

Figure 4.8: Figure showing the Phase View where the phases are compared

The following tab, the "Phase View", provides slightly more depth by comparing phases. Shown in Figure 4.8.

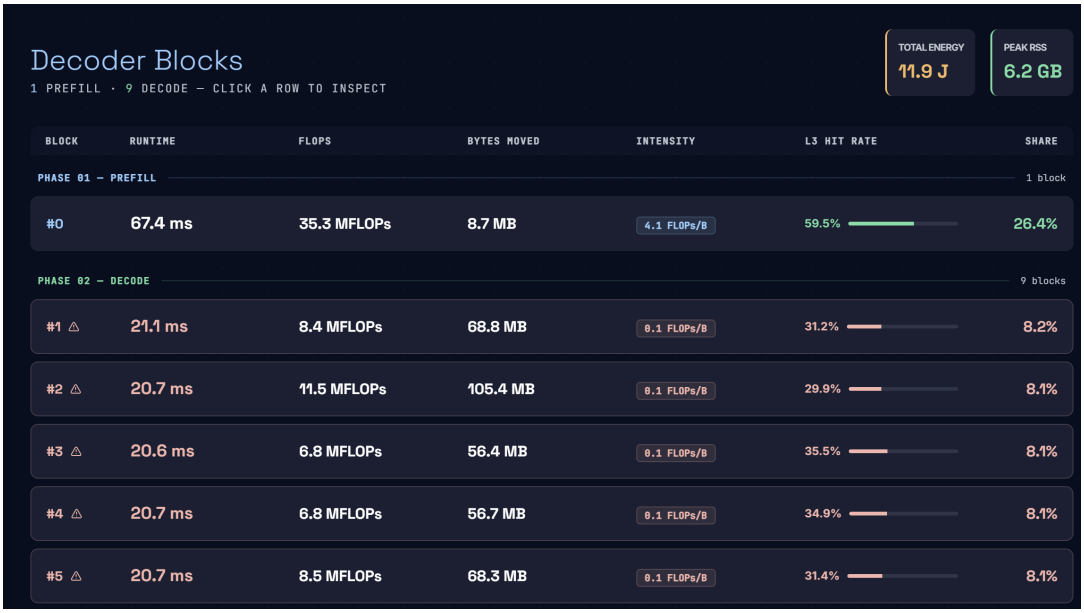

Figure 4.9: Figure showing the Decode block view

The third tab "Decoder Block", shown in Figure 4.9 provides a statistical browser where the user can dive into the inference statistics to find more precise data, and after selection of a certain block, the user can see data about the layers pertaining to that specific decoder block, see Figure 4.10.

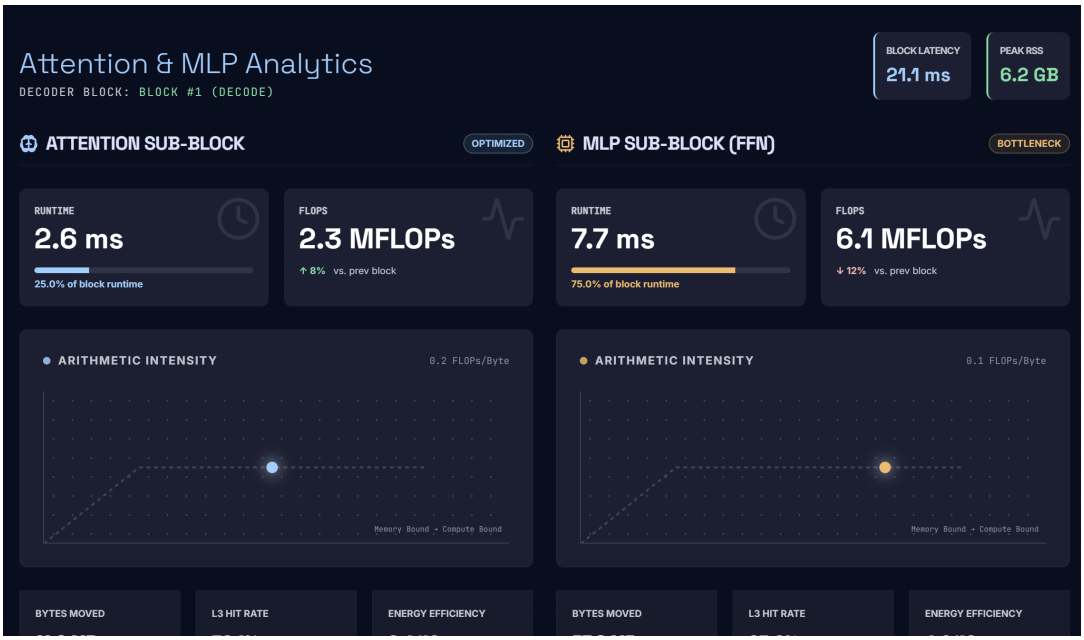

Figure 4.10: Showcasing Attention and MLP analytics for a specific decoder block.

4.4 Roofline Analysis

The tool provides higher-level analysis derived from the collected data. A Roofline model can be generated at four levels of granularity: the entire inference run, the prefill and decode phases separately, individual decoder blocks and specific layers within a given decoder blocks, allowing memory- and compute-bound behaviour to be localised at varying levels of detail.

Case Study: F16 vs Q8 on Intel Core i7-1185G7

The roofline model was applied to two quantization formats of the Qwen2.5-1.5B-Instruct model: 16-bit floating point (F16) and 8-bit quantization (Q8). Both models were executed with the prompt: "Tell me a story" and 128 generated tokens on an Intel Core i7-1185G7 processor. Memory bandwidth was measured using the STREAM benchmark, resulting in 31.47 GB/s for F16 and 31.66 GB/s for Q8.

Metric	F16	Q8
Peak FLOPS (GFLOPS)	499.2	499.2
Memory bandwidth (GB/s)	31.47	31.66
Ridge point (FLOP/byte)	15.86	15.77

Table 4.8: Hardware ceiling for both model configurations.

Model	Phase	OI (FLOP/byte)	Achieved (GFLOPS)	Bound
F16	Entire program	1.28	10.97	Memory
F16	Prefill	17.40	55.35	Compute
F16	Decode	1.19	10.33	Memory
Q8	Entire program	0.81	4.75	Memory
Q8	Prefill	9.08	18.42	Memory
Q8	Decode	0.76	4.50	Memory

Table 4.9: Roofline results for F16 and Q8 across inference phases.

Both models are memory-bound during decode and across the full inference run. The decode OI values of 1.19 FLOP/byte for F16 and 0.76 FLOP/byte for Q8 are well below their respective ridge points, indicating that performance is limited by memory bandwidth rather than compute throughput.

The prefill phase shows different behavior. F16 reaches an OI of 17.40 FLOP/byte, exceeding the ridge point and becoming compute-bound, while Q8 remains memory-bound at 9.08 FLOP/byte. Prefill processes the entire prompt in a batch, increasing arithmetic reuse and therefore OI compared to decode.

F16 achieves higher throughput overall, reaching 10.97 GFLOPS compared to 4.75 GFLOPS for Q8. During decode, the achieved throughput is 10.33 GFLOPS for F16 and 4.50 GFLOPS for Q8. Although Q8 reduces memory traffic through smaller model weights, the lower achieved throughput suggests overhead associated with dequantization.

4.5 Heatmap Analysis

The heatmap visualizations provide operation-level insight into how execution time, L3 cache pressure, and IPC are distributed across transformer layers and inference phases. The full set of heatmaps is included in Appendix C, while the phase runtime heatmap is presented here as the primary result since it most directly reflects the performance trends identified in the roofline analysis.

The heatmaps were generated using the Llama-3.2-1B-Instruct-Q4_K_M model executed through `llama.cpp`, using a two-turn conversation session with the prompts "Hi" and "Good". The KV-cache was configured to use FP16 precision for both the K-cache and V-cache. The token limit per response was set to 10 000 tokens, while the temperature parameter was fixed at 0 to ensure deterministic outputs. Hardware performance counters were collected using `PAPI` in multibatch mode, covering cache, instruction, and floating-point events.

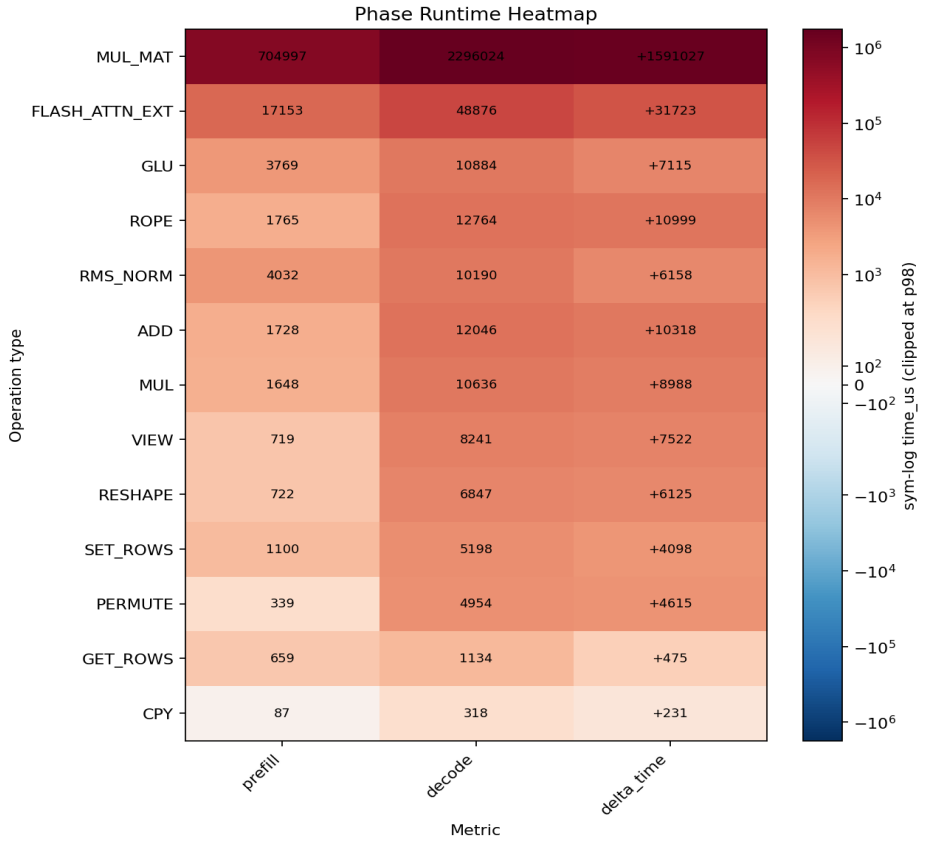

Figure 4.11: Phase runtime heatmap comparing aggregated execution time between prefill and decode. The delta column represents the signed difference between phases.

