CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# Performance Bottleneck Evaluation of llama.cpp on Jetson and H100

Master's thesis in Computer science and engineering

Mingqi Yu
Yifan Tang

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

Master's thesis 2026

# Performance Bottleneck Evaluation of llama.cpp on Jetson and H100

Mingqi Yu
Yifan Tang

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Performance Bottleneck Evaluation of llama.cpp on Jetson and H100

Mingqi Yu and Yifan Tang



Supervisor: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering
Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering

Master'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

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2026

Performance Bottleneck Evaluation of llama.cpp on Jetson and H100

Mingqi Yu
Yifan Tang
Department of Computer Science and Engineering
Chalmers University of Technology

# Abstract

Efficient large language model inference depends strongly on the interaction between workload shape, numerical precision, serving configuration, and hardware platform. This thesis evaluates the performance bottlenecks of `llama.cpp` on two contrasting NVIDIA platforms: the datacenter-class H100 GPU and the edge-oriented Jetson AGX Orin. The study uses Llama 3.1 8B models in BF16, Q8_0, and Q4_K_M formats, and separates inference into prefill and decode phases using controlled single-sequence workloads and concurrent serving experiments.

The evaluation first establishes baseline performance across balanced, prefill-heavy, and decode-heavy workloads. It then applies targeted profiling with Nsight Systems, server-side timing logs, and power measurements to explain the observed behavior. On H100, the results show a stable phase-dependent precision trade-off: BF16 is most effective for long-prefill workloads because execution is dominated by optimized BF16 GEMM and attention kernels, while Q4_K_M is more favorable for decode-heavy workloads where execution shifts to repeated matrix-vector kernels. Flash Attention further improves long-prefill throughput, but has a smaller effect on decode.

On Jetson Orin, the dominant tuning problem is different. Performance and energy efficiency depend strongly on the selected power mode. The results show that 50W provides a strong energy-oriented operating point, while MAX mode gives the highest throughput and lowest latency. Orin also shows power-mode-dependent precision behavior: Q8_0 remains competitive at lower power, while Q4_K_M becomes more favorable for decode at higher power modes. Concurrent serving experiments further reveal a trade-off between throughput, energy per token, and time to first token.

Overall, this thesis shows that inference optimization cannot rely on a single global configuration. Instead, effective deployment requires phase-aware, platform-aware, and power-aware tuning. The final guidelines recommend precision, Flash Attention, power mode, and concurrency settings based on the dominant workload and deployment objective.

Keywords: LLM inference, llama.cpp, H100, Jetson Orin, profiling, tuning.

# Acknowledgements

The authors would like to express their sincere gratitude to their supervisor and examiner, Ahmed Ali-Eldin Hassan, for his guidance, support, and valuable feedback throughout this thesis project. His insights and encouragement have been highly appreciated and have helped shape the direction and quality of this work.

The authors would also like to thank the CSE Department for providing technical support and access to the resources required to carry out the experiments in this thesis. Their assistance was essential for completing the practical part of this project.

Mingqi Yu, Yifan Tang, Gothenburg, 2026-06-23

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

## 1.1 Motivation

Large language models (LLMs) are increasingly deployed in both cloud and edge environments. In cloud settings, datacenter GPUs such as the NVIDIA H100 provide high compute throughput and memory bandwidth, making it possible to serve many requests with high aggregate throughput. In edge settings, platforms such as NVIDIA Jetson AGX Orin make local inference possible under tighter constraints on power, thermal headroom, memory capacity, and cost. These two deployment regimes have very different hardware characteristics, but both require efficient inference.

Inference efficiency is important for several reasons. First, LLM serving can be expensive because each request may involve billions of model parameters and many sequential token-generation steps. Second, latency matters for interactive applications, where users are sensitive to time to first token and response streaming speed. Third, power and energy consumption are increasingly important, especially for edge devices and sustainability-aware datacenter deployment. Improving inference performance therefore requires understanding not only how fast a model runs, but also where time and energy are spent.

This thesis studies `llama.cpp`, a widely used inference framework for running LLMs on different hardware backends. `llama.cpp` is particularly relevant because it supports both high-performance GPUs and resource-constrained devices, and because it exposes several practical tuning dimensions, including numerical precision, Flash Attention, context size, and serving parallelism. However, these tuning choices do not have a single globally optimal setting. A configuration that works well for one platform or workload phase may be inefficient for another.

A central observation behind this thesis is that LLM inference is not a homogeneous workload. Prompt processing, or prefill, has different computational characteristics from token-by-token generation, or decode. Similarly, a datacenter GPU and an edge device may expose different bottlenecks even when running the same model and inference framework. This motivates a phase-aware and platform-aware bottleneck evaluation of `llama.cpp` on NVIDIA H100 and Jetson Orin.

## 1.2 Problem Statement

Many performance studies report aggregate throughput or end-to-end latency, but these metrics alone do not explain why a given configuration performs well or poorly. For example, a model format may provide high throughput during prefill but perform worse during decode. A high-concurrency serving configuration may improve aggregate throughput while increasing time to first token. A power mode may increase raw performance but reduce energy efficiency.

The problem addressed in this thesis is therefore not only to measure `llama.cpp` performance, but also to identify the dominant bottlenecks under different workloads, precision formats, serving configurations, and hardware platforms. The study focuses on two contrasting systems: NVIDIA H100 as a datacenter-class GPU and Jetson AGX Orin as an edge-oriented platform. By comparing these platforms under a shared experimental framework, the thesis aims to determine how bottlenecks shift between prefill and decode, between precision formats, and between power and concurrency settings.

The final goal is to turn profiling results into practical tuning guidelines. Rather than proposing a new inference engine or modifying low-level CUDA kernels, this thesis focuses on profiling-driven configuration tuning. This includes choosing appropriate precision formats, Flash Attention settings, serving concurrency levels, and Orin power modes based on the target workload and deployment objective.

## 1.3 Research Questions

This thesis is guided by the following research questions:

- **RQ1:** How do prefill and decode differ in their performance characteristics when running `llama.cpp`?
- **RQ2:** How do precision formats such as BF16, `Q8_0`, and `Q4_K_M` affect performance and energy efficiency across different workload phases?
- **RQ3:** How do bottlenecks differ between NVIDIA H100 and Jetson AGX Orin under matched `llama.cpp` workloads?
- **RQ4:** How do serving concurrency and Jetson Orin power modes affect throughput, latency, stability, and energy per generated token?
- **RQ5:** What practical tuning choices can be derived from profiling results for datacenter and edge deployment?

## 1.4 Contributions

The main contributions of this thesis are:

- A phase-aware benchmarking framework for evaluating `llama.cpp` across prefill-heavy, decode-heavy, and balanced workloads.

- A cross-platform comparison of `llama.cpp` on NVIDIA H100 and Jetson AGX Orin using the same model family, precision formats, and workload structure.
- A detailed H100 profiling analysis showing that BF16 is favorable for long-prefill workloads, while `Q4_K_M` is favorable for decode-heavy workloads, due to different dominant CUDA kernel families.
- A Jetson Orin profiling analysis showing that performance and energy efficiency are strongly affected by power mode, and that precision-format behavior can change between low-power and high-performance operating points.
- A serving-oriented analysis of concurrency, throughput, time to first token, and energy per generated token on both platforms.
- A set of practical tuning guidelines for selecting precision format, Flash Attention, serving concurrency, and Orin power mode based on workload phase and deployment objective.

## 1.5 Thesis Structure

The remainder of this thesis is organized as follows.

Chapter 2 presents the background needed for the study, including the LLM inference pipeline, the distinction between prefill and decode, the architecture of `llama.cpp`, precision formats, Flash Attention, serving concurrency, and power-related metrics.

Chapter 3 describes the experimental methodology. It introduces the hardware platforms, software stack, model variants, workload design, benchmarking tools, serving experiments, power measurement methods, and profiling methodology.

Chapter 4 presents the baseline results on H100 and Jetson Orin. These results establish the main phase-dependent, precision-dependent, and power-dependent trends before deeper profiling is applied.

Chapter 5 presents the deep profiling and bottleneck analysis. It explains the H100 results using Nsight Systems traces and kernel-level observations, and analyzes the Orin results using serving logs, power-mode behavior, and `tegrastats` time-series data.

Chapter 6 presents targeted tuning and optimization results. It summarizes how profiling results are used to select practical configurations for H100 and Jetson Orin.

Chapter 7 discusses the broader implications of the findings, compares the optimization problems on H100 and Orin, and describes limitations and threats to validity.

Chapter 8 concludes the thesis and summarizes the final answers to the research questions.

# 2

# Background

## 2.1 LLM Inference Pipeline

Large language model (LLM) inference is typically autoregressive. Given an input prompt, the model first processes the prompt tokens and then generates output tokens one at a time. Each newly generated token is appended to the context and becomes part of the input for the next generation step. This structure makes inference different from a single feed-forward neural network call: the cost depends not only on the model size, but also on the number of prompt tokens, the number of generated tokens, the numerical format, and the serving configuration.

Modern LLM inference pipelines are built on the Transformer architecture, whose attention-based design makes prompt-parallel prefill and autoregressive decode natural performance units for analysis [1].

For decoder-only transformer models, inference is commonly separated into two phases: *prefill* and *decode*. Prefill processes the input prompt and builds the initial key-value (KV) cache. Decode then repeatedly uses this KV cache to generate one new token at a time. This distinction is important because the two phases expose different performance behavior. Prefill often provides more parallelism because many prompt tokens can be processed together. Decode is more sequential and latency-sensitive because each step depends on the previous generated token. Therefore, a single end-to-end tokens-per-second number can hide important bottlenecks.

### 2.1.1 Prefill Phase

The prefill phase processes the full input prompt before generation begins. For a prompt of length $L$, the model computes the hidden states for the prompt tokens and stores the attention keys and values in the KV cache. This cache is reused later during decode so that the model does not need to recompute attention states for all previous tokens at every generation step.

Prefill is often compute-rich compared with decode. Since many tokens are available at once, matrix multiplication and attention operations can be executed with more parallel work. On a high-end GPU, this can map well to optimized GEMM and attention kernels. However, the exact performance depends on sequence length, numerical precision, attention implementation, and backend kernel efficiency. Long-prefill workloads are especially useful in this thesis because they stress prompt-side

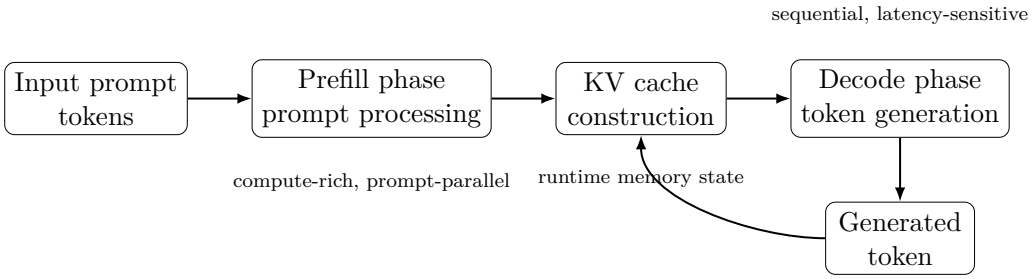


Figure 2.1: Simplified LLM inference pipeline separating prefill and decode. Prefill processes the prompt and builds the KV cache, while decode repeatedly generates one token and updates the cache.

processing and reveal whether a platform benefits more from high-throughput dense computation or from reduced memory footprint.

