Universiteit Leiden

Master Computer Science

CPU vs. NPU: A Systematic Performance Evaluation of Mobile LLM Inference on Snapdragon Heterogeneous SoC

Name: Pu Li
Student ID: s3997944

Date: 08/04/2026

Specialisation: Data Science: Computer Science

1st supervisor: Qinyu Chen
2nd supervisor: Chang Gao (TU Delft)

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Deploying large language models (LLMs) on mobile devices demands efficient utilization of heterogeneous computing resources, yet systematic performance characterization across CPU and NPU backends remains limited. This thesis constructs a reproducible benchmarking framework based on llama.cpp and the Snapdragon 8 Gen 3 platform (Hexagon v75 NPU), and proposes a fine-grained profiling methodology combining operator-level instrumentation with an OPMASK-based pipeline decomposition that isolates the NPU inference process into communication, quantization, and computation stages.

Experiments across four models (Llama 3.1-8B, Llama 3.2-3B, Qwen3-4B, Qwen3-8B) under varied runtime configurations (batch size, prompt length, CPU threads, and NPU offloading layers) reveal a stage-specific performance reversal: multi-core CPUs outperform the NPU in the compute-intensive Prefill stage (NPU 1.27–1.62× slower), while the NPU achieves a 1.5–1.7× speed advantage in the memory-bandwidth-bound Decode stage. OPMASK analysis shows that core computation (i.e., the actual execution time of operators on the NPU) dominates NPU execution (85–99% of total time), but Decode-stage communication overhead surges to 9.9–13.0%. High scheduling overhead for lightweight operators (e.g., `RMS_NORM` and `ADD`), where the dispatch time reaches 8–22× the actual execution time, combined with cross-backend fallback penalties limits end-to-end Decode speedup to only 1.05–1.20×. Based on these findings, a split-phase heterogeneous deployment strategy is proposed, and key optimization paths, including operator ecosystem completion and deep operator fusion, are identified.

Contents

1 Introduction

In recent years, large language models (LLMs) have achieved remarkable progress in natural language processing tasks [16] and have gradually expanded from cloud-based inference to edge devices and mobile deployment [19]. Performing LLM inference on mobile devices not only reduces latency and enhances user privacy but also decreases dependence on network connectivity, attracting broad attention from both academia and industry. However, constrained by the computational capability, memory bandwidth, and power budget of mobile devices, achieving efficient LLM inference in resource-limited environments remains a challenging problem.

Existing research on mobile LLM inference primarily focuses on three directions. The first direction is model-level optimization, including quantization and compression techniques, which reduces computational overhead by lowering model precision or parameter scale [5, 12, 20]. The second direction is system-level inference framework design, which improves inference efficiency through optimized execution scheduling, memory management, parallelism strategies, and dedicated hardware design [10, 15, 24]. The third direction is hardware-oriented performance evaluation and benchmarking, measuring and comparing inference performance across different devices and runtime configurations [18, 25, 4]. These studies provide an important foundation for mobile LLM deployment, yet certain limitations remain.

In terms of hardware evaluation, Zhang and Huang [25] compared the inference performance of CPU and GPU backends on an iOS platform, revealing a counter-intuitive phenomenon in which the CPU can outperform GPU acceleration under specific configurations. They further identified matrix multiplication as the primary inference bottleneck and highlighted the significant impact of thread configuration on performance. This work challenged the conventional assumption that "GPU is always optimal," but its scope was limited to a single iOS platform and a single model scale, without analyzing NPU backend behavior or delving into operator-level execution details. Overall, the existing body of work still exhibits the following shortcomings. First, most studies evaluate performance solely through aggregate metrics (such as latency or throughput), lacking fine-grained analysis of the internal mechanisms of inference, particularly at the operator level. Second, systematic comparisons between CPUs and dedicated accelerators (such as NPUs) on mobile heterogeneous platforms remain insufficient, especially reproducible analyses under a unified experimental framework. Moreover, existing studies rarely distinguish between different stages of LLM inference (e.g., Prefill and Decode), offering limited insight into the performance differences and their causes— factors that directly affect hardware selection and parameter configuration strategies in practice.

To address these issues, this thesis targets mobile heterogeneous computing platforms (CPU and Hexagon NPU) and constructs a reproducible LLM inference performance benchmarking framework, along with a fine-grained profiling methodology for systematic analysis of the inference process. Specifically, this work builds upon the Snapdragon 8 Gen 3 platform and extends the llama.cpp inference framework by introducing operator-level profiling mechanisms and an OP-MASK pipeline decomposition method. This decomposes the NPU inference process into three stages—communication, quantization, and computation— enabling quantitative analysis of queue time and computation time. On this basis, we further compare the performance of CPU and NPU under various runtime configurations (such as batch size, prompt length, and thread count) as well as the overall system power consumption, and attribute the performance differences and energy

efficiency bottlenecks observed in the Prefill and Decode stages.

The main contributions of this thesis are as follows:

1. A reproducible benchmarking framework and fine-grained profiling methodology are constructed. This combines operator-level instrumentation with OPMASK (a low-level execution control mechanism that selectively isolates hardware pipeline stages) to enable systematic throughput evaluation and stage-wise modeling of the NPU process.

2. A systematic performance and energy evaluation is conducted, comparing the CPU and Hexagon NPU across multiple models and runtime configurations. This quantifies both inference throughput patterns and the total on-device system power consumption.

3. The performance reversal phenomenon between CPU and NPU across the Prefill and Decode stages is revealed. Its causes are analyzed from three perspectives: operator execution efficiency, scheduling overhead, and cross-backend fallback.

4. Based on experimental results, the key performance bottlenecks in mobile LLM deployment are summarized, and optimization recommendations with practical guidance are provided.

The remainder of this thesis is organized as follows. Section 2 introduces background and related work. Section 3 describes the experimental platform and methodology. Section 4 presents the system-level performance and power consumption comparison results. Section 5 provides the micro-architectural and operator-level analysis. Section 6 discusses optimization strategies and future work. Section 7 concludes the thesis.

2 Background and Related Work

2.1 Mobile LLM Inference and Optimization

2.1.1 Mobile Model Architecture Evolution

In recent years, models optimized for on-device deployment have emerged, such as Llama 3.2 [14], MiniCPM [23], and Gemma [6]. Although these models have been compressed in terms of parameter scale, their performance on complex reasoning tasks often falls short of cloud-based large models. Consequently, how to improve inference efficiency through algorithmic methods under the limited computation and memory bandwidth constraints of edge devices has become a core research focus.

2.1.2 Model Quantization Techniques

Quantization is a key technique for enabling on-device deployment, aiming to reduce memory access overhead by lowering the bit-width of weights and activations.

- **Mainstream algorithms:** Currently, widely adopted post-training quantization (PTQ) algorithms include GPTQ [5] and AWQ [12], which achieve 4-bit weight compression using a small amount of calibration data.

- **Precision optimization:** To address the outlier problem in activations, SmoothQuant [20] and DuQuant [11] smooth the distribution through mathematical transformations. QuaRot [3] and SpinQuant [13] further introduce rotation transforms, targeting full compression of weights, activations, and even the KV cache to 4-bit while maintaining high accuracy.
- **Quantization granularity:** Research has shown that traditional per-channel quantization can cause severe accuracy loss in on-device inference tasks. As a result, fine-grained group quantization has gradually become the standard configuration for mobile deployment [5, 12].

2.1.3 On-Device Inference Frameworks and Heterogeneous Scheduling

At the system implementation level, llama.cpp [7] serves as a representative framework from the open-source community, implementing multi-backend abstraction support through the GGML library.

- **Heterogeneous cooperation:** Existing works such as llm.npu [21] explore the potential of NPU-accelerated Prefill stages, while HeteroInfer [4] attempts tensor-partitioned cooperative inference between GPU and NPU.
- **Breaking bottlenecks:** Zhang and Huang [25] revealed the competitive advantage of mobile CPUs in certain scenarios. For memory-constrained Decode stages, PowerInfer-2 [22] and PowerServe [17] improve throughput through sparse operator optimization and floating-point/fixed-point hybrid partitioning strategies, respectively. Additionally, speculative decoding [9] and test-time scaling [8] provide new optimization paths for balancing inference performance and generation quality.

2.2 Mobile Heterogeneous Computing Platforms and NPU Architecture

For hardware acceleration of mobile LLM inference, the Snapdragon 8 Gen 3 platform used in this study integrates the Hexagon v75 NPU, which belongs to the Hexagon Tensor Processor (HTP) architecture. The NPU employs a typical "vector + matrix" hybrid computing architecture, whose core components are illustrated in Figure 1.

2.2.1 Core Computation Units

The computational resources of the Hexagon NPU consist of three types of cores working in coordination:

- **Scalar Core:** Comprises 6 to 8 hardware threads and employs a Very Long Instruction Word (VLIW) architecture. It is responsible for program logic control, operator scheduling, and non-vectorized instruction execution.
- **Vector Unit (HVX):** The Hexagon Vector eXtension primarily handles general-purpose vector operations (such as normalization and activation functions). Each HVX unit is equipped with 32 vector registers of 1024-bit width (128 bytes).

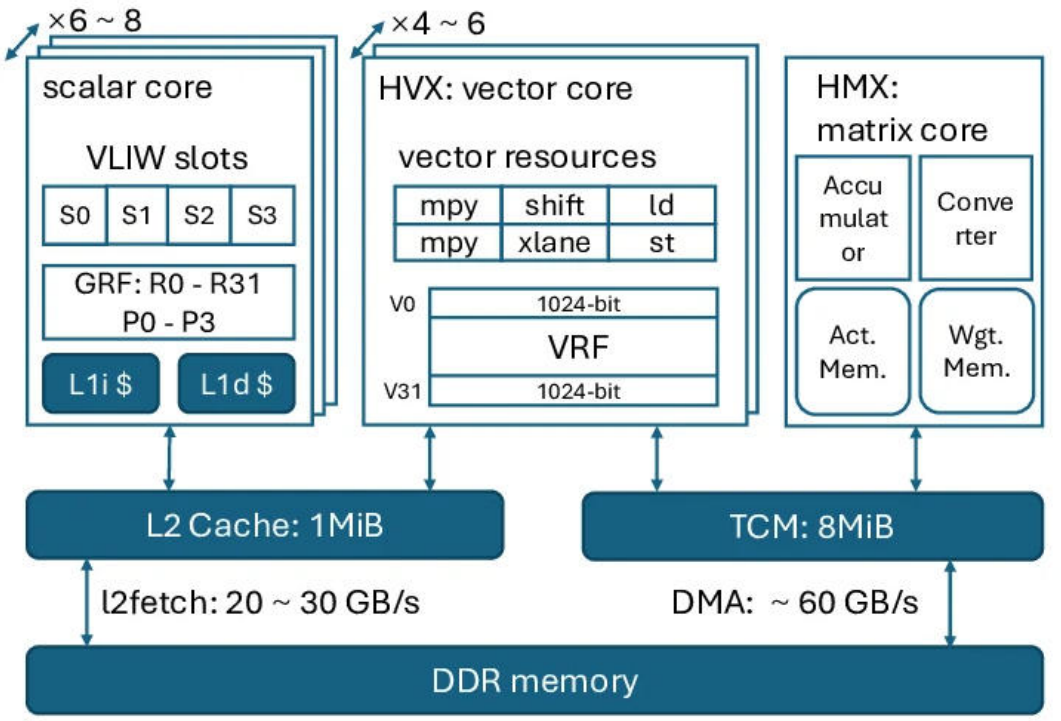


Figure 1: Hexagon NPU Architecture. Source: [8].

- **Matrix Unit (HMX):** The Hexagon Matrix eXtension is the core component specifically hardened for matrix multiplication (GEMM) within the NPU. HMX supports multiple computation precisions including INT4, INT8, INT16, and FP16, and serves as the primary compute engine for dense tensor operations in LLM inference.

2.2.2 Memory Subsystem and Execution Model

To mitigate the limited memory bandwidth bottleneck on mobile devices, the Hexagon NPU introduces a multi-level storage architecture:

- **Memory hierarchy:** This includes a 1 MiB shared L2 cache and 8 MiB of on-chip Tightly Coupled Memory (TCM). TCM is a software-managed on-chip SRAM that supports low-latency, high-speed access for HVX and HMX, thereby reducing frequent scheduling to DDR main memory.
- **Execution mechanism:** Unlike the SIMT (Single Instruction, Multiple Threads) model commonly used in GPUs, the NPU employs a SIMD (Single Instruction, Multiple Data)-based execution model. By reducing hardware-level control logic overhead, the NPU exhibits higher execution efficiency and better energy efficiency when processing regular tensor computations compared to GPUs.

The above hardware features form the foundation for NPU-accelerated LLM inference. The specific communication protocols between CPU and NPU, operator scheduling, and weight repack-

ing execution details are introduced in Section 2.3.3 in conjunction with the llama.cpp framework implementation.

2.3 The llama.cpp Inference Framework

The performance evaluation experiments in this study are based on the llama.cpp [7] inference framework. llama.cpp is an open-source C/C++ LLM inference engine whose underlying numerical computation relies on the GGML tensor computation library. This section introduces the overall architecture of the framework, the inference execution flow, and the working mechanism of its Hexagon NPU backend, providing the necessary technical background for subsequent experimental analysis.

2.3.1 GGML Computation Graph and Backend Abstraction

GGML is the underlying tensor computation library used by llama.cpp. Its core design is based on a **static computation graph** mechanism. The model's forward pass is represented as a directed acyclic graph (DAG), where each node corresponds to a tensor operation operator (e.g., matrix multiplication `MUL_MAT`, normalization `RMS_NORM`, rotary position encoding `ROPE`, attention computation `FLASH_ATTN_EXT`, etc.), and edges represent tensor data dependency relationships. For each inference call, the framework first constructs the complete computation graph based on the model architecture, then hands it to the execution engine for computation.

GGML adopts a **backend abstraction** architecture to support heterogeneous computing. Each hardware platform (CPU, GPU, NPU, etc.) is encapsulated as an independent backend, with each backend implementing a unified interface, including operator support query (`supports_op`), buffer allocation (`buffer_alloc`), and graph computation (`graph_compute`). This design allows upper-level model code to remain agnostic to underlying hardware differences, enabling the same computation graph to execute on different backends.

In heterogeneous execution scenarios, GGML uses a **backend scheduler** to partition and assign the computation graph. Before graph computation, the scheduler queries each operator one by one to determine whether the target backend supports it: if supported, the operator is assigned to the corresponding backend; if not, it falls back to the default CPU backend. When operators handled by different backends coexist in a computation graph, the scheduler splits the graph into multiple sub-graphs (splits) and schedules them for execution in dependency order. Switching between sub-graphs involves synchronization waits and data transfers, introducing additional overhead. This will be analyzed in detail in Section 5.

2.3.2 Inference Execution Flow

The LLM inference process based on the llama.cpp framework can be divided into three main phases: initialization, Prefill, and Decode. The overall execution flow is shown in Figure 2.