Figure 4.11 shows that `MUL_MAT` is the dominant operation in both prefill and decode. The operation accumulates substantially higher runtime during decode, increasing from 0.705 s to 2.296 s, corresponding to a runtime delta of +1.591 s. `FLASH_ATTN_EXT` is the second most time-consuming operation, while operations such as `ROPE`, `ADD`, and `GLU` contribute comparatively smaller portions of the total runtime.

All operation types exhibit higher runtime during decode than prefill, showing that decode accumulates more execution time across all measured operations. This behavior is consistent with autoregressive inference, where decode repeatedly executes the transformer layers for each generated token, while prefill processes the input prompt in a single pass.

The appendix heatmaps show similar patterns for memory pressure and IPC. The memory heatmaps indicate that `MUL_MAT` also dominates L3 cache misses across nearly all transformer layers, suggesting that matrix multiplication is both compute-intensive and memory-intensive. The IPC heatmaps show that `MUL_MAT` achieves high IPC values during prefill, but experiences reduced IPC during decode, indicating increased memory pressure and lower execution efficiency.

5
Discussion

5.1 Interpretation of Results

The results demonstrate that the tool enables hardware-level analysis of CPU-based LLM inference in `llama.cpp` across multiple levels of granularity. The four views each produce distinct and internally consistent measurements, and together they allow a user to move from a high-level overview of an entire inference session down to the behavior of individual tensor operations.

The output from the **Top View** and **Phase View** provides an immediate overview of where time and energy are spent across the inference session, while the **Decoder Block View** allows the user to isolate individual blocks and compare their subcomponents. The **Tensor Operation View** provides the finest level of detail, enabling operation-level hardware counter measurements.

The roofline plots and heatmaps demonstrate that the collected profiling data can be transformed into higher-level visual representations. Rather than requiring users to inspect raw counter values manually, the tool presents the measurements in forms that make differences between phases, layers, and operation types easier to compare.

Overall, the results suggest that the tool achieves its goal of making hardware performance data accessible for CPU-based inference in `llama.cpp`, providing information that a user could act on to guide optimization decisions or get a better understanding how LLM inference interacts with hardware.

5.2 Discussion of Research Questions

This section discusses the extent to which the tool fulfills its intended purpose in relation to the research questions stated in Section 1.2. These concern the collection of hardware performance data at multiple levels of granularity, the structuring and visualization of that data, and the application of roofline analysis to characterize inference behavior.

5.2.1 Hardware Performance Data Collection

The tool collects hardware performance data at four levels of granularity. At the finest level, the **Tensor Operation View** instruments every individual tensor op-

eration and, because each tensor operation is associated with a tensor name, its measurements can be aggregated to approximate coarser granularities. However, the accuracy of this aggregation varies. Since measurements are started and stopped around each tensor operation, instrumentation overhead is introduced at every invocation. Through the cross-view validation tests mentioned in Section 3.5 with results displayed in Appendix B, this overhead manifests as an increase in both measured runtime and L1 cache behavior relative to the **Top View**, which is expected given the additional code executed around each tensor operation.

The three coarser views **Top View**, **Phase View**, and **Decoder Block View** each target a different level of granularity and include metrics unique to their scope. Through the cross-view validation tests mentioned in Section 3.5 with results displayed in Appendix B, these views produce stable measurements that agree with each other within 5% and do not introduce the same overhead complications observed in the **Tensor Operation View**. For this reason, using a combination of all four views is preferable to relying solely on aggregated **Tensor Operation View** data to represent coarser granularities, as each view provides its own metrics more reliably at its intended level of abstraction.

5.2.2 Data Structuring and Visualization

The `SQLite` database schema enabled flexible aggregation across runs, phases, layers, and operation types, allowing the same collected data to support both heatmap visualizations and roofline analysis without repeated inference runs.

The heatmap results identified `MUL_MAT` as the dominant operation across execution time, memory pressure, and IPC, reinforcing the conclusion that matrix multiplication is the primary bottleneck during CPU-based LLM inference. The phase comparison heatmaps further showed that the decode generally introduces higher runtime and memory pressure across most operations, consistent with the memory-bound behavior observed in the roofline analysis.

5.2.3 Roofline Analysis

The results show that roofline analysis can identify performance bottlenecks at multiple levels of granularity, including inference phases, decoder blocks, and individual transformer layers. By relating measured floating-point throughput to estimated memory traffic, the operational intensity of each component can be compared against the hardware ridge point to determine whether performance is limited by computation or memory bandwidth.

The measurements also align with expected transformer behavior. Prefill generally exhibits higher operational intensity, while decode remains more memory-bound due to sequential token generation and repeated memory accesses.

5.3 Evaluation of the Tool

The tool is capable of collecting large volumes of performance data across multiple levels of granularity during `llama.cpp`-based inference. Its measurements are currently limited to CPU-related metrics and hardware performance counters, with no support for GPU profiling or other accelerators. Within those bounds, the tool is flexible in terms of which `PAPI` events are collected, as the user can specify any event available on the host hardware. However, certain features such as the roofline model and heatmap visualizations depend on specific counters being available. If the required counters are not supported by the hardware, the affected visualizations will not be generated, though the rest of the tool will function normally.

Due to the limited number of `PAPI` counters that can be measured simultaneously, the tool addresses this constraint by replaying the inference run once per event group, so that all requested counters can be collected even if the user only interacts with the program once. Cross-view validation results, shown in Appendix B, indicate that deviations between replay runs generally remain within 5%, suggesting sufficient reliability for analyzing how CPU-based inference interacts with the hardware.

The tool presents both a terminal interface and a web-based graphical user interface, which allows it to be run on remote machines while still enabling the user to analyse the output locally, increasing the overall flexibility of the tool.

5.4 Limitations

This section discusses the limitations of the work and their impact on the results. These include constraints related to hardware, the accuracy and availability of performance counters, and the scope of the implementation. The implications of these limitations for the validity and generalizability of the results are considered.

Hardware performance counters

The test processor provides a limited number of physical hardware counter registers per core, which restricts how many `PAPI` events can be measured simultaneously. As a result, collecting all metrics required for the roofline analysis, floating-point operations and memory traffic, required separate profiling runs. While the same model, prompt, and generation parameters were used across runs, this introduces a source of measurement uncertainty since system conditions may vary slightly between executions.

Instrumentation overhead

Some hardware counters, particularly L1 cache metrics, are affected by instrumentation overhead when collected at the finest granularity through the **Tensor Operation View**, which limits their reliability at that level of measurement. This effect

is visible in the consistency results presented in Appendix B.

Tool support and validation

Due to limited hardware support on the processor used in this project, memory traffic could not be measured directly from the memory controller. Instead, it was approximated using L3 cache accesses from `PAPI`, which introduces uncertainty in the operational intensity values, as discussed in Section 3.3.1. Attempts to validate this approximation against a ground-truth measurement on a separate machine were unsuccessful, as L3 cache miss counters were unavailable through `PAPI` on that system. Consequently, the accuracy of the memory traffic approximation remains unverified.

5.5 Future Work

- **LLM architectures:** Extend profiling to sparse and mixture-of-experts models by recording metadata such as selected experts, routing scores, active experts, layers, and tokens.
- **Python package:** Package the tool for PyPI with a notebook-friendly API for loading data, running experiments, and visualising results.
- **Roofline accuracy:** Improve the roofline implementation by reducing the dependency on approximating DRAM traffic from L3 cache misses.
- **Measurement averaging:** Repeat each `PAPI` event group run multiple times and average the results to reduce run-to-run variance.
- **View consolidation:** Combine the **Top View**, **Phase View**, **Decoder Block View**, and **Tensor Operation View** probes into a single executable to reduce replay runs.
- **Graphical user interface:** Extend the interface with extrapolated metrics, saved runs, model comparisons, and search functionality.

6
Conclusion

This thesis presented a profiling and visualization tool for analyzing CPU-based LLM inference in `llama.cpp`. The tool combines hardware performance counters collected through `PAPI` and Linux `perf` with structured data storage and interactive visualization tools to support analysis of transformer inference behavior.

