*electronics*

MDPI

*Article*

# Prompt-Shape- and Prefix-Cache-Aware CPU/NPU Routing for Large Language Model Inference on Edge SoCs

**Qiuchang Han, Yitian Mao, Qinyu Guo, Kouan Hao and Liang Wang ***

National Key Laboratory of Space Security, Shanghai Xiaoyuan Innovation Center, Minhang District, Shanghai 201100, China; 20113020006@fudan.edu.cn (Q.H.); maoyt1124@163.com (Y.M.); 18729573275@163.com (Q.G.); haokouan@126.com (K.H.)
* Correspondence: lllwwwwang@gmail.com

**Abstract**

Large language model (LLM) inference on edge systems must exploit heterogeneous central processing unit (CPU) and neural processing unit (NPU) resources under tight memory and latency constraints. Because prefilling and autoregressive decoding have different computational characteristics, the fastest backend depends on the request shape and prefix-cache reuse. This work proposes request-level CPU/NPU routing for the RK3588 system on chip (SoC). Prompt-shape-aware routing (PSR) uses prompt length and generation budget to select a backend for no-cache requests, while prefix-cache-aware routing (PCAR) additionally models reusable-prefix length, remaining prefill workload, full-context effects, and cache-path state with backend-specific latency models. On 49 no-cache held-out workloads, PSR achieves 97.96% backend-selection accuracy and a relative oracle loss of 0.121% $\pm$ 0.010%. On 45 independent cache-aware held-out workloads, accounting for the remaining prefill workload reduces relative oracle loss from 4.926% $\pm$ 0.378% for PSR(raw) to 0.425% $\pm$ 0.060% for PSR(eff). The complete PCAR model further reduces it to 0.205% $\pm$ 0.043%, improves selection accuracy from 93.33% to 95.56%, and provides an additional 0.22% cumulative-latency reduction over that of PSR(eff). These results show that jointly modeling request shape and prefix-cache state enables near-oracle heterogeneous LLM inference routing on edge SoCs.

**Keywords:** edge LLM; heterogeneous inference; CPU/NPU routing; prompt cache; cache-aware scheduling; RK3588

## 1. Introduction

Large language models (LLMs) have been widely adopted in natural language understanding, code generation, intelligent assistants, and on-device intelligence [1–3]. As model size and computational complexity continue to increase, deploying LLMs on resource-constrained edge devices remains challenging. Compared with cloud graphics processing unit (GPU) clusters, edge systems on chips (SoCs) are subject to tighter constraints on memory capacity, memory bandwidth, compute capability, and power budget [4,5]. Therefore, achieving low-latency and efficient LLM inference under limited hardware resources is a key problem for edge AI systems.

To reduce the deployment cost of LLMs at the edge, prior work has primarily focused on model compression, low-bit quantization, and memory optimization [6–13]. LLM.int8() reduces inference memory requirements through mixed-precision 8-bit matrix multiplication [6]; ZeroQuant, SmoothQuant, GPTQ, and AWQ improve the usability of low-bit

Check for updates

Academic Editor: Arkaitz Zubiaga

Received: 4 September 2026
Revised: 3 October 2026
Accepted: 6 October 2026
Published: 9 October 2026

**Copyright:** © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

quantization through post-training quantization, activation smoothing, quantization-error compensation, and salient-weight protection, respectively [7–10]. QLoRA and SpQR further explore 4-bit and sparse quantized representations [11,12]. To address limited device dynamic random-access memory (DRAM) capacity, LLM in a Flash stores model parameters in flash memory and loads them on demand, thereby reducing runtime memory pressure [13]. Overall, these methods reduce model storage, computation, and data-movement costs and provide an important foundation for deploying LLMs on resource-constrained devices.

Beyond model compression, the stage-wise characteristics of LLM inference provide additional opportunities for runtime optimization. Transformer autoregressive inference generally consists of a prefill stage and a decode stage. The prefill stage processes the complete input prompt all at once and exhibits relatively high parallelism, whereas the decode stage generates tokens sequentially and continuously depends on the existing context, making it more constrained by sequential dependencies and memory-access overhead [14–17]. Building on these characteristics, FlashAttention reduces attention-related memory traffic through input/output (I/O)-aware attention [14], vLLM improves KV-cache management through PagedAttention [15], and Orca and FastServe optimize serving performance through continuous batching and request scheduling [16,17]. FlexGen and DeepSpeed-Inference further study data movement across GPU, central processing unit (CPU), and storage tiers [18,19], while Splitwise and Sarathi-Serve exploit the differences between the prefill and decode stages through phase separation or chunked prefilling [20,21]. These studies show that LLM inference performance depends not only on the model itself but also on input length, generation budget, KV-cache state, and the computational characteristics of different inference stages.

Unlike most existing methods designed for GPU or multi-GPU serving environments [15–21], modern edge SoCs often integrate both a general-purpose CPU and a dedicated neural processing unit (NPU). These backends differ substantially in parallel-compute capability, memory-access patterns, operator support, and runtime overhead [22–25], so a single fixed backend is not optimal for all request shapes. In general, requests with long prompts and small generation budgets have a larger prefill share and are more likely to benefit from highly parallel backends; as the generation budget increases, the cumulative decoding cost becomes more important, and a backend with better decoding efficiency may become preferable [20,21]. On the RK3588 platform evaluated in this work, this difference specifically appears as a clear NPU advantage during prefill and a higher decode throughput for the optimized CPU backend, as quantified later in Section 4.2.

Hardware-aware compilation and heterogeneous deployment frameworks, such as TVM, TensorRT, MLC-LLM, and WebLLM, have demonstrated the value of backend optimization and cross-platform adaptation [26–29]. However, these systems mainly focus on operator-level optimization, compilation, GPU acceleration, or portable deployment rather than on request-level CPU/NPU backend selection caused by dynamically changing request shapes and cache states on edge SoCs. In practical LLM applications, prompt lengths and generation budgets vary substantially, while multi-turn conversations, fixed system prompts, and retrieval-augmented generation (RAG) requests often contain repeated prefixes that trigger KV-cache reuse. After a cache hit, the original prompt length no longer equals the amount of prefill computation that must be executed for the current request, and the relative CPU/NPU performance can change accordingly. This motivates a request-level routing mechanism for edge SoCs that jointly accounts for request shape and prefix-cache state.

To address these challenges, this paper proposes a prompt-shape- and prefix-cache-aware request-level CPU/NPU routing method for edge SoCs. The method does not modify the model architecture or require KV-state migration between the CPU and NPU; instead, it

selects one backend for the complete request before execution, based on observable request features. We first characterize the performance differences between the CPU and NPU during the prefill and decode stages on RK3588 and propose prompt-shape-aware routing (PSR) for no-cache requests. We then enable native prompt caching on both llama.cpp and RKLLM and introduce prefix-cache-aware routing (PCAR) for prefix-reuse scenarios such as multi-turn dialogue, fixed system prompts, and RAG. PCAR explicitly represents the remaining prefill workload after cache reuse and further uses backend-specific latency models to capture the different effects of full-context decoding, cache size, and cache-path overhead on CPU and NPU latency. The evaluation uses strictly separated calibration/held-out sets for PSR and development/final sets for PCAR, with all routing rules and model parameters frozen before held-out evaluation.

The main contributions of this work are as follows:

(1) We characterize and exploit the complementary prefill–decode performance of the CPU and NPU on an edge SoC and formulate a complete-request CPU/NPU heterogeneous routing framework.
(2) We propose prompt-shape-aware routing (PSR) for no-cache requests. PSR uses prompt length and the maximum number of new tokens to construct a lightweight lookup-based routing rule, and its near-oracle backend-selection capability is validated on independent held-out workloads.
(3) We propose prefix-cache-aware routing (PCAR), which jointly uses prompt length, reusable-prefix length, remaining prefill length, the maximum number of new tokens, and cache-path state to predict CPU/NPU request latency using backend-specific models. We also introduce a PSR(eff) ablation to separate the contribution of remaining-prefill correction from that of full backend-specific latency modeling.
(4) We validate the proposed routing methods on RK3588 using the native prompt-cache mechanisms of llama.cpp and RKLLM. PSR uses a strictly separated calibration/held-out design, whereas PCAR uses a strictly separated development/final design.