Initialization phase. This phase comprises backend registration, model loading, and context creation. Upon framework startup, the system automatically loads and registers available hardware

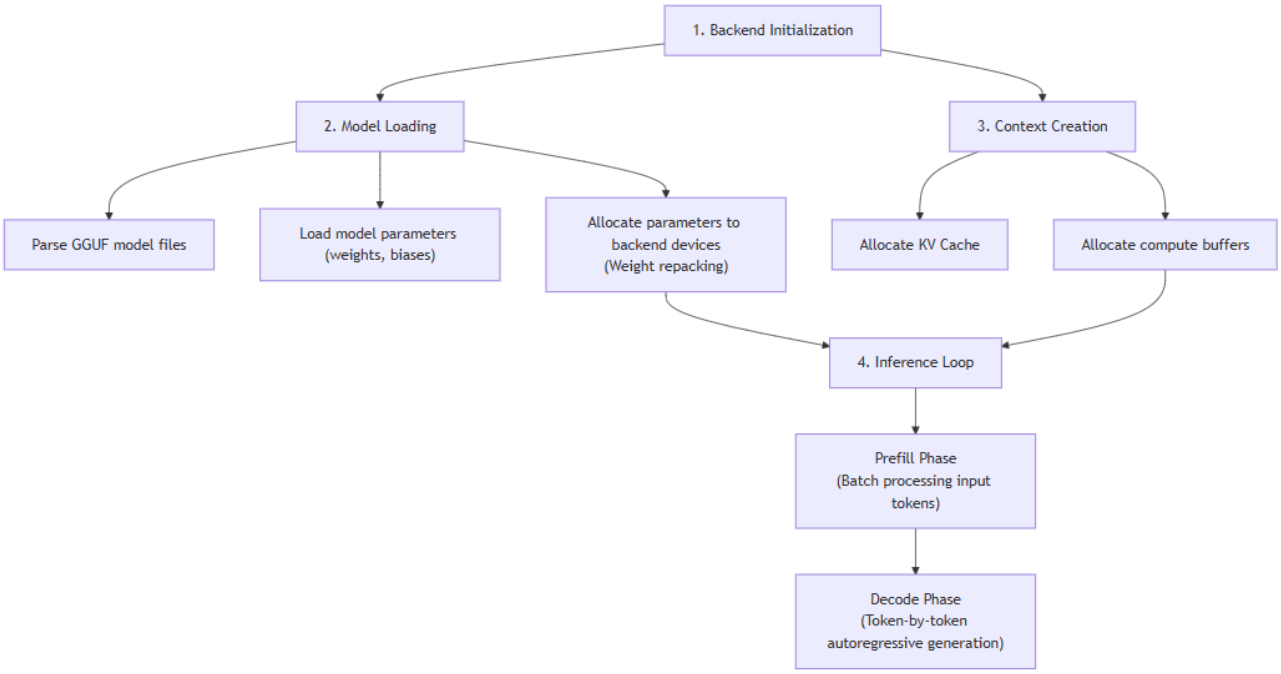


Figure 2: Overall inference execution flow of llama.cpp.

backends (such as the CPU backend, Hexagon backend, etc.). The framework then parses the GGUF-format model file, reading model hyperparameters (such as embedding dimension, number of attention heads, number of Transformer layers, etc.), vocabulary, and weight tensors. Based on the user-specified NPU offloading layer count (`-ngl` parameter), the framework assigns the weights of the corresponding layers to the target device's buffer, while retaining the remaining layers on the CPU side. The context creation step then allocates the KV cache and temporary buffers required for forward computation according to the specified context window size.

Inference calls. All inference calls are uniformly executed through the `llama_decode()` function. Regardless of whether it is the Prefill or Decode stage, each call goes through the following steps: (1) constructing the computation graph for the current batch based on the model architecture; (2) the backend scheduler partitions and assigns the computation graph to each backend according to operator support; (3) each backend sequentially executes its assigned sub-graph. The two stages differ in the scale of the input batch:

- **Prefill stage:** The input prompt is fed into `llama_decode()` in batches according to the `batch_size` parameter. Each batch processes multiple tokens, with the core computation being matrix-matrix multiplication (GEMM), making this a compute-intensive task. This stage completes the full forward pass over the entire input sequence and constructs the KV cache.
- **Decode stage:** Output tokens are generated autoregressively one by one. Each step calls `llama_decode()` to process only 1 token, with the core computation degenerating to matrix-vector multiplication (GEMV), making this a memory-bandwidth-bound task. Each compu-

tation step in the Decode stage depends on the KV cache accumulated from previous steps, imposing strict sequential dependencies.

2.3.3 Hexagon NPU Backend Execution Mechanism

The Hexagon backend of llama.cpp implements a complete operator scheduling and execution pipeline for the Qualcomm Hexagon NPU (HTP). The operation of this backend involves cooperative work between the CPU side (Host) and the NPU side (Skel). The core execution path is shown in Figure 3.

Weight repacking (REPACK). During the model loading phase, weight tensors assigned to the Hexagon backend are repacked into an NPU-optimized data layout. For example, Q4_0 quantized weights are converted to Q4x4x2 interleaved format to match the data access pattern of HVX vector instructions, thereby improving computational efficiency.

CPU–NPU communication mechanism. The CPU and NPU communicate through the `dspqueue` asynchronous communication framework. `dspqueue` is implemented based on the FastRPC protocol and shared memory, using a producer-consumer model. The CPU side encapsulates each operator as a request packet (containing the opcode, tensor pointers, and control flags) and submits it to the shared memory queue; the NPU side runs a resident callback function that reads requests from the queue and dispatches them for execution. Note that the "dsp" in the name `dspqueue` is a legacy naming convention from the Hexagon SDK; the actual communication target is the HTP processor.

Operator execution. All operators in the Hexagon backend are implemented using HVX Intrinsics (C-language inline functions for the Hexagon vector extensions), where each intrinsic function maps directly to a processor instruction. For core operators such as matrix multiplication, the implementation code directly operates on the 1024-bit vector registers of HVX (see Section 2.2.1), using instructions such as `Q6_Vw_vrmpy_VbVb` to complete multiple groups of quantized integer multiply-accumulate operations in a single cycle. During computation, the NPU uses on-chip VTCM (Vector Tightly Coupled Memory) to store temporary data, reducing memory access latency.

Graph optimization. Before executing the computation graph, the Hexagon backend performs two types of optimization on the operator sequence: (1) **Operator fusion**—merging adjacent lightweight operators (e.g., `ADD`+`MUL`, `RMS_NORM`+`MUL`) into a single composite operation to reduce the number of CPU–NPU communications; (2) **Operator reordering**—arranging `MUL_MAT` operators that share the same input tensor together, so that quantized activation values already loaded into VTCM can be reused, thereby avoiding redundant dynamic quantization operations.

Operator fallback mechanism. When the computation graph contains operators not supported by the Hexagon backend (e.g., `FLASH_ATTN_EXT` in the current version), the scheduler assigns these operators to the CPU backend for execution. Fallback causes the computation graph

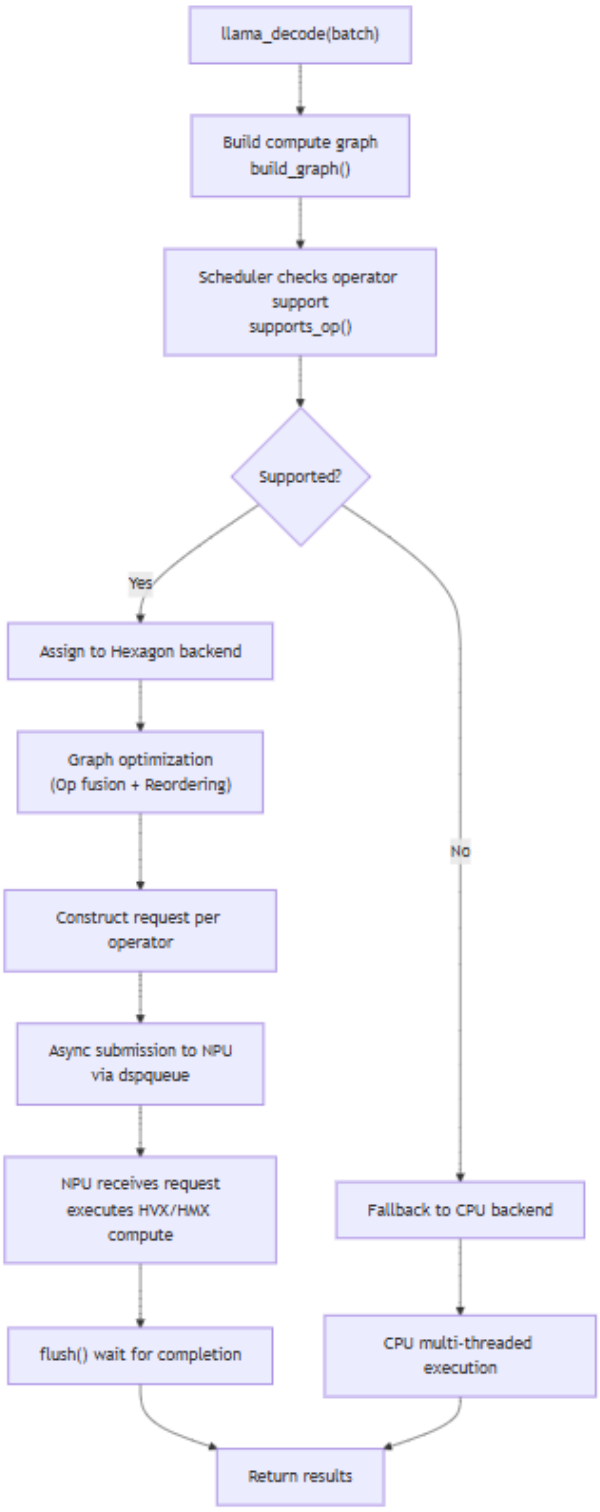


Figure 3: Hexagon NPU backend execution flow within `llama_decode()`.

to be split into multiple sub-graphs: NPU sub-graph → CPU sub-graph → NPU sub-graph. During sub-graph switching, the CPU must wait for the NPU side to complete all computations in the preceding sub-graph (synchronous wait), and when necessary perform cache coherency maintenance and data format conversion. The overhead of such cross-backend switching will be quantified and discussed in the experimental analysis of Section 5.

3 Experimental Platform and Methodology

This section presents the experimental platform and unified methodological framework used for the system-level performance analysis, including the hardware and software environment, variable design and control strategy, inference protocol and statistical methods, and micro-level profiling methodology. All performance evaluations are conducted under unified experimental conditions to ensure comparability and reproducibility.

3.1 Hardware and Software Environment

All experiments are conducted on the same hardware platform. The configuration is summarized in Table 1.

Table 1: Experimental Platform Configuration

Item	Configuration
CPU Model	Snapdragon 8 Gen 3
CPU Cores	2 + 1 + 5
NPU Model	Hexagon v75
Memory	16 GB
Operating System	Android 15
Inference Framework Version	tag: b7588

The Snapdragon 8 Gen 3 is equipped with the Hexagon v75 NPU, which belongs to Qualcomm's Hexagon Tensor Processor (HTP) architecture. HTP is an evolution of the traditional Hexagon DSP, retaining the HVX (Hexagon Vector eXtensions) vector extensions while adding the HMX (Hexagon Matrix eXtensions) matrix acceleration unit. Throughout this paper, "NPU" refers to the Hexagon HTP.

The inference engine used is llama.cpp (version listed in Table 1), a local inference engine supporting GGUF-format models. llama.cpp is compiled into an Android ARM64 binary via the cross-compilation toolchain provided by Qualcomm (including Android NDK and Hexagon SDK), with the Hexagon backend enabled (`GGML_HEXAGON=ON`). The build artifacts include the main program `llama-completion` and its dependent shared libraries (`libggml-hexagon.so`, `libggml-htp-v75.so`, etc.), which are pushed to the device via ADB (Android Debug Bridge) for execution. All models use Q4_0 quantization format to eliminate performance differences caused by varying precision levels. Model loading and execution procedures are kept consistent to avoid interference from different runtime interfaces or compilation options.

No changes are made to system power modes or scheduling policies during the experiments, and no additional acceleration options are enabled. The phone is not operated during testing, ensuring that all models and parameter configurations are evaluated under identical system conditions.

3.2 System-Level Variable Design and Control Strategy

To systematically analyze the impact of key parameters on inference performance, a single-variable control strategy is adopted. When scanning one variable, all other parameters are held fixed so that performance changes can be clearly attributed.

The following four categories of variables are analyzed:

Prompt Length. The input token count is varied across multiple scales to cover both short-sequence and long-sequence scenarios. The specific values are listed in Table 2. In the CPU threads and batch size experiments, the prompt length is fixed at approximately 1281/1282 tokens.

Table 2: Prompt Length Scanning Range

Model	Token Counts
Llama Series	{82, 162, 322, 642, 1282, 2562}
Qwen Series	{81, 161, 321, 641, 1281, 2561}

CPU Threads. The number of CPU threads is scanned over {1, 2, 3, 4, 6}, allowing analysis of how CPU parallelism affects throughput in both the Prefill and Decode stages.

Batch Size. Batch size is scanned over {1, 4, 8, 16, 32, 128, 256}. In specific experiments, it is fixed at 128 depending on the study objective.

NPU Offloading Layers, ngl. The parameter `ngl` denotes the number of Transformer layers offloaded to the NPU for execution. When `ngl = 0`, the model runs entirely on the CPU. The scanning ranges for different models are shown in Table 3.

Table 3: NPU Offloading Layer Range

Model	ngl Values
Llama-3.2-3B	{0, 14, 27, 29}
Llama-3.1-8B	{0, 15, 31, 33}
Qwen3-4B	{0, 11, 23, 35, 37}
Qwen3-8B	{0, 11, 23, 35, 37}

In single-variable experiments, varying `ngl` is used to analyze the effect of the heterogeneous computation ratio on system performance.

3.3 System-Level Inference Protocol and Statistical Method

The inference process is divided into two stages:

- **Prefill stage:** The full input prompt undergoes a complete forward pass and the KV cache is constructed.
- **Decode stage:** Output tokens are generated autoregressively in a token-by-token manner.

Throughput (tokens/s) is recorded separately for both stages, along with model load time (Load Time, ms).

The number of generated tokens is fixed at `n_predict = 256` for all experiments. To avoid bias in Decode throughput estimation from short generation sequences, the `--ignore-eos` flag is used to force the model to generate the complete sequence.

Each parameter configuration is executed 10–15 times. Final results are reported as the arithmetic mean over valid samples. For anomalous samples, the IQR (Interquartile Range) method is used to filter out records with significantly abnormal throughput values. All statistical results are computed based on filtered mean values.

Experiment execution is driven by automated scripts. The scripts read parameter combinations (including model path, ngl, Batch Size, thread count, etc.) from predefined CSV configuration files, and sequentially invoke the on-device `llama-completion` tool via ADB shell to perform inference. The complete log of each run (including framework output and operator-level profiling information) is saved as an individual file. The scripts include a checkpoint recovery mechanism that records completed task IDs, allowing resumption from the last interruption point to avoid redundant runs. An interval is set between experiments to reduce the impact of device overheating on results.