The tool supports profiling at multiple levels of granularity, ranging from complete inference runs to individual tensor operations. By separating measurements into Top View, Phase View, Decoder Block View, and Tensor Operation View, it becomes possible to relate low-level hardware behavior to specific parts of the inference pipeline. This enables analysis of runtime, cache activity, memory traffic, energy usage, and CPU utilization across different models and quantization formats.

The results show that the tool can collect, structure, and visualize performance measurements for local LLM inference on CPUs. The graphical dashboard, heatmaps, and roofline visualizations help users interpret runtime behavior and identify performance bottlenecks during inference.

The roofline analysis further showed that different inference phases exhibit different performance characteristics. Prefill generally achieved higher operational intensity, while decode was more strongly affected by memory bandwidth due to repeated accesses to model weights and the KV-cache. This demonstrates how the tool can be used to distinguish between compute-bound and memory-bound behavior.

Although the tool provides detailed profiling capabilities, the work is limited to CPU inference in `llama.cpp`, and some measurements are affected by hardware counter limitations and instrumentation overhead. Nevertheless, the project demonstrates that combining hardware performance counters with structured profiling and visualization provides valuable insight into the behavior of transformer-based LLM inference on CPUs.

References

[1] W. X. Zhao et al., "A Survey of Large Language Models," arXiv:2303.18223, 2023. Available: https://arxiv.org/abs/2303.18223

[2] H. Wang, H. Chai, X. Tu, D. Chen, H. Ke, and K. Han, "lm-Meter: Unveiling Runtime Inference Latency for On-Device Language Models," arXiv:2510.06126, 2025. Available: https://arxiv.org/abs/2510.06126

[3] H. Shen, H. Chang, B. Dong, Y. Luo, and H. Meng, "Efficient LLM Inference on CPUs," arXiv:2311.00502, 2023. Available: https://arxiv.org/abs/2311.00502

[4] Z. Yuan et al., "LLM Inference Unveiled: Survey and Roofline Model Insights," arXiv:2402.16363, 2024. Available: https://arxiv.org/abs/2402.16363

[5] G. Gerganov, "llama.cpp," 2023. Available: https://github.com/ggml-org/llama.cpp

[6] S. Browne, C. Deane, G. Ho, P. Mucci, and D. Terpstra, "PAPI: Portable interface to hardware performance counters," in *Proceedings 2nd Department of Defense HPCMP Users Group Conference*, IEEE, 1999, pp. 7–10. Available: https://icl.utk.edu/papi/

[7] Linux perf wiki Contributors, "PerfWiki: Linux profiling with performance counters," 2024. Available: https://perfwiki.github.io/main/ Accessed: Feb. 2026.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. Available: https://dl.acm.org/doi/10.1145/1498765.1498785

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[10] R. Sennrich, B. Haddow, and A. Birch, "Neural Machine Translation of Rare Words with Subword Units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, ACL, 2016, pp. 1715–1725. Available: https://arxiv.org/abs/1508.07909

[11] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, USENIX, 2024,

pp. 193–210. Available: https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[12] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The Curious Case of Neural Text Degeneration," in *International Conference on Learning Representations (ICLR)*, 2020. Available: https://arxiv.org/abs/1904.09751

[13] Y. Xu, N. K. Khaira, and T. Singh, "KV Cache Optimization Strategies for Scalable and Efficient LLM Inference," arXiv:2603.20397, 2026. Available: https://arxiv.org/abs/2603.20397

[14] M. Tanej, M. Sonsare, and J. Mooney, "GPU-Accelerated INT8 Quantization for KV Cache Compression in Large Language Models," arXiv:2601.04719, 2026. Available: https://arxiv.org/abs/2601.04719

[15] Hugging Face, "Quantization - Hugging Face Optimum," 2024. Available: https://huggingface.co/docs/optimum/concept_guides/quantization Accessed: Apr. 29, 2026.

[16] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," 2021. Available: https://arxiv.org/abs/2103.13630 Accessed: Feb. 13, 2026.

[17] G. Gerganov, "ggml tensor library," 2022. Available: https://github.com/ggml-org/ggml

[18] Hugging Face, "GGUF," 2023. Available: https://huggingface.co/docs/hub/gguf Accessed: Apr. 29, 2026.

[19] James R. Reinders, "Intel® AVX-512 Instructions," Intel Developer Zone, June 2017. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html

[20] H. K. Pyla, B. Ramesh, C. J. Ribbens, and S. Varadarajan, "ScALPEL: A Scalable Adaptive Lightweight Performance Evaluation Library for application performance monitoring," arXiv:0903.0035, 2009. Available: https://arxiv.org/abs/0903.0035

[21] G. Gerganov, "llama.cpp: simple-chat example," 2023. Available: https://github.com/ggml-org/llama.cpp/tree/master/examples/simple-chat

[22] IEEE and The Open Group, "IEEE Standard for Information Technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7," IEEE Std 1003.1-2017, 2017. Available: https://ieeexplore.ieee.org/document/8277153

[23] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "RAPL in Action: Experiences in Using RAPL for Power Measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no. 2, article 9, Apr. 2018. Available: https://doi.org/10.1145/3177754

[24] J. D. McCalpin, "STREAM: Sustainable memory bandwidth in high performance computers," 1991–2007. Available: https://www.cs.virginia.edu/stream/

[25] Google Labs, "From idea to app: Introducing Stitch, a new way to design UIs," Google Developers Blog, May 2025. Available: https://developers.googleblog.com/stitch-a-new-way-to-design-uis/

A

AI Usage

In accordance with the guidelines set forth by Chalmers University of Technology regarding the use of artificial intelligence in academic work, this section outlines the extent and nature of AI assistance utilized during the project.

Generative AI tools were employed strictly as auxiliary instruments to accelerate prototyping, support the iterative design process, and assist with troubleshooting. The specific tools and their applications were as follows:

- **Ideation and Problem Solving:** Conversational language models, including ChatGPT, Gemini, and Claude, were used to brainstorm, discuss architectural concepts, and evaluate different technical approaches. These tools served as sounding boards in a rapid, iterative process to validate ideas before implementation.
- **UI Prototyping:** The graphical user interface (GUI) was initially designed using handcrafted paper prototypes. The generative AI platform v0.dev was subsequently used to translate these physical sketches into foundational frontend code, which was then manually refined, adjusted, and integrated into the React application. The final iteration of the User Interface was developed by using Google Stitch to generate initial wireframes that later was imported into figma for manual refinement, and finally Google Gemini was used to generate the foundations of the react application.
- **Code Experimentation:** Command-line AI assistants, specifically the Gemini CLI and Claude Code, were utilized to rapidly prototype code snippets and test potential solutions. This allowed the team to explore various implementation strategies efficiently before drawing concrete conclusions and writing the final production code.
- **Debugging and Error Resolution:** When encountering unexpected behavior or errors, AI tools were used to analyze stack traces and log outputs. This helped the team identify syntax and logical errors much faster, significantly reducing the time spent on initial troubleshooting, though all proposed fixes were independently verified before being applied.
- **Text Refinement and Proofreading:** During the drafting phase, language models such as Gemini, ChatGPT, and Claude were utilized to proofread the text and improve overall sentence structure. Furthermore, Grammarly was actively used throughout the writing process to immediately identify and correct basic spelling and grammatical mistakes.

B

Cross-View measurement validation

This appendix presents the detailed results from the cross-view measurement validation described in Section 3.5.2. The validation compares measurements collected at different instrumentation granularities using runtime measurements and `PAPI` hardware counters. The shared run configuration used across all tests was:

- **Prompt:** "Tell me a story that is a 1000 words long!"
- **Token limit:** 64 tokens
- **K-cache type:** f16
- **V-cache type:** f16

The models used were:

- Llama-3.2-1B-Instruct-Q4_K_M.gguf
- Llama-3.2-1B-Instruct-Q8_0.gguf
- Llama-3.2-1B-Instruct-f16.gguf
- qwen2.5-1.5b-instruct-q4_0.gguf
- qwen2.5-1.5b-instruct-q8_0.gguf
- qwen2.5-1.5b-instruct-fp16.gguf

B.1 Baseline validation with a single `PAPI` event

This test was conducted with a single `PAPI` event, `PAPI_L3_TCM`, active across all executables simultaneously. Each executable was run once per model with the shared configuration described above. Running with only one event in order to avoid counter multiplexing, providing a baseline comparison between views under minimal instrumentation load. All ratios are computed by dividing the metric gathered by the **Top View** executable by the corresponding summed metric from the other view.