### 2.1.2 Decode Phase

The decode phase generates tokens sequentially. At each step, the model receives the most recently generated token, attends over the existing KV cache, computes logits, selects the next token, and updates the cache. This loop repeats until the requested number of tokens is generated or a stopping condition is reached.

Decode differs from prefill in two important ways. First, it has less parallelism at the request level because each token depends on the previous one. Second, it repeatedly accesses model weights and the growing KV cache for a relatively small amount of new computation. As a result, decode can expose different bottlenecks from prefill, such as matrix-vector style execution, memory access, scheduling overhead, or per-token synchronization. In serving workloads, decode also strongly affects inter-token latency and user-perceived responsiveness.

## 2.2 llama.cpp Architecture

`llama.cpp` is an open-source C/C++ inference framework for running LLMs on a wide range of hardware platforms. It supports CPU execution as well as several accelerator backends, including CUDA for NVIDIA GPUs. The framework is useful for this thesis because it exposes both high-level model execution and low-level backend behavior, making it possible to study inference performance across datacenter and edge devices.

llama.cpp is an open-source C/C++ inference runtime for large language models, which makes it a practical software substrate for cross-platform performance analysis [2].

At a high level, `llama.cpp` can be understood as two interacting layers. The first layer is the model logic layer, which manages the inference workflow. The second layer is the tensor computation and backend layer, implemented through `ggml` and hardware-specific backends such as `ggml-cuda`. This separation is useful for profiling

because some costs come from model-level control logic, while others come from backend tensor execution.

### 2.2.1 Model Logic Layer

The model logic layer is responsible for the end-to-end inference workflow. It loads model weights, initializes runtime state, manages the context and KV cache, handles tokenization and sampling, and coordinates prompt processing and token generation. In server mode, this layer also handles incoming requests, request scheduling, streaming output, and per-request state.

From a performance perspective, this layer contributes both one-time and repeated costs. One-time costs include model loading, memory allocation, and context initialization. Repeated costs include building compute graphs, dispatching work to the backend, sampling tokens, and managing active requests. In single-sequence benchmarking, these overheads may be small compared with GPU kernel execution. In concurrent serving, however, scheduling, queueing, and request management can become important, especially at high parallelism.

### 2.2.2 ggml and CUDA Backend

The lower layer of `llama.cpp` is built around `ggml`, which represents tensor operations and constructs computation graphs for inference. When CUDA is enabled, performance-critical tensor operations are dispatched to CUDA kernels through the `ggml-cuda` backend. These kernels include dense matrix multiplication, quantized matrix multiplication, matrix-vector operations, normalization, RoPE, attention-related operations, and data conversion kernels.

This backend layer is central to the bottleneck analysis in this thesis. Different workloads and formats lead to different dominant kernel families. For example, long-prefill workloads can be dominated by GEMM and attention kernels, while decode-heavy workloads can shift toward repeated matrix-vector kernels. Quantized formats introduce different CUDA paths from higher-precision formats, often including quantized multiplication, dequantization, or fixup kernels. Therefore, profiling the backend kernel composition helps explain why a format that is efficient in one phase may not be efficient in another.

## 2.3 Precision Formats

LLM deployment commonly uses different numerical formats to trade off memory footprint, arithmetic throughput, and model quality. This thesis focuses on three formats: BF16, `Q8_0`, and `Q4_K_M`.

Prior work has shown that carefully designed low-precision representations can preserve model quality while substantially reducing inference memory requirements, making precision a system-level tuning axis rather than a purely numerical choice [3].

BF16 is a 16-bit floating-point format with the same exponent width as FP32 but fewer mantissa bits. It preserves a wide dynamic range while reducing storage and bandwidth requirements compared with FP32. On modern datacenter GPUs, BF16 can also benefit from specialized hardware support, making it attractive for dense matrix operations.

`Q8_0` and `Q4_K_M` are quantized formats used in `llama.cpp`. `Q8_0` stores weights with approximately 8-bit quantization and generally keeps more numerical information than 4-bit formats. `Q4_K_M` is a 4-bit groupwise quantized format that substantially reduces model weight size. Lower-bit formats reduce memory footprint and can reduce the amount of data moved per token, but they may also introduce additional quantization-related computation and specialized kernel paths.

Post-training quantization methods such as GPTQ demonstrated that accurate 3-bit and 4-bit weight quantization can substantially reduce memory footprint while retaining strong generation quality [4].

SmoothQuant further showed that activation outliers are central to practical low-precision deployment, which is important when interpreting why some quantized formats help certain hardware and phases more than others [5].

Activation-aware methods such as AWQ also highlight that the practical value of low-bit inference depends not only on compression ratio but also on hardware-friendly kernel realization [6].

For this reason, quantization does not guarantee a uniform speedup. A quantized format can be beneficial in memory-sensitive or decode-heavy regimes, while a higher-precision format can be faster when the workload maps well to optimized dense computation. The central question is therefore not which precision is globally best, but which precision is best for a given phase, platform, and deployment objective.

## 2.4 Flash Attention

Standard attention can require substantial memory traffic, especially for long sequences. Flash Attention is an IO-aware exact attention algorithm designed to reduce memory reads and writes between GPU memory and on-chip memory by using tiling and recomputation strategies [7]. Subsequent work on FlashAttention-2 showed that better work partitioning and parallelism can further improve the efficiency of exact attention on modern GPUs [8].In practice, this can improve attention performance and reduce memory overhead for long-context workloads.

In this thesis, Flash Attention is relevant mainly because the tested workloads separate short and long prompt processing. Long-prefill scenarios stress attention over many prompt tokens, so enabling Flash Attention is expected to have the largest effect there. Decode is different: each decode step uses only one new query token and reuses the existing KV cache. Therefore, Flash Attention may still help decode, but its effect is usually smaller than in long prefill. This phase-dependent behavior makes Flash Attention a useful example of why kernel-level optimizations should be evaluated separately for prefill and decode.

## 2.5 Serving, Batching, and Concurrency

Single-sequence benchmarking is useful for isolating prefill and decode behavior, but real inference systems often serve multiple requests at the same time. In this thesis, concurrent serving is evaluated using `llama-server` and a custom client that sends multiple parallel completion requests. The number of simultaneous requests is referred to as the concurrency level, or `npl` in the experiment logs.

Concurrency can improve aggregate throughput by giving the backend more work to process and by increasing hardware utilization. However, higher concurrency also increases queueing, per-request latency, KV-cache pressure, and scheduling complexity. Therefore, the best concurrency level is not necessarily the maximum one. A useful serving configuration must balance aggregate throughput with user-facing latency.

Several metrics are used to describe serving behavior. Aggregate throughput measures the total number of generated tokens per second across successful requests. Time to first token (TTFT) measures the delay between submitting a request and receiving the first generated token. Inter-token latency (ITL) measures the spacing between generated tokens after the first token. Percentile metrics such as median TTFT or p95 TTFT are useful because serving latency can have long tails. In this thesis, concurrency is treated as a tuning parameter rather than a fixed background setting.

## 2.6 Power and Energy Metrics

Performance alone is not sufficient to evaluate inference systems, especially when comparing a datacenter GPU with an edge device. This thesis therefore also considers power and approximate energy efficiency.

Average power is measured differently on the two platforms. On H100, GPU power is sampled using NVIDIA tooling such as `nvidia-smi`. On Jetson Orin, power and system utilization are collected using `tegrastats`, which reports device-level power, GPU activity, CPU activity, memory use, and temperature-related information. These measurements are used to compute approximate energy per generated token:

$$\text{J/token} \approx \frac{\text{Average power (W)}}{\text{Throughput (tokens/s)}}. \tag{2.1}$$

This metric is useful for comparing configurations within the same platform, such as different Orin power modes or different serving concurrency levels. However, it should be interpreted carefully. Average power is measured over a run, while prefill and decode may consume power differently within that run. Therefore, the reported J/token values are coarse efficiency indicators rather than exact phase-specific energy accounting. In addition, H100 board power and Orin tegrastats-reported power are not identical measurement sources, so cross-platform energy comparisons must be made with caution.

## 2.7 Related Work

Prior work on LLM inference has shown that performance depends on both model-level and system-level choices. The separation between prefill and decode is especially important. Splitwise argues that prompt processing and token generation expose different resource requirements and proposes phase-specific resource management [9]. Sarathi-Serve studies the throughput-latency trade-off created by interleaving prefill and decode work in serving systems, showing that scheduling policy can strongly affect serving capacity and latency [10]. More recent systems such as DistServe go further by disaggregating prefill and decode across different GPUs, emphasizing that phase interference can itself become a dominant systems bottleneck in high-throughput serving [11].

Memory management is also a major issue in LLM serving. PagedAttention, introduced in the vLLM system, treats KV-cache memory using a block-based management strategy inspired by virtual memory, reducing fragmentation and improving serving efficiency [12]. This line of work highlights that inference bottlenecks are not only caused by neural network kernels, but also by runtime state management and serving-level scheduling.

Energy-aware LLM serving has also received increasing attention. Recent studies on datacenter GPUs emphasize that quantization, batching, and request shaping affect both throughput and energy efficiency [13]. For edge devices, prior evaluations of LLM inference on Jetson platforms show that power mode, quantization, batch size, sequence length, and memory constraints can all change latency, throughput, and energy behavior [14]. These studies motivate treating H100 and Jetson Orin as different optimization problems rather than simply comparing their absolute throughput.

This thesis builds on these ideas by performing a matched, phase-aware evaluation of `llama.cpp` on H100 and Jetson Orin. Unlike work focused only on one platform or one serving engine, the goal here is to connect baseline measurements, deep profiling, power measurements, and targeted tuning into a single cross-platform bottleneck analysis. The emphasis is not only on reporting which configuration is fastest, but also on explaining why different platforms, phases, and precision formats expose different bottlenecks.

# 3

# Methodology

## 3.1 Overview

The methodology of this thesis is designed to connect high-level benchmark observations with lower-level bottleneck explanations. Rather than relying only on aggregate throughput, the evaluation separates LLM inference into prefill and decode phases, varies workload shape and precision format, and compares behavior across a datacenter GPU and an edge platform.