3.4 Micro-Level Profiling Methodology

The system-level analysis in Section 4 focuses on macro-level metrics such as end-to-end throughput and latency, while the micro-level analysis in Section 5 requires drilling down into the NPU pipeline internals and individual operator execution times. This section describes the methodology adopted to enable these two levels of micro-level profiling, including framework-level logging enhancements, the NPU pipeline stage separation mechanism, and operator-level performance metric extraction.

3.4.1 Framework Logging Enhancements

The Hexagon backend of llama.cpp already includes built-in operator-level profiling functionality that records timing data for each operator. However, the original implementation does not annotate which inference stage (Prefill or Decode) an operator belongs to. The CPU backend provides no operator-level performance logging at all. To enable cross-backend, stage-aware fine-grained performance analysis, the following modifications are made to the framework:

1. **Inference stage annotation.** A global computation stage state variable is introduced at the GGML core layer, with values `PREFILL` or `DECODE`. The upper-level inference logic sets this state before each computation graph execution based on the current input batch

size (batch processing indicates Prefill; single-token processing indicates Decode). Both the Hexagon and CPU backends read this state when outputting logs, thereby annotating each operator record with its corresponding inference stage.

2. **CPU backend operator-level timing.** Timing logic is added to the CPU backend's computation graph execution loop: a start timestamp is recorded before each operator executes, and an end timestamp is recorded after the multi-thread synchronization barrier. By collecting time after the barrier, the recorded duration reflects the actual time for all compute threads to complete the operator. Only the main thread records logs to avoid duplicate output from multiple threads. This feature is enabled via the environment variable `GGML_CPU_PROFILE=1` and incurs no overhead when disabled.

3. **Hexagon backend profiling configuration.** Operator-level profiling in the Hexagon backend is enabled via the `GGML_HEXAGON_PROFILE=1` environment variable, which records each operator's queue scheduling time (call-usec) and NPU-side actual computation time (op-usec). Additionally, since the Hexagon backend uses an asynchronous execution mode by default (operators are submitted without immediately waiting for completion), operator timing recorded under this mode may be inaccurate. To obtain reliable per-operator timing data, the profiling experiments also set `GGML_HEXAGON_OPSYNC=1`, which forces each operator to complete synchronously on the NPU side before returning, ensuring that op-usec and call-usec measurements accurately reflect the true latency of each individual operator.

These modifications do not affect the framework's computational logic or numerical results; they only produce log output when profiling is enabled.

3.4.2 NPU Pipeline Decomposition via OPMASK

To quantify the overhead proportion of each stage within the NPU execution pipeline, this study leverages the `GGML_HEXAGON_OPMASK` environment variable provided by the llama.cpp Hexagon backend to perform progressive masking tests on the pipeline.

A complete operator execution on the NPU involves three sub-stages, as summarized in Table 4.

Table 4: NPU Pipeline Sub-stages and OPMASK Bit Mapping

Stage	OPMASK Bit	Function
Queue (Comm.)	`0x1`	CPU constructs request, sends to NPU via FastRPC, awaits return
Quantize	`0x2`	NPU-side dynamic quantization of input activations (e.g., F32 → INT8)
Compute	`0x4`	NPU executes matrix operations using HVX/HMX units

By setting different OPMASK values, the above stages can be progressively enabled or disabled. The independent latency of each stage is then isolated using the differential method, as shown in Table 5.

Table 5: OPMASK Experimental Conditions

Condition	OPMASK	Enabled Stages	Observation Target
E0 (baseline)	`0x0`	None	CPU-side graph traversal and scheduling overhead
E1	`0x1`	Queue	CPU $\leftrightarrow$ NPU communication round-trip overhead
E2	`0x3`	Queue + Quantize	Comm. + dynamic quantization cumulative overhead
E3 (default)	`0x7`	All	Full NPU execution time

The independent latency of each stage is computed using the following differential formulas:

$$T_{\text{comm}} = E1 - E0 \tag{1}$$
$$T_{\text{quant}} = E2 - E1 \tag{2}$$
$$T_{\text{compute}} = E3 - E2 \tag{3}$$

Note that when OPMASK is not equal to `0x7`, the NPU computation is partially truncated and the returned data is invalid. Therefore, these experiments focus solely on timing metrics and do not examine the correctness of inference output. The experimental results of this method are discussed in detail in Section 5.1.

3.4.3 Operator-Level Performance Metric Extraction

When profiling is enabled, each inference run generates a log file containing per-operator timing records. Each log entry includes the following key fields:

- **Operator type** (e.g., `MUL_MAT`, `RMS_NORM`, `ROPE`, etc.)
- **Inference stage** (`PREFILL` or `DECODE`)
- **Execution backend** (NPU or CPU)
- **op-usec:** actual execution time of the operator on the compute unit (microseconds)
- **call-usec** (NPU operators only): total call time from CPU-side request initiation to NPU result return (microseconds), encompassing communication, quantization, and computation combined

The difference between call-usec and op-usec reflects the communication and scheduling overhead at the CPU–NPU boundary. For operators executed on the CPU backend (including fallback operators), only op-usec is recorded, since the CPU backend does not have an equivalent call-usec concept.

Log data is parsed and aggregated automatically via Python scripts. The scripts extract the above fields from logs of multiple repeated experiments, group statistics by model, inference stage,

operator type, and execution backend, and compute per-group average latency, invocation count, and total latency. The processed data is used for the CPU vs. NPU operator-level performance comparison analysis in Section 5.

3.5 Power Consumption Experimental Design and Measurement Methodology

In addition to system-level throughput and latency experiments, this thesis further conducts supplementary measurements of the end-device inference energy consumption of Llama-3.2-3B under different `ngl` configurations. The goal of this experiment is not to measure the absolute hardware power rail of the CPU or NPU, but to compare the relative energy consumption of different execution modes under the same device and running conditions.

The power consumption experiments are conducted on an iQOO 12 smartphone, equipped with the Snapdragon 8 Gen 3 SoC and Android 15. The evaluated model is the Q4_0 quantized version of Llama-3.2-3B. The inference program is `llama-completion`. Experiments execute inference commands on the device via wireless ADB. Since retail Android devices cannot directly access the underlying power rails of the CPU or NPU without root access, this thesis utilizes the battery statistics interface exposed by the Android system for energy estimation [1, 2].

This power consumption experiment serves as a supplementary study and is conducted exclusively on Llama-3.2-3B. The experiment fixes `batch-size=128`, `threads=6`, `ctx-size=4096`, `n_predict=256`, `NDEV=1`, and `Device=HTP0`. The input length is set to three levels: 256, 512, and 1024. For each input length, three offloading configurations are tested: `ngl=0`, `ngl=14`, and `ngl=99`. Each configuration group is repeated 8 times.

This experiment adopts a single-window sampling approach. Before each single experiment begins, the script executes `adb shell dumpsys batterystats --reset` to reset the current statistics window. Subsequently, a single `llama-completion` inference pass is executed on the smartphone. Following the inference, the script calls `adb shell dumpsys batterystats --write` and `adb shell dumpsys batterystats --usage`, while recording the battery status via `adb shell dumpsys battery` before and after the experiment [1]. All experiments are automated via scripts, which read a predefined parameter table, sequentially issue inference commands, and save individual logs along with group-level aggregated results.

This thesis uses two group-level metrics for analysis. The first metric is `GroupChargeCounterDelta`, which is calculated from the difference in the `Charge counter` before and after a group of experiments, reflecting the total power drain during the 8 iterations. The second metric is `SumSingleShellUidMah`, obtained by summing the estimated energy consumption corresponding to `UID 2000` (`adb shell`) across the 8 experiments within the group. The former is closer to the actual battery drain during the entire group of experiments. The latter is a software-estimated value from Android `batterystats`, but it demonstrates good stability in the current experiments and thus serves as an auxiliary metric.

The experiments also recorded `ComputedDrainMah`, which is derived from the `Computed drain` at the top of `batterystats --usage`. However, under a single-experiment window, this value fluctuates significantly and repeatedly produced 0 mAh or extremely low values. This phenomenon indicates its sensitivity to short time windows. Consequently, this thesis does not use `Computed-`

`DrainMah` as a primary analytical metric.

It should be noted that the experimental results reflect relative energy consumption differences visible at the Android system level, rather than the absolute hardware power of the CPU or NPU. This method is suitable for comparing the relative energy changes of different `ngl` configurations under identical experimental conditions, but cannot serve as a direct substitute for external power meters or chip-level power rail measurements.

4 System-Level Performance Analysis

This section presents the system-level performance evaluation results across all four models under a unified experimental framework. Three sets of controlled experiments are conducted: prompt length vs. ngl, CPU threads vs. ngl, and batch size vs. ngl. Prefill throughput and decode throughput serve as the primary evaluation metrics.

4.1 Impact of Prompt Length on System Performance

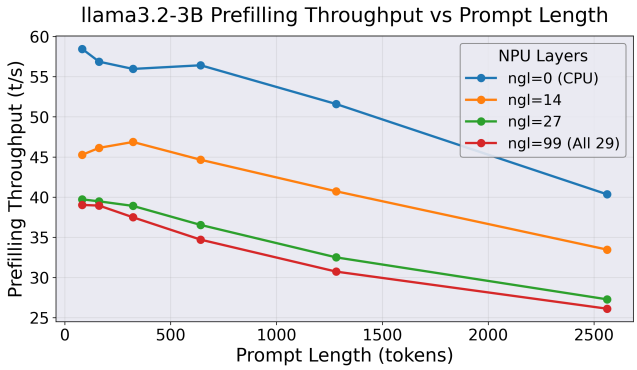
To evaluate the influence of input sequence length on heterogeneous inference performance, we measure the throughput of both the Prefill and Decode stages under six discrete prompt lengths. For each model, performance differences across different NPU offloading configurations (`ngl`) are compared. All four models follow the same testing protocol and statistical procedure, and all reported results are averaged over 15 repeated runs.

4.1.1 Prefill Stage

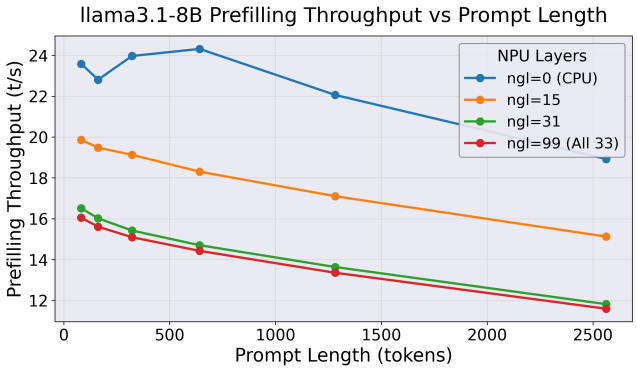
Figure 4 illustrates the Prefill throughput of the four models under different prompt lengths. A consistent trend is observed across all models: throughput decreases as the prompt length increases, and the decline gradually stabilizes once the sequence length reaches approximately 1280 tokens. This behavior reflects the nature of the Prefill stage, where the full input sequence is processed in a single forward pass and the KV cache is constructed. As sequence length grows, the scale of attention computation and intermediate tensor memory access increases proportionally, directly reducing the number of tokens processed per unit time.

When comparing different `ngl` configurations, a consistent ordering is observed. For all four models, `ngl = 0` (CPU-only execution) achieves the highest throughput. Any non-zero `ngl` configuration results in lower throughput, and throughput further decreases as `ngl` increases. This pattern holds for both the Llama and Qwen series. The result indicates that, on the current hardware platform, the dominant cost in the Prefill stage is not limited by per-layer matrix computation capability, but is strongly influenced by data scheduling and cross-device communication overhead. The additional overhead introduced by heterogeneous offloading outweighs the potential computational benefit provided by the NPU, preventing effective acceleration.

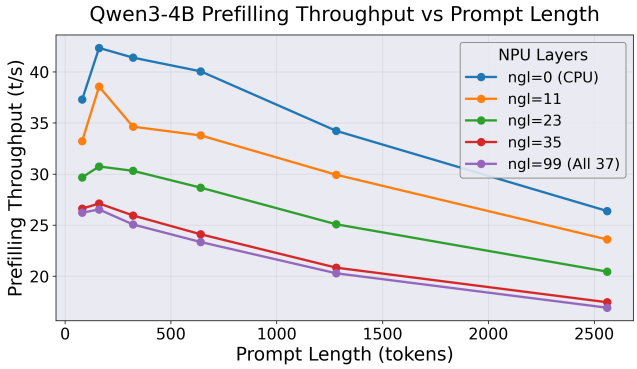
Model scale also affects Prefill behavior. The 3B and 4B models achieve higher throughput than the 8B models at the same prompt length. For 8B models, throughput degrades more noticeably under long-sequence conditions, indicating that larger parameter sizes increase per-layer computation and memory pressure, which amplifies performance degradation for long prompts.