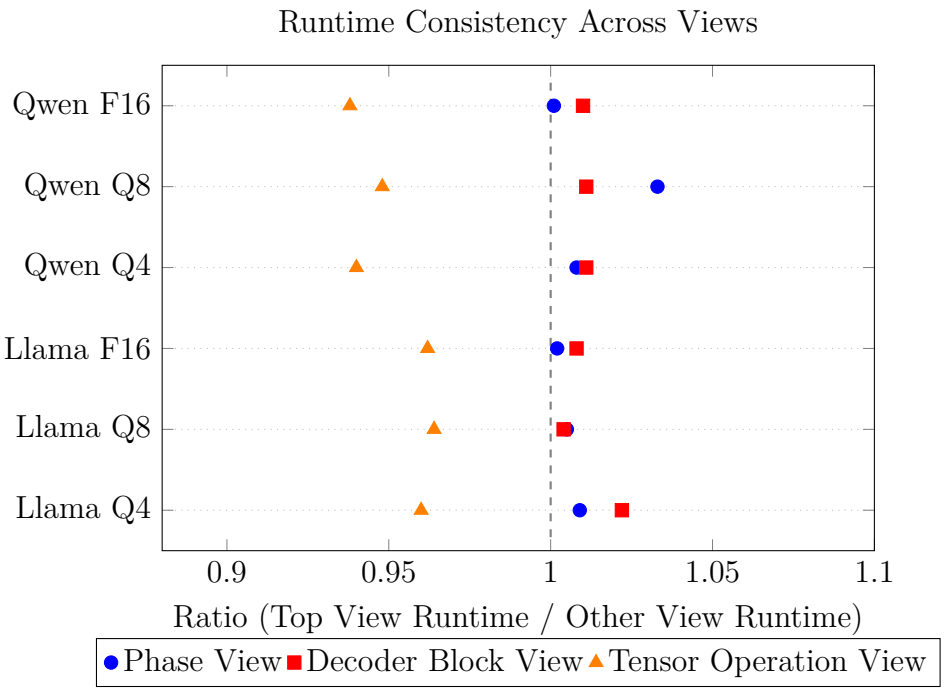

Figure B.1: Runtime ratio of each view relative to the Top View.

Figure B.1 shows the runtime ratios for all three views relative to the **Top View**. The **Phase View** and **Decoder Block View** both produce ratios slightly above 1.0, meaning their summed runtimes are marginally lower than the **Top View** measurement. The **Tensor Operation View** consistently yields ratios in the range 0.94–0.96 across all models and quantization levels, indicating that its summed per-tensor runtimes are approximately 5% higher than the **Top View**.

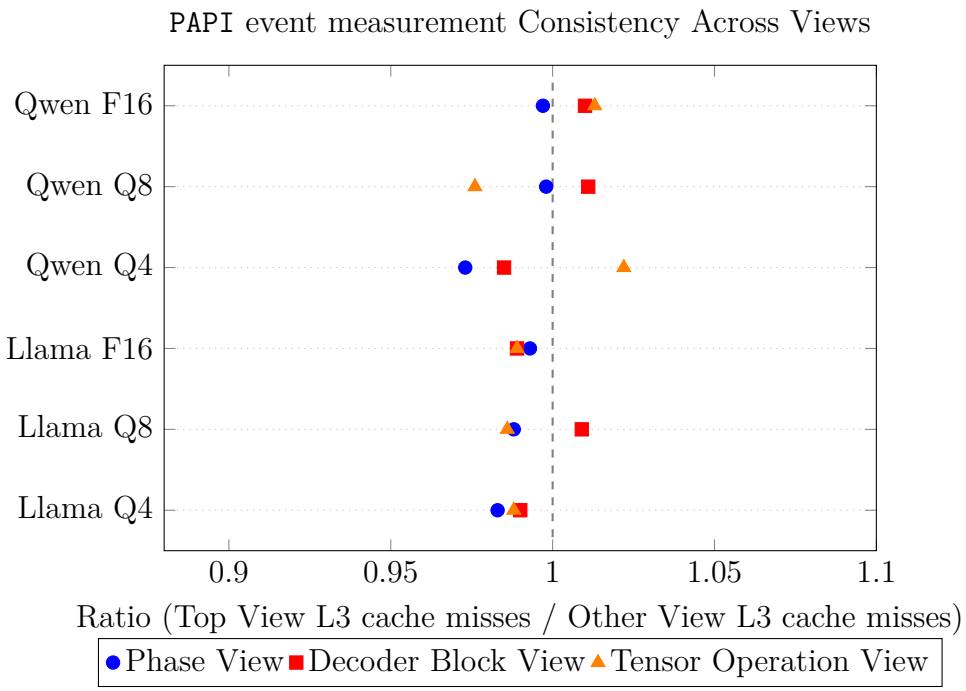

Figure B.2: Single PAPI event ratio of each view relative to the Top View, event used `PAPI_L3_TCM`.

Figure B.2 shows the `PAPI_L3_TCM` ratios for the same test. All three views produce ratios within 3% of 1.0, with no consistent directional bias observed across models or quantization levels.

B.2 Validation under multiple simultaneous `PAPI` events

These tests were conducted to further examine measurement behaviour when multiple `PAPI` events are active simultaneously, introducing a greater possibility of counter multiplexing. Two separate tests were performed, each using a different set of events.

The first test collected three cache miss counters: `PAPI_L3_TCM`, `PAPI_L2_DCM`, and `PAPI_L1_DCM` across all four views using the same run configuration and models as the single-event test. The ratios for each cache level and each view relative to the **Top View** are presented in Table B.1.

		Top / Phase ratios				Top / Dec. block ratios				Top / Tensor Op Ratios			
Model	**Q**	RT	L3	L2	L1	RT	L3	L2	L1	RT	L3	L2	L1
Llama 1B	Q4	1.005	0.994	1.005	0.999	1.014	1.012	1.016	1.017	0.941	1.023	1.001	0.940
	Q8	0.987	0.999	1.004	1.012	1.002	1.005	1.006	1.014	0.941	1.011	0.999	0.934
	F16	0.992	1.009	0.999	0.998	1.010	0.998	1.002	1.003	0.953	0.985	0.990	0.942
Qwen 1.5B	Q4	1.003	0.985	0.995	0.998	1.022	0.998	1.012	1.010	0.920	0.997	0.990	0.942
	Q8	0.988	1.071	1.029	1.023	1.004	1.009	1.011	0.010	0.906	0.975	0.989	0.914
	F16	1.001	1.003	0.999	1.003	1.005	1.001	1.003	1.006	0.922	0.997	0.989	0.935

Table B.1: Top View and other views cache miss comparison for Llama (1B) and Qwen (1.5B) at 64 generated tokens. Runtime in ms, cache misses in millions, ratios relative to Top View.

The **Phase View** and **Decoder Block View** remain within 3% of the **Top View** for all cache levels and all models. The **Tensor Operation View** shows runtime ratios of approximately 0.90–0.95, while its L3 and L2 cache miss counts remain closer to 1.0. The L1 cache miss counts for the **Tensor Operation View** show somewhat larger deviations compared to L3 and L2, as illustrated in Figure B.3.

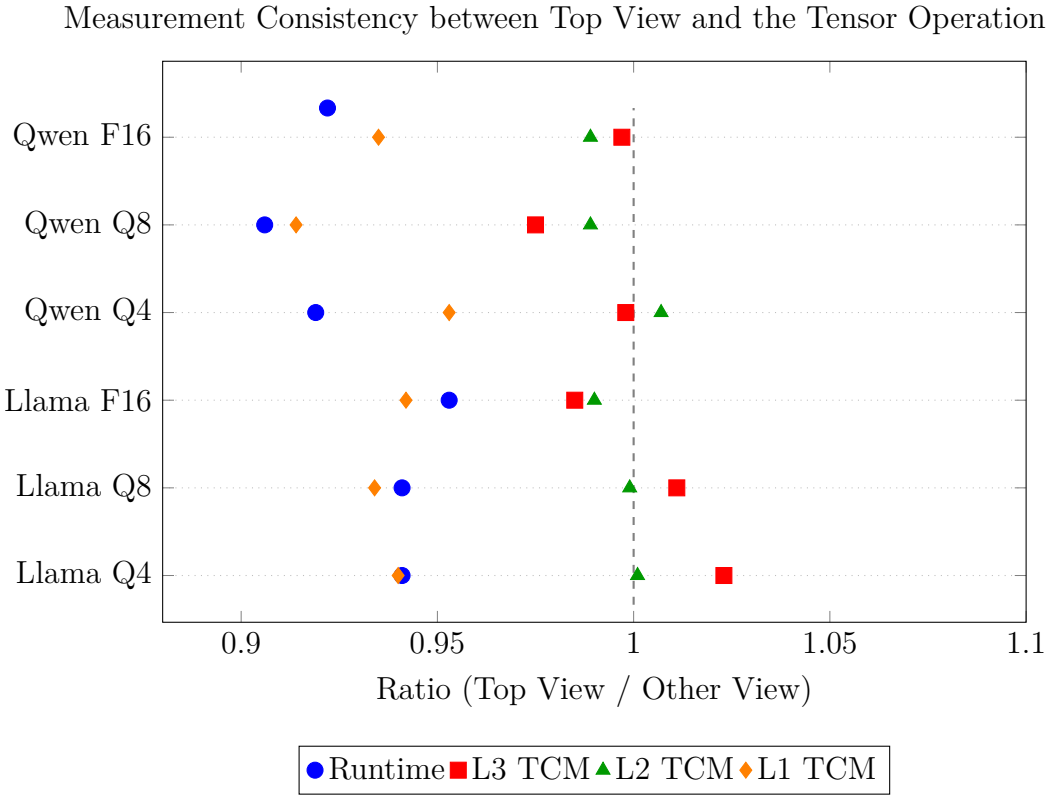

Figure B.3: Consistency ratios between the Top View and Tensor Operation View for runtime and `PAPI` cache-miss measurements. Values near 1.0 indicate strong agreement.

The second test was designed to examine the impact of running the maximum number of `PAPI` events that the hardware supports simultaneously. The events selected

were: `PAPI_L1_TCM`, `PAPI_L2_DCM`, `PAPI_L2_TCA`, `PAPI_L3_DCA`, `PAPI_L1_DCM`, and `PAPI_L1_ICM`. Since this test is specifically aimed at observing the effect of simultaneously collecting the maximum number of supported events, it only compares the **Top View** and the **Tensor Operation View**, as these two executables differ most in how long their measurement windows remain open.