## 2. Related Work and Positioning

Edge-LLM optimization has primarily focused on model compression, low-bit quantization, and data loading under limited memory. Methods such as LLM.int8(), SmoothQuant, GPTQ, AWQ, QLoRA, SpQR, and LLM in a Flash reduce model storage, computation, or runtime memory cost by lowering weight or activation precision, improving quantized representations, or exploiting hierarchical storage [6–13]. These methods mainly optimize the deployment cost of the model itself, whereas this work studies how to dynamically select an execution backend based on request characteristics within a fixed CPU/NPU deployment stack and quantization configuration.

A second line of work focuses on LLM serving and runtime optimization, including attention memory traffic, KV-cache management, continuous batching, resource scheduling, and prefill–decode phase separation. FlashAttention, vLLM, Orca, FastServe, FlexGen, DeepSpeed-Inference, Splitwise, and Sarathi-Serve improve inference efficiency from these perspectives [14–21]. These systems are primarily designed for GPU or multi-GPU environments and often rely on GPU-side resource scheduling or phase splitting. In contrast, this work targets complete-request selection between CPU and NPU on an edge SoC and does not split prefill and decode across devices, which is compatible with deployment conditions in which black-box NPU runtimes such as RKLLM do not expose an interoperable cross-backend KV-state migration interface.

Traditional heterogeneous-computing and deep-learning deployment frameworks, including TVM, TensorRT, MLC-LLM, WebLLM, and HeteroCL, mainly address hardware-aware compilation, kernel optimization, static execution planning, and cross-platform

backend adaptation [26–30]. Unlike these systems, our method neither regenerates operator execution plans nor modifies the underlying NPU runtime; instead, it performs request-level backend selection on top of an existing CPU/NPU deployment stack. Specifically, PSR uses prompt length and generation budget to capture the prefill–decode performance trade-off in no-cache requests, while PCAR further incorporates reusable-prefix length, remaining prefill workload, and cache state and uses separately calibrated CPU/NPU latency models to describe backend-specific responses to the same cache-reuse state. Thus, the focus of this work is not further optimizing a single backend, but selecting the more suitable backend for the entire request according to dynamic request shape and cache state.

## 3. Materials and Methods

### *3.1. Framework Overview*

We propose a prompt-shape- and prefix-cache-aware request-level CPU/NPU routing framework for edge SoCs. The framework does not modify the model architecture or the underlying NPU runtime; instead, it selects either a complete CPU path or a complete NPU path before request execution. As shown in Figure 1, the system consists of request analysis, request-level prefix-cache information, feature construction, PSR/PCAR routing, cache-enabled heterogeneous backends, and offline profiling and model-calibration modules. During online execution, the request-analysis module first obtains the input prompt length $p$ and the maximum number of new tokens $g$. The cache-management module provides the cache state $c$ and reusable-prefix length $p_{\text{cache}}$, from which the remaining prefill length $p_{\text{eff}}$ is derived.

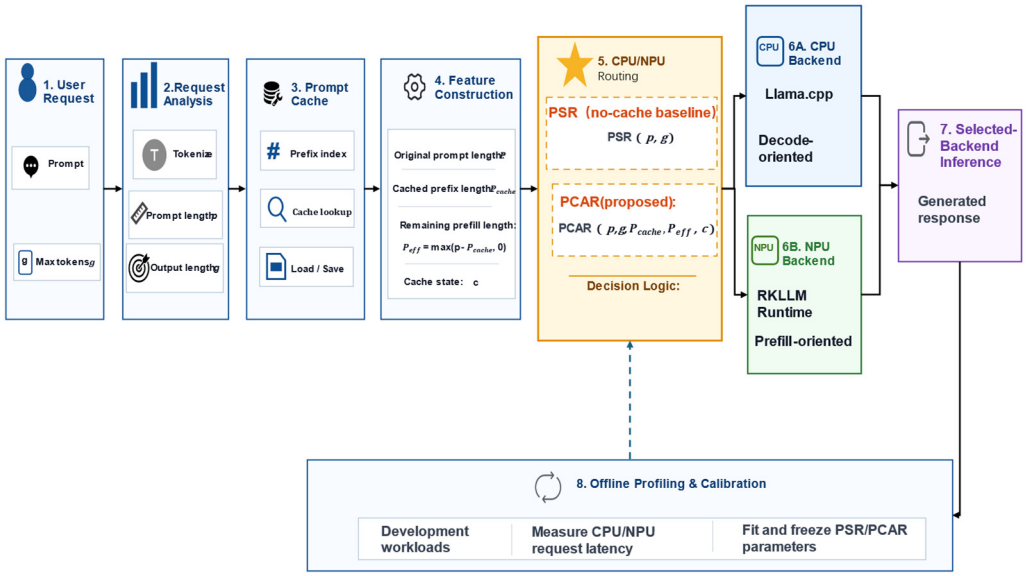



**Figure 1.** Prompt-shape- and prefix-cache-aware request-level CPU/NPU routing framework. PSR routes no-cache requests using prompt length $p$ and the maximum number of new tokens $g$. PCAR further incorporates request-level cache state $c$, reusable-prefix length $p_{\text{cache}}$, and remaining prefill length $p_{\text{eff}}$; predicts request latency using backend-specific CPU and NPU latency models; and selects the backend with the lower predicted latency. PSR rules are calibrated offline on calibration workloads, while PCAR parameters are fitted offline on development workloads; both are frozen before the corresponding held-out evaluation. Abbreviations: CPU, central processing unit; NPU, neural processing unit; PSR, prompt-shape-aware routing; PCAR, prefix-cache-aware routing.

The router selects the decision path according to the available cache information. For no-cache requests, PSR queries a frozen routing rule using only $(p, g)$; when cache information is available, PCAR uses $p$, $g$, $p_{\text{cache}}$, $p_{\text{eff}}$, and the cache state as shared request-

level features, feeds them into backend-specific latency models for the CPU and NPU, and selects the backend with the lower predicted request latency. The selected CPU request is executed by llama-server, whereas the selected NPU request is executed by RKLLM Runtime; both backends use their native prompt-cache mechanisms. PSR rules are calibrated offline using only the calibration set, and PCAR parameters are fitted offline using only the development set; both are frozen before their respective held-out evaluations.

### *3.2. Problem Formulation*

We consider an edge SoC with two inference backends, the CPU and NPU. For any LLM inference request, the system selects an execution backend $b$ from {CPU, NPU}. Let $p$ denote the input prompt length, $g$ the maximum number of new tokens, and $c$ the cache state. When a reusable prefix exists, its length is denoted by $p_{cache}$, and the remaining prefill length is defined as $p_{eff} = \max(p - p_{cache}, 0)$. Here, $p_{eff}$ represents the amount of prefill work that still needs to be executed for the current request. Because the cached KV state remains part of the full context, decoding cost can still depend on the original prompt length $p$. The request-level feature vector is therefore represented as $x = (p, g, c, p_{cache}, p_{eff})$. For backend $b$, let the request latency be $T_b(x)$. The ideal backend is the one with the minimum actual latency, as defined in (1):

$$b^* = arg \min_{b \in \{CPU,NPU\}} T_b(x) \tag{1}$$

Because the actual request latencies of the CPU and NPU are unavailable before execution, the oracle decision in (1) cannot be directly used for online routing. We therefore construct deployable approximate routing strategies from offline profiling. PSR selects a backend from $(p, g)$ in the no-cache setting, whereas PCAR additionally uses $p_{\text{cache}}$, $p_{\text{eff}}$, and cache state to predict the request latencies of the two backends.