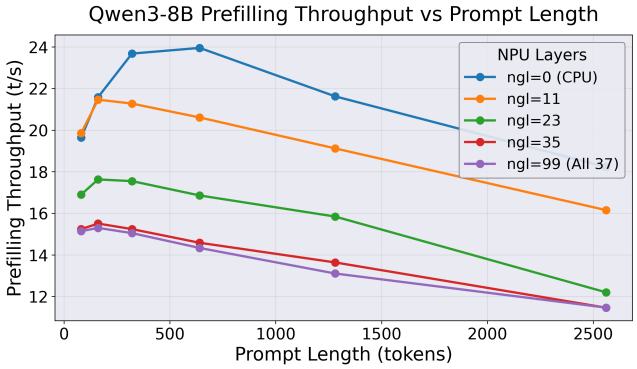
(a) Llama-3.2-3B



(b) Llama-3.1-8B



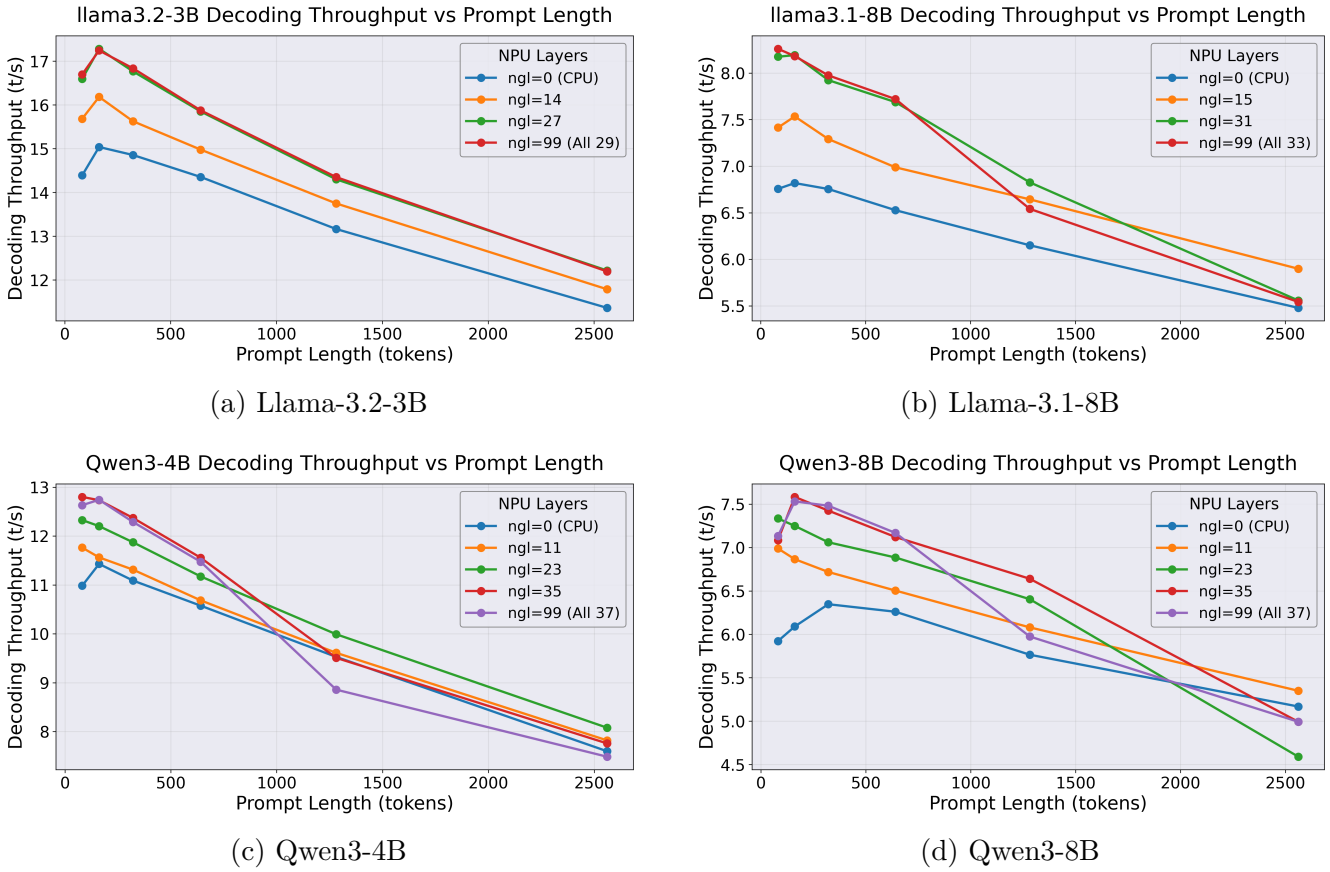
(c) Qwen3-4B



(d) Qwen3-8B

Figure 4: Prefill throughput (tokens/s) as a function of prompt length under different NPU layer configurations (ngl). Subfigures (a)–(d) correspond to Llama-3.2-3B, Llama-3.1-8B, Qwen3-4B, and Qwen3-8B, respectively.

4.1.2 Decode Stage



(a) Llama-3.2-3B

(b) Llama-3.1-8B

(c) Qwen3-4B

(d) Qwen3-8B

Figure 5: Decode throughput (tokens/s) as a function of prompt length under different NPU layer configurations (ngl). Subfigures (a)–(d) correspond to Llama-3.2-3B, Llama-3.1-8B, Qwen3-4B, and Qwen3-8B, respectively.

Figure 5 presents the Decode throughput of the four models across different prompt lengths. As the prompt length scales from roughly 80 tokens to 2560 tokens, Decode throughput exhibits a notable decline, with percentage drops ranging from 20% to 40% across different models and offloading configurations. Interestingly, contrary to the intuition that the Decode stage might be less sensitive to prompt length than the Prefill stage due to KV cache reuse, empirical data reveals that their performance degradation proportions are relatively comparable. For multiple setups (such as the Qwen3 series), the relative drop in Decode throughput even exceeds that of the Prefill stage. A primary mechanism contributing to this phenomenon is the fundamental shift in hardware bottlenecks between the two stages. Unlike the Prefill stage, which is predominantly compute-bound due to matrix-matrix computations, the Decode stage operates as a memory-bandwidth-bound task involving primarily matrix-vector computations. During autoregressive generation, the processor must fetch the entire model weights and the accumulated KV cache from off-chip main memory (DRAM) to on-chip caches (VTCM) for every newly generated token. While modern mobile SoCs provide substantial theoretical memory bandwidth (e.g., approximately 76 GB/s for the Snapdragon 8 Gen 3), the linearly growing KV cache steadily increases the absolute memory

read volume per token. This continuous increase in memory access overhead offsets the theoretical bandwidth advantage, resulting in a relative decline in Decode throughput that is comparable to, or in some cases exceeds, that of the Prefill stage.

The effect of NPU offloading in the Decode stage differs from that in the Prefill stage. For Llama-3.1-8B and Llama-3.2-3B, all non-zero `ngl` configurations improve throughput across the entire length range, and higher `ngl` generally corresponds to higher Decode throughput. This indicates that single-step matrix computation becomes the dominant workload in this stage, allowing the NPU to contribute effectively to acceleration.

For Qwen3-4B and Qwen3-8B, an optimal offloading range is observed. Intermediate `ngl` values provide relatively stable performance across multiple lengths, while the maximum `ngl` configuration shows diminishing or reversed advantages in the long-sequence region. This suggests that, in the Qwen models, Decode performance is constrained by both computation and data path overhead, and the offloading ratio must balance computational gains against communication costs.

Taken together, the impact of prompt length exhibits clear stage-specific characteristics. In the Prefill stage, CPU-only execution achieves the highest throughput on the current platform, and heterogeneous offloading does not provide benefits. In the Decode stage, NPU offloading can improve throughput for certain models, but the degree of improvement depends on model architecture and sequence length. These results indicate that system optimization for long-prompt scenarios should analyze the bottlenecks of the two stages separately rather than applying a uniform offloading strategy.

4.2 Impact of CPU Threads on System Performance

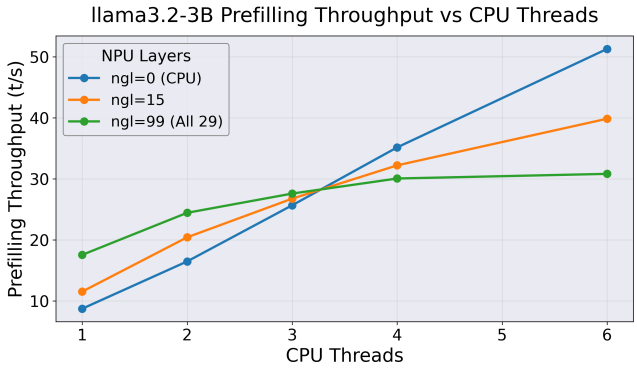
To analyze the effect of CPU parallelism on system throughput, we fix the prompt length at approximately 1281/1282 tokens and scan the number of CPU threads over {1, 2, 3, 4, 6}. Throughput is measured separately for the Prefill and Decode stages under different `ngl` configurations. All reported results are averaged over ten repeated runs.

4.2.1 Prefill Stage

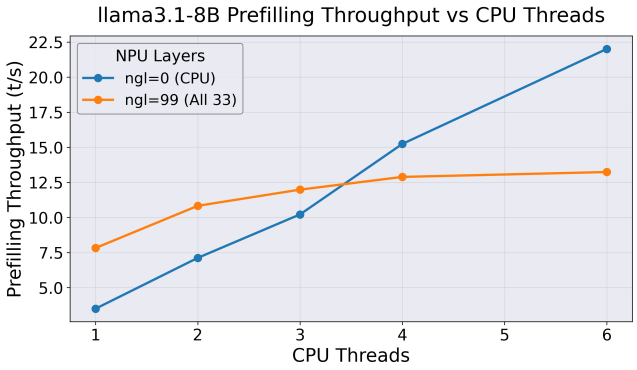
Figure 6 shows the Prefill throughput of the four models under different thread counts. Under `ngl = 0`, all models exhibit near-linear thread scalability. As the number of threads increases from 1 to 6, throughput consistently improves, with the most significant gains observed between 1 and 4 threads. For example, on Llama-3.2-3B, Prefill throughput increases from approximately 9 tokens/s in the single-thread setting to over 50 tokens/s with 6 threads, demonstrating strong scalability. Similar trends are observed for Qwen3-4B and Qwen3-8B.

As `ngl` increases, the scalability pattern changes. For Llama-3.2-3B and the Qwen models, Prefill throughput still increases with thread count under higher `ngl` settings, but the improvement becomes significantly weaker and tends to plateau beyond 4 threads. For instance, under `ngl = 29` in Llama-3.2-3B, increasing threads from 4 to 6 brings little additional gain, indicating that CPU computation is no longer the dominant bottleneck.

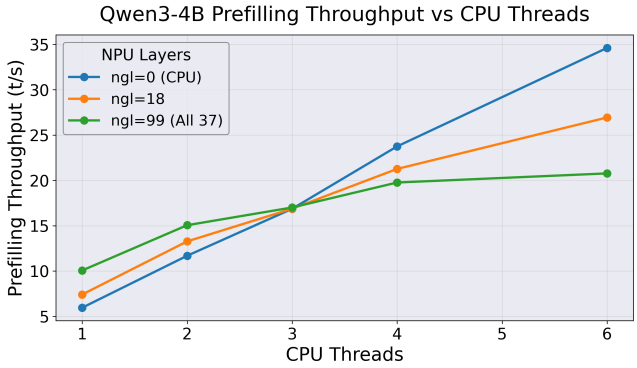
A similar pattern appears in Llama-3.1-8B. Thread scalability is strongest when `ngl = 0`, while under higher offloading (e.g., `ngl = 33`), the slope of throughput growth with respect to thread



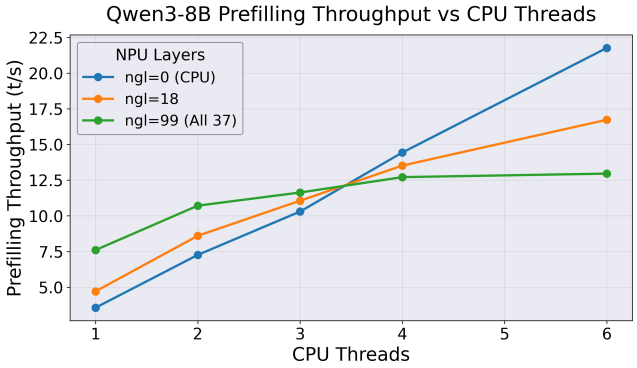
(a) Llama-3.2-3B



(b) Llama-3.1-8B



(c) Qwen3-4B



(d) Qwen3-8B

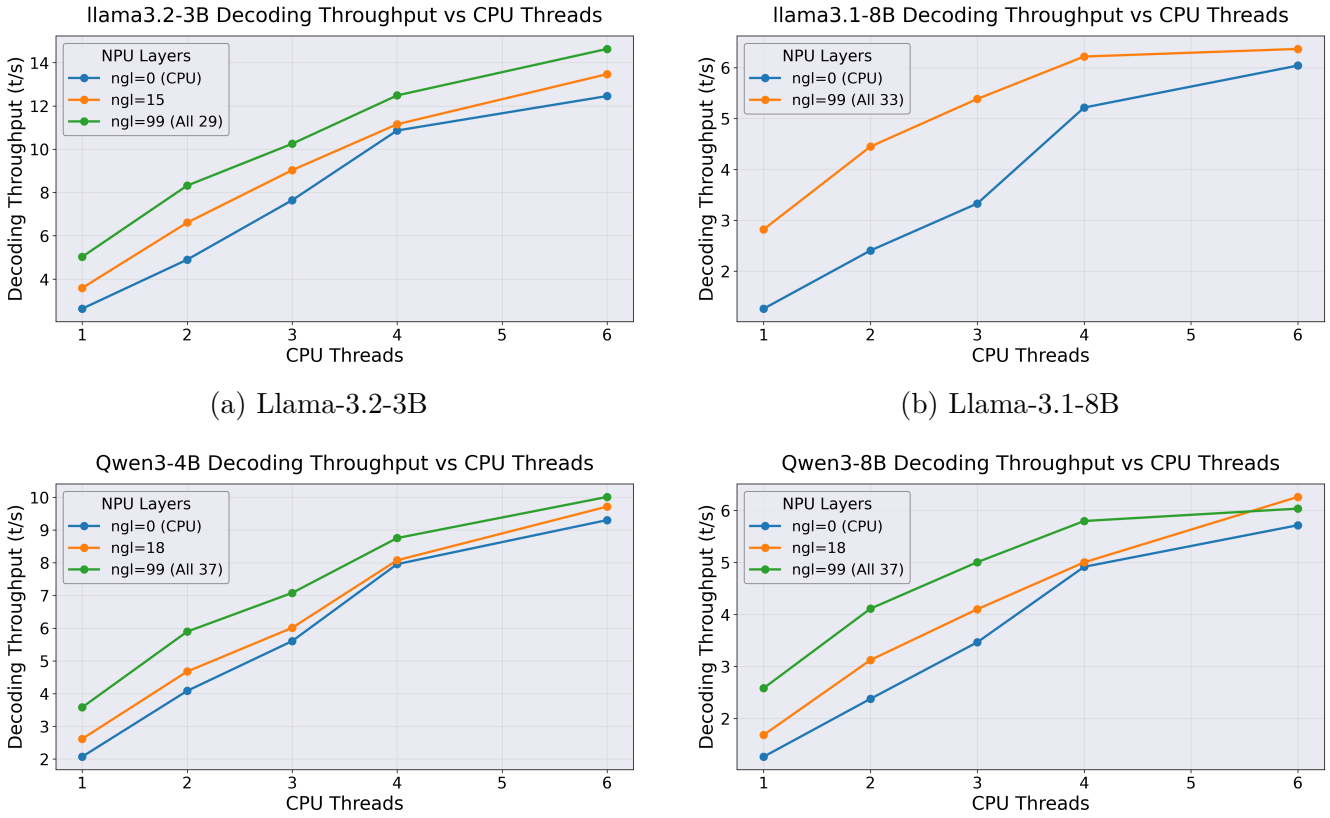
Figure 6: Prefill throughput (tokens/s) as a function of CPU thread count under different NPU layer configurations (ngl). Subfigures (a)–(d) correspond to Llama-3.2-3B, Llama-3.1-8B, Qwen3-4B, and Qwen3-8B, respectively. Results are averaged over 10 runs per configuration.

count decreases noticeably. Qwen3-8B shows the same behavior: once a large portion of layers is offloaded to the NPU, the influence of CPU threads on overall Prefill performance is substantially reduced.

These observations indicate that the Prefill stage achieves high parallel utilization under CPU-only execution. As computation is progressively offloaded to the NPU, the marginal contribution of additional CPU threads decreases, and the system bottleneck shifts toward heterogeneous scheduling and data transfer overhead.

4.2.2 Decode Stage

Figure 7 presents Decode throughput under different thread counts. Compared with the Prefill stage, Decode exhibits weaker scalability, although throughput still increases monotonically with thread count.