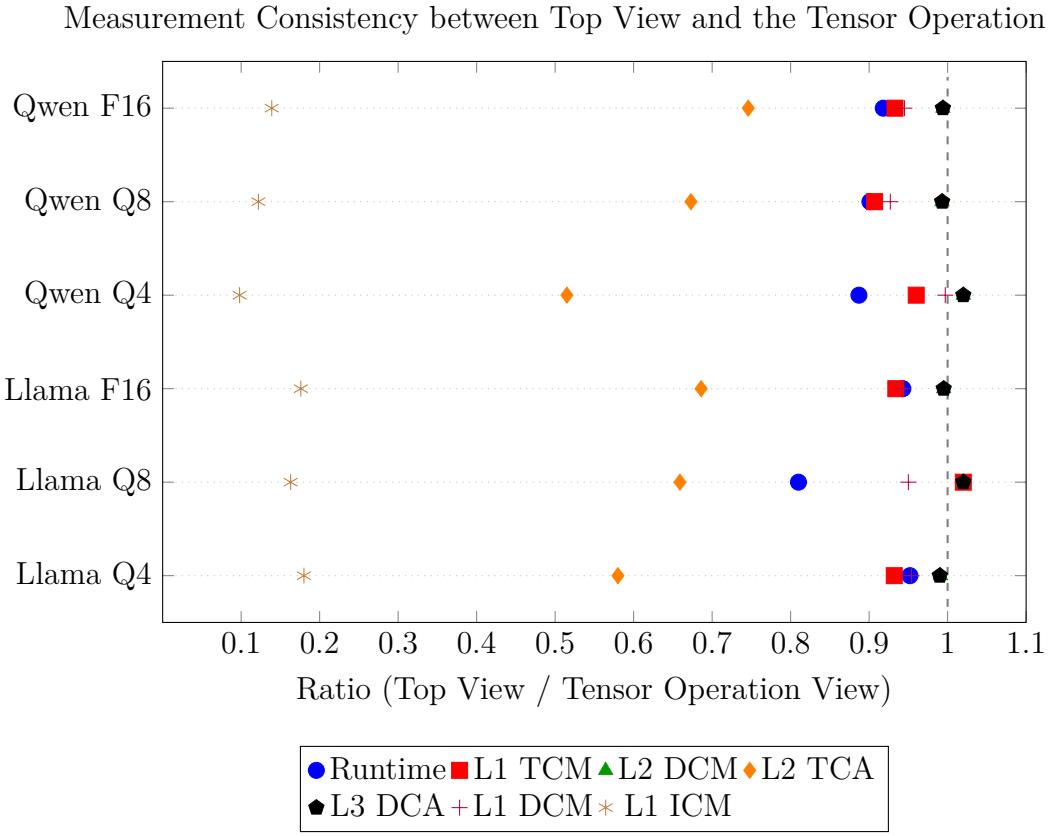

Figure B.4: Consistency ratios between the Top View and Tensor Operation View for runtime and `PAPI` cache measurements. Values near 1.0 indicate strong agreement.

As shown in Figure B.4, most metrics remain within a few percent of 1.0, consistent with the results from the previous tests. However, two events deviate substantially: `PAPI_L2_TCA` yields ratios in the range 0.51–0.75, and `PAPI_L1_ICM` produces ratios as low as 0.10–0.18 across all models. The remaining events: `PAPI_L1_TCM`, `PAPI_L2_DCM`, `PAPI_L3_DCA`, and `PAPI_L1_DCM` all stay within approximately 10% of 1.0.

C

Heatmap

This appendix contains the remaining heatmap visualizations referenced in Section 4.5. These figures provide additional layer-level and phase-level views of execution time, memory pressure, and IPC across transformer operations.

C.1 Layer-Level Heatmaps

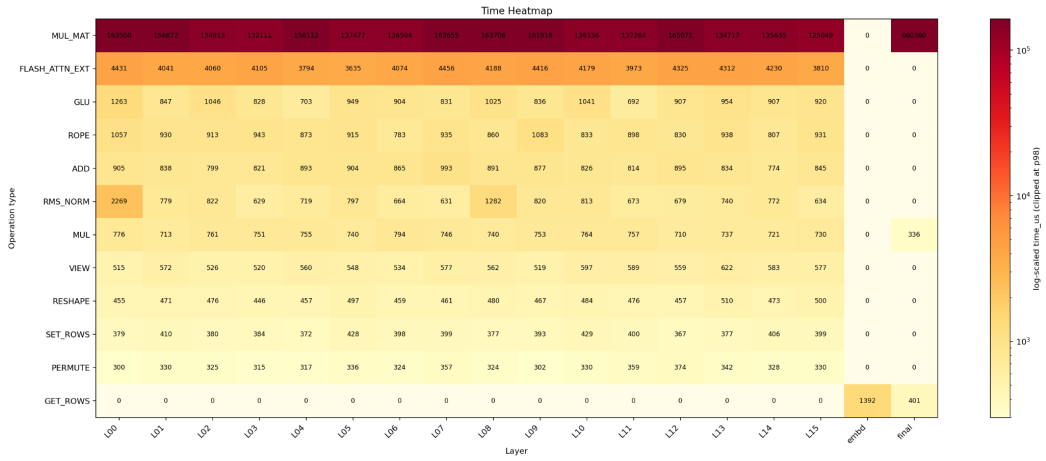

Figure C.1: Layer-level runtime heatmap showing execution time distribution across transformer layers.

The runtime heatmap confirms that `MUL_MAT` dominates execution time across nearly all transformer layers, with per-layer values generally ranging between approximately 123,000 and 183,500 μs, and a notably higher value in the final layer. `FLASH_ATTN_EXT` is the second most time-consuming operation, while operations such as `PERMUTE` and `RESHAPE` contribute comparatively little runtime.

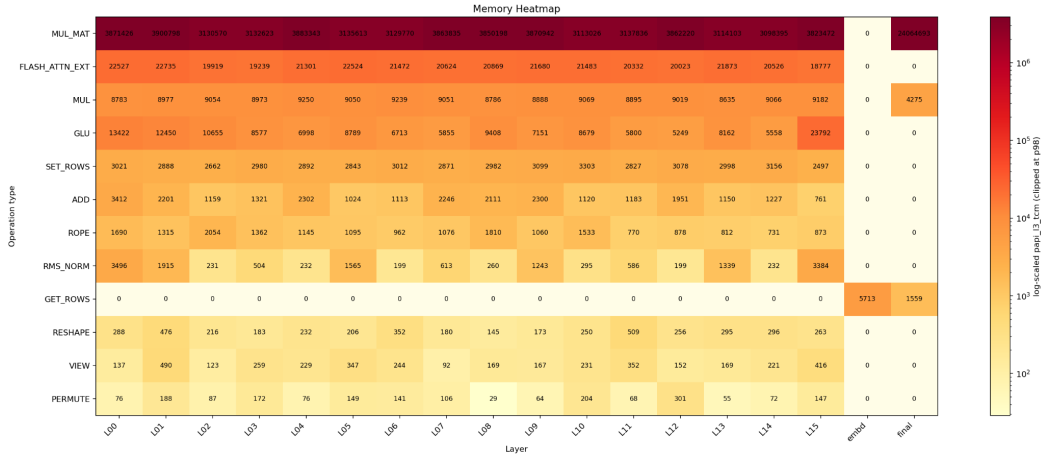

Figure C.2: Layer-level memory heatmap showing L3 cache miss distribution across transformer layers.

The memory heatmap shows that `MUL_MAT` produces by far the highest L3 cache miss counts across all layers, with per-layer values ranging from approximately 3,099,395 to 3,900,798, and a substantially higher count in the final layer. `FLASH_ATTN_EXT` and `GLU` follow at considerably lower levels, while operations such as `PERMUTE` and `VIEW` exhibit minimal cache pressure.

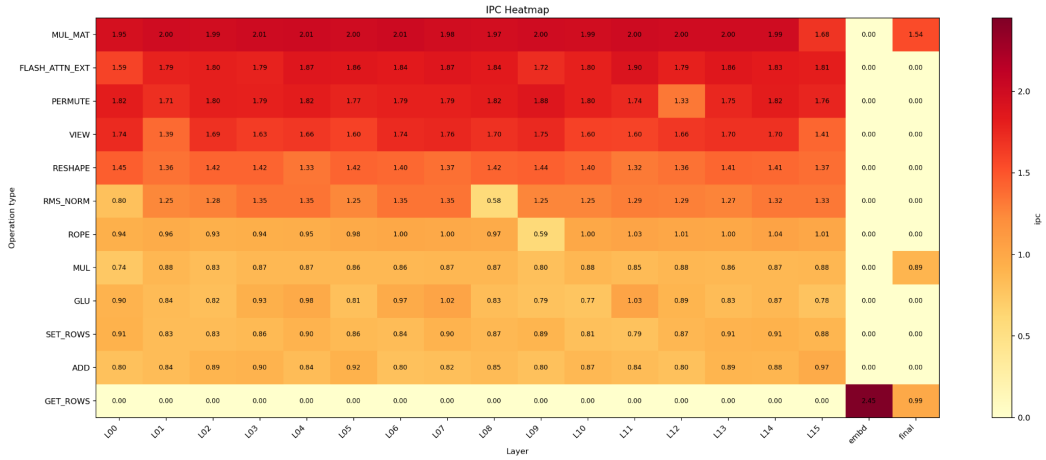

Figure C.3: Layer-level IPC heatmap showing instruction throughput across transformer layers.