The experimental workflow has four main stages. First, baseline measurements are collected using `llama-bench` for single-sequence inference and `llama-server` for concurrent serving. Second, the baseline results are analyzed to identify representative configurations where performance differences are large or practically important. Third, targeted deep profiling is performed on selected H100 and Jetson Orin cases using Nsight Systems, server timing logs, and power/utilization time-series data. Finally, the profiling results are used to derive small configuration-level tuning decisions, such as choosing a precision format, enabling Flash Attention, selecting a serving concurrency level, or choosing an Orin power mode.

The methodology is intentionally not a full exhaustive search over all possible configurations. Instead, the goal is to establish a reproducible baseline, select important representative cases, explain the observed bottlenecks, and derive practical tuning guidelines.

## 3.2 Hardware Platforms

The experiments are conducted on two NVIDIA platforms with different deployment characteristics.

**NVIDIA H100** is used as the datacenter-class platform. It provides high compute throughput, high memory bandwidth, and a large GPU memory capacity. In this thesis, H100 serves as the high-performance reference platform. It is expected to expose bottlenecks related to phase-specific GPU utilization, precision format, kernel efficiency, and serving parallelism.

**Jetson AGX Orin** is used as the edge-oriented platform. Compared with H100, Orin operates under tighter power, thermal, and memory constraints. It also sup-

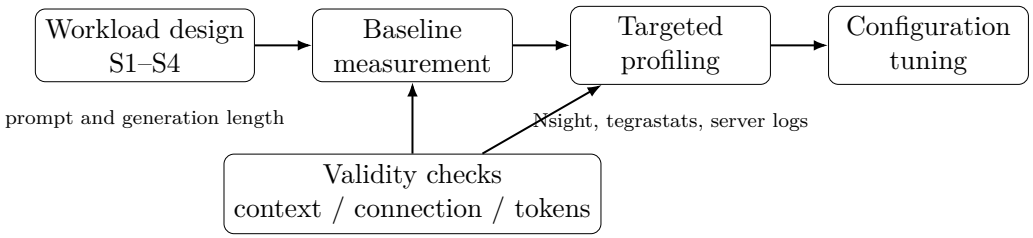


Figure 3.1: Experimental workflow used in this thesis. Controlled workloads are first measured as baselines, then representative cases are profiled and converted into configuration-level tuning decisions.

ports multiple power modes, making it suitable for studying how operating conditions affect inference performance and energy efficiency. In this thesis, Orin is not treated simply as a slower version of H100. Instead, it is evaluated as a platform where the optimal precision format, concurrency level, and power mode may change depending on workload and deployment objective.

The contrast between these platforms is central to the thesis. H100 represents a datacenter setting where the main tuning problem is often phase-aware use of precision and serving parallelism. Orin represents an edge setting where power mode and energy efficiency become first-order considerations.

## 3.3 Software Stack and Model Variants

All experiments use `llama.cpp` with CUDA support enabled. Two execution modes are used:

- `llama-bench` for single-sequence benchmarking;
- `llama-server` together with a custom client for concurrent serving experiments.

The model family used throughout the experiments is Llama 3.1 8B. Using the Llama 3 family as a fixed model family is methodologically reasonable because it represents a contemporary decoder-only Transformer workload with long-context and multilingual capabilities [15].Keeping the model family fixed allows the study to isolate the effects of workload shape, precision format, hardware platform, serving configuration, and power mode.

Three precision formats are evaluated:

- **BF16**, used as the higher-precision reference format;
- `Q8_0`, used as an intermediate quantized format;
- `Q4_K_M`, used as a lower-bit quantized format with reduced model memory footprint.

For concurrent serving, most experiments focus on `Q4_K_M`. This format is the most practical for high-concurrency serving in the current setup because it reduces model

memory footprint and leaves more memory headroom for the KV cache and multiple active requests.

Flash Attention is evaluated as a runtime option. In single-sequence experiments, selected workloads are run with Flash Attention disabled and enabled in order to measure its effect on prefill and decode separately.

## 3.4 Workload Design

The workload design separates prompt length and generation length in order to expose different inference regimes. Four main workload scenarios are used:

Table 3.1: Workload scenarios used in the thesis.

| Scenario | Description | Prompt tokens | Generated tokens |
|---|---|---|---|
| S1 | Balanced / ShortQA | 128 | 128 |
| S2 | Prefill-heavy / RAG-like | 2048 | 128 |
| S3 | Pure prefill stress test | 4096 | 1 |
| S4 | Decode-heavy / LongGen | 128 | 1024 |

S1 is used as a balanced reference workload and is the main scenario for concurrent serving. S2 and S3 stress prompt processing and are used to study prefill behavior. S4 stresses long token generation and is used to study decode behavior.

In some serving-style profiling runs, the actual tokenized prompt length can differ from the nominal prompt-token target because prompts are generated using a text-based approximation. When this occurs, the run is labeled and interpreted according to its actual measured token count. Strict S3 experiments use `pp4096/tg1` in `llama-bench`.

## 3.5 Single-Sequence Benchmarking

Single-sequence experiments are performed using `llama-bench`. This tool reports separate prompt-processing and token-generation throughput values. In this thesis, these are interpreted as:

- **prefill throughput**, corresponding to prompt processing;
- **decode throughput**, corresponding to token generation.

For each model format and workload scenario, `llama-bench` is run with CUDA offloading enabled. Flash Attention is evaluated using the runtime flag for selected cases. The number of repeated measurements is chosen to reduce short-run noise, and median or average values are reported depending on the output format and post-processing step.

Single-sequence benchmarking is used for two purposes. First, it establishes the baseline ranking of precision formats across prefill-heavy and decode-heavy work-

loads. Second, it provides representative cases for deeper Nsight Systems profiling, such as BF16 and `Q4_K_M` under S3 and S4.

## 3.6 Concurrent Serving Experiments

Concurrent serving experiments are performed using `llama-server` and a custom Python client. The server is started with a selected model, context size, and number of parallel slots. The client sends multiple parallel completion requests to the server and records request-level latency and generation information.

The main serving workload is S1, with approximately 128 prompt tokens and 128 generated tokens. The primary concurrency levels are:

$$npl \in \{1, 8, 32, 64, 128\}. \tag{3.1}$$

Here, `npl` refers to the number of concurrent client requests and is aligned with the server parallelism setting for the corresponding experiment. The client uses streaming completion requests with deterministic generation settings. Prompt caching is disabled so that requests do not benefit from reused prompt state across runs.

For each serving run, the client records:

- number of successful and failed requests;
- total generated tokens;
- aggregate generated-token throughput;
- per-request total time;
- time to first token (TTFT);
- inter-token latency (ITL), when valid;
- generated token count per request.

The aggregate serving throughput is computed as:

$$\text{Throughput} = \frac{\text{total generated tokens from successful requests}}{\text{wall time of the batch}}. \tag{3.2}$$

This definition reflects the total useful generated-token rate across all concurrent requests.

## 3.7 Power and Energy Measurement

Power and energy behavior are measured differently on the two platforms.

On H100, GPU power is sampled using NVIDIA tooling, such as `nvidia-smi`, during benchmark or serving runs. The sampled power values are averaged over the run and combined with throughput to compute approximate energy per generated token.

On Jetson Orin, power and system utilization are recorded using `tegrastats`. For Orin experiments, full time-series logs are collected whenever possible. These logs include power-related values, GPU utilization, CPU utilization, memory usage, temperature, and other platform-level information depending on what `tegrastats` reports.

Approximate energy per generated token is computed as:

$$\text{J/token} \approx \frac{\text{average measured power (W)}}{\text{generated-token throughput (tokens/s)}}. \tag{3.3}$$

This metric is used as a coarse efficiency indicator. It is most reliable for comparing configurations within the same platform, such as different Orin power modes or different H100 concurrency levels. It is not interpreted as exact phase-specific energy accounting, because power is sampled over an entire run and may not separate prefill and decode energy precisely. Cross-platform energy comparisons are also treated with caution because H100 and Orin use different power measurement sources.

## 3.8 Deep Profiling Methodology

Targeted deep profiling is used to explain the most important baseline observations. The profiling methodology differs slightly between H100 and Orin because the main bottleneck questions differ across platforms.

On H100, Nsight Systems is used to collect CUDA API summaries, GPU kernel summaries, and GPU memory operation summaries. Graph-level Nsight traces are used to obtain profiling information with relatively low runtime perturbation. Node-level CUDA Graph traces are used only for selected cases where the internal kernel composition of CUDA Graphs is needed. In particular, node-level traces are used to inspect decode-heavy S4 execution and identify whether BF16 and `Q4_K_M` are dominated by different matrix-vector kernel families.

The H100 profiling cases are selected to explain:

- why BF16 is faster in long-prefill workloads;
- why `Q4_K_M` is faster in decode-heavy workloads;
- how Flash Attention changes long-prefill execution;
- how serving behavior changes across concurrency levels.

On Jetson Orin, deep profiling places stronger emphasis on power-mode behavior and system-level time-series data. `tegrastats` logs are collected for the selected Orin cases, and Nsight Systems graph-level traces are used where feasible. The Orin profiling cases are selected to explain:

- why 50W is an energy-oriented operating point;
- why MAX mode gives the highest throughput and lower TTFT;
- how npl64 and npl128 differ in throughput, TTFT, and energy;
- how precision-format ranking changes across power modes.

Node-level traces are not used as the main profiling method for Orin serving, because the main Orin questions concern power, latency, utilization, and energy trade-offs rather than fine-grained CUDA Graph kernel composition.

## 3.9 Measurement Corrections and Validity Checks

Several corrections and validity checks are applied to avoid misleading results.

First, high-concurrency serving requires sufficient context capacity per parallel slot. For npl128 experiments, a larger context size is used so that each request has enough context budget for the prompt and generated tokens. This prevents high-concurrency results from being confounded by premature truncation.

Second, the client connection limit is explicitly removed for high-concurrency runs. The Python client uses an `aiohttp` connector with no connection cap, so that requested concurrency levels such as 128 are not limited by the client-side HTTP connection pool.

Third, generated-token validation is performed. For each serving case, the total number of generated tokens is checked against the expected number:

$$\text{expected generated tokens} = \text{number of successful requests} \times \text{requested generated tokens}. \tag{3.4}$$

This ensures that throughput values are not inflated or distorted by truncated requests.

Fourth, failed requests are recorded separately from successful requests. Throughput is computed only over successful generated tokens, while the number of failures is reported as a stability metric.

Fifth, the inter-token latency implementation in the client is corrected so that token intervals are computed from consecutive token timestamps rather than from absolute timestamp values. When older logs contain invalid ITL values, ITL is not used in the analysis.