### *3.3. Stage-Aware Latency Model*

LLM autoregressive inference typically consists of prefill and decode stages with substantially different computational characteristics. The prefill stage processes the complete input sequence in parallel and therefore exhibits relatively high parallelism, whereas the decode stage generates tokens sequentially and continuously depends on the existing context. Consequently, the relative performance of the CPU and NPU can differ substantially across the two stages. On the RK3588 deployment stack evaluated in this work, the NPU is better suited to highly parallel prefilling, while the optimized CPU incurs lower per-step generation overhead during decoding. To explain this stage complementarity from the perspective of request shape, we decompose the request latency on backend $b$ into prefill cost, decode cost, and other fixed request overheads, as shown in (2).

$$T_b(p, g) = T_{pre,b}(p) + T_{dec,b}(g) + T_{o,b} \tag{2}$$

Because the prefill workload mainly grows with prompt length $p$, whereas cumulative decode workload is primarily controlled by the maximum number of new tokens $g$, (2) can be approximated by (3). $R_{\text{pre},b}$ and $R_{\text{dec},b}$ denote the effective prefill and decode throughputs of backend $b$, respectively; $T_{o,b}$ denotes backend-specific fixed request overhead within the warm-runtime timing boundary that is not explicitly captured by the two stage terms, excluding model initialization, teardown, and first-time cache construction.

$$T_b(p, g) \approx \frac{p}{R_{pre,b}} + \frac{g}{R_{dec,b}} + T_{o,b} \tag{3}$$

Equation (3) is intended only to explain how changes in the prefill and decode workload affect backend selection rather than to precisely predict request latency. This first-order

model does not explicitly capture higher-order changes in decode latency as the context length grows; PCAR introduces context-dependent interaction terms in the subsequent empirical latency model. According to this model, the optimal backend can be expressed as

$$b^* = arg \min_{b \in \{CPU, NPU\}} T_b(p, g) \tag{4}$$

According to the model, an NPU is more likely to achieve lower end-to-end latency when the request is prefill-dominated; as the generation budget increases, the cumulative decoding cost grows, and the relative advantage of the CPU can increase. We therefore use prompt length $p$ and the maximum number of new tokens $g$ as lightweight request-shape features to construct PSR. PCAR further incorporates cache workload and context-dependent features and empirically calibrates request latency for the two backends using the development workloads.

### *3.4. PSR: Prompt-Shape-Aware CPU/NPU Routing*

Based on the stage-wise analysis in Section 3.3, we propose prompt-shape-aware routing (PSR) for CPU/NPU backend selection in no-cache scenarios. PSR is not intended to precisely predict request latency online; instead, it uses the prefill–decode balance implied by prompt length and generation budget to select a more suitable backend through a lightweight rule.

For a no-cache request, there is no reusable prefix, and thus, $p_{\text{eff}} = p$. Prompt length $p$ primarily characterizes prefill scale, while the maximum number of new tokens $g$ represents the cumulative decode budget. PSR, therefore, uses $(p, g)$ as its runtime routing feature. For each request combination $(p_i, g_j)$, the oracle backend is defined as

$$b^*(p_i, g_j) = arg \min_{b \in \{CPU, NPU\}} T_b(p_i, g_j) \tag{5}$$

During offline calibration, we measure CPU and NPU request latency on the calibration workloads. For each request combination $(p_i, g_j)$, the corresponding oracle backend is determined by (5). The resulting two-dimensional backend map exhibits a clear structural boundary: under small generation budgets, prefill accounts for a larger fraction of the request, and the NPU is more likely to be preferable; as $g$ increases, the cumulative decode cost rises, and the CPU's decode-efficiency advantage becomes increasingly important.

Based on this structural boundary, for each fixed generation budget $g_j$, PSR searches a candidate prompt-length threshold $P$ and constructs a binary backend-selection rule according to (6). The final threshold $P_j$ is obtained by minimizing cumulative request latency on the calibration set according to (7). The candidate set also includes the degenerate all-CPU and all-NPU strategies to cover cases in which no internal switching threshold exists within the tested range.

$$b_P(p_i, g_j) = \begin{cases} NPU, & p_i \geq P, \\ CPU, & p_i < \mathrm{P} \end{cases} \tag{6}$$

$$P_j = arg \min_P \sum_i T_{b_P(p_i, g_j)}(p_i, g_j) \tag{7}$$

At runtime, the scheduler only needs $p$ and $g$ to query the frozen PSR rule; the CPU and NPU do not need to be executed simultaneously. Thus, a backend-selection process that would otherwise require online dual-backend measurement is converted into a lookup decision over $(p, g)$. Compared with fixed CPU-only or NPU-only execution, PSR can adapt the backend to the request shape, allowing long-prompt, small-generation-budget requests to exploit the NPU's prefill advantage while avoiding the NPU's slower decoding

for requests with larger generation budgets. The resulting frozen PSR rules are summarized in Table 1. In the current implementation, PSR is defined only for the calibrated generation-budget values $g \in \{32, 64, 128, 192, 256, 384, 512\}$; therefore, the held-out evaluation assesses generalization to unseen prompt lengths rather than unseen generation budgets. Additional $g$ values require corresponding offline profiling and threshold calibration; interpolation or continuous handling of arbitrary $g$ is not evaluated in this work.

**Table 1.** Prompt-shape-aware routing (PSR) rules in the no-cache setting.

| Maximum New Tokens $g$ (Tokens) | NPU-Selection Condition | CPU-Selection Condition |
|---|---|---|
| 32 | $p \geq 256$ tokens | $p < 256$ tokens |
| 64 | $p \geq 512$ tokens | $p < 512$ tokens |
| 128 | $p \geq 1536$ tokens | $p < 1536$ tokens |
| 192 | $p \geq 2048$ tokens | $p < 2048$ tokens |
| 256/384/512 | No NPU selection in the tested range | CPU selected for all tested $p$ |

Note: All prompt-length thresholds and generation-budget values in this table are expressed in tokens. The thresholds were obtained from the 56 calibration workload configurations and were frozen before held-out evaluation.

### *3.5. PCAR: Prefix-Cache-Aware CPU/NPU Routing*

PSR is designed for no-cache requests and bases its decisions on the full prompt length $p$ and generation budget $g$. However, requests in multi-turn dialogue, fixed-system-prompt, RAG, and continuous-task settings often contain reusable prefixes. After a cache hit, part of the KV state no longer needs to be recomputed, so the full prompt length no longer directly reflects the actual prefill workload of the current request. PCAR, therefore, introduces the reusable-prefix length $p_{cache}$ and uses $p_{eff} = \max(p - p_{cache}, 0)$ to represent the remaining prefill workload after cache reuse. Importantly, $p_{cache}$ and $p_{eff}$ are shared request-level features rather than backend-specific quantities; backend-specific responses to the same cache state are captured by separate model parameters.

Using $p_{\text{eff}}$ alone is still insufficient to describe backend latency for cache-enabled requests. Decoding continues to operate over the full context, and the two runtimes can respond differently to cache size and cache-path overhead. PCAR, therefore, adopts the backend-specific empirical model in (8), which jointly captures remaining-prefill cost, basic decode cost, context-dependent decode variation, cache-size-related overhead, and fixed cache-path overhead.

$$\hat{T}_b = \beta_{0,\text{b}} + \beta_{1,\text{b}} p_{eff} + \beta_{2,\text{b}} g + \beta_{3,\text{b}} pg + \beta_{4,\text{b}} p_{\text{cache}} + \beta_{5,b} I(c) \tag{8}$$

where

$$I(\text{c}) = \begin{cases} 1, & \text{c} \in \{prefix\text{-}hit, exact\text{-}hit\}, \\ 0, & \text{c} = no\text{-}cache. \end{cases} \tag{9}$$

In (8), $\beta_{0,\text{b}}$ denotes backend-specific fixed request overhead; $\beta_{1,\text{b}} p_{eff}$ captures the remaining-prefill cost; $\beta_{2,\text{b}} g$ and $\beta_{3,\text{b}} pg$ model the basic decode cost and its dependence on the full context length, respectively; and $\beta_{4,b} p_{cache}$ and $\beta_{5,b} I(c)$ capture cache-size-related overhead and fixed cache-path overhead. The CPU and NPU parameters are fitted separately from measured request latency on the development workloads and frozen before the final held-out evaluation.