The IPC heatmap shows that `MUL_MAT` achieves the highest IPC values across most layers, consistently around 1.95–2.01, indicating efficient instruction throughput during matrix multiplication. `FLASH_ATTN_EXT` and `PERMUTE` also maintain relatively high IPC values. In contrast, operations such as `GLU`, `MUL`, and `ADD` exhibit lower IPC, indicating lower instruction throughput.

C.2 Phase Comparison Heatmaps

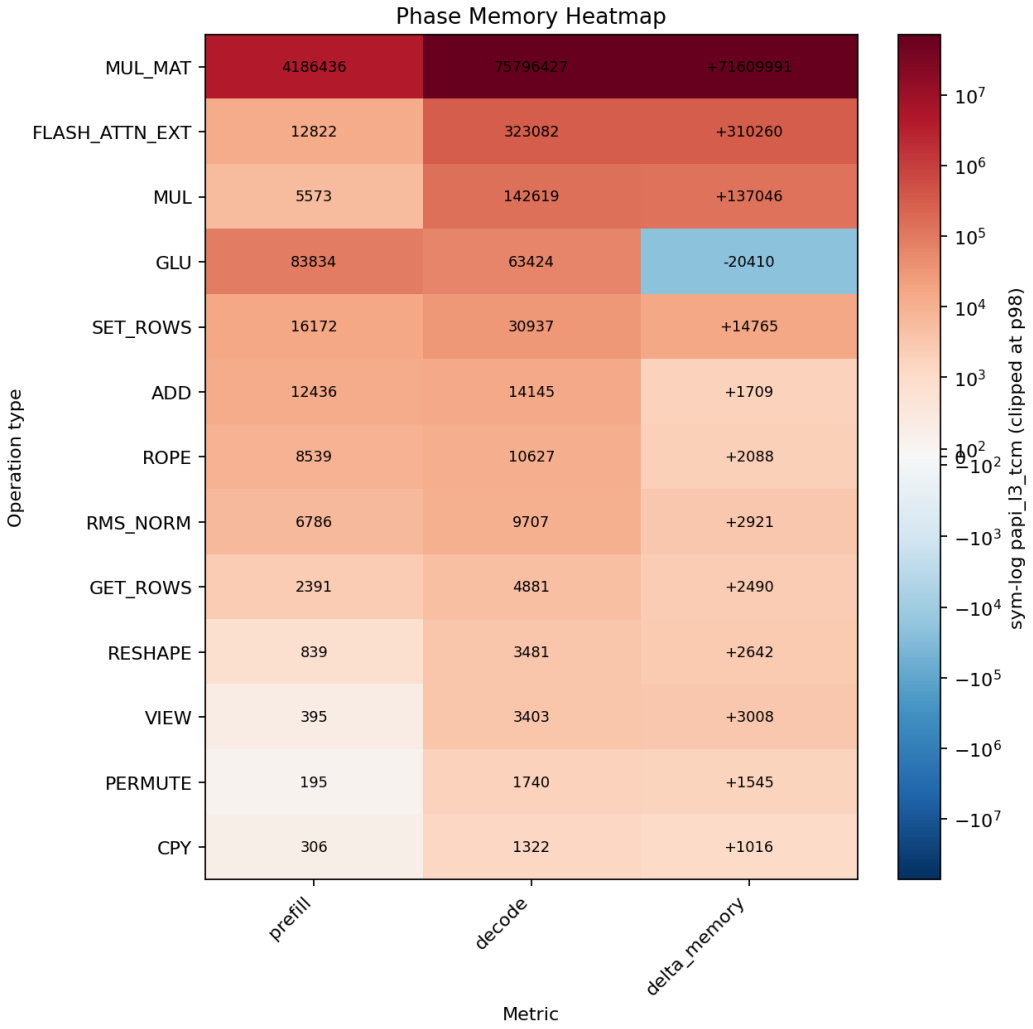

Figure C.4: Phase memory heatmap comparing aggregated L3 cache misses between prefill and decode. The delta column represents the signed difference between phases.

The phase memory heatmap shows substantially higher memory pressure during decode, particularly for `MUL_MAT`, which increases from 4,186,436 cache misses during prefill to 75,796,427 during decode, corresponding to a delta of +71,609,991. This behavior is consistent with repeated weight accesses during autoregressive generation. `GLU` is a notable exception, exhibiting a negative delta of −20,410, indicating slightly fewer cache misses during decode than prefill.

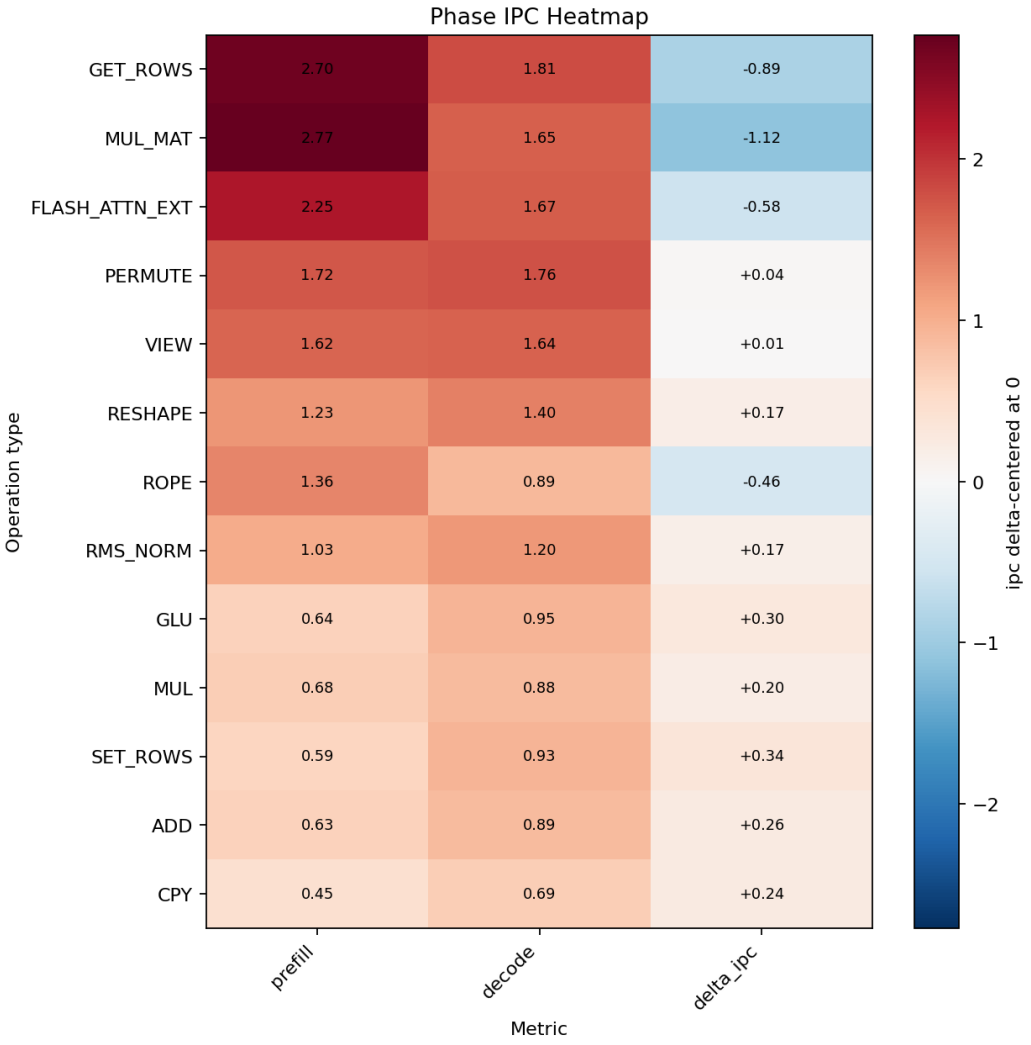

Figure C.5: Phase IPC heatmap comparing aggregated IPC between prefill and decode. The delta column represents the signed difference between phases.

The phase IPC heatmap shows that `MUL_MAT` achieves the highest IPC during prefill at 2.77, but drops to 1.65 during decode, corresponding to a delta of −1.12. `GET_ROWS` and `FLASH_ATTN_EXT` show similar reductions. In contrast, operations such as `PERMUTE` and `VIEW` maintain near-identical IPC across phases, with deltas of +0.04 and +0.01 respectively, indicating stable instruction throughput regardless of inference phase.

D

Default PAPI Events

This appendix lists the default PAPI hardware performance counters used by the profiling framework for each profiling view.