Sixth, profiler overhead is treated explicitly. Node-level CUDA Graph traces can perturb runtime, especially when a workload contains many small kernels. Therefore, throughput values are reported from baseline or graph-level runs, while node-level traces are used only to interpret kernel composition.

Finally, model loading is distinguished from steady-state inference. Full-process Nsight traces often include host-to-device transfers corresponding to model loading. These memory-copy summaries are not interpreted as steady-state inference bottlenecks unless the trace explicitly isolates the inference section.

## 3.10 Data Processing

The raw experimental data consists of benchmark logs, server logs, client result JSON files, Nsight Systems statistics, and power/utilization time-series logs. These files are parsed and summarized into tables for analysis.

For single-sequence experiments, the main reported metrics are:

- prefill throughput;
- decode throughput;
- average power, when available;
- approximate J/token.

For concurrent serving experiments, the main reported metrics are:

- aggregate generated-token throughput;
- median TTFT;
- p95 TTFT, where applicable;
- number of failed requests;
- total generated tokens;
- average power;
- approximate J/token.

For H100 deep profiling, Nsight Systems summaries are used to identify dominant CUDA kernel families and major CUDA API activity. For Orin, `tegrastats` logs are parsed to obtain time-series behavior of power, GPU activity, CPU activity, memory usage, and temperature.

The final analysis reports median values across repeated runs where possible. When a run is used only for profiling rather than performance measurement, this is stated explicitly. In particular, node-level traces are used for qualitative kernel composition and not for final throughput reporting.

# 4

# Baseline Results

## 4.1 H100 Baseline Results

This section presents the baseline results on NVIDIA H100. The goal of the baseline stage is to establish the main phase-dependent and precision-dependent trends before applying deeper profiling. The H100 baseline uses `llama-bench` for single-sequence experiments and `llama-server` with a custom client for concurrent serving.

### 4.1.1 Phase-Aware Performance

Table 4.1 shows the H100 single-sequence baseline with Flash Attention enabled. The results separate prompt processing throughput from token generation throughput. In this thesis, prompt processing is interpreted as prefill throughput, while token generation is interpreted as decode throughput.

The first observation is that prefill throughput is much higher than decode throughput across all formats and scenarios. For example, in S3, BF16 reaches approximately 14923 tokens/s during prefill but only approximately 148 tokens/s during decode. This confirms that prefill and decode should not be collapsed into one aggregate throughput number.

The second observation is that decode throughput remains relatively stable across scenarios for a given precision format, whereas prefill throughput changes more strongly with prompt length and precision. This reflects the different structure of the two phases: prefill can process many prompt tokens together, while decode generates tokens sequentially.

### 4.1.2 Precision Trade-offs

The H100 baseline shows a clear phase-dependent precision trade-off. In all prefill measurements in Table 4.1, BF16 gives the highest prefill throughput. This is especially visible in the prefill-heavy workloads. In S2, BF16 reaches 15713.1 tokens/s, compared with 9905.3 tokens/s for `Q8_0` and 8415.6 tokens/s for `Q4_K_M`. In S3, BF16 again achieves the highest prefill throughput.

Decode shows the opposite trend. Across all four scenarios, `Q4_K_M` gives the highest decode throughput, followed by `Q8_0`, while BF16 is slowest. For example, in S4,

Table 4.1: H100 single-sequence baseline with Flash Attention enabled. Throughput is reported in tokens/s.

| Scenario | Metric | BF16 | Q8_0 | Q4_K_M |
|---|---|---|---|---|
| S1 128:128 | Prefill tok/s | 8944.4 | 6069.9 | 5086.6 |
| | Decode tok/s | 152.2 | 197.3 | 202.9 |
| | Avg power W | 238.8 | 251.1 | 190.6 |
| S2 2048:128 | Prefill tok/s | 15713.1 | 9905.3 | 8415.6 |
| | Decode tok/s | 152.2 | 196.4 | 203.1 |
| | Avg power W | 266.4 | 258.1 | 235.0 |
| S3 4096:1 | Prefill tok/s | 14923.3 | 9691.7 | 8362.5 |
| | Decode tok/s | 148.3 | 175.5 | 190.2 |
| | Avg power W | 188.0 | 221.5 | 230.8 |
| S4 128:1024 | Prefill tok/s | 9339.5 | 6063.5 | 5200.0 |
| | Decode tok/s | 150.7 | 195.6 | 203.3 |
| | Avg power W | 361.2 | 364.2 | 331.1 |

Table 4.2: H100 BF16 strict long-prefill comparison with Flash Attention disabled and enabled.

| Setting | Prefill tok/s | Decode tok/s |
|---|---|---|
| BF16, FA off, pp4096/tg1 | 10815.47 | 133.54 |
| BF16, FA on, pp4096/tg1 | 14731.42 | 136.10 |

`Q4_K_M` reaches 203.3 tokens/s in decode, compared with 195.6 tokens/s for `Q8_0` and 150.7 tokens/s for BF16.

This means that H100 does not have a single best precision format across all phases. BF16 is the best choice for prefill-heavy workloads, while `Q4_K_M` is the best choice for decode-heavy workloads. This observation motivates the later deep profiling analysis, which examines the dominant CUDA kernel families behind these differences.

### 4.1.3 Flash Attention Baseline

Flash Attention is evaluated as a runtime option because it is expected to affect attention-heavy prompt processing. Table 4.2 shows a strict long-prefill comparison for BF16 using `pp4096/tg1`. This setting isolates the prefill-heavy case more directly than shorter prompt configurations.

Enabling Flash Attention improves strict long-prefill throughput from 10815.47 tokens/s to 14731.42 tokens/s, an improvement of approximately 36%. The decode result in this table should not be over-interpreted because the workload generates only one token, so decode timing is sensitive to short-run noise. The main conclusion is that Flash Attention provides a substantial benefit for long-prefill workloads on H100.

Table 4.3: H100 `Q4_K_M` S1 concurrent serving baseline under corrected setup.

| npl | Throughput tok/s | Median TTFT s | Failures |
|---|---|---|---|
| 1 | 184.92 | 0.037 | 0 |
| 32 | 1129.71 | 0.554 | 0 |
| 64 | 1152.57 | 0.854 | 0 |
| 128 | 923.67 | 1.985 | 0 |

### 4.1.4 Concurrent Serving Baseline

In addition to single-sequence benchmarking, H100 is evaluated under concurrent serving using `llama-server`. The concurrent serving sweep focuses on `Q4_K_M`, because it is the most practical format for high-concurrency serving in the current setup. Table 4.3 summarizes the validated S1 serving results.

The serving baseline shows that very low concurrency underutilizes H100. Increasing concurrency from npl1 to npl32 or npl64 greatly improves aggregate throughput. However, increasing to npl128 does not continue the same trend: throughput decreases compared with npl64, and TTFT increases substantially. After correcting the high-concurrency setup, npl128 no longer fails, but it remains a degraded serving point in terms of useful throughput and latency.

The main baseline conclusion for H100 serving is therefore that moderate-to-high concurrency is necessary to use the GPU effectively, but blindly maximizing concurrency is not beneficial. The npl32–npl64 range provides a better practical serving region than npl128 for the tested S1 workload.

## 4.2 Jetson Orin Baseline Results

Jetson AGX Orin is evaluated under multiple power modes: 15W, 30W, 50W, and MAX mode. This is necessary because Orin performance depends not only on workload and precision format, but also on the available power budget. The Orin baseline therefore focuses on how precision ranking, throughput, latency, and energy efficiency change across operating modes.

### 4.2.1 Single-Sequence Results Across Power Modes

Table 4.4 summarizes Orin S3 prefill throughput with Flash Attention enabled. S3 is used as the prefill-heavy stress case. The table shows how the prefill precision ranking changes as the available power budget increases.

At 15W, BF16 clearly gives the highest prefill throughput. At 30W, BF16 still leads, but `Q8_0` becomes closer. At 50W and MAX mode, BF16 and `Q8_0` are nearly tied for long-prefill throughput, while `Q4_K_M` remains lower. This indicates that the prefill precision ranking on Orin changes with power mode and is not identical to H100.

Table 4.5 summarizes S4 decode throughput across power modes. S4 is the decode-

Table 4.4: Jetson Orin S3 prefill throughput across power modes with Flash Attention enabled.

| Power mode | BF16 | Q8_0 | Q4_K_M |
|---|---|---|---|
| 15W | 155.82 | 136.17 | 116.47 |
| 30W | 214.72 | 209.36 | 183.29 |
| 50W | 554.51 | 553.37 | 459.41 |
| MAX | 860.78 | 861.39 | 793.96 |

Table 4.5: Jetson Orin S4 decode throughput across power modes with Flash Attention enabled.

| Power mode | BF16 | Q8_0 | Q4_K_M |
|---|---|---|---|
| 15W | 3.11 | 3.87 | 3.81 |
| 30W | 4.46 | 6.12 | 5.70 |
| 50W | 7.45 | 12.70 | 14.50 |
| MAX | 8.94 | 16.14 | 21.58 |

heavy workload with a short prompt and long generation.

The decode results show a clear power-mode-dependent transition. At 15W and 30W, `Q8_0` gives the highest decode throughput. At 50W and MAX mode, `Q4_K_M` becomes the fastest decode format. This is an important Orin-specific result: unlike H100, where `Q4_K_M` is consistently best for decode, Orin's decode precision ranking depends on the power mode.

The single-sequence baseline therefore shows that Orin requires both phase-aware and power-mode-aware analysis. BF16 remains strong for prefill, especially at lower power, while the best decode format shifts from `Q8_0` at lower power to `Q4_K_M` at higher power.

### 4.2.2 Concurrent Serving Results Across Power Modes

Concurrent serving on Orin is evaluated primarily using `Q4_K_M` on S1. This setup is used because `Q4_K_M` provides the most practical memory footprint for high-concurrency serving. Table 4.6 summarizes the validated serving results used in the final analysis.

The serving results show several trends. First, higher power modes improve throughput and reduce TTFT. Moving from 30W npl64 to 50W npl64 more than doubles throughput, and MAX mode provides the highest throughput among the tested cases.

Second, 50W remains a strong energy-oriented operating mode. Although MAX mode gives higher throughput and lower TTFT, it also consumes substantially more power. For the validated S1 serving cases, 50W gives lower J/token than MAX at both npl64 and npl128.