At runtime, PCAR computes the predicted latency for both backends and selects the following:

$$b_{PCAR} = \arg \min_{b \in \{CPU, NPU\}} \hat{T}_b \tag{10}$$

PCAR does not require the CPU and NPU to share or migrate KV state. Given the request-level cache state and reusable-prefix length, the method predicts request latency for both backends and selects one backend for the complete request before execution. If cache metadata is unavailable, the system falls back to PSR. The online PCAR decision consists only of a fixed number of arithmetic operations and one backend comparison. The online decision procedure is summarized in Algorithm 1.

**Algorithm 1.** PCAR: Prefix-Cache-Aware CPU/NPU Routing

Input:
$p$ prompt length
$g$ maximum number of new tokens (generation budget)
$c$ cache state
$p_{\text{cache}}$ reusable prefix length
$\beta_{0,\text{b}}, \ldots, \beta_{5,\text{b}}$ frozen model parameters
Output:
$b_{PCAR}$ selected backend
if cache metadata is unavailable then
    $b_{PCAR} \leftarrow \text{PSR}(p, g)$
    return $b_{PCAR}$
end if
if $c$ = no-cache then
    $p_{\text{cache}} \leftarrow 0$
end if
$p_{eff} \leftarrow \max(p - p_{\text{cache}}, 0)$
I(c) $\leftarrow$ 0 if c = no-cache else 1
for b $\in$ {CPU, NPU} do
    $\widehat{T}_b \leftarrow \beta_{0,\text{b}} + \beta_{1,\text{b}} p_{eff} + \beta_{2,\text{b}} g + \beta_{3,\text{b}} pg + \beta_{4,\text{b}} p_{\text{cache}} + \beta_{5,b} I(c)$
end for
$b_{PCAR} \leftarrow arg \min_{b \in \{CPU,NPU\}} \widehat{T}_b$
return $b_{PCAR}$

## 4. Results

### *4.1. Experimental Setup*

All experiments were conducted on an ATK-DLRK3588 board equipped with an RK3588 SoC and 16 GB of memory. The NPU backend uses RKLLM Runtime v1.2.2 with the W8A8-quantized model deepseek1p5b_w8a8_normal_core3_v122.rkllm and all three NPU cores. The CPU backend uses an optimized AArch64 llama.cpp implementation with DeepSeek-R1-Distill-Qwen-1.5B.Q4_K_M.gguf, ARMv8.2-A dot-product instructions, and Q4_K $\times$ Q8_K feed-forward network (FFN) optimization. The CPU llama-server runs with four inference threads and a maximum context length of 4096 tokens. All experiments use the same reflash-v096 system environment. Because the two backends use different quantization formats and runtimes, the comparison reflects request-level engineering performance of the actual RK3588 deployment stacks rather than pure hardware performance under strictly identical quantization and operator implementations. The primary latency metric is warm-runtime request latency after model loading. CPU requests are executed through llama-server's/completion interface and measured using client-side wall-clock time, which includes local interface and server-bookkeeping overhead; model and server initialization are excluded. For NPU no-cache requests, latency is measured using run_wall_ms reported by RKLLM Runtime, while for cache-hit requests, it is defined as $T_{\text{NPU}} = T_{\text{load_cache}} + T_{\text{run}}$.

Model initialization, teardown, and first-time cache construction are excluded. Therefore, the reported latencies represent the warm-runtime request-level performance of the two practical deployment stacks rather than strictly interface-normalized compute time. A sensitivity analysis further confirms that the difference in timing boundaries does not change the oracle backend decisions on the PSR held-out workloads.

Request-level routing features are determined using a unified token-counting convention. $p$ denotes prompt length, and $g$ denotes the maximum number of new tokens. Cache-enabled requests additionally define reusable-prefix length $p_{\text{cache}}$ and remaining prefill length $p_{\text{eff}}$. The CPU and NPU runtimes may report slightly different prefill token counts because of special tokens and internal counting conventions; these runtime counts are used only for behavior diagnostics and are not routing inputs. The cache-reuse ratio is defined as $r = p_{\text{cache}}/p$. We evaluate $r \in \{0, 0.25, 0.50, 0.75, 1.00\}$, corresponding to no-cache, partial prefix-hit, and exact-hit conditions. For each cache-aware workload, the CPU and NPU use the same logical prompt, reusable-prefix content, cache-reuse ratio, and generation budget, but each backend materializes and maintains its cache independently using its native runtime mechanism. On the CPU side, the reusable prefix—or the full prompt for an exact hit—is first issued to llama-server with prompt caching enabled to populate the designated cache slot before the target request is executed. On the NPU side, the same logical prefix—or the full prompt for an exact hit—is first processed through the RKLLM prompt-cache save interface to create a backend-native cache file, which is then loaded for the target request. No KV-cache state is shared or migrated between the two backends. Therefore, the cache-aware evaluation represents a controlled steady-state condition in which an equivalent reusable prefix has already been materialized independently on both backends before the target request is measured. PSR uses a strictly separated calibration/held-out design. The calibration set combines $p \in \{128, 256, 512, 768, 1024, 1536, 2048, 3072\}$ with $g \in \{32, 64, 128, 192, 256, 384, 512\}$, producing 56 no-cache workloads. The held-out set uses prompt lengths $p \in \{192, 384, 640, 896, 1280, 1792, 2560\}$ with the same $g$ set, producing 49 workloads. Each backend-workload pair is repeated three times, resulting in 336 backend measurements for calibration and 294 for held-out evaluation. All PSR rules are generated only from the calibration set and frozen before held-out evaluation. PCAR uses an independent development/final design. The development set contains $p \in \{512, 1024, 2048, 3072\}$, $g \in \{64, 256, 512\}$, and five cache-reuse ratios, yielding 60 workloads and 360 backend measurements. The final held-out set uses independent prompt content and previously unseen $p \in \{896, 1280, 2560\}$ and $g \in \{128, 192, 384\}$ values within the tested range, together with the same five cache-reuse ratios, yielding 45 workloads and 270 backend measurements. All PCAR parameters are frozen before final evaluation. To support reproducibility, the benchmark measurements generated in this study, together with the PSR/PCAR model-fitting and routing-evaluation scripts, are publicly archived in Zenodo [31].

Frozen routing strategies are evaluated by offline replay over the measured dual-backend latency matrix. For each workload, a strategy first selects a backend from the request features, and its performance is then computed using the measured request latency of that selected backend; the CPU and NPU are not executed simultaneously online. The no-cache evaluation compares CPU-only, NPU-only, prompt-only, PSR, and oracle, whereas the cache-aware evaluation compares CPU-only(cache), NPU-only(cache), PSR(raw), PSR(eff), PCAR, and oracle. PSR(raw) applies the frozen PSR rule to the original $(p, g)$; PSR(eff) does not refit the rule, but instead applies the same frozen PSR rule to $(p_{\text{eff}}, g)$, isolating the contribution of remaining-prefill correction. Both strategies use the native prompt cache

of the selected backend during execution. For workload $i$, let $b$ denote backend $\overline{T_{i,b}}$ mean request latency over the three repetitions. The oracle backend is then defined as

$$b_i^* = \arg \min_{b \in \{CPU, NPU\}} \overline{T_{i,b}} \tag{11}$$

For routing strategy $S$, cumulative request latency in repetition $k$ is defined as

$$T_S^{(k)} = \sum_{i=1}^{N} T_{i,b_{i,S}}^{(k)} \tag{12}$$

where $b_{i,S}$ denotes the backend selected by strategy $S$ for workload $i$. The corresponding oracle cumulative latency is defined as $T_{\text{oracle}}^{(k)} = \sum_{i=1}^{N} T_{i,b_i^*}^{(k)}$. The relative oracle loss is then defined as

$$L_{\text{oracle}}^{(k)}(S) = \frac{T_S^{(k)} - T_{\text{oracle}}^{(k)}}{T_{\text{oracle}}^{(k)}} \times 100\%. \tag{13}$$

Backend-selection accuracy is defined as

$$A(S) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(b_{i,S} = b_i^*) \times 100\%. \tag{14}$$

Each backend-workload combination is measured three times. Cumulative request latency and relative oracle loss are computed independently for the three repetition rounds and reported as mean ± standard deviation. Backend-selection accuracy is computed against the oracle backend determined from the mean latency across the three repetitions. Because CPU request latency includes local/completion interface overhead while NPU latency is measured using timing exposed by RKLLM Runtime, the results should be interpreted as request-level performance of the two practical deployment stacks rather than as strictly interface-normalized compute-kernel time. The complete platform, model, and workload configuration is summarized in Table 2.