(a) Llama-3.2-3B

(b) Llama-3.1-8B

(c) Qwen3-4B

(d) Qwen3-8B

Figure 7: Decode throughput (tokens/s) as a function of CPU thread count under different NPU layer configurations (ngl). Subfigures (a)–(d) correspond to Llama-3.2-3B, Llama-3.1-8B, Qwen3-4B, and Qwen3-8B, respectively. Results are averaged over 10 runs per configuration.

Under `ngl = 0`, all models benefit from additional threads. For example, in Llama-3.2-3B, Decode throughput increases from approximately 2.5 tokens/s with one thread to over 12 tokens/s with six threads. The growth is clear but smaller than in the Prefill stage. Similar behavior is observed in Qwen3-4B and Qwen3-8B.

When `ngl` increases, Decode shows a different pattern from Prefill. For the Llama models, higher `ngl` configurations consistently achieve higher throughput across all thread counts, and increasing threads continues to provide gains. This indicates that Decode in these models can benefit from both CPU parallelism and NPU offloading simultaneously. In contrast, for Qwen3-8B, although larger `ngl` improves overall throughput, the thread scalability curve becomes flatter beyond 4 threads, suggesting that part of the computational load is already dominated by the NPU.

Overall, while Decode remains positively responsive to thread increases, its scalability is consistently lower than that of Prefill. This aligns with its computation pattern: Decode operates in a token-by-token autoregressive manner with smaller per-step workloads, limiting the amount of parallel work that can be distributed across threads.

CPU thread scalability exhibits clear stage-specific differences. Under `ngl = 0`, the Prefill stage achieves high parallel utilization, with throughput increasing approximately linearly as the number of threads grows. Although the Decode stage also benefits from multithreading, its scalability remains consistently lower than that of Prefill.

As `ngl` increases, the influence of CPU threads on overall performance gradually weakens, particularly in the Prefill stage. This indicates that heterogeneous offloading reshapes the system bottleneck distribution, making thread count no longer the primary limiting factor.

Model size further modulates this behavior. Larger models obtain more substantial gains from multithreading under CPU-only execution, while their benefits converge more rapidly under high `ngl` configurations. These results demonstrate a clear interaction between thread configuration and offloading strategy: the effectiveness of thread optimization depends on how computational workload is distributed between the CPU and the NPU.

4.3 Impact of Batch Size on System Performance

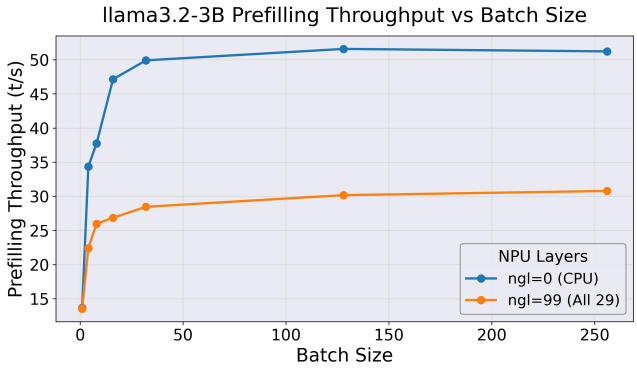
To evaluate the effect of batch size on inference performance, we fix the prompt length at approximately 1281/1282 tokens and scan Batch Size across multiple values. Throughput is measured separately for the Prefill and Decode stages under different `ngl` configurations. All reported results are averaged over ten repeated runs.

4.3.1 Prefill Stage

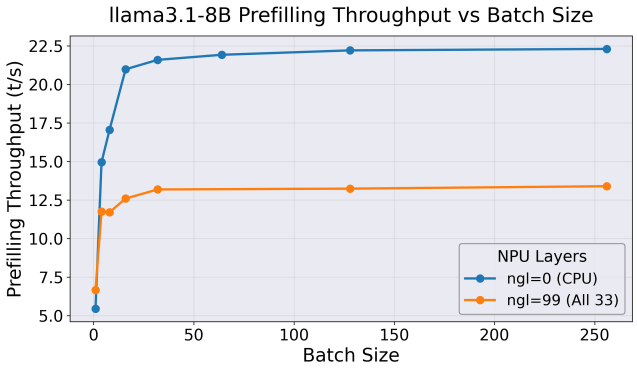
As shown in Figure 8, the Prefill stage exhibits a consistent scaling trend across all models. Under CPU-only execution (`ngl = 0`), throughput increases significantly as Batch Size grows from 1 to 8 or 16. For example, Llama-3.2-3B and Qwen3-4B show rapid throughput improvement in the small-batch range, and similar behavior is observed for Qwen3-8B and Llama-3.1-8B.

When Batch Size further increases to 32 and beyond, throughput growth slows down and gradually enters a plateau region for most models. Minor fluctuations appear in some configurations at large batch sizes, indicating that the system approaches computational or memory resource limits in the medium-batch range.

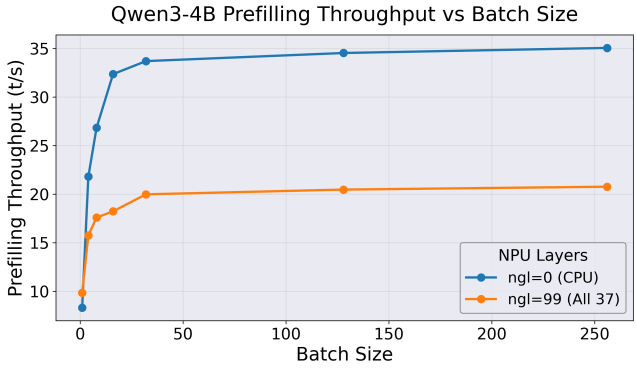
This behavior is closely related to the computation structure of the Prefill stage. Prefill performs a full forward pass over the input sequence, dominated by large-scale matrix multiplications



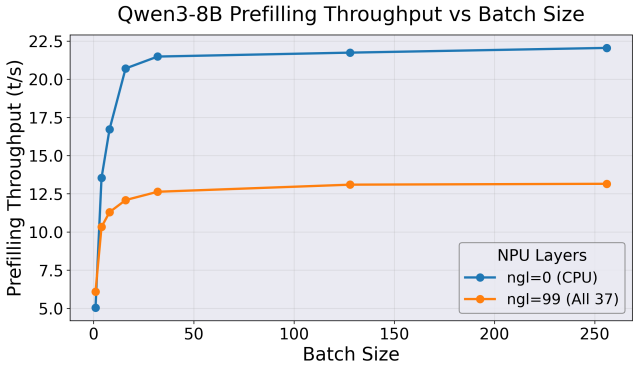
(a) Llama-3.2-3B



(b) Llama-3.1-8B



(c) Qwen3-4B



(d) Qwen3-8B

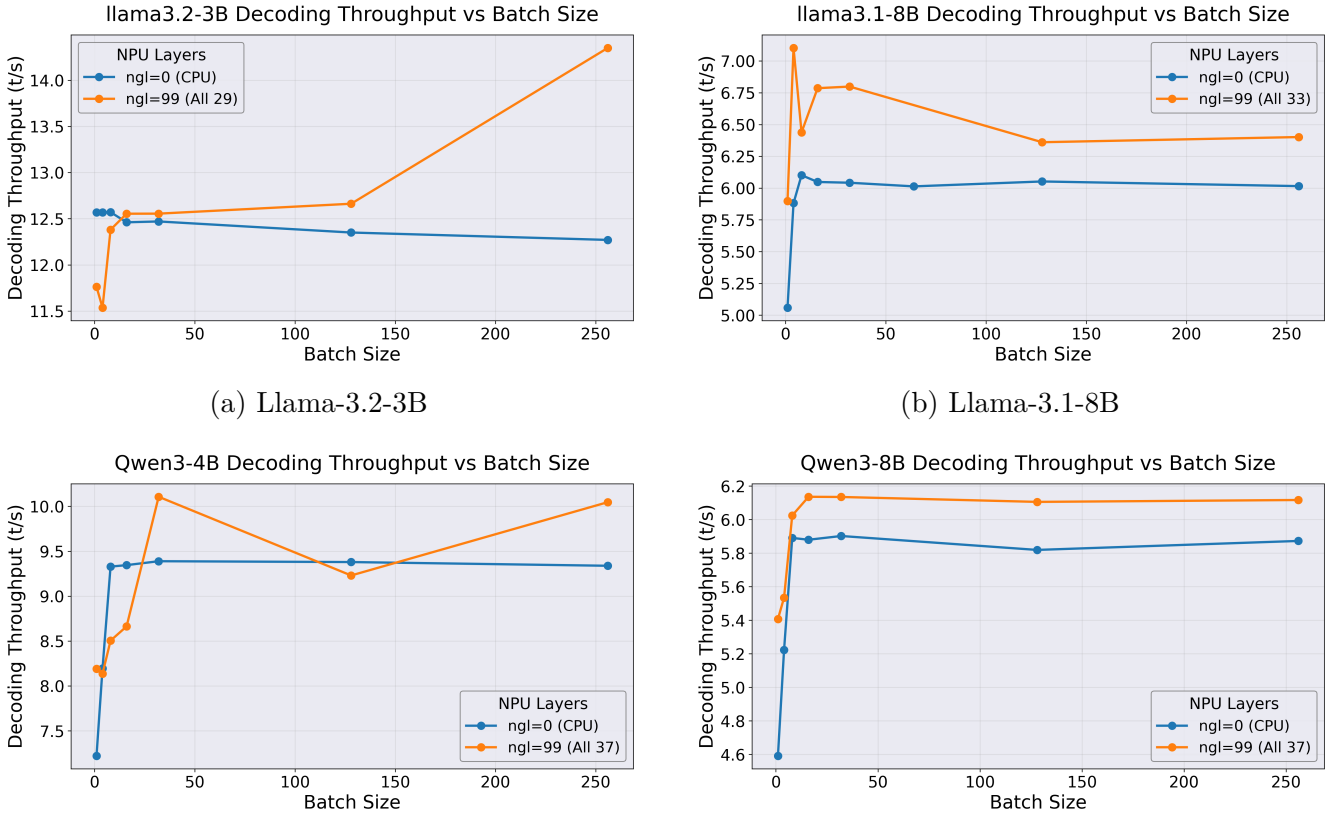
Figure 8: Prefill throughput (tokens/s) as a function of batch size under different NPU layer configurations (ngl). Subfigures (a)–(d) correspond to Llama-3.2-3B, Llama-3.1-8B, Qwen3-4B, and Qwen3-8B, respectively.

and attention operations. Increasing Batch Size effectively enlarges matrix dimensions, improving hardware utilization and yielding significant gains in the small-batch range. However, as batch size continues to grow, memory bandwidth pressure and KV cache usage increase rapidly. Cache contention and data access latency become the dominant constraints, leading to diminishing returns.

Under higher `ngl` configurations, the benefit of batch scaling becomes weaker. As more layers are offloaded to the NPU, throughput stabilizes earlier in the medium-batch range, as illustrated by the high-`ngl` curves in Figure 8. This indicates that once part of the workload is migrated to the NPU, the bottleneck shifts toward heterogeneous scheduling and data path efficiency.

4.3.2 Decode Stage

The Decode stage is generally less responsive to Batch Size than the Prefill stage, as shown in Figure 9. Under CPU-only execution, throughput increases noticeably when Batch Size grows from 1 to 8 or 16, but typically reaches a plateau at larger batch sizes.



(a) Llama-3.2-3B

(b) Llama-3.1-8B

(c) Qwen3-4B

(d) Qwen3-8B

Figure 9: Decode throughput (tokens/s) as a function of batch size under different NPU layer configurations (ngl). Subfigures (a)–(d) correspond to Llama-3.2-3B, Llama-3.1-8B, Qwen3-4B, and Qwen3-8B, respectively.

For Qwen3-4B and Qwen3-8B, Decode throughput reaches a peak in the medium-batch range and then changes only marginally. For Llama-3.1-8B and Llama-3.2-3B, throughput growth is more

gradual and does not exhibit the same magnitude of improvement as in Prefill. Minor fluctuations appear in some high-batch configurations, suggesting that Decode is more sensitive to scheduling and cache overhead.

This pattern arises from the autoregressive structure of Decode. Although batch processing allows parallel generation of multiple sequences, each step remains constrained by sequence-level dependencies, resulting in a smaller parallel granularity compared to Prefill. Consequently, the effective batch scaling range is narrower and reaches saturation earlier.

Under high `ngl` configurations, batch scalability in Decode further weakens. Most models exhibit stable throughput once Batch Size reaches a moderate level, indicating that when part of the computation is handled by the NPU, simply increasing batch size does not sustain throughput growth.

Overall, Batch Size has a stronger impact on the Prefill stage than on the Decode stage. Under CPU-only execution, moderate batch sizes significantly improve throughput, while gains converge at larger batch values. As `ngl` increases, the performance benefit of batch scaling diminishes, indicating that the system bottleneck gradually shifts from pure computation to heterogeneous scheduling and data path efficiency. In practice, the effective scaling region is concentrated in the small-to-medium batch range, beyond which sustained performance gains are difficult to obtain.

4.3.3 Micro-Level Analysis: Impact of Batch Size on Operator Performance

To investigate the underlying causes of the stage-specific performance differences brought by increasing Batch Size, this section further extracts operator scheduling logs from CPU-only execution and analyzes the impact of Batch Size from the perspective of individual operators.

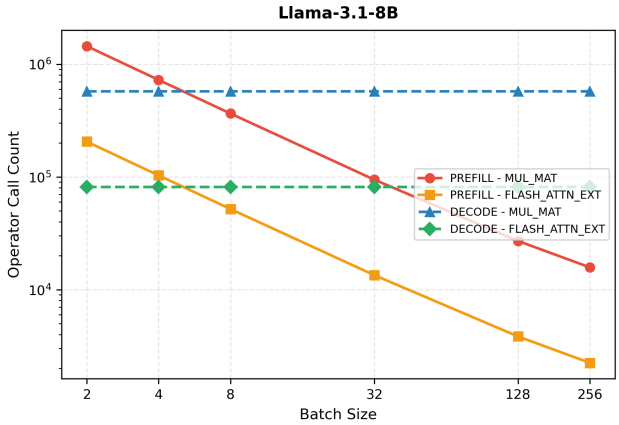
Operator-level analysis reveals that during the Prefill stage, as Batch Size increases, the invocation count of major operators (such as `MUL_MAT`, `GLU`, `MUL`, etc.) in the model's forward pass decreases significantly. This greatly amortizes the fixed overhead associated with each operator invocation, including framework scheduling and memory allocation costs. Although the merging of many small matrix operations into batched operations causes the per-invocation computation time (Op-usec) to increase, the overall reduction in scheduling overhead from the sharp decrease in invocation count far outweighs the increase in pure computation time. The combined effect leads to a substantial reduction in total node latency. According to experimental measurements, taking the increase from Batch Size 2 to 256 as an example, the average total Prefill latency across all four evaluated models decreases by 68.55% to 77.95%. This directly explains, from a micro-level mechanism, the rapid throughput increase observed in the macro experiments within the small-to-medium batch range.