Third, the interpretation of npl128 changes under the corrected setup. Earlier high-concurrency runs showed failures or incomplete generation. After increasing context

Table 4.6: Validated Jetson Orin `Q4_K_M` S1 serving results across selected power modes and concurrency levels.

| Case | Throughput tok/s | Median TTFT s | Power W | J/token |
|---|---|---|---|---|
| 15W, npl64 | 29.90 | 74.75 | 4.86 | 0.1590 |
| 30W, npl64 | 60.60 | 37.23 | 6.25 | 0.1032 |
| 50W, npl1 | 13.96 | 0.42 | 16.81 | 1.2118 |
| 50W, npl64 | 143.70 | 14.39 | 12.54 | 0.0874 |
| 50W, npl128 | 154.00 | 26.33 | 12.29 | 0.0800 |
| MAX, npl64 | 216.70 | 9.09 | 24.47 | 0.1126 |
| MAX, npl128 | 234.60 | 16.87 | 24.42 | 0.1043 |

size, removing the client connection cap, and validating generated token counts, npl128 completes successfully. However, npl128 also increases TTFT substantially. Therefore, npl128 is not best described as a failure region in the corrected setup; it is better interpreted as a higher-throughput but latency-degraded operating point.

The Orin serving baseline therefore exposes an objective-dependent trade-off. For latency-sensitive use, npl64 is more balanced. For throughput-oriented or energy-per-token-oriented use, npl128 may be attractive, but only if the higher TTFT is acceptable. For power-mode selection, 50W is energy-oriented, while MAX is performance-oriented.

## 4.3 Initial Cross-Platform Observations

The baseline results already reveal that H100 and Jetson Orin expose different optimization problems.

On H100, the dominant baseline trend is phase-dependent precision selection. BF16 is consistently strongest in prefill-heavy workloads, while `Q4_K_M` is consistently strongest in decode-heavy workloads. This suggests that the H100 tuning problem is mainly phase-aware: the best precision format depends on whether the workload is dominated by prefill or decode.

On Jetson Orin, the situation is more power-dependent. Prefill and decode still behave differently, but the best precision format can change with power mode. In particular, `Q8_0` is stronger for decode at 15W and 30W, while `Q4_K_M` becomes stronger at 50W and MAX mode. This means that Orin tuning must consider both inference phase and operating point.

The serving results also differ across platforms. H100 requires enough concurrency to avoid underutilization, but excessive concurrency can reduce useful throughput and increase TTFT. Orin also benefits from concurrency, but power mode becomes a first-order factor. On Orin, 50W provides a strong energy-oriented operating point, while MAX mode provides the highest throughput and lower latency at the cost of higher power.

Overall, the baseline results motivate the deeper profiling performed in the next chap-

ter. The baseline measurements show what configurations perform well or poorly; the deep profiling analysis explains why these differences appear.

# 5

# Deep Profiling and Bottleneck Analysis

The baseline results in the previous chapter show the main performance trends, but they do not fully explain why those trends appear. This chapter uses targeted profiling to connect the observed performance behavior to lower-level causes. The goal is not to profile every configuration in the full experimental matrix. Instead, representative cases are selected to explain the most important bottleneck shifts.

For H100, the focus is on CUDA kernel composition and serving concurrency behavior. For Jetson Orin, the focus is on power-mode behavior, serving latency, energy efficiency, and system-level utilization from `tegrastats`. The results in this chapter are used later in Chapter 6 to derive practical tuning guidelines.

## 5.1 H100 Deep Profiling

The H100 baseline shows a stable phase-dependent pattern: BF16 is strongest in prefill-heavy workloads, while `Q4_K_M` is strongest in decode-heavy workloads. Targeted Nsight Systems profiling is used to explain this difference.

Graph-level Nsight Systems traces are used for performance-oriented profiling, while node-level CUDA Graph traces are used only when the internal kernel composition of CUDA Graphs is needed. Node-level traces can introduce non-negligible overhead, especially for workloads with many short kernels. Therefore, node-level throughput is not used as a final performance number.

### 5.1.1 Why BF16 Wins Long Prefill

A strict long-prefill workload, `pp4096/tg1`, is used to analyze H100 prefill behavior. Under this setting, BF16 reaches 14731.42 tokens/s, while `Q4_K_M` reaches 8199.06 tokens/s. The purpose of the profiling trace is to explain this large gap.

Table 5.1 summarizes the dominant kernel families observed in the graph-level Nsight Systems traces.

In the BF16 trace, the largest kernel family is the H100-optimized BF16 GEMM path, represented by `sm90_xmma_gemm_bf16...` kernels. Flash Attention kernels

Table 5.1: Dominant H100 long-prefill kernel families from graph-level profiling.

| Format | Dominant kernel families | Interpretation |
|---|---|---|
| BF16 | BF16 GEMM + Flash Attention | Dense prefill maps well to H100 kernels |
| Q4_K_M | Quantized matmul + fixup kernels | Quantized path adds specialized overhead |

Table 5.2: Dominant H100 S4 decode kernel families from node-level traces.

| Format | Dominant decode kernels | Share of GPU kernel time |
|---|---|---|
| BF16 | `mul_mat_vec_f` family | 86.9% |
| Q4_K_M | `mul_mat_vec_q` family | 75.5% |
| Q4_K_M | `quantize_q8_1` | 8.2% |

such as `flash_attn_ext_f16` also appear among the dominant kernels. This indicates that long-prefill execution provides enough parallel work for dense GEMM-style kernels and attention kernels to be effective.

In the `Q4_K_M` trace, the dominant kernels are instead quantized matrix multiplication kernels such as `mul_mat_q`, together with stream fixup and quantization helper kernels. This explains why the smaller model footprint of `Q4_K_M` does not make it faster for prefill on H100. The prefill phase benefits more from the efficient BF16 dense execution path than from the reduced weight size of the quantized format.

The bottleneck interpretation is therefore that H100 long-prefill performance is dominated by large matrix and attention operations, where BF16 is able to use highly optimized GPU kernels. This explains the prefill precision ranking observed in the baseline results.

### 5.1.2 Why Q4_K_M Wins Decode

The decode-heavy S4 workload shows the opposite precision ranking. As shown in the baseline results, `Q4_K_M` achieves higher decode throughput than BF16 in S4. To understand why, node-level CUDA Graph traces are used to inspect the internal kernel composition of the decode loop.

Table 5.2 summarizes the dominant decode kernel families.

The node-level traces show that decode is no longer dominated by large GEMM-style kernels. Instead, it is dominated by repeated matrix-vector style kernels. BF16 decode is dominated by BF16 matrix-vector kernels, while `Q4_K_M` decode is dominated by quantized matrix-vector kernels and quantization helpers.

This explains why the precision ranking changes between prefill and decode. BF16 is favorable when the workload exposes large dense matrix operations, as in long prefill. In decode, the execution becomes token-by-token and matrix-vector dominated, making the quantized path more favorable.

This result also shows why a single precision format is not globally optimal on H100. The preferred format depends on the dominant inference phase.

### 5.1.3 Flash Attention in Long Prefill

Flash Attention is evaluated as a targeted kernel-level option for long-prefill workloads. The strict BF16 `pp4096/tg1` comparison in Table 4.2 shows that enabling Flash Attention increases prefill throughput from 10815.47 tokens/s to 14731.42 tokens/s.

The profiling traces explain this improvement. With Flash Attention disabled, the trace contains traditional attention and softmax-related kernels such as `soft_max_f32`. With Flash Attention enabled, the attention path changes to kernels such as `flash_attn_ext_f16` and related fixup kernels.

The important point is that Flash Attention mainly improves the long-prefill phase. It changes the attention execution path and reduces attention-side overhead for long prompts. It does not change the broader H100 phase conclusion: BF16 remains the preferred format for long prefill, while `Q4_K_M` remains preferred for decode-heavy workloads.

### 5.1.4 Serving Concurrency Behavior

H100 serving behavior is analyzed using corrected `Q4_K_M` S1 serving measurements. The corrected setup uses sufficient context size, removes the client-side connection limit, and validates that each request generates the full target length.

As shown in Table 4.3, the serving behavior has three regimes. At npl1, throughput is low because the GPU is underutilized. At npl32 and npl64, throughput increases substantially, indicating that additional concurrency helps expose useful work to the GPU. At npl128, the corrected setup no longer fails, but throughput decreases relative to npl64 and TTFT increases substantially.

The bottleneck interpretation is that H100 needs enough concurrency to avoid underutilization, but excessive concurrency does not continue improving useful throughput. For this workload, npl128 is better described as a degraded high-concurrency region rather than a failure region. The useful serving region is around npl32–npl64, where throughput is high without the same TTFT penalty observed at npl128.

## 5.2 Jetson Orin Deep Profiling

The Orin baseline results show that power mode is a first-order factor. The same workload can have different throughput, energy efficiency, and latency depending on the selected operating point. Unlike H100, where the main bottleneck explanation is based on CUDA kernel families, the Orin analysis relies more on serving logs, generated-token validation, and `tegrastats` time-series data.

The final Orin experiments use corrected high-concurrency settings. In particular, larger context sizes are used for high parallelism, client-side connection limits are removed, and generated-token counts are validated. Under this corrected setup, npl128 no longer fails in the tested S1 serving cases.

### 5.2.1 Power-Mode Behavior

The Orin S1 serving results show that increasing the power mode improves throughput and TTFT, but does not always improve energy efficiency. As shown in Table 4.6, MAX mode provides the highest throughput and lowest TTFT among the tested serving cases, while 50W provides better J/token.

This behavior is different from a simple "higher power is always better" interpretation. MAX mode is performance-oriented: it improves raw throughput and latency, but at substantially higher power. In contrast, 50W is energy-oriented: it provides high throughput while keeping J/token lower than MAX mode.

The low-power modes show the opposite side of the trade-off. At 15W and 30W, throughput is much lower and TTFT is much higher. These modes reduce power draw, but they are not attractive for latency-sensitive serving under the tested workload.

### 5.2.2 npl64 vs. npl128 Serving Trade-off

The corrected Orin results change the interpretation of npl128. Earlier experiments suggested that npl128 was unstable, but after fixing context size and client-side connection limits, npl128 completes successfully for S1. The key trade-off is therefore not failure versus success, but throughput and energy versus TTFT.

At both 50W and MAX mode, npl128 gives higher aggregate throughput and lower J/token than npl64. However, npl128 also increases TTFT substantially. This indicates that npl128 is a latency-degraded high-concurrency point.

The profiling interpretation is that npl128 can be useful when the objective is aggregate throughput or energy per generated token, but it is less suitable when the objective is interactive responsiveness. npl64 remains the more latency-balanced operating point.

### 5.2.3 Precision Shift Across Power Modes

The Orin S4 decode results show a power-mode-dependent precision shift. At 30W, `Q8_0` has higher decode throughput than `Q4_K_M`. At 50W and MAX mode, `Q4_K_M` becomes faster. In all tested S4 decode cases, `Q4_K_M` has lower J/token than `Q8_0`.