**Table 2.** Experimental platform, models, and workload configuration. Abbreviations: SoC, system on chip; CPU, central processing unit; NPU, neural processing unit.

| Item | Configuration |
| --- | --- |
| SoC | ATK-DLRK3588 Board/RK3588 |
| Memory | 16 GB |
| System | reflash-v096 |
| CPU frequency policy | schedutil; maximum 1.8/2.304/2.352 GHz across three CPU frequency domains |
| NPU frequency policy | rknpu_ondemand; maximum 1.0 GHz |
| Power/frequency mode | Dynamic DVFS; no fixed-frequency performance mode |
| CPU runtime | Custom optimized AArch64 llama.cpp/llama-server; 4 threads; upstream Git revision not embedded in deployed binary |
| CPU runtime arguments | -c 4096 -t 4 -np 1 --cache-prompt --no-cache-idle-slots --no-webui |
| CPU model | DeepSeek-R1-Distill-Qwen-1.5B.Q4_K_M.gguf |
| CPU decoding | n_predict = g; temperature = 0; non-streaming; no stop strings; ignore_eos = true for corrected fixed-budget PSR profiling |
| NPU runtime | RKLLM Runtime v1.2.2; three NPU cores |

**Table 2.** *Cont.*

| Item | Configuration |
| --- | --- |
| NPU model | deepseek1p5b_w8a8_normal_core3_v122.rkllm; W8A8 |
| NPU decoding | max_new_tokens = g; top-k = 1; top-p = 0.95; temperature = 0.8; repeat penalty = 1.1; frequency/presence penalty = 0; skip_special_token = true |
| Maximum context | 4096 tokens |
| Warm-up | Model/runtime warm-up and cache-construction preparation are excluded from target-request latency |
| Cache reuse ratio | 0%, 25%, 50%, 75%, 100% |
| Repetitions | 3 |

### *4.2. Stage-Wise CPU/NPU Characterization*

Figure 2 summarizes the stage-wise throughput characteristics of the CPU and NPU backends. As shown in Figure 2a, NPU prefill throughput is substantially higher than CPU throughput across all tested prompt lengths. Except for small fluctuations at short prompts, prefill throughput on both backends generally decreases as prompt length increases, while the NPU retains a clear advantage at every tested point. This result is consistent with the higher degree of parallelism in the prefill stage and shows that the NPU can exploit this parallelism effectively on the evaluated RK3588 stack.

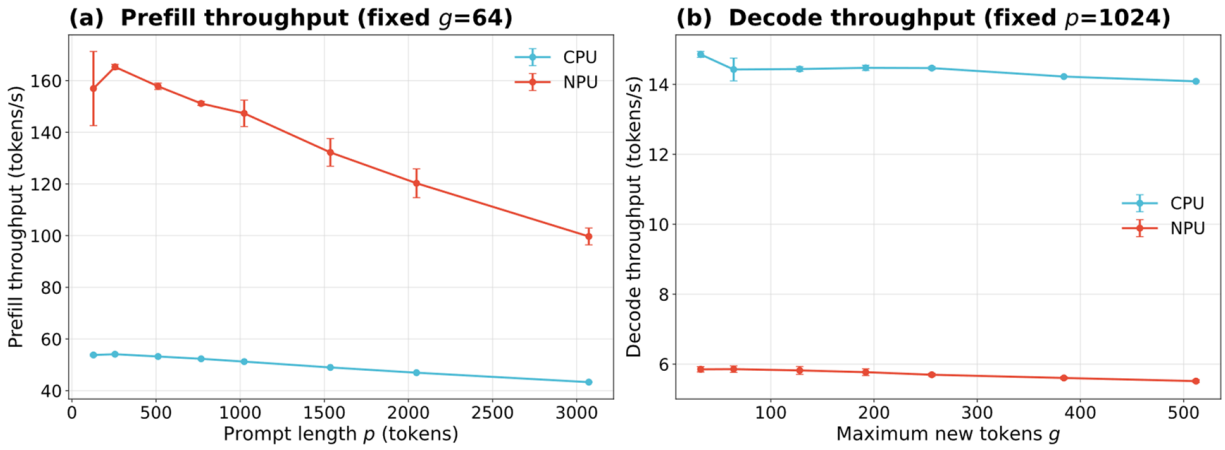



**Figure 2.** Stage-wise throughput comparison of CPU and NPU. (**a**) Prefill throughput versus prompt length $p$, with the maximum number of new tokens fixed at $g = 64$. (**b**) Decode throughput versus the maximum number of new tokens $g$, with prompt length fixed at $p = 1024$. Each point shows the mean of three repeated measurements, and the error bars indicate the standard deviation.

In contrast, CPU decode throughput is consistently higher than NPU decode throughput on the evaluated RK3588 deployment stack, as shown in Figure 2b. As the maximum number of new tokens increases, decode time accumulates and constitutes a larger fraction of total request latency. Therefore, the NPU's prefill advantage does not necessarily translate into lower overall latency for requests with larger generation budgets.

These results show that there is no globally optimal backend on the evaluated platform. The NPU is better suited to prefill-dominated requests with long prompts and small generation budgets, whereas the CPU is better suited to decode-dominated requests with larger generation budgets. This stage-wise complementarity provides the direct experimental motivation for the prompt-shape-aware PSR policy.

### *4.3. No-Cache Prompt-Shape-Aware Routing Results*

To examine whether prompt length $p$ and the maximum number of new tokens $g$ can effectively characterize the CPU/NPU switching boundary, we first evaluate PSR in the no-cache setting. The PSR rule is calibrated offline using only the 56 calibration workloads defined in Section 4.1 and then independently evaluated, after freezing, on 49 held-out workloads that were not used for calibration.

As shown in Figure 3, the oracle backend distribution on the calibration set forms a clear structural boundary in the $(p, g)$ space. The frozen PSR rules are listed in Table 1. For $g$ = 32, 64, 128, 192, the prompt-length thresholds for choosing the NPU are $p$ = 256, 512, 1536, and 2048, respectively; for $g$ = 256, 384, and 512, the CPU is selected for all prompt lengths within the calibration range. This trend is consistent with the stage-wise complementarity shown in Figure 2: the prefill stage makes a greater contribution under small generation budgets, whereas the CPU's decode stage advantage increasingly dominates total request latency as $g$ grows.

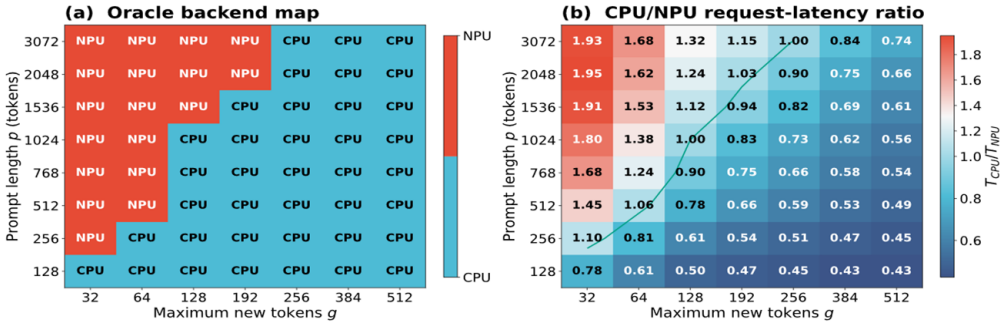



**Figure 3.** Oracle backend distribution under no-cache calibration workloads. (**a**) Oracle backend for each $(p, g)$ pair, determined from the mean measured request latency of CPU and NPU over three repetitions. (**b**) CPU/NPU request-latency ratio T_CPU/T_NPU. Values greater than 1 indicate that the NPU is faster, values below 1 indicate that the CPU is faster, and the contour at 1 denotes the approximate backend switching boundary.