D.1 Top View

```
TOP_VIEW_PAPI_EVENTS = [
    "PAPI_L3_TCM",
    "PAPI_L2_TCM",
    "PAPI_L1_TCM"
]
```

D.2 Phase View

```
PHASE_VIEW_PAPI_EVENTS = [
    "PAPI_FP_OPS",
    "PAPI_L3_TCM",
    "PAPI_L3_TCA",
    "PAPI_TOT_INS",
    "PAPI_TOT_CYC"
]
```

D.3 Decoder Block and Tensor Operation Views

The Decoder Block View and Tensor Operation View use the same default PAPI event configuration.

```
DECODER_BLOCK_VIEW_PAPI_EVENTS =
TENSOR_OP_VIEW_PAPI_EVENTS = [

    # L1 Cache
    "PAPI_L1_DCM",   # L1 data cache misses
    "PAPI_L1_ICM",   # L1 instruction cache misses
    "PAPI_L1_TCM",   # L1 total cache misses

    # L2 Cache
    "PAPI_L2_DCM",   # L2 data cache misses
```

```
    "PAPI_L2_ICM",   # L2 instruction cache misses
    "PAPI_L2_TCM",   # L2 total cache misses
    "PAPI_L2_DCA",   # L2 data cache accesses
    "PAPI_L2_TCA",   # L2 total cache accesses
    "PAPI_L2_DCR",   # L2 data cache reads
    "PAPI_L2_TCR",   # L2 total cache reads
    "PAPI_L2_LDM",   # L2 load misses

    # L3 Cache
    "PAPI_L3_DCM",   # L3 data cache misses
    "PAPI_L3_ICM",   # L3 instruction cache misses
    "PAPI_L3_TCM",   # L3 total cache misses
    "PAPI_L3_DCA",   # L3 data cache accesses
    "PAPI_L3_TCA",   # L3 total cache accesses
    "PAPI_L3_DCR",   # L3 data cache reads
    "PAPI_L3_TCR",   # L3 total cache reads
    "PAPI_L3_LDM",   # L3 load misses

    # TLB
    "PAPI_TLB_DM",   # Data TLB misses
    "PAPI_TLB_IM",   # Instruction TLB misses
    "PAPI_TLB_TL",   # Total TLB misses

    # FLOPS
    "PAPI_FP_OPS",

    # IPC
    "PAPI_TOT_INS",
    "PAPI_TOT_CYC"
]
```

E

Graphical User Interface

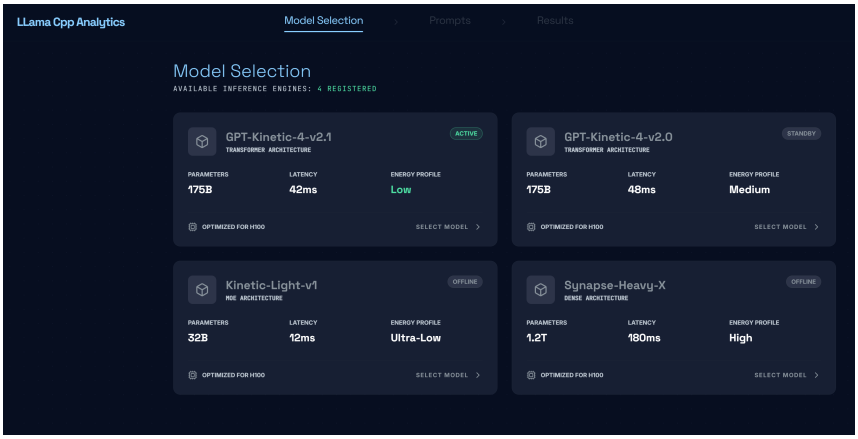

Figure E.1: Figure - Model Selection Page

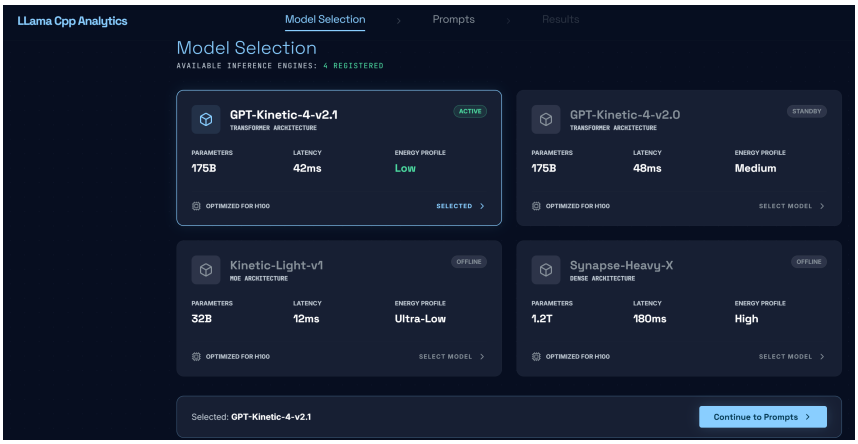

Figure E.2: Figure - Model Selection Page (Selected)

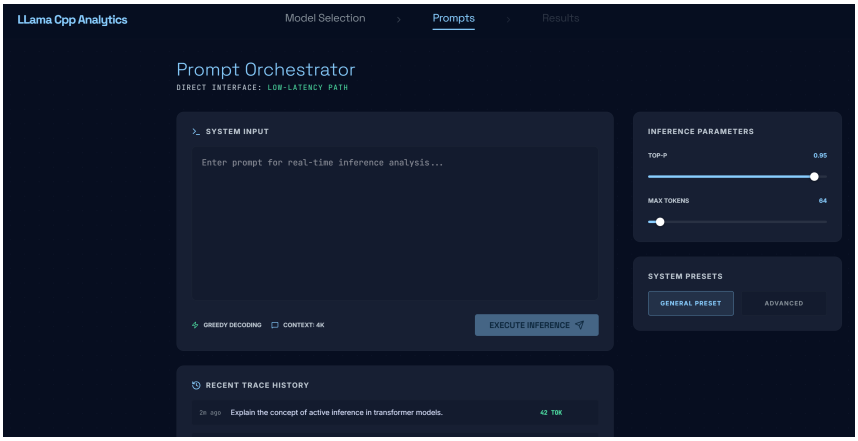

Figure E.3: Figure - Prompts Page

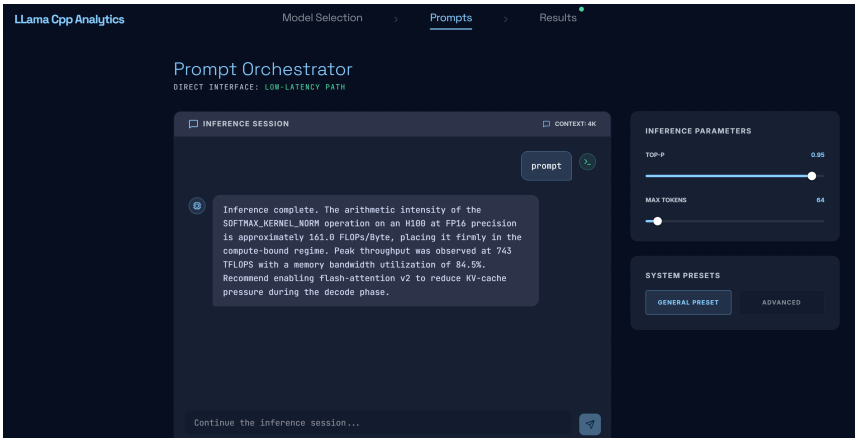

Figure E.4: Figure - Prompts Page (Conversation mode)