This result shows that Orin precision selection cannot be copied directly from H100. On H100, `Q4_K_M` consistently wins decode. On Orin, the decode ranking depends on power mode. At lower power, `Q8_0` remains throughput-competitive, while at higher power, `Q4_K_M` becomes both faster and more energy-efficient.

The long-prefill representative Orin cases also show that BF16 remains strong for prompt processing. However, these Orin long-prefill cases are serving-style representative runs rather than strict `pp4096/tg1` pure prefill runs. They are therefore used to support the qualitative prefill trend, while the main strict prefill evidence comes from H100.

Table 5.3: Representative Jetson Orin `tegrastats` summary for S1 serving.

| Case | Reported power W | GR3D utilization | Max GPU temp C |
|---|---|---|---|
| 15W, npl64 | 4.78 | 94% | 45.3 |
| 30W, npl64 | 6.03 | 91% | 47.2 |
| 50W, npl64 | 12.21 | 86% | 50.3 |
| 50W, npl128 | 12.11 | 87% | 52.0 |
| MAX, npl64 | 23.84 | 88% | 56.5 |
| MAX, npl128 | 23.88 | 81% | 59.7 |

### 5.2.4 Power, Utilization, and Temperature Timelines

The `tegrastats` logs provide system-level context for the Orin results. Table 5.3 summarizes representative time-series statistics for selected S1 serving cases.

The time-series data supports three interpretations. First, GPU utilization is already high in the high-concurrency serving cases, so the difference between 50W and MAX is not simply whether the GPU is active. It is more related to available power and clock headroom. Second, MAX mode roughly doubles reported power compared with 50W, but it does not provide a proportional reduction in J/token. This explains why MAX is faster but not energy-optimal. Third, npl128 does not greatly increase reported power within the same power mode, but it does increase TTFT. This suggests that the npl128 penalty is more related to serving latency and scheduling pressure than to a simple increase in power draw.

The temperature values remain within a reasonable range in the tested runs, and no thermal failure is observed. However, MAX mode runs at higher power and temperature, reinforcing the interpretation that MAX is a performance-oriented setting rather than an energy-oriented setting.

## 5.3 Cross-Platform Bottleneck Comparison

The profiling results show that H100 and Jetson Orin require different optimization strategies.

On H100, the main bottleneck shift is phase-specific. Long-prefill workloads are dominated by BF16 GEMM and attention kernels, making BF16 and Flash Attention effective. Decode-heavy workloads shift toward repeated matrix-vector kernels, where `Q4_K_M` becomes more favorable. H100 serving also needs enough concurrency to avoid underutilization, but too much concurrency can reduce useful throughput and increase TTFT.

On Orin, the main bottleneck shift is power-mode-specific. The same workload changes behavior depending on whether the device is running at 15W, 30W, 50W, or MAX mode. 50W provides an energy-oriented operating point, while MAX mode provides the highest throughput and lower TTFT. Precision behavior is also power-mode-dependent: `Q8_0` can be competitive at lower power, while `Q4_K_M` becomes more favorable at higher power.

These differences show that Orin is not simply a lower-throughput H100. H100 tuning is primarily phase-aware. Orin tuning must be phase-aware, power-mode-aware, and latency-aware. The next chapter turns these profiling observations into practical tuning guidelines.

# 6

# Targeted Tuning and Optimization

## 6.1 Scope of Tuning

The tuning in this thesis is based on the bottleneck analysis from the previous chapters. The goal is not to redesign `llama.cpp`, implement new CUDA kernels, or introduce a new serving scheduler. Instead, the optimization scope is limited to *profiling-driven configuration tuning*.

The tuning decisions considered in this chapter are:

- enabling Flash Attention for long-prefill workloads;
- selecting precision format according to the dominant inference phase;
- selecting serving concurrency according to throughput and TTFT trade-offs;
- selecting Jetson Orin power mode according to energy and latency objectives;
- selecting precision format on Orin according to both phase and power mode.

This scope is intentional. The profiling results show that a significant part of the performance difference comes from choosing the right runtime configuration for the workload and platform. Therefore, the optimization contribution of this thesis is to turn the measured bottlenecks into practical deployment choices.

Measurement corrections, such as increasing context size, removing client-side connection limits, validating full generated-token counts, and fixing invalid ITL computation, are treated as validity fixes rather than optimizations. Their purpose is to ensure that the measurements are reliable. The tuning results below are based on the corrected and validated measurements.

## 6.2 H100 Tuning

The H100 results show that the main tuning problem is phase-aware configuration selection. Long-prefill workloads benefit from BF16 and Flash Attention, while decode-heavy workloads benefit from `Q4_K_M`. For concurrent serving, the GPU needs enough parallel work to avoid underutilization, but the maximum tested concurrency is not the best practical point.

Table 6.1 summarizes the H100 tuning decisions derived from the profiling results.

Table 6.1: H100 tuning decisions derived from profiling.

| Objective | Tuned choice | Evidence from profiling |
|---|---|---|
| Long-prefill performance | Enable Flash Attention | Strict BF16 long-prefill throughput improves by about 36% when Flash Attention is enabled. |
| Prefill-heavy workload | Use BF16 | Long prefill is dominated by BF16 GEMM and attention kernels, and BF16 is about 1.8× faster than `Q4_K_M` in the strict S3 case. |
| Decode-heavy workload | Use `Q4_K_M` | S4 decode is dominated by matrix-vector kernels, where `Q4_K_M` is about 1.3× faster than BF16. |
| Concurrent serving | Use npl32–npl64 | npl1 underutilizes H100, while npl128 gives lower throughput and higher TTFT than npl64 under the corrected setup. |

### 6.2.1 Flash Attention for Long Prefill

The first H100 tuning decision is to enable Flash Attention for long-prefill workloads. As shown in the strict S3 comparison, enabling Flash Attention increases BF16 prefill throughput from 10815.47 tokens/s to 14731.42 tokens/s, corresponding to an improvement of approximately 36.2%.

This tuning is phase-specific. The main benefit appears in long prefill, where attention over a long prompt becomes important. The result should not be interpreted as an equally large decode improvement, since the strict S3 workload only generates one token. Therefore, the practical guideline is to enable Flash Attention when the workload contains substantial prompt processing.

### 6.2.2 Phase-Aware Precision Selection

The second H100 tuning decision is precision selection. The deep profiling results show that H100 should not use one precision format for all workload phases.

For prefill-heavy workloads, BF16 is the tuned choice. The long-prefill traces show that BF16 maps well to H100's optimized dense GEMM and attention kernels. In contrast, the `Q4_K_M` path is dominated by quantized matrix multiplication and fixup kernels, which do not outperform BF16 in prefill.

For decode-heavy workloads, `Q4_K_M` is the tuned choice. The decode traces show that token generation is dominated by repeated matrix-vector operations, where the quantized path is more favorable. This explains why `Q4_K_M` becomes faster than BF16 in S4 decode, even though it is slower in long prefill.

The resulting H100 precision guideline is:

- use BF16 for prefill-heavy workloads;
- use `Q4_K_M` for decode-heavy workloads.

Table 6.2: Jetson Orin tuning decisions derived from profiling.

| Objective | Tuned choice | Evidence from profiling |
|---|---|---|
| Energy-oriented serving | 50W | 50W gives lower J/token than MAX in the validated S1 serving runs. |
| Performance-oriented serving | MAX mode | MAX gives the highest throughput and lower TTFT among the tested power modes. |
| Latency-balanced serving | npl64 | npl64 has lower TTFT than npl128 at both 50W and MAX. |
| Throughput / energy-oriented serving | npl128, if TTFT is acceptable | npl128 improves throughput and J/token under corrected setup, but increases TTFT substantially. |
| Decode at lower power | `Q8_0` for throughput, `Q4_K_M` for energy | At 30W, `Q8_0` has higher S4 decode throughput, while `Q4_K_M` has lower J/token. |
| Decode at 50W / MAX | `Q4_K_M` | At 50W and MAX, `Q4_K_M` is both faster and more energy-efficient than `Q8_0`. |

### 6.2.3 Serving Concurrency Selection

The third H100 tuning decision is serving concurrency. The corrected serving measurements show that very low concurrency underutilizes H100, while excessive concurrency can reduce useful throughput and increase latency.

npl1 gives very low TTFT, but it does not provide enough work to utilize the GPU efficiently. npl32 and npl64 form the practical high-throughput serving region. npl64 gives the highest measured throughput in the corrected S1 serving experiment, while npl32 provides high throughput with lower TTFT. npl128 completes successfully after the measurement corrections, but it gives lower throughput and higher TTFT than npl64.

Therefore, the tuning guideline is not to maximize concurrency blindly. For the tested H100 `Q4_K_M` S1 serving workload, npl32–npl64 is the preferred serving region.

## 6.3 Jetson Orin Tuning

The Jetson Orin tuning problem is different from H100. On Orin, power mode is a first-order variable, and the best configuration depends strongly on whether the deployment objective is energy efficiency, throughput, or latency.

Table 6.2 summarizes the Orin tuning decisions derived from the corrected serving and precision-shift results.

### 6.3.1 Power-Mode Selection

The first Orin tuning decision is power-mode selection. The corrected serving results show that 50W and MAX serve different deployment objectives.

50W is the energy-oriented setting. It provides much higher throughput than 15W or 30W, while keeping J/token lower than MAX in the tested serving cases. MAX mode is the performance-oriented setting. It gives the highest throughput and lower TTFT, but at substantially higher power draw.

This means that Orin should not be evaluated or deployed only under MAX mode. If the goal is lowest energy per generated token, 50W is preferable. If the goal is maximum throughput or lower latency and higher power consumption is acceptable, MAX is preferable.

### 6.3.2 Concurrency Selection

The second Orin tuning decision is serving concurrency. Earlier measurements suggested that npl128 was unstable, but the corrected setup shows that npl128 can complete successfully when context size and client connection limits are handled properly. The interpretation therefore changes from failure to trade-off.

npl128 improves aggregate throughput and J/token compared with npl64 at both 50W and MAX. However, it also increases median TTFT substantially. This makes npl128 useful only when throughput or energy per generated token is more important than responsiveness. For latency-sensitive serving, npl64 is the more balanced point.

The Orin concurrency guideline is therefore:

- use npl64 when latency balance matters;
- use npl128 only when higher throughput or lower J/token is worth the TTFT cost.

### 6.3.3 Power-Mode-Aware Precision Selection

The third Orin tuning decision is precision selection. Unlike H100, Orin does not have a fixed decode precision ranking across all operating points.

At 30W, `Q8_0` gives higher S4 decode throughput than `Q4_K_M`. However, `Q4_K_M` already has lower J/token. At 50W and MAX mode, `Q4_K_M` becomes both faster and more energy-efficient. This shows that Orin precision tuning must account for power mode.