On the independent held-out set, CPU-only and NPU-only incur relative oracle losses of 9.105% ± 0.041% and 38.909% ± 0.490%, with backend-selection accuracies of 69.39% and 30.61%, respectively. Prompt-only, which uses only prompt length, yields a single threshold of $p = 3072$ on the calibration set. Because all prompt lengths in the held-out set are below this threshold, prompt-only degenerates to CPU-only. This result shows that prompt length alone is insufficient to capture the backend switching caused by changes in generation budget.

In contrast, PSR achieves a backend-selection accuracy of 97.96% on the 49 held-out workloads, with a relative oracle loss of only 0.121% ± 0.010%. Its cumulative request latency is 1746.87 ± 1.32 s, compared with 1744.75 ± 1.44 s for the oracle. The difference between the mean cumulative latencies is small relative to the run-to-run variability observed across the repeated measurements and is therefore not interpreted as a precise performance gap. Taken together, the high backend-selection accuracy and low relative oracle loss indicate that jointly considering prompt length and generation budget closely reproduces the oracle backend-switching behavior associated with changes in the prefill–decode workload balance. The held-out routing comparison is summarized in Figure 4.

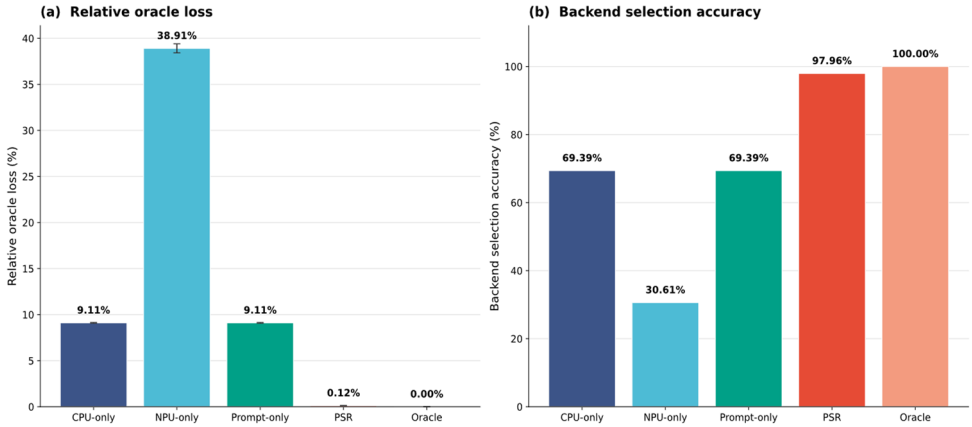



**Figure 4.** Performance comparison of routing strategies on the independent no-cache held-out set. (**a**) Relative oracle loss; (**b**) backend-selection accuracy. Prompt-only uses prompt length $p$ alone, whereas PSR jointly considers prompt length $p$ and the maximum number of new tokens $g$. Error bars in panel (**a**) indicate the standard deviation across the three cumulative repetition results.

### *4.4. Cache-Aware Routing Results*

PSR selects a backend based on prompt length $p$ and the maximum number of new tokens $g$, and accurately captures the CPU/NPU switching boundary in the no-cache setting. Prefix caching, however, reuses existing KV states and reduces the amount of prefill work that must be re-executed relative to the full prompt length. As the cache-reuse ratio increases, the NPU's prefill advantage can diminish while the CPU's decode advantage accounts for a larger fraction of total request latency. Accordingly, this section first characterizes the native prompt-cache behavior of both backends, then evaluates the PCAR latency model, and finally assesses cache-aware routing on the independent held-out workloads.

#### 4.4.1. Native Prompt-Cache Behavior on Both Backends

To characterize the effect of prefix caching on actual execution, the CPU backend uses the native prompt cache of llama.cpp/llama-server, whereas the NPU backend uses the prompt-cache save/load interface provided by RKLLM Runtime. Figure 5 reports a representative development workload under the five cache-reuse ratios defined in Section 4.1. As cache reuse increases from 0% to 100%, the runtime-reported remaining prefill tokens decrease continuously on both the CPU and NPU, and the corresponding prefill latency also decreases markedly. Under exact-hit, repeated prefill is completely eliminated on the NPU, while only a very small amount of runtime processing remains on the CPU. In contrast, with the same generation budget, the decode cost does not decrease proportionally with the cache-reuse ratio.

These results show that prompt caching primarily eliminates repeated prefill stages rather than shortening the full context used by the autoregressive decode stage. More importantly, even though the remaining prefill workload decreases in a similar manner on both the CPU and NPU, the resulting request-latency responses differ across the two backends. As cache reuse increases, a request can therefore shift from prefill-dominated to decode-dominated, which may change the optimal backend. This observation directly motivates PCAR to account for remaining prefill workload, full context length, and generation budget jointly.

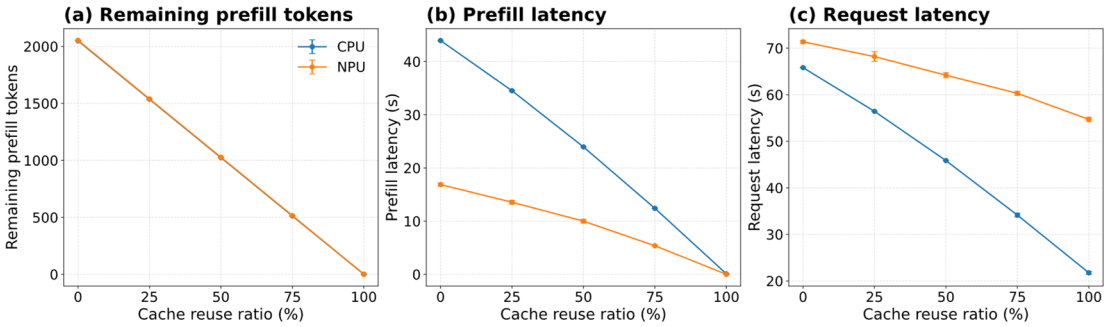



**Figure 5.** Native prompt-cache behavior of CPU and NPU. (**a**) Runtime-reported remaining prefill tokens at different cache-reuse ratios; CPU values are from llama-server prompt_n and NPU values are from RKLLM Runtime prefill_tokens. (**b**) Prefill latency. (**c**) Cache-enabled request latency. Cache-reuse ratios are 0%, 25%, 50%, 75%, and 100%, where 100% corresponds to exact-hit. Each point shows the mean of three repeated measurements, and error bars indicate the standard deviation.

#### 4.4.2. PCAR Model Calibration and Internal Validation

Using the development set defined in Section 4.1, we calibrate separate cache-aware latency models for both the CPU and NPU. Each workload-backend pair is fitted using the mean request latency over three repetitions, and all model parameters are frozen before final held-out evaluation. Figure 6 compares measured and predicted request latencies. The CPU model achieves a root mean square error (RMSE) of 1.216 s, a mean absolute error (MAE) of 1.007 s, and an $R^2 = 0.9980$; the corresponding NPU values are 1.372 s, 1.079 s, and 0.9989. The predictions closely match the measurements for both backends, indicating that the low-dimensional feature set captures latency variation induced jointly by prompt length, generation budget, and cache reuse.