In contrast, when the system enters the Decode stage, due to the strict sequential dependency of autoregressive generation, the model degenerates to token-by-token generation. In principle, regardless of the Batch Size configured during Prefill, the operator structure and invocation count in the Decode stage remain identical, and its throughput should be insensitive to this parameter. However, experimental results (e.g., Llama-3.1-8B and the Qwen3 series) reveal a counter-intuitive phenomenon: under very small Batch Size configurations (e.g., BS=2 or 4), the average total Decode latency shows noticeable degradation, approximately 30% higher than under BS=128 or 256 configurations.

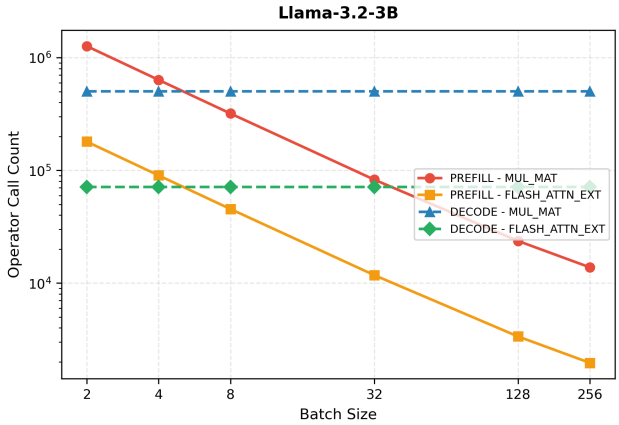
Analysis indicates that the "latency penalty" caused by small Batch Size is primarily driven by degradation in the single-invocation cost (Single Cost) of operators. Taking Llama-3.1-8B as an example, the average execution time of operators such as `FLASH_ATTN_EXT` at BS=2 is nearly twice that at BS=256. We hypothesize that the root cause of this anomaly lies in **KV Cache memory fragmentation**. Since `--batch-size` controls the data ingestion step size during the Prefill stage, processing approximately 1280 tokens of prompt requires fewer than 6 large-block KV Cache insertions when BS=256, whereas BS=2 triggers over 600 fine-grained consecutive allocations. Such high-frequency fragmented allocation is likely to break the spatial locality of memory. Upon entering the Decode stage, the autoregressive attention operation must traverse this large and potentially fragmented cache layer by layer, which can trigger frequent CPU cache misses. Longer input texts further amplify this locality penalty, causing a batch processing strategy originally associated only with Prefill to leave a performance "aftereffect" in the Decode stage. This suggests that setting Batch Size appropriately not only amortizes scheduling overhead but also helps maintain a well-structured underlying memory layout.

Model	Phase	BS=2 (ms)	BS=4 (ms)	BS=8 (ms)	BS=32 (ms)	BS=256 (ms)	Reduction (%)
Llama-3.1-8B	PREFILL	263181	129433	85859.5	59111.8	58023.3	78
Llama-3.1-8B	DECODE	62345.6	64482.8	50040.2	43005.6	42892.9	31.2
Llama-3.2-3B	PREFILL	79641.3	38728.5	33381.8	27476.9	25048.8	68.5
Llama-3.2-3B	DECODE	22703.7	21576.8	21551.6	21662	21510.6	5.3
Qwen3-4B	PREFILL	129895	52163.7	46925.7	41172.1	37307.2	71.3
Qwen3-4B	DECODE	41513	28608.6	28718.4	28594.2	28811.3	30.6
Qwen3-8B	PREFILL	265448	110064	84302.9	60030.9	59114.9	77.7
Qwen3-8B	DECODE	65423.6	52512.2	45472.5	45064.2	45444.3	30.5

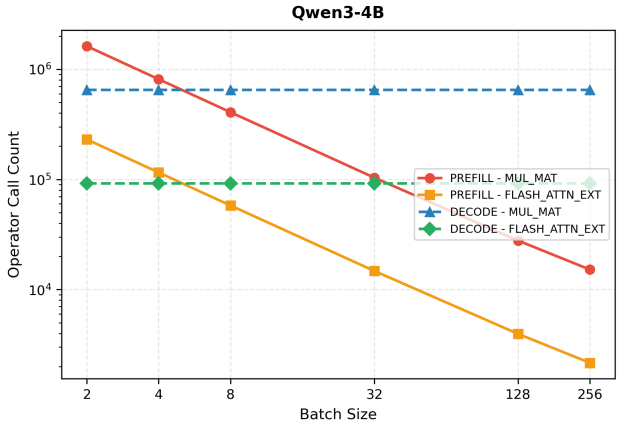
Figure 10: Impact of Batch Size on operator-level latency under CPU-only execution. The table shows total Prefill and Decode latency changes as Batch Size increases from 2 to 256.



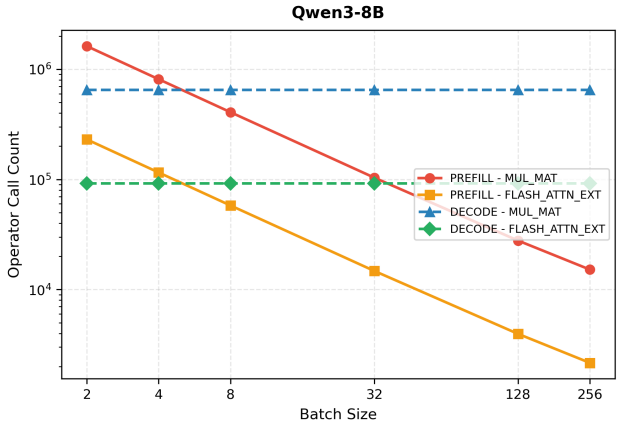
(a) Llama-3.1-8B



(b) Llama-3.2-3B



(c) Qwen3-4B



(d) Qwen3-8B

Figure 11: Operator invocation count comparison during the Prefill stage under different Batch Size configurations. Subfigures (a)–(d) show how major operator call counts decrease as Batch Size increases from 2 to 256.

4.4 Power Consumption Experimental Results and Phenomenon Analysis

To supplement the system-level performance results, this thesis further compares the group-level energy consumption of Llama-3.2-3B under different `ngl` configurations. Experimental results indicate that under current testing conditions, higher `ngl` settings do not exhibit lower group-level energy consumption compared to `ngl=0`. In fact, under most conditions, energy consumption is higher for `ngl=14` and `ngl=99`.

Table 6 presents the group-level results of the power experiments. For clarity, `GroupChargeCounterDelta` is presented in mAh. As shown in the table, with the increase of prompt length, the group-level total energy consumption across all three `ngl` configurations generally rises. Furthermore, under most length conditions, higher `ngl` configurations correspond to higher `GroupChargeCounterDelta` and `SumSingleShellUidMah`.

Table 6: Group-level energy consumption of Llama-3.2-3B under different prompt lengths and offloading configurations.

Prompt Length	ngl	GroupChargeCounterDelta (mAh)	SumSingleShellUid (mAh)	Group Elapsed Time (s)
256	0	64.000	45.90	320.310
256	14	75.000	53.72	342.518
256	99	78.000	45.55	362.555
512	0	78.000	43.09	458.935
512	14	94.000	49.43	425.294
512	99	103.000	62.29	448.837
1024	0	93.000	60.84	465.993
1024	14	133.000	77.07	571.692
1024	99	141.000	78.43	699.772

This phenomenon is quite evident from the group-level battery drain `GroupChargeCounterDelta`. Using `ngl=0` as the baseline, when prompt length is 256, the total power drain for the `ngl=14` and `ngl=99` groups increases by approximately 17.19% and 21.88%, respectively. When prompt length is 512, this increment grows to 20.51% and 32.05%. At a prompt length of 1024, the increase further expands to 43.01% and 51.61%. The overall trend of the auxiliary metric `SumSingleShellUidMah` remains largely consistent. Specifically at prompt lengths of 512 and 1024, higher `ngl` configurations consistently result in higher group-level estimated energy consumption.

This result suggests that, under the current platform and implementation, increasing the `ngl` parameter does not automatically translate to lower on-device total energy consumption. More accurately, the current experiment reflects not "whether the NPU is used", but rather "how the total system energy consumption changes when a subset of layers is offloaded to the NPU under the current backend implementation and operator coverage."

Combining the operator-level analysis and OPMASK experiments from Section 5 provides a consistent explanation for this phenomenon. First, the current Hexagon backend cannot cover the

complete LLM computation graph. Several critical operators still cannot be executed on the NPU and require falling back to the CPU. Taking `FLASH_ATTN_EXT` as an example, the current backend has not incorporated it into the Hexagon support set; therefore, under high `ngl` configurations, such operators are still executed by the CPU. At this point, the system does not enter a "pure NPU inference" state, but rather forms a mixed CPU–NPU execution path.

Second, mixed execution introduces additional overhead for graph partitioning and boundary synchronization. For unsupported operators, the scheduler must split the computation graph into NPU-routing and CPU-routing subgraphs, and perform synchronizations, buffer coherency maintenance, and data format conversions between them. This means the CPU not only needs to execute the fallback operators themselves, but also participates in backend switching and data scheduling. Operator-level experiments indicate that the execution time of some fallback operators under the "NPU-fallback-CPU" path is higher than under the pure CPU path, suggesting that the fallback penalty is not just "still computed by CPU," but also encompasses significant cross-backend execution overhead.

Third, the OPMASK experiments show that the NPU path inherently includes communication and quantization overheads. Experimental results demonstrate that while the primary bottleneck in the Prefill stage remains the NPU-side computation itself, the Decode stage suffers from much more significant communication costs. Although communication and quantization time may not dominate the total time, they constitute an extra fixed cost for high `ngl` configurations relative to the pure CPU path. When the computational benefits acquired by offloading specific layers are insufficient to offset these costs, both group execution time and group energy consumption can increase.

Finally, the energy consumption results are directly influenced by changes in total execution time. Total energy is determined by both average power and time. If higher `ngl` settings lead to increased group running time, higher group power consumption is expected. In our data, the total elapsed time of several high-`ngl` groups is higher than that of `ngl=0`, aligning with the trend of elevated group-level energy consumption.

Based on the existing evidence, this thesis posits a reasonable explanation: the current NPU backend's incomplete operator coverage leads to significant CPU fallback and cross-backend switching under high `ngl` configurations. Meanwhile, the CPU side is still burdened with fallback computation, scheduling, and communication tasks. Along with longer group execution times under high `ngl` settings across multiple experiments, the NPU execution path ultimately manifests as higher group-level energy consumption.

It is important to point out that the above explanation remains a mechanistic inference drawn from performance experiments and backend implementations, rather than a causal conclusion directly proven by chip-level power rail measurements. Current experiments cannot directly isolate the absolute power contributions of the CPU versus NPU on non-root retail devices, nor easily obtain accurate power profiles separately for the Prefill and Decode stages. Therefore, the explanation inside this thesis should be viewed as a logically sound hypothesis consistent with operator-level evidence, rather than direct observations of the underlying power distribution. To further validate this assessment, the following chapter will expand the analysis focusing on pipeline stage overheads and operator fallback behaviors.

5 Micro-Architectural and Operator-Level Analysis

Chapter 4 presented the system-level throughput and group-level energy consumption results. Both types of results collectively indicate that the current Hexagon execution path exhibits distinct differences across various stages and configurations. To explain these phenomena, this chapter begins with the heterogeneous execution mechanism to thoroughly decompose the micro-architectural execution of LLM inference on the Hexagon NPU. We first isolate the end-to-end latency into three core components—communication, quantization, and computation—using controlled masking tests. We then combine operator-level profiling logs to compare the per-invocation overhead of critical operators across different hardware backends, quantifying the extra performance penalty introduced by the fallback path.

5.1 NPU Pipeline Decomposition via OPMASK

In the Hexagon backend implementation of llama.cpp, the CPU uses the `dspqueue` mechanism (an API name inherited from the Hexagon SDK; the actual communication target is the NPU) to asynchronously dispatch operator computation primitives to the NPU. A complete NPU operator execution typically involves three key sub-stages: (1) control information and tensor communication between CPU and NPU (Hexagon HTP), (2) dynamic quantization of high-precision floating-point activations to low-level fixed-point formats, and (3) core matrix computation using the tensor acceleration units. Since these operations form a tightly coupled execution pipeline at the hardware level, conventional framework-level logs cannot accurately isolate the independent latency of each stage. To enable fine-grained decomposition, we leverage a low-level debug environment variable (`GGML_HEXAGON_OPMASK`) control mechanism that progressively enables internal NPU processing logic through parameter bit flags.

Through detailed analysis of the backend dispatch source code (i.e., the implementation of the graph dispatch function `ggml_hexagon_dispatch_op`), specific control flags can determine the execution depth when constructing low-level execution requests. Four quantitative masking conditions are carefully designed for the test:

1. **OPMASK=0x0** serves as the control baseline. By intercepting the request queue write instruction via a flag bit, the NPU is completely excluded from the pipeline, thereby isolating the pure CPU-side graph framework scheduling overhead and the execution time of non-offloaded operators.

2. **OPMASK=0x1** enables only the queue handshake dispatch command, but uses internal flags to force-skip subsequent quantization and multiply-accumulate computation. The difference between this condition and the baseline represents the full request round-trip communication overhead across the CPU–NPU heterogeneous boundary.

3. **OPMASK=0x3** additionally restores the dynamic tensor format conversion loop after dispatch. The reintroduction of this function precisely maps the cost of fixed-point and floating-point activation type conversion.

4. **OPMASK=0x7** enables the full system-level execution pipeline under normal scheduling. Comparison with the previous condition accurately derives the actual hardware computation time.

After preprocessing with median filtering to remove outlier deviations, this mechanism fully characterizes the workload distribution of each pipeline component and identifies potential communication bottlenecks.

The data shows that the core computation unit is the primary bottleneck of the entire pipeline. Across multiple test models including Llama-3.2 and Qwen series, in both the Prefill and Decode stages, pure core computation time consistently accounts for 85% to 99% of the total NPU execution time. This core bottleneck does not change with model architecture evolution. Additionally, the analysis shows that the overhead of dynamic quantization (converting high-precision floats to low-level fixed-point formats) is minimal. The on-the-fly quantization behavior based on low-level instructions accounts for only approximately 0.4% to 2.5% of total NPU runtime, which is nearly negligible. This confirms that the current framework's asynchronous dynamic quantization strategy for mobile deployment is highly efficient and stable, introducing no significant performance penalty.

Meanwhile, the pipeline decomposition directly reveals the large difference in CPU–NPU boundary communication overhead between the two inference stages. During the Prefill stage, the model processes a large volume of data in a single pass with extremely high computation density, so the communication latency between CPU and NPU is masked by the massive computation time, with communication overhead accounting for only about 0.2%. However, during the Decode stage, the LLM switches to a token-by-token iterative generation process. This computation pattern degrades from large-scale parallel matrix multiplication to high-frequency matrix-vector operations, causing the CPU to dispatch large numbers of fine-grained computation subgraphs to the NPU at high frequency. This high-frequency, fine-grained scheduling creates severe boundary "communication congestion," pushing the heterogeneous interconnection cost to approximately one-tenth of total latency (up to 9.9%–13.0%). In particular, for larger 8B models, the frequent short-graph dispatch causes data blocking at the boundary, significantly limiting the system's overall inference throughput ceiling.