The resulting Orin precision guideline is:

- at lower power, `Q8_0` can be throughput-competitive for decode;
- at 50W and MAX, `Q4_K_M` is the preferred decode format;
- for energy-oriented decode, `Q4_K_M` is generally more favorable in the tested cases.

## 6.4 Practical Guidelines

The tuning results can be summarized as deployment guidelines for the two platforms.

### H100 Guidelines

- Enable Flash Attention for long-prefill workloads.
- Use BF16 for prefill-heavy workloads.
- Use `Q4_K_M` for decode-heavy workloads.
- Avoid npl1 for serving because it underutilizes the GPU.
- Use npl32 or npl64 as practical high-throughput serving points.
- Do not blindly increase concurrency to npl128; under the corrected setup it completes successfully, but gives lower throughput and higher TTFT than npl64 for the tested S1 workload.

### Jetson Orin Guidelines

- Use 50W for energy-oriented serving.
- Use MAX mode for maximum throughput and lower TTFT when power consumption is less constrained.
- Use npl64 when latency balance matters.
- Use npl128 only when higher aggregate throughput or lower J/token is more important than TTFT.
- Select precision format based on both inference phase and power mode.
- For decode-heavy workloads, `Q8_0` can be throughput-competitive at lower power, while `Q4_K_M` is preferable at 50W and MAX.

### Overall Guideline

The main outcome of the tuning analysis is that LLM inference should not be configured using a single global setting. H100 benefits most from phase-aware precision selection, Flash Attention for long-prefill workloads, and moderate-to-high serving concurrency. Jetson Orin requires phase-aware, power-mode-aware, and latency-aware tuning. The best configuration depends on the deployment objective: throughput, latency, or energy efficiency.

# 7

# Discussion

## 7.1 Summary of Findings

The results of this thesis show that `llama.cpp` inference cannot be characterized by a single throughput number or a single best configuration. The main reason is that LLM inference contains different execution regimes. Prefill and decode expose different bottlenecks, and the best precision format, serving configuration, and power mode depend on which regime dominates the workload.

On H100, the most important finding is the stable phase-dependent precision trade-off. BF16 is most favorable for long-prefill workloads, while `Q4_K_M` is most favorable for decode-heavy workloads. The deep profiling results explain this difference. Long prefill maps well to optimized BF16 GEMM and Flash Attention kernels, while decode-heavy execution shifts toward repeated matrix-vector kernels, where the quantized `Q4_K_M` path becomes more favorable. This means that H100 tuning should be phase-aware: BF16 should be selected for prefill-heavy workloads, while `Q4_K_M` should be selected for decode-heavy workloads.

The H100 serving results also show that concurrency must be selected carefully. Very low concurrency underutilizes the GPU, while very high concurrency does not necessarily improve useful throughput. Under the corrected setup, npl128 no longer fails, but it gives lower throughput and higher TTFT than npl64 for the tested S1 workload. Therefore, the practical serving region on H100 is around npl32–npl64 rather than the maximum tested concurrency.

On Jetson Orin, the main finding is that power mode is a first-order tuning variable. The Orin results show that 50W is the energy-oriented operating point, while MAX mode is the performance-oriented operating point. MAX provides the highest throughput and lower TTFT, but it also consumes substantially more power. In contrast, 50W provides a better energy-per-token trade-off in the tested serving workloads.

The Orin results also show that precision behavior is power-mode-dependent. In the S4 decode-heavy workload, `Q8_0` gives higher throughput than `Q4_K_M` at 30W, while `Q4_K_M` becomes faster at 50W and MAX mode. In energy terms, `Q4_K_M` is more favorable in the tested S4 decode cases. This differs from H100, where `Q4_K_M` consistently wins decode. Therefore, Orin requires both phase-aware and power-mode-aware tuning.

A further finding is that high-concurrency behavior must be interpreted carefully. Earlier high-concurrency runs appeared unstable, but corrected experiments with larger context size, removed client connection limits, and full generated-token validation changed the interpretation. In the final Orin results, npl128 is stable for S1 and can improve throughput and J/token, but it also substantially increases TTFT. Thus, npl128 is not a simple failure region; it is a latency-degraded high-concurrency point.

## 7.2 H100 and Orin as Different Optimization Problems

The comparison between H100 and Jetson Orin shows that the two platforms require different tuning strategies. H100 is a datacenter-class GPU with high compute throughput and memory bandwidth. Its main optimization problem in this thesis is selecting the right precision format and serving concurrency for the dominant workload phase. The key question on H100 is therefore: *which phase dominates, and how should precision and concurrency be selected for that phase?*

For H100, long-prefill workloads benefit from BF16 and Flash Attention because they expose enough dense parallel work for the GPU. Decode-heavy workloads behave differently because token generation is sequential and dominated by repeated matrix-vector operations. This makes `Q4_K_M` more favorable for decode. The serving results then add another dimension: concurrency must be high enough to avoid underutilization, but not so high that TTFT increases and throughput degrades.

Jetson Orin exposes a different optimization problem. Because Orin is an edge-oriented platform with configurable power modes, the best operating point depends strongly on the energy-latency-throughput objective. The same model and workload can behave differently at 15W, 30W, 50W, and MAX mode. This makes power mode a central part of the tuning problem rather than a background implementation detail.

The Orin results also show that the platform is not simply a slower version of H100. If Orin were only a uniformly slower H100, the same precision rankings and serving conclusions would hold at a lower absolute throughput. Instead, the results show that the decode precision ranking changes with power mode, and that power-mode selection changes the energy and latency trade-off. This means that cross-platform inference tuning must consider architectural and operating constraints, not only absolute throughput differences.

Overall, H100 tuning is primarily phase-aware, while Orin tuning is phase-aware, power-mode-aware, and latency-aware. This distinction is one of the central outcomes of the thesis.

## 7.3 Interpretation of the Tuning Results

The targeted tuning results should be understood as configuration-level optimization. The thesis does not claim to introduce a new inference engine, a new CUDA kernel, or a new scheduling algorithm. Instead, it shows that careful configuration choices can produce meaningful improvements or better trade-offs when guided by profiling.

For H100, enabling Flash Attention is a clear example of a targeted tuning choice. The strict long-prefill experiment shows a substantial prefill improvement when Flash Attention is enabled. However, this optimization is phase-specific: it is most useful for long-prefill workloads and should not be interpreted as an equally large decode optimization.

Phase-aware precision selection is another tuning result. Choosing BF16 for long prefill and `Q4_K_M` for long decode is not simply a restatement of the benchmark table; it is supported by the profiling results that show different dominant kernel families in the two phases. This makes the recommendation more robust than choosing a precision format based only on aggregate throughput.

For Orin, power-mode selection is the most important tuning decision. The results show that 50W and MAX serve different objectives. 50W is more energy-oriented, while MAX is more performance-oriented. Similarly, npl64 and npl128 represent different serving trade-offs. npl64 is more latency-balanced, while npl128 can improve throughput and J/token at the cost of higher TTFT. The correct choice depends on whether the target deployment prioritizes responsiveness, throughput, or energy efficiency.

These tuning results demonstrate the value of profiling-driven analysis. Without separating phases, measuring power, validating concurrency behavior, and checking generated-token completeness, several conclusions would have been misleading. In particular, npl128 would have been incorrectly described as a failure region rather than a latency-degraded high-concurrency point under corrected setup.

## 7.4 Threats to Validity

Several threats to validity should be considered when interpreting the results.

First, the energy measurements are approximate. Energy per token is computed using average measured power and generated-token throughput. This provides a useful efficiency indicator, but it is not exact phase-specific energy accounting. Prefill and decode may have different power behavior within a single run, and this is not fully separated by the current measurement method.

Second, H100 and Orin use different power measurement sources. H100 power is sampled using NVIDIA GPU tooling, while Orin power is collected from `tegrastats`. These sources are useful within each platform, but they are not identical measurement methods. Therefore, the thesis emphasizes within-platform comparisons, such

as Orin 50W versus MAX, more strongly than absolute cross-platform energy comparisons.

Third, profiler overhead can affect measured performance. This is especially important for node-level CUDA Graph tracing. In this thesis, node-level traces are used only to inspect kernel composition, not to report final throughput. Throughput conclusions are based on baseline or graph-level runs.

Fourth, high-concurrency serving results are sensitive to measurement setup. Context size, client connection limits, timeout behavior, and generated-token validation can all affect whether a high-concurrency run is valid. This thesis addresses this threat by using corrected high-concurrency settings and validating that requests generate the full target length. However, it also shows that uncorrected high-concurrency experiments can be misleading.

Fifth, the Orin long-prefill representative profiling cases are not always strict `pp4096/tg1` pure prefill runs. They are useful for understanding long-prefill-like behavior, but they should not be interpreted as identical to the strict H100 S3 setup unless the exact prompt and generation lengths match.

Finally, the workload set is controlled and synthetic. S1–S4 are designed to isolate balanced, prefill-heavy, and decode-heavy regimes, but they do not cover all real-world prompt distributions, multi-turn conversations, or bursty production traffic. The results should therefore be interpreted as controlled profiling and tuning evidence, not as a complete production serving benchmark.

## 7.5 Limitations

The first limitation is model coverage. The thesis focuses on Llama 3.1 8B. This keeps the study controlled and makes cross-platform comparison manageable, but the conclusions may differ for substantially smaller or larger models. Larger models may create different memory pressure, KV-cache behavior, or offloading constraints, especially on Orin.

The second limitation is precision coverage. The study evaluates BF16, `Q8_0`, and `Q4_K_M`. These formats are sufficient to show important precision trade-offs, but they do not cover all quantization schemes supported by `llama.cpp` or other inference frameworks. Other formats may expose different performance-quality trade-offs.

The third limitation is that the thesis does not evaluate model quality. This limitation is important because edge-oriented quantized deployments often require simultaneous consideration of energy efficiency, latency, and output accuracy rather than throughput alone [16]. The study focuses on performance, energy, latency, and bottlenecks. It does not measure accuracy, perplexity, or downstream task quality under different quantization formats. Therefore, precision recommendations should be interpreted as performance-oriented recommendations, not as complete quality-performance trade-off decisions.

The fourth limitation is that the optimization scope is configuration-level tuning.

This is intentional, but it means the thesis does not implement new CUDA kernels, new memory managers, or new serving schedulers. The results show how to choose better existing configurations, not how to redesign the inference engine.

The fifth limitation is the depth of platform-specific profiling. H100 deep profiling uses Nsight Systems and node-level traces for selected cases, which allows kernel family analysis. Orin profiling relies more on `tegrastats`, server logs, and timing measurements. This is appropriate for the Orin power-mode analysis, but it means the Orin kernel-level explanation is less detailed than the H100 kernel-level explanation.