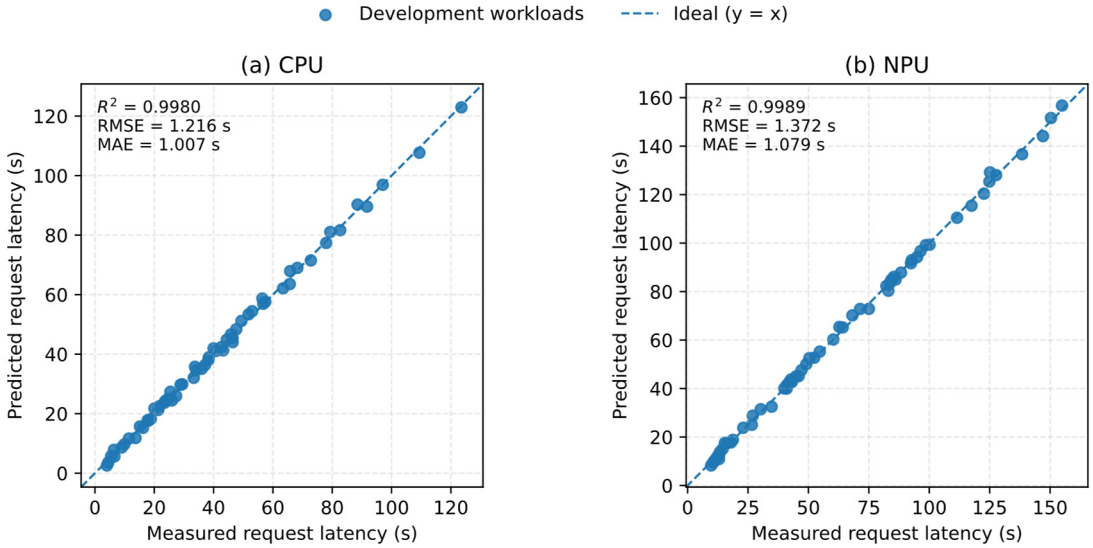



**Figure 6.** Fit of the PCAR cache-aware latency models on the development set. (**a**) Measured versus predicted request latency for the CPU backend. (**b**) Measured versus predicted request latency for the NPU backend. Each point represents the three-run mean for one workload, and the dashed line indicates the ideal relationship y = x.

To test whether the models depend excessively on a particular prompt length or cache ratio, we further perform leave-one-prompt-length-out and leave-one-cache-ratio-out validation. Within the development workload family, both validation procedures maintain 100% backend-selection accuracy and 0% relative oracle loss. Because these are internal stability checks within the development workload family, final performance

is assessed on the independent final held-out set. The CPU and NPU models are fitted independently using unweighted ordinary least squares (OLS), without regularization or sample weighting. For each backend, the three repeated latency measurements of each development workload are first averaged, yielding 60 observations per backend for model fitting. The resulting frozen backend-specific coefficients are reported in Table 3.

**Table 3.** Frozen backend-specific PCAR coefficients obtained from the development set.

| Parameter | Meaning | Unit | CPU | NPU |
|---|---|---|---|---|
| $\beta_0$ | intercept | ms | −5041.47 | −5413.74 |
| $\beta_1$ | $p_{\text{eff}}$ coefficient | ms/token | 24.71 | 11.27 |
| $\beta_2$ | $g$ coefficient | ms/token | 58.56 | 148.53 |
| $\beta_3$ | $p \times g$ coefficient | ms/token$^2$ | 0.014 | 0.033 |
| $\beta_4$ | $p_{\text{cache}}$ coefficient | ms/token | 1.15 | 1.54 |
| $\beta_5$ | cache-hit fixed term | ms | 2846.87 | 2324.03 |

#### 4.4.3. Final Held-Out Cache-Aware Routing Performance

After freezing the PCAR parameters, we perform final evaluation on the held-out set defined in Section 4.1 and compare CPU-only(cache), NPU-only(cache), PSR(raw), PSR(eff), PCAR, and oracle. As shown in Figure 7a, their cumulative request latencies are 1610.05 ± 0.57 s, 2324.18 ± 18.16 s, 1641.98 ± 3.70 s, 1571.55 ± 1.21 s, 1568.11 ± 1.61 s, and 1564.89 ± 2.10 s, respectively. Relative to cache-unaware PSR(raw), PCAR reduces cumulative latency by 4.50%; relative to PSR(eff), which only corrects the routing input using remaining prefill length, PCAR provides a further reduction of approximately 0.22%.

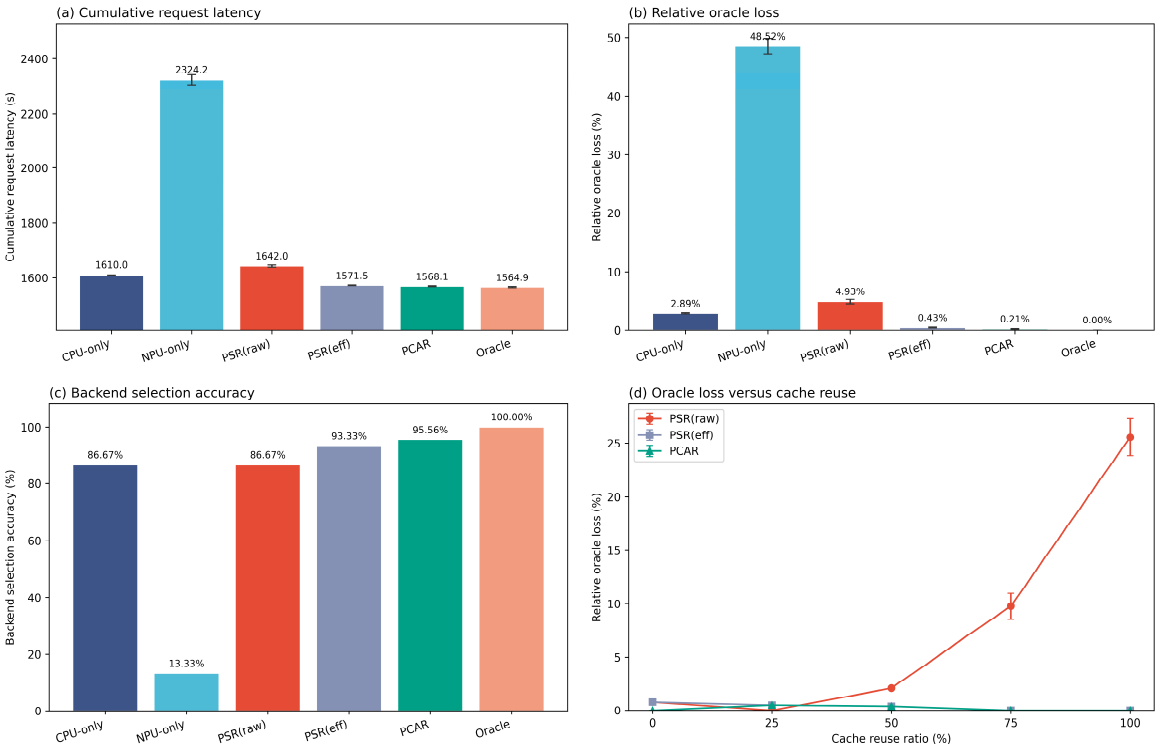



**Figure 7.** Cache-aware routing performance on the independent final held-out set. (**a**) Cumulative request latency of CPU-only(cache), NPU-only(cache), PSR(raw), PSR(eff), PCAR, and oracle. (**b**) Relative oracle loss. (**c**) Backend-selection accuracy. (**d**) Relative oracle loss of PSR(raw), PSR(eff), and PCAR at different cache-reuse ratios. PSR(raw) routes using the original $(p, g)$, whereas PSR(eff) applies the same frozen PSR rule to $(p_{\text{eff}}, g)$; both use the native prompt cache of the selected backend during execution. Error bars in panels (**a**,**b**,**d**) indicate the standard deviation across the three repetition results.

Figure 7b,c further illustrate the contributions of the two levels of cache awareness. The relative oracle losses of PSR(raw), PSR(eff), and PCAR are 4.926% ± 0.378%, 0.425% ± 0.060%, and 0.205% ± 0.043%, respectively, while their backend-selection accuracies are 86.67%, 93.33%, and 95.56%. Thus, replacing $p$ with $p_{\text{eff}}$ removes most of the cache-unaware routing error, while the backend-specific PCAR model further improves decisions near the backend switching boundary. The oracle selects CPU for 39 of the 45 workloads and NPU for 6, whereas PCAR selects CPU for 41 workloads and NPU for 4.