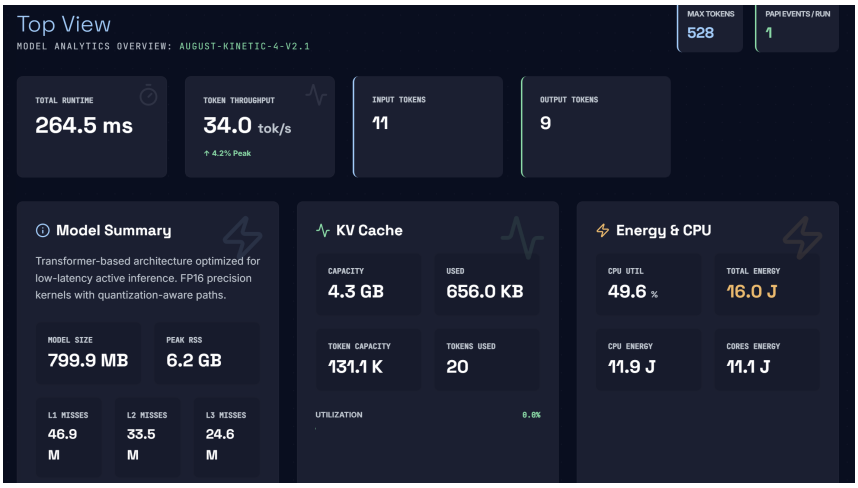

Figure E.5: Figure - Results page - Top View

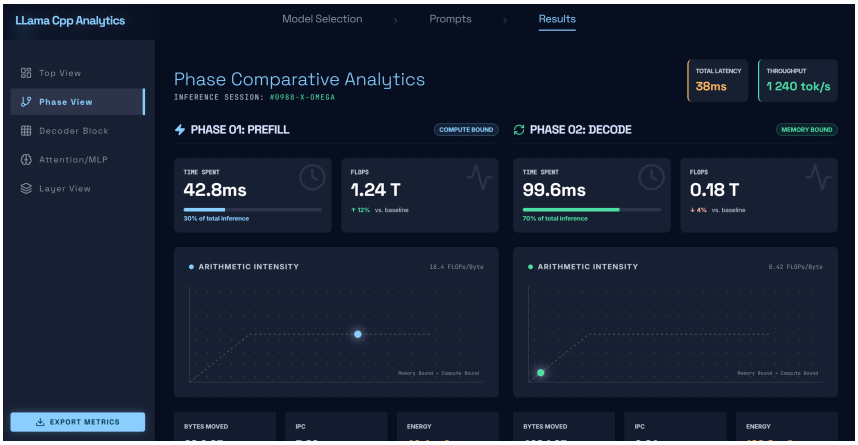

Figure E.6: Figure - Results page - Phase View

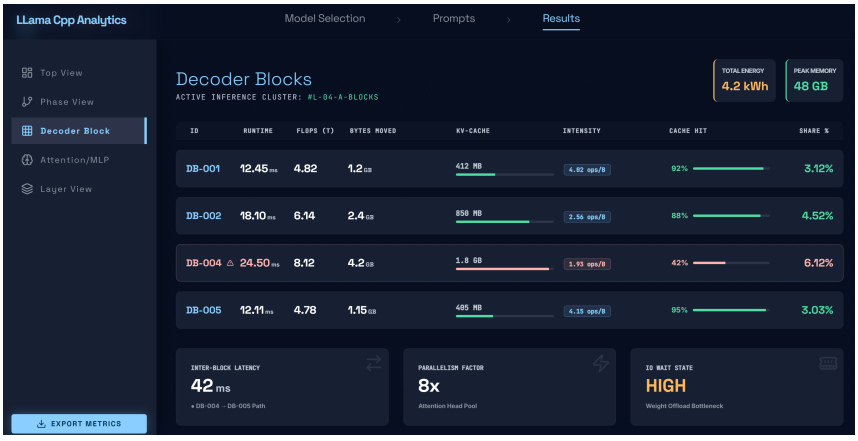

Figure E.7: Figure - Results page - Decoder Block

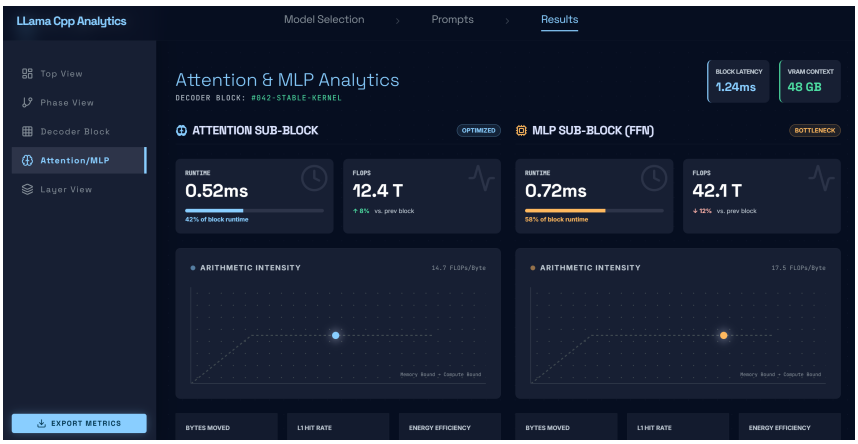

Figure E.8: Figure - Results page - Attention and MLP

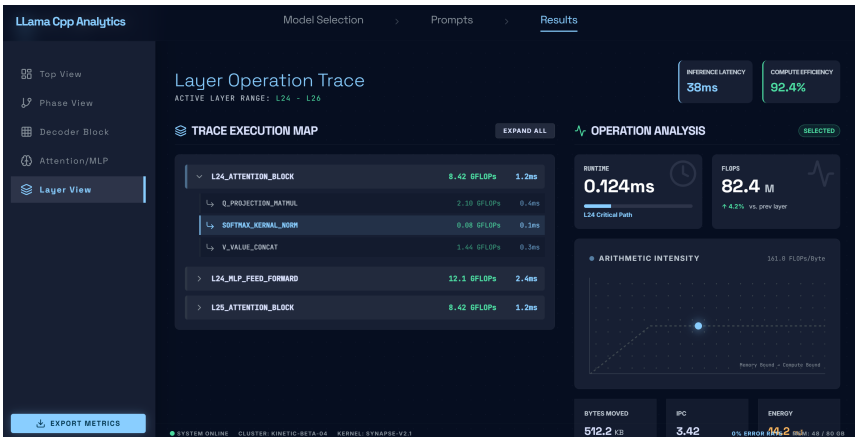

Figure E.9: Figure - Results page - Layer View

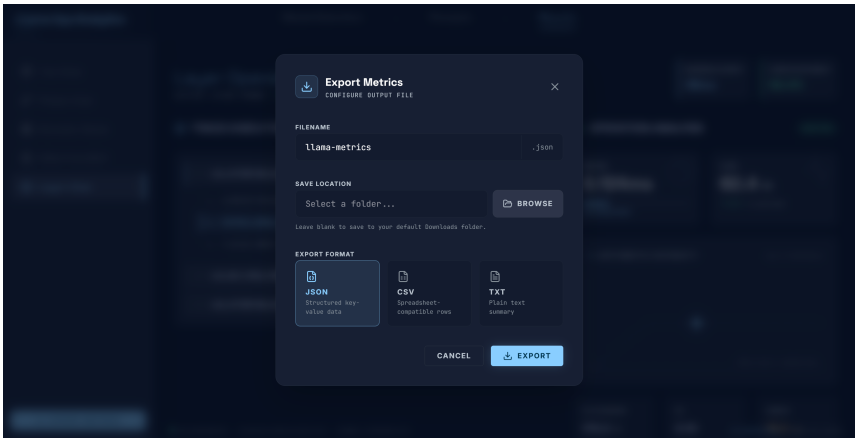

Figure E.10: Figure - Results page - Exporting Metrics

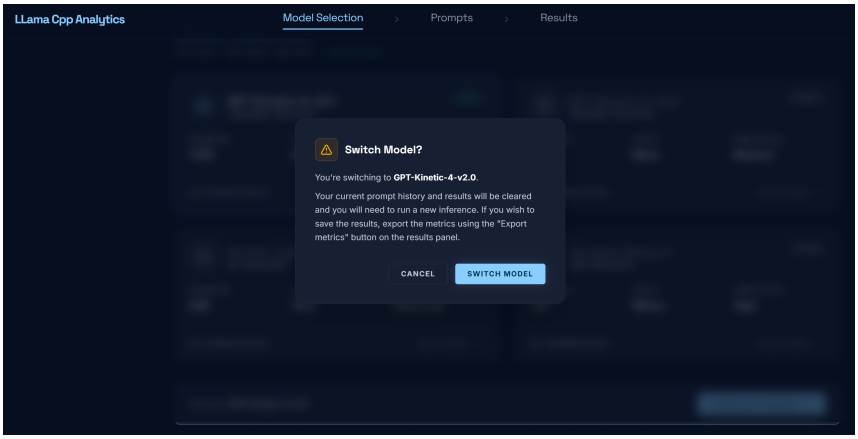

Figure E.11: Figure - Model Selection Page - Switching models

F

Detailed description of utility scripts

F.1 Event retriever

Since the number of hardware performance counters available on a processor is limited, not all `PAPI` events can be collected in a single run. The event retriever module is responsible for partitioning a given list of `PAPI` events into valid groups that can each be collected in a single executable invocation. The module first queries the system using `papi_avail` to determine which events are available on the current hardware and how many native hardware counters each event requires. Events are then packed into groups using a best-fit bin packing algorithm, where the bin capacity is the number of hardware counters reported by the system. Each proposed group is validated using `papi_event_chooser` to check for hardware conflicts that bin packing alone cannot detect. Events that conflict are moved to an overflow list and re-packed separately. Once all groups are validated, a merging pass attempts to combine smaller groups together as long as the combined cost stays within the counter capacity and the merged group passes validation. The result is a minimal set of runs needed to collect all requested events. It is also possible to regulate the total amount of events a group can contain, the reason behind limiting the number of events per group is to reduce multiplexing overhead.

F.2 Run Handler

The run handler module orchestrates the data collection pipeline by invoking a given executable with the correct arguments and coordinating the multi-batch runs required to collect all `PAPI` events. It is configured via a `Config` data class that specifies the model path, prompt, prediction length, cache types, and any custom events to collect. Each executable is represented by a `Run_type` enum value that bundles its output file path together with a default set of `PAPI` events suited to that level of granularity, though custom events can be provided instead. For default event sets see Appendix D. When `run_view` is called, the event retriever module is used to partition the requested events into valid groups, and the executable is then invoked once per group via `subprocess`. The first invocation is always interactive and launches with the `-collect-prompts` flag so that the user's prompts are saved during the conversation. All subsequent invocations consume these saved prompts automatically via the `-user-prompts` argument with console output suppressed,

ensuring that every batch run replays the exact same conversation. For the **Tensor Operation View**, results are directed to a SQLite database rather than a JSON file using the `-use-db` and `-no-csv` flags. The `-temp 0` flag is passed to all invocations to ensure deterministic output.

F.3 JSON complementation

After all executables have finished, the JSON output files produced by the **Phase View** and **Decoder Block View** executables are enriched with additional derived metrics that cannot be computed inside the executables themselves, as they require data from multiple sources or multiple runs. For the **Phase View**, derived metrics include instructions per cycle (IPC), floating point operations per second (FLOP/s), last level cache (LLC) miss rate, bytes moved estimated from LLC misses multiplied by the 64-byte cache line size, operational intensity, and average power in watts derived from the RAPL energy readings. Per-operation breakdowns are also added by querying the **Tensor Operation View** SQLite database, giving each phase a breakdown of time share, count, IPC, bytes moved, and operational intensity per `ggml` operation type. For the **Decoder Block View**, the same derived metrics are computed per block, along with a runtime share percentage relative to the total runtime across all blocks. Each block is additionally enriched with subcomponent metrics that separate the attention and feed-forward network (FFN) contributions by matching tensor names from the SQLite database against known naming patterns for each layer, giving a per-block breakdown of runtime, FLOPs, bytes moved, operational intensity, cache behaviour, and IPC for both the attention and FFN subcomponents independently.