Overall, the OPMASK-based analysis confirms that NPU backend execution in both Prefill and Decode stages is dominated by pure computation, but also exposes the significantly increased heterogeneous boundary communication penalty during the Decode stage. However, pipeline stage proportions alone cannot fully explain the core phenomenon observed in the macro experiments: why the NPU overall outperforms the CPU in the Decode stage (where communication overhead is high), while the CPU significantly outperforms the NPU in the Prefill stage (where communication overhead is low). This indicates that the fundamental cause of the performance reversal lies in differences in operator-level execution efficiency. The next section therefore drills down to the operator level, using fine-grained computational profiling to fully reveal the physical attribution of this reversal.

The OPMASK experimental results are presented below.

Model	Phase	Communication	Quantization	Computation	Total NPU
Llama-3.2-3B	Prefill	58.4	73.7	7636.9	7769.1
Llama-3.2-3B	Decode	1494.2	115.6	9903.3	11513.1
Qwen3-4B	Prefill	16.3	42.4	10380.5	10439.2
Qwen3-4B	Decode	1899.1	325.5	12689.1	14913.6
Llama-3.1-8B	Prefill	98.8	382.0	17780.1	18260.9
Llama-3.1-8B	Decode	2393.4	556.9	19777.9	22728.2
Qwen3-8B	Prefill	587.7	-310.8*	18087.4	18364.2
Qwen3-8B	Decode	2283.6	118.9	20577.2	22979.8

Figure 12: Estimated time breakdown for each pipeline stage across four models in Prefill and Decode stages. *Caused by measurement noise; this component accounts for only 1.7% of the Prefill stage.

Model	Phase	Communication %	Quantization %	Computation %
Llama-3.2-3B	Prefill	0.8%	0.9%	**98.3%**
Llama-3.2-3B	Decode	13.0%	1.0%	**86.0%**
Qwen3-4B	Prefill	0.2%	0.4%	**99.4%**
Qwen3-4B	Decode	12.7%	2.2%	**85.1%**
Llama-3.1-8B	Prefill	0.5%	2.1%	**97.4%**
Llama-3.1-8B	Decode	10.5%	2.5%	**87.0%**
Qwen3-8B	Prefill	3.2%	≈0%*	**98.5%**
Qwen3-8B	Decode	9.9%	0.5%	**89.5%**

Figure 13: Ratio of each pipeline stage (communication, quantization, computation) for four models in Prefill and Decode stages.

5.2 NPU vs. CPU Operator-Level Performance Analysis

The macro-level end-to-end tests and low-level pipeline decomposition together reveal a counter-intuitive phenomenon: in the compute-intensive Prefill stage, multi-core CPU inference speed significantly exceeds that of the Hexagon NPU, whereas in the Decode stage with relatively smaller computation volume, the NPU shows an acceleration advantage—though not at an ideal scale. This section investigates the underlying mechanisms driving this CPU–NPU performance reversal. Based on operator-granularity physical timing data, we analyze the limitations exposed by the NPU when handling the autoregressive generation framework, particularly its inherent shortcomings in computational characteristic matching, operator fusion, and heterogeneous scheduling.

5.2.1 Core Operator Computation Differences

In the **Prefill stage**, the model processes the entire long-sequence prompt in a single parallel pass. At this point, the core operator `MUL_MAT` receives large-scale matrix-matrix multiplication workloads. Experimental profiling data shows that facing such "low-frequency, large-volume" dense computation, the high-frequency CPU using 6 physical cores demonstrates stronger data throughput capability. In comparison, the NPU operator exhibits pipeline execution efficiency bottlenecks when processing equivalently large matrices. Across all four evaluated models (Llama-3 and Qwen3 series), the NPU's actual `MUL_MAT` execution overhead in the Prefill stage is 1.27× to 1.62× slower than the CPU. This across-the-board disadvantage in operator-level computation efficiency is the core reason why the NPU substantially trails the CPU in overall Prefill latency.

Beyond this, differences in operator implementation maturity are another important factor holding back NPU performance. The Hexagon backend of llama.cpp implements all operators from scratch using HVX Intrinsics (C-level inline functions that map directly to Hexagon vector processor instructions), whose optimization depth has not yet reached the level of the CPU-side implementations that have undergone extensive community iteration. Meanwhile, the on-chip tightly coupled vector memory (VTCM) of the Hexagon HTP has limited capacity; when processing the very large matrix multiplications characteristic of the Prefill stage, the efficiency of data tiling and transfer strategies directly affects pipeline utilization. Moreover, LLMs such as Llama are based on the Transformer block paradigm, whose 1D tensor computations and attention score calculation patterns differ significantly from the regular 2D convolution operations in traditional vision models. Some operators still have room for optimization on the NPU backend. For instance, rotary position encoding (`ROPE`) runs up to 6.5× slower than on the CPU. Other operators, such as the attention mechanism (`FLASH_ATTN_EXT`), have not yet been implemented in the Hexagon backend of llama.cpp and cannot be offloaded to the Hexagon NPU; during execution, they are forced to fall back from the NPU to the CPU. This fallback process not only fails to utilize NPU compute power but also incurs additional overhead under the unified memory architecture: the NPU typically requires data to reside in physically contiguous memory with specific alignment (e.g., FastRPC buffers), and fallback forces cross-unit cache coherency synchronization, tensor reordering, and extra memory copy (`CPY`) overhead, making its processing cost even 1.5× higher than running the operator natively on the CPU. This dual penalty mechanism further amplifies the NPU's disadvantage in the Prefill stage.

When the system enters the **Decode stage**, the inference paradigm shifts to token-by-token autoregressive generation (generating a single token per step). At this point, the core bottleneck

MUL_MAT undergoes a dimensional reduction, degenerating from matrix-matrix multiplication to matrix-vector multiplication. Operator-level profiling precisely reveals that the Hexagon NPU possesses a clear low-latency advantage for such small-scale matrix-vector operations: its MUL_MAT actual execution time is 1.5× to 1.7× faster than the CPU running at full speed (i.e., execution time is approximately 60% of CPU time). It is precisely because the NPU achieves a speed reversal on the dominant operator that it is able to outperform the CPU in overall Decode performance.

5.2.2 Lightweight Operator Scheduling and Cross-Backend Fallback Penalty

Despite the NPU's absolute computational advantage on the core multiply-accumulate operator during Decode, the total end-to-end speedup ratio remains only 1.05× to 1.20×, far below expectations. The physical bottleneck here confirms the OPMASK-based analysis from Section 5.1: **the high-frequency scheduling overhead of lightweight operators and the fallback synchronization penalty**. When facing the extremely fine-grained per-step operations of the Decode stage, for lightweight operators such as normalization (RMS_NORM) or addition (ADD), the NPU's pure computation time (Op-usec) may be only a few microseconds, while the graph dispatch and communication time (Call-usec) at the CPU–NPU boundary can be 8× to 22× higher than the actual computation time. The continuous dispatch of massive lightweight operators in small fragments wastes time that should have been used for acceleration on inter-chip bus waiting and communication congestion. Combined with the fallback penalty from the attention operator (FLASH_ATTN_EXT) and the output layer's MUL_MAT, this substantially offsets the NPU's 1.7× core acceleration advantage, preventing the mobile deployment acceleration from reaching its theoretical upper bound.

Table 7 presents the MUL_MAT operator-level timing comparison during the Prefill stage. The NPU is consistently slower than the CPU across all four models.

Table 7: MUL_MAT operator latency comparison — Prefill stage.

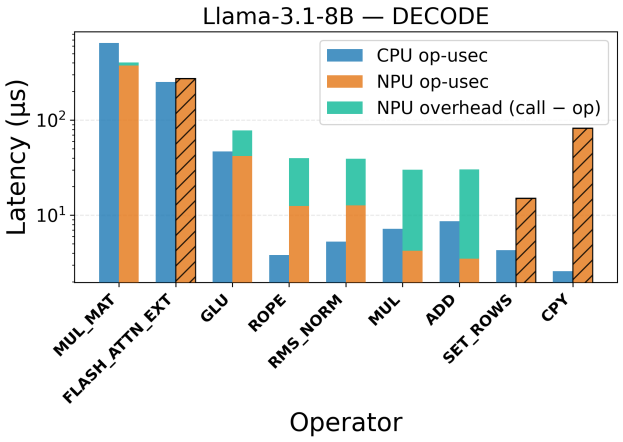
Model	CPU (μs)	NPU op (μs)	NPU call (μs)	NPU/CPU
Llama-3.2-3B	5,896	9,453	9,541	1.62×
Llama-3.1-8B	16,085	20,291	20,378	1.27×
Qwen3-4B	7,181	10,267	10,352	1.44×
Qwen3-8B	12,033	18,353	18,444	1.53×

Table 8 shows the corresponding comparison for the Decode stage, where the NPU demonstrates a clear advantage.

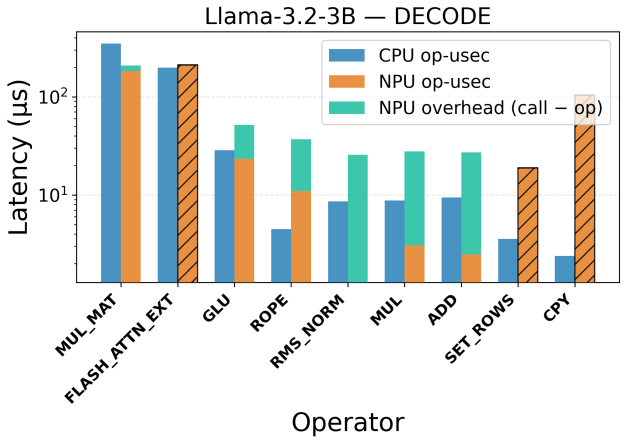
Figure 14 visualizes the per-operator latency breakdown during the Decode stage, highlighting the discrepancy between call-usec and op-usec for lightweight operators.

Table 8: `MUL_MAT` operator latency comparison — Decode stage.

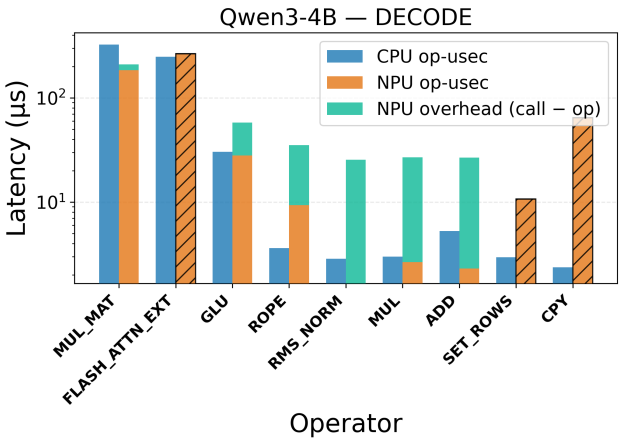
Model	CPU (μs)	NPU op (μs)	NPU call (μs)	CPU/NPU
Llama-3.2-3B	347.65	182.56	207.57	1.67×
Llama-3.1-8B	646.02	376.34	403.67	1.60×
Qwen3-4B	323.98	184.57	209.68	1.55×
Qwen3-8B	589.44	332.41	359.16	1.64×



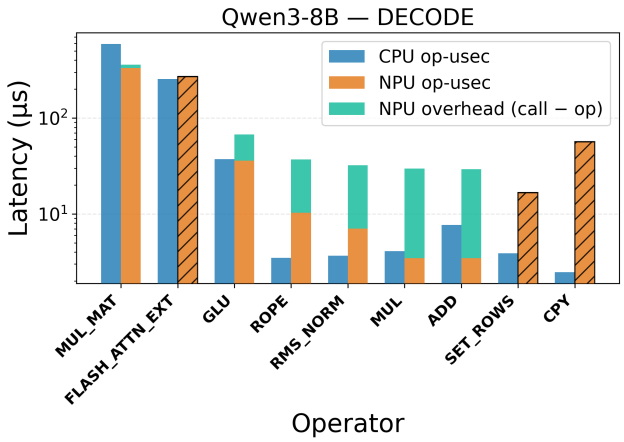
(a) Llama-3.1-8B



(b) Llama-3.2-3B



(c) Qwen3-4B



(d) Qwen3-8B

Figure 14: Decode stage per-operator latency comparison (op-usec vs. call-usec) for CPU and NPU backends. Subfigures (a)–(d) correspond to Llama-3.1-8B, Llama-3.2-3B, Qwen3-4B, and Qwen3-8B, respectively.

6 Discussion: Optimization Strategies and Deployment Trade-offs

Based on the system-level and micro-architectural analysis of CPU and Hexagon NPU behavior in mobile LLM inference presented in the preceding sections, this section proposes targeted optimization strategies for heterogeneous computing deployment. Experiments have demonstrated that relying entirely on a single compute unit or adopting a coarse-grained static offloading strategy cannot achieve optimal performance under the constrained conditions of mobile platforms. Therefore, this section focuses on how to leverage the complementary strengths of heterogeneous scheduling and deep operator-level fusion to break through the current compute bottlenecks on mobile platforms.

6.1 Analysis of Results and Comparison with Prior Work

The experimental results on the Snapdragon 8 Gen 3 platform validate the core finding of prior work [25]: on mobile devices, multi-core CPUs exhibit strong competitiveness in LLM inference and can even outperform heterogeneous acceleration units in certain scenarios. However, through operator-level profiling, this study further extends the boundary of this conclusion:

1. **Deep attribution of the performance reversal phenomenon:** While prior work revealed the potential of CPUs, this study pinpoints the logic behind the performance reversal where the CPU dominates in the Prefill stage and the NPU dominates in the Decode stage. Experimental data shows that the Prefill bottleneck lies not only in the additional overhead from operator fallback, but also in the fact that the Hexagon NPU's computation efficiency for large-scale matrix-matrix multiplication is still lower than that of high-frequency multi-core CPUs. This indicates that current mobile NPU compute architectures still lack adequate adaptation for the long-sequence parallel workloads specific to LLMs.

2. **Quantification of scheduling overhead:** Unlike prior work that focused only on macro-level throughput, this study uses Call-usec and Op-usec metrics to reveal the "scheduling tax" of the NPU during the Decode stage. Even though the NPU's matrix-vector multiplication is faster than the CPU, the acceleration benefit is substantially diluted by high-frequency fine-grained cross-boundary scheduling. This finding points the direction for operator optimization in future inference frameworks.

6.1.1 Memory-Architectural Roots of the Decode-Stage NPU Advantage

A question that warrants deeper examination is why the Hexagon NPU consistently outperforms the multi-core CPU on the dominant `MUL_MAT` operator during the Decode stage. Since decode-stage matrix-vector multiplication is widely recognized as memory-bandwidth bound [16], and both processing units share the same LPDDR5x memory subsystem with a common peak bandwidth of approximately 76.8 GB/s, one might expect both to be equally constrained by the DRAM bandwidth ceiling. However, operator-level measurements reveal that the NPU achieves 1.5× to 1.7× lower per-invocation latency across all four evaluated models (Table 8).