As shown in Figure 7d, PSR(raw) is highly sensitive to the degree of cache reuse. At cache-reuse ratios of 0%, 25%, 50%, 75%, and 100%, its relative oracle losses are 0.79%, 0%, 2.11%, 9.78%, and 25.57%, respectively. In contrast, the corresponding PSR(eff) losses are 0.79%, 0.50%, 0.39%, 0%, and 0%, while PCAR yields 0%, 0.50%, 0.39%, 0%, and 0%. These results further show that explicitly representing the remaining prefill workload accounts for most of the benefit of cache-aware routing, while PCAR further improves cumulative performance and backend-selection accuracy. Note that the nine workloads with $r = 0$ in Figure 7d belong to the final cache-aware set, whereas Figure 4 uses 49 no-cache PSR held-out workloads; therefore, the PSR losses in the two figures are not directly comparable.

PCAR makes only two misclassifications in the final evaluation, at ($r$ = 25%, $p$ = 2560, $g$ = 192) and ($r$ = 50%, $p$ = 2560, $g$ = 128). Both lie in crossover regions, where the CPU and NPU latencies are close, indicating that the remaining errors are concentrated near the backend switching boundary.

## 5. Discussion

The results show that request-level CPU/NPU backend selection on the evaluated RK3588 deployment stacks is strongly influenced by the relative contribution of the prefill and decode stages within each request. On the evaluated RK3588 stack, the NPU advantage primarily comes from highly parallel prefilling, whereas the optimized CPU provides higher throughput during autoregressive decoding. A fixed backend, therefore, cannot accommodate all combinations of prompt length and generation budget, and the routing boundary learned by PSR can be interpreted as the point at which the relative stage costs favor one backend over the other. A secondary Qwen2.5-0.5B sensitivity experiment exhibits the same qualitative prefill–decode-driven routing structure but shifted switching thresholds; profiling the same 56 $(p, g)$ request shapes, corresponding to 336 CPU/NPU measurements with three repetitions, required approximately 167.7 min (2.79 h) of cumulative sequential profiling time.

Prefix caching further changes this stage-cost balance. Figure 5 shows that the remaining prefill workload decreases similarly on both the CPU and NPU as cache reuse increases, but the resulting request-latency responses differ. The final held-out ablation shows that simply replacing $p$ with $p_{\text{eff}}$ in the frozen PSR rule reduces the relative oracle loss from 4.926% to 0.425%, demonstrating that correctly representing the remaining prefill workload is the main source of the cache-aware routing benefit. PCAR further models the full-context effects, generation budget, and cache-path overhead, reducing the loss to 0.205% and improving backend-selection accuracy from 93.33% to 95.56%. Thus, PCAR does not merely replace $p$ with $p_{\text{eff}}$; rather, it further captures backend-specific latency responses to the same cache state using cache-aware features. The reported routing performance is obtained by offline replay over real dual-backend measurements, and the online execution overhead of the router itself has not been separately measured.

This work uses complete-request routing rather than splitting prefilling and decoding across different backends. Although phase-level migration could, in principle, exploit CPU/NPU complementarity more aggressively, llama.cpp and RKLLM Runtime use different quantization formats, runtime interfaces, and KV-cache representations, and the

current platform does not expose a directly interoperable mechanism for cross-backend KV-state migration. Complete-request routing is therefore more directly deployable under a black-box NPU runtime.

This study has several limitations. First, the full PSR/PCAR evaluation is conducted on a single RK3588 platform using the 1.5B model and one RKLLM runtime version, while the Qwen2.5-0.5B experiment serves only as a secondary model-size sensitivity check. Accordingly, the observed stage-wise ordering—particularly the higher decode throughput of the optimized CPU—should be interpreted as a property of the evaluated deployment stack rather than as a universal CPU/NPU characteristic. The proposed routing framework does not assume a fixed backend ordering; instead, the PSR thresholds and PCAR parameters are obtained through offline profiling and must be recalibrated when the hardware, model, quantization configuration, or runtime changes. In the present configuration, full recalibration involves 336 backend measurements for PSR and 360 for PCAR. Second, the CPU and NPU deployments use Q4_K_M and W8A8 quantization, respectively; the reported comparison, therefore, reflects request-level performance of the actual deployment stacks rather than pure hardware performance under strictly controlled quantization and operator implementations. Third, the cache-aware evaluation represents a controlled steady-state condition in which the same logical reusable prefix has already been materialized independently in the native cache of each backend. No KV-cache representation is shared or migrated between the CPU and NPU. This setup isolates backend latency responses under an equivalent prefix-reuse opportunity, but it does not model backend-asymmetric cache residency. In a practical multi-turn system, a reusable prefix may reside only on the backend that processed an earlier request, while the other backend may require cache reconstruction. Incorporating backend-specific cache availability, cache-construction cost, eviction policy, and request history, together with concurrent workloads, energy efficiency, and output-quality constraints, remains an important direction for future work.

## 6. Conclusions

This paper studies request-level CPU/NPU backend selection for LLM inference on a resource-constrained edge SoC. Stage-wise characterization on RK3588 shows that the NPU provides higher throughput during highly parallel prefilling, whereas the optimized CPU provides higher generation throughput during autoregressive decoding, demonstrating that a fixed backend cannot efficiently serve all request shapes. Based on this complementarity, we propose two request-level routing methods. PSR handles no-cache requests using prompt length and the maximum number of new tokens, while PCAR additionally incorporates reusable-prefix length, remaining prefill workload, and cache-path state, predicting cache-enabled CPU/NPU request latency using backend-specific models. Both methods select one backend for the complete request, and neither requires cross-backend KV-cache migration.

On 49 no-cache held-out workloads not used for rule calibration, PSR achieves 97.96% backend-selection accuracy, with a relative oracle loss of only 0.121% $\pm$ 0.010%. On 45 final cache-aware held-out workloads, PSR(raw), PSR(eff), and PCAR achieve relative oracle losses of 4.926% $\pm$ 0.378%, 0.425% $\pm$ 0.060%, and 0.205% $\pm$ 0.043%, respectively. Thus, accounting for the remaining prefill workload provides most of the cache-aware routing benefit, while the complete PCAR model further improves backend-selection accuracy from 93.33% to 95.56% and reduces cumulative latency by an additional 0.22% relative to the results for PSR(eff).

On the evaluated RK3588 deployment stacks, these results indicate that effective CPU/NPU routing depends on matching backend stage characteristics to request shape and prefix-cache state, while quantitative switching thresholds remain model- and runtime-

dependent. Future work will investigate additional models and hardware platforms, concurrent workloads, asymmetric cache residency, energy efficiency, and multi-objective routing.

**Author Contributions:** Conceptualization, Q.H. and L.W.; methodology, Q.H. and L.W.; software, Q.H.; validation, Q.H.; formal analysis, Q.H.; investigation, Q.H.; data curation, Q.H., Y.M., Q.G. and K.H.; writing—original draft preparation, Q.H.; writing—review and editing, Q.H. and L.W.; visualization, Q.H.; supervision, L.W. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was supported by the Shanghai Xiaoyuan Innovation Center Youth Fund (No. 2026-0005).

**Data Availability Statement:** The original data presented in this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.23117825 [31]. The repository includes the benchmark prompt files and manifests, PSR and PCAR measurement datasets, latency-collection scripts, and scripts for model fitting, routing evaluation, and figure reproduction.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 5998–6008.
2. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
3. Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; et al. LLaMA: Open and efficient foundation language models. *arXiv* **2023**, arXiv:2302.13971. [CrossRef]
4. Zheng, Y.; Chen, Y.; Qian, B.; Shi, X.; Shu, Y.; Chen, J. A review on edge large language models: Design, execution, and applications. *ACM Comput. Surv.* **2025**, *57*, 209. [CrossRef]
5. Navardi, M.; Aalishah, R.; Fu, Y.; Lin, Y.; Li, H.; Chen, Y.; Mohsenin, T. GenAI at the edge: Comprehensive survey on empowering edge devices. *Proc. AAAI Symp. Ser.* **2025**, *5*, 180–187. [CrossRef]
6. Dettmers, T.; Lewis, M.; Belkada, Y.; Zettlemoyer, L. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 30318–30332. [CrossRef]
7. Yao, Z.; Aminabadi, R.Y.; Zhang, M.; Wu, X.; Li, C.; He, Y. ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 27168–27183. [CrossRef]
8. Xiao, G.; Lin, J.; Seznec, M.; Wu, H.; Demouth, J.; Han, S. SmoothQuant: Accurate and