## 7.6 Future Work

Several extensions are possible.

First, future work could evaluate more model sizes and architectures. Testing larger models would show whether the same phase-aware and power-mode-aware conclusions hold when memory pressure becomes stronger, especially on Orin. Another direction is to extend the evaluation from single-device execution to low-resource distributed settings, where recent systems have shown that 30B–70B-scale inference can be spread across heterogeneous home clusters [17].

Second, future work could include model quality evaluation. Combining throughput, latency, energy, and quality would allow a more complete precision-format recommendation. This would be especially important when comparing `Q8_0` and `Q4_K_M`.

Third, future work could use more realistic serving traces. The current workloads are controlled and useful for isolating bottlenecks, but production traffic often contains variable prompt lengths, variable output lengths, bursts, and multi-turn interactions. A trace-driven serving evaluation would show how the tuning guidelines behave under more realistic conditions.

Fourth, future work could perform deeper Orin kernel-level profiling. The current Orin analysis focuses on power modes, system utilization, and serving behavior. More detailed CUDA-level profiling on Orin could help explain the power-mode-dependent precision shift more directly.

Fifth, future work could implement actual runtime adaptation. A natural extension is to explore activation-aware hybrid execution, as recent local inference systems show that exploiting activation locality can materially reduce memory pressure and improve throughput on constrained hardware [18]. For example, an inference server could select precision, concurrency, or power mode dynamically based on workload phase, latency target, or power budget. This thesis provides evidence for such policies, but does not implement an adaptive runtime system.

Finally, future work could investigate more advanced optimizations inside `llama.cpp`, such as improved scheduling, memory reuse, KV-cache management, or platform-specific kernel tuning. Future work could also evaluate edge-specific low-bit kernels rather than relying only on generic runtime backends, since specialized mixed-

precision matrix multiplication support can substantially change the effectiveness of aggressive quantization [19]. These are beyond the scope of this thesis, but the profiling results identify where such optimizations may be useful.

# 8

# Conclusion

## 8.1 Answers to Research Questions

This thesis evaluated the performance bottlenecks of `llama.cpp` on NVIDIA H100 and Jetson AGX Orin. The study used phase-aware workloads, multiple precision formats, concurrent serving experiments, power measurements, and targeted profiling to understand how inference behavior changes across platforms and operating conditions.

**RQ1: How do prefill and decode differ in their performance characteristics when running `llama.cpp`?**

The results show that prefill and decode behave very differently. Prefill generally achieves much higher throughput because it processes many prompt tokens together and exposes more parallel work to the backend. Decode is slower and more latency-sensitive because it generates tokens sequentially. On H100, profiling shows that long prefill is dominated by GEMM and attention kernels, while decode-heavy execution is dominated by repeated matrix-vector kernels. This confirms that aggregate tokens/s alone is not sufficient for understanding LLM inference performance.

**RQ2: How do precision formats such as BF16, `Q8_0`, and `Q4_K_M` affect performance and energy efficiency across different workload phases?**

The best precision format depends on the inference phase. On H100, BF16 is best for long-prefill workloads, while `Q4_K_M` is best for decode-heavy workloads. The profiling results explain this behavior: BF16 uses efficient dense GEMM and attention kernels during prefill, while `Q4_K_M` is more favorable during decode because execution shifts toward repeated matrix-vector operations. On Orin, precision behavior is also phase-dependent, but additionally changes with power mode. For example, `Q8_0` is throughput-competitive for decode at lower power, while `Q4_K_M` becomes faster and more energy-efficient at 50W and MAX mode.

**RQ3: How do bottlenecks differ between NVIDIA H100 and Jetson AGX Orin under matched `llama.cpp` workloads?**

H100 and Orin expose different tuning problems. H100 is mainly limited by phase-specific execution behavior and serving utilization. Its optimization problem is primarily phase-aware: choose BF16 and Flash Attention for long prefill, choose `Q4_K_M` for decode-heavy workloads, and use enough concurrency to avoid underutilization.

Orin is not simply a slower H100. Its behavior depends strongly on power mode, and the best configuration depends on the trade-off between throughput, TTFT, and energy per token. Orin therefore requires power-mode-aware and latency-aware tuning in addition to phase-aware analysis.

**RQ4: How do serving concurrency and Jetson Orin power modes affect throughput, latency, stability, and energy per generated token?**

Serving concurrency improves aggregate throughput up to a useful region, but higher concurrency can substantially increase TTFT. On H100, npl32–npl64 provides the practical high-throughput region, while npl128 is degraded under the corrected setup because it gives lower throughput and higher TTFT than npl64. On Orin, npl128 is valid after correcting the measurement setup and can improve throughput and J/token, but it also significantly increases TTFT. Orin power modes also strongly affect the result: 50W is the most energy-oriented operating point in the tested serving cases, while MAX mode provides the highest throughput and lower TTFT at higher power.

**RQ5: What practical tuning choices can be derived from profiling results for datacenter and edge deployment?**

The final tuning guidelines are platform-specific. For H100, the recommended choices are to enable Flash Attention for long-prefill workloads, use BF16 for prefill-heavy workloads, use `Q4_K_M` for decode-heavy workloads, and use npl32–npl64 for practical concurrent serving. For Jetson Orin, the recommended choices depend on deployment objective: use 50W for energy-oriented serving, use MAX mode for highest throughput and lower TTFT, use npl64 when latency balance matters, and use npl128 only when higher throughput or lower J/token is worth the TTFT cost. Precision selection on Orin should also consider power mode.

## 8.2 Main Contributions

The thesis provides a phase-aware performance evaluation of `llama.cpp` on both a datacenter GPU and an edge platform. By separating prefill and decode, the study shows that LLM inference cannot be summarized by one aggregate throughput metric.

Besides, it provides a cross-platform comparison between NVIDIA H100 and Jetson AGX Orin under a shared experimental framework. The comparison shows that H100 and Orin require different tuning strategies: H100 is primarily phase-aware, while Orin is phase-aware, power-mode-aware, and latency-aware.

It uses targeted deep profiling to explain the observed baseline behavior. On H100, Nsight Systems traces show why BF16 wins long prefill and why `Q4_K_M` wins decode. On Orin, serving logs and `tegrastats` time-series data show how power mode and concurrency affect throughput, TTFT, power, and energy per token.

The thesis identifies and corrects several measurement validity issues, including high-concurrency context sizing, client-side connection limits, generated-token validation,

and invalid ITL computation. These corrections change the interpretation of high-concurrency behavior, especially for npl128. It turns profiling observations into practical configuration-level tuning guidelines. These guidelines do not require modifying `llama.cpp` kernels or model architecture. Instead, they show how existing runtime choices can be selected more effectively based on workload phase, platform, and deployment objective.

## 8.3 Final Remarks

The central conclusion of this thesis is that efficient LLM inference requires platform-aware and phase-aware tuning. A configuration that is optimal for one phase, platform, or objective may be suboptimal for another. On H100, the key is to match precision format and serving concurrency to the dominant workload phase. On Jetson Orin, the key is to additionally account for power mode and latency-energy trade-offs.

The results also show the value of profiling-driven analysis. Baseline measurements reveal what happens, but profiling explains why it happens. This distinction is important because practical tuning decisions should be based on bottleneck behavior, not only on aggregate throughput. By combining baseline benchmarking, targeted profiling, power measurement, and small configuration-level optimizations, this thesis provides a practical methodology for evaluating and tuning `llama.cpp` inference on both datacenter and edge hardware.

# Bibliography


[1] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[2] G. Gerganov and contributors, *llama.cpp: Llm inference in c/c++*, `https://github.com/ggml-org/llama.cpp`, Accessed: 2026-05-29, 2023.

[3] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "LLM.int8(): 8-bit matrix multiplication for transformers at scale," *arXiv preprint arXiv:2208.07339*, 2022. arXiv: `2208.07339`.

[4] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022. arXiv: `2210.17323`.

[5] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "SmoothQuant: Accurate and efficient post-training quantization for large language models," *arXiv preprint arXiv:2211.10438*, 2022. arXiv: `2211.10438`.

[6] J. Lin et al., "AWQ: Activation-aware weight quantization for llm compression and acceleration," *arXiv preprint arXiv:2306.00978*, 2023. arXiv: `2306.00978`.

[7] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 16 344–16 359.

[8] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023. arXiv: `2307.08691`.

[9] P. Patel et al., "Splitwise: Efficient generative llm inference using phase splitting," in *Proceedings of the 51st Annual International Symposium on Computer Architecture*, 2024, pp. 118–132.

[10] A. Agrawal et al., "Taming throughput-latency tradeoff in llm inference with sarathi-serve," in *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[11] Y. Zhong et al., "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," *arXiv preprint arXiv:2401.09670*, 2024. arXiv: `2401.09670`.

[12] W. Kwon et al., "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ACM, 2023, pp. 611–626. DOI: `10.1145/3600006.3613165`.

[13] J. Delavande, R. Pierrard, and S. Luccioni, "Understanding efficiency: Quantization, batching, and serving strategies in llm energy use," *arXiv preprint arXiv:2601.22362*, 2026. arXiv: `2601.22362`.

[14] M. Arya and Y. Simmhan, "Understanding the performance and power of llm inferencing on edge accelerators," *arXiv preprint arXiv:2506.09554*, 2025. arXiv: 2506.09554.

[15] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, et al., "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024. arXiv: 2407.21783.

[16] E. J. Husom et al., "Sustainable llm inference for edge ai: Evaluating quantized llms for energy efficiency, output accuracy, and inference latency," *arXiv preprint arXiv:2504.03360*, 2025. arXiv: 2504.03360.

[17] Z. Li, T. Li, W. Feng, M. Guizani, and H. Yu, "PRIMA.CPP: Speeding up 70b-scale llm inference on low-resource everyday home clusters," *arXiv preprint arXiv:2504.08791*, 2025. arXiv: 2504.08791.

[18] Y. Song, Z. Mi, H. Xie, and H. Chen, "PowerInfer: Fast large language model serving with a consumer-grade gpu," *arXiv preprint arXiv:2312.12456*, 2023. arXiv: 2312.12456.

[19] J. Wei et al., "T-MAC: Cpu renaissance via table lookup for low-bit llm deployment on edge," *arXiv preprint arXiv:2407.00088*, 2024. arXiv: 2407.00088.

# A

# Appendix 1

This appendix includes the main scripts used for benchmarking, concurrent serving experiments, power monitoring, and Jetson Orin profiling. The source code is included to improve reproducibility and to make the experimental workflow explicit.

The complete source code is available at: `https://github.com/millqi/Performance-Bottleneck-Evaluation-of-llama.cpp-on-Jetson-and-H100`.