This advantage can be attributed primarily to fundamental differences in memory hierarchy design between the two processing units. The Cortex-X4 and A720 CPU cores rely on a traditional multi-level cache hierarchy (per-core L1/L2 caches and a 12 MB shared L3 cache) that is managed entirely by hardware. During the Decode stage, model weights exhibit a pure streaming access pattern: each weight element is read exactly once per generated token and is never reused. This pattern represents the worst case for a cache-based architecture, as weight data pollutes the L1/L2/L3 caches without benefiting from temporal locality; given that the model weights (e.g., Llama-3.2-3B Q4_0 $\approx$ 1.9 GB) far exceed the L3 capacity, most weight accesses are likely to miss in the shared cache and be serviced from DRAM. Furthermore, the SoC-wide cache coherency protocol introduces additional per-transaction overhead even when only six cores are active.

In contrast, the Hexagon NPU employs a software-managed scratchpad memory (Vector Tightly Coupled Memory, VTCM) combined with a dedicated DMA engine [4]. Rather than relying on speculative caching, the DMA engine performs asynchronous bulk transfers of weight tiles from DRAM into VTCM, employing double-buffering techniques to overlap data movement with computation. Because scratchpad placement is explicitly scheduled by the compiler and firmware rather than speculatively managed, the NPU likely avoids much of the cache-miss-related penalty seen in the CPU path and reduces cache coherency overhead. Moreover, the DMA engine issues large, contiguous burst transfers that are inherently more efficient for the DRAM memory controller than the cache-line-granularity requests (64 bytes) generated by the CPU's load/store units. This difference in request granularity, combined with the elimination of prefetcher inaccuracies, likely allows the NPU to sustain a higher effective fraction of the peak DRAM bandwidth in this workload. This interpretation is also consistent with the observed per-invocation latency gap, although the precise bandwidth partitioning policy of the SoC is not publicly documented.

This analysis also explains why the NPU's advantage disappears in the Prefill stage. During prefill, the input sequence length (batch-size = 128 tokens in our configuration) transforms `MUL_MAT` from a matrix-vector operation into a matrix-matrix operation, dramatically increasing arithmetic intensity. Each weight element loaded from DRAM is reused across all input tokens, meaning the CPU's cache hierarchy becomes an advantage rather than a liability: weights persist in L1/L2 cache and amortize the DRAM access cost over hundreds of multiply-accumulate operations. In this compute-bound regime, the aggregate throughput of six high-frequency Cortex cores exceeds the NPU's computation capacity, producing the observed CPU advantage of 1.27× to 1.62×.

6.2 Deployment Strategy: Split-Phase Heterogeneous Deployment

Based on the differentiated performance characteristics of the Prefill and Decode stages analyzed above, a "Split Deployment" strategy may be considered. This section discusses this strategy from three perspectives: applicability, implementation path, and benefit boundaries.

6.2.1 Performance Pain Points and Strategy Motivation

The detailed experiments in preceding sections expose two key pain points of on-device LLM inference:

1. **NPU compute shortfall and fallback penalty in Prefill:** When facing long-sequence input prefill computation, the core operator (matrix-matrix multiplication) operates at large

scale. Logs show that the Hexagon NPU's computation efficiency for such large matrices is lower than that of multi-core high-frequency CPUs, and it frequently triggers cross-boundary fallback to the CPU for unsupported operators (such as attention layers). The penalty from heterogeneous cache synchronization and tensor copying results in poor NPU performance in this stage.

2. **Scheduling tax and communication congestion in Decode:** Upon entering the autoregressive decoding stage, the computation pattern degenerates to fine-grained matrix-vector multiplications. Although the NPU's core operator physical latency is significantly faster than the CPU, the framework's high-frequency dispatch of large numbers of lightweight operators (such as `RMS_NORM`) causes frequent communication at the CPU–NPU boundary. Communication cost (up to 13%) substantially erodes the NPU's theoretical acceleration benefit.

Given these asymmetric performance characteristics, adopting a single fixed global offloading strategy (e.g., permanently offloading 30 Transformer layers to the NPU) is clearly not optimal.

6.2.2 Dynamic Pipeline Implementation Concept

The core idea of split-phase heterogeneous deployment is: **breaking the static hardware binding during the model execution lifecycle and performing a compute device hot swap at the boundary before and after the first token is generated.**

Phase 1: CPU-dominated Prefill. Upon receiving a long-text prompt as input, the system may temporarily bypass the fixed `--ngl` offloading configuration and prioritize binding the computation graph to multi-core CPU execution. This phase leverages the CPU's stability in large-scale contiguous memory access and large matrix computation to reduce the Time To First Token (TTFT) and generate the complete KV cache in main memory.

Phase 2: NPU-dominated Decode. After Prefill completes and token-by-token generation begins, the system can trigger an execution state migration at the phase boundary, allowing the Decode stage to be primarily handled by the NPU for matrix-vector computations. This phase requires combined operator fusion and in-graph reordering to reduce the high-frequency dispatch of lightweight operators and cross-boundary communication.

Implementing this "Split Deployment" requires the underlying inference engine (e.g., llama.cpp) to introduce a phase-aware routing mechanism in the execution graph, and where feasible, to implement low-copy KV access paths between the CPU and NPU sides. Given the current hardware and framework characteristics, this strategy appears feasible, but systematic experiments are still needed to confirm its stability and acceleration effect.

6.3 Future Research Directions

Based on existing experimental data, optimization of mobile LLM inference should not remain at the level of static parameter tuning, but should advance in two directions: dynamic cooperation and low-level infrastructure optimization.

6.3.1 Multi-Backend Cooperation and Energy-Aware Scheduling

Current inference frameworks mostly adopt a heterogeneous strategy involving the CPU and a single acceleration unit (e.g., NPU). Future research may explore incorporating the GPU into a unified scheduling domain, building a "CPU-GPU-NPU" tri-backend cooperation mechanism that leverages the GPU's bandwidth advantage for certain matrix operation scales (especially in long-sequence scenarios) to enable more flexible operator dispatch. Meanwhile, the ultimate goal of mobile deployment is to achieve a balance among performance, power consumption, and thermal metrics. Future schedulers should establish multi-dimensional cost models that sense device battery level and thermal throttling state in real time, dynamically adjusting the participation ratio of heterogeneous units to ensure stability during sustained inference.

6.3.2 Operator Ecosystem Completion and Deep Fusion Strategies

The upper bound of dynamic scheduling benefits is limited by the underlying operator support rate. If critical operators (such as the attention mechanism) continue to fall back to the CPU, the advantages of the heterogeneous system will be greatly diminished. Therefore, future work should focus on:

- **Improving operator coverage:** Prioritize native NPU support for LLM core operators such as `FLASH_ATTN`, eliminating cross-boundary synchronization and data reordering penalties.
- **Increasing fusion granularity:** Addressing the "scheduling tax" problem identified in Section 5.2.2, use compiler optimizations to fuse consecutive lightweight operators (such as `RMS_NORM` and `ADD`) into coarse-grained computation blocks, reducing the frequency of graph dispatch from CPU to NPU.

6.4 Limitations

Despite the fine-grained performance profiling provided by this study, the following limitations exist due to experimental constraints:

1. **Hardware platform timeliness:** This study is based on the Snapdragon 8 Gen 3 platform released in 2023. As of 2026, mobile chip architectures have undergone significant iterations (e.g., substantial increases in NPU matrix compute power and native hardware support for Transformer operators). Newer platforms (e.g., Snapdragon 8 Gen 5 or equivalent) may have already mitigated the matrix multiplication inefficiency and scheduling latency issues observed in this study.
2. **Simplified thermal management:** During experiments, reasonable inter-task intervals (approximately 2 seconds) are set to allow the device temperature to return to normal levels, thereby avoiding severe thermal throttling interference. However, under extreme workloads involving large-scale, sustained inference, the impact of temperature fluctuations on scheduling benefits still requires more rigorous monitoring data.

3. **Model architecture diversity:** Experiments primarily cover mainstream dense architecture models. As the deployment demand for Mixture-of-Experts (MoE) models on mobile devices increases, the impact of their sparse memory access patterns on heterogeneous backends warrants further exploration.

7 Conclusion

This thesis addresses the performance evaluation of large language model inference on mobile heterogeneous computing platforms. A reproducible benchmarking framework based on llama.cpp is constructed, supporting systematic evaluation across multiple models (Llama 3.1-8B, Llama 3.2-3B, Qwen3-4B, Qwen3-8B), multiple backends (CPU and Hexagon NPU), and multiple runtime parameter combinations. On this basis, a fine-grained profiling methodology based on operator-level profiling and OPMASK pipeline decomposition is proposed, decomposing the NPU inference pipeline into communication, quantization, and computation stages, enabling quantitative analysis of each stage's latency contribution.

The experimental results reveal the following core findings. First, the Prefill and Decode stages exhibit a significant performance reversal phenomenon between CPU and NPU: in the Prefill stage, multi-core CPUs process large-scale matrix-matrix multiplications more efficiently than the current NPU implementation (NPU is 1.27–1.62× slower), whereas in the Decode stage, the NPU demonstrates a 1.5–1.7× speed advantage in matrix-vector multiplication. Second, the OPMASK-based pipeline decomposition shows that core computation accounts for 85%–99% of total NPU execution time, dynamic quantization overhead is negligible (0.4%–2.5%), but the CPU–NPU communication overhead in the Decode stage surges to 9.9%–13.0%, becoming a major factor limiting NPU acceleration. Third, the high-frequency scheduling of lightweight operators produces a severe "scheduling tax" (call-usec reaches 8–22× of op-usec), which, combined with the cross-backend synchronization penalty from attention operator fallback, limits the NPU's end-to-end speedup in the Decode stage to only 1.05–1.20×, far below the theoretical acceleration upper bound of the core operator. Fourth, system-level experiments confirm the differentiated impact patterns of batch size, prompt length, and thread count on throughput, and reveal that small batch size configurations impose approximately 30% additional latency penalty on the Decode stage through KV cache fragmentation. Fifth, power consumption analysis shows that under the current heterogeneous execution path with incomplete operator coverage, increasing the number of NPU offload layers has not effectively reduced the total power consumption on the end side. On the contrary, due to the cross-end scheduling overhead of operator fallback, the underlying communication cost, and the increase in the overall group running time, the total power consumption at the group level has increased significantly (with the highest increase reaching 51%).

These findings provide clear optimization directions for mobile LLM deployment. Accordingly, this thesis proposes a split-phase heterogeneous deployment strategy (Prefill stage dominated by CPU, Decode stage dominated by NPU), and identifies that operator ecosystem completion (especially native NPU support for the attention operator) and operator fusion (reducing the high-frequency dispatch of lightweight operators) are the key paths to breaking through the current performance bottlenecks. Future research should further explore CPU–GPU–NPU multi-backend cooperative scheduling mechanisms and energy-aware dynamic resource allocation strategies, aim-

ing to achieve a better balance among performance, power consumption, and thermal constraints.

References

[1] Android Developers. Android `dumpsys` documentation. https://developer.android.com/tools/dumpsys, 2025.

[2] Android Open Source Project. Power profiles for android. https://source.android.com/docs/core/power, 2025.

[3] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. 2024. URL: https://arxiv.org/abs/2404.00456, arXiv:2404.00456.

[4] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. Characterizing mobile SoC for accelerating heterogeneous LLM inference. In *Proceedings of the 31st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2025. URL: https://dl.acm.org/doi/abs/10.1145/3731569.3764808, doi:10.1145/3731569.3764808.

[5] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[6] Gemma Team. Gemma: Open models based on Gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024. URL: https://arxiv.org/abs/2403.08295.

[7] ggml-org. llama.cpp. GitHub repository, 2024. URL: https://github.com/ggml-org/llama.cpp.

[8] Zixu Hao, Jianyu Wei, Tuowei Wang, Minxing Huang, Huiqiang Jiang, Shiqi Jiang, Ting Cao, and Ju Ren. Scaling LLM test-time compute with mobile NPU on smartphones. *arXiv preprint arXiv:2509.23324*, 2025. URL: https://arxiv.org/abs/2509.23324.

[9] Tanishq Kumar, Tri Dao, and Avner May. Speculative speculative decoding. *arXiv preprint arXiv:2603.03251*, 2026. URL: https://arxiv.org/abs/2603.03251.

[10] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023. URL: https://arxiv.org/abs/2309.06180.

[11] Haokun Lin, Haobo Xu, Yichen Wu, Jingzhi Cui, Yingtao Zhang, Linzhan Mou, Linqi Song, Zhenan Sun, and Ying Wei. Duquant: Distributing outliers via dual transformation makes stronger quantized llms. *arXiv preprint arXiv:2406.01721*, 2024.

[12] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. In *Proceedings of Machine Learning and Systems*, volume 6, pages 87–100, 2024.

[13] Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations. 2025. URL: https://arxiv.org/abs/2405.16406, arXiv:2405.16406.

[14] Meta AI. LLaMA 3.2: Vision and edge-optimized models for multimodal and mobile AI. Meta AI Blog, 2024. URL: https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/.

[15] MLC AI. MLC-LLM. GitHub repository, 2024. URL: https://github.com/mlc-ai/mlc-llm.

[16] James Pan and Guoliang Li. A survey of LLM inference systems. *arXiv preprint arXiv:2506.21901*, 2025. URL: https://arxiv.org/abs/2506.21901.

[17] PowerServe Project. PowerServe. GitHub repository, 2025. URL: https://github.com/powerserveproject/PowerServe.

[18] Vijay Janapa Reddi, David Kanter, Peter Mattson, Jared Duke, Thai Nguyen, Ramesh Chukka, Ken Shiring, Koan-Sin Tan, Mark Charlebois, William Chou, et al. MLPerf mobile inference benchmark. *arXiv preprint arXiv:2012.02328*, 2020. URL: https://arxiv.org/abs/2012.02328.

[19] Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. Empowering edge intelligence: A comprehensive survey on on-device AI models. *ACM Computing Surveys*, 2025. URL: https://arxiv.org/abs/2503.06027, doi:10.1145/3724420.

[20] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[21] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. llm.npu: Fast on-device LLM inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 445–462, 2025. URL: https://dl.acm.org/doi/abs/10.1145/3669940.3707239, doi:10.1145/3669940.3707239.

[22] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. PowerInfer-2: Fast large language model inference on a smartphone. *arXiv preprint arXiv:2406.06282*, 2024. URL: https://arxiv.org/abs/2406.06282.

[23] Tianyu Yu, Zefan Wang, Chongyi Wang, Fuwei Huang, Wenshuo Ma, Zhihui He, Tianchi Cai, Weize Chen, Yuxiang Huang, Yuanqian Zhao, et al. MiniCPM-V 4.5: Cooking efficient MLLMs via architecture, data, and training recipe. *arXiv preprint arXiv:2509.18154*, 2025. URL: https://arxiv.org/abs/2509.18154.

[24] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, et al. Flightllm: Efficient large language model inference with a complete mapping flow on fpgas. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 223–234, 2024.

[25] Haolin Zhang and Jeff Huang. Challenging GPU dominance: When CPUs outperform for on-device LLM inference. *arXiv preprint arXiv:2505.06461*, 2025. URL: https://arxiv.org/abs/2505.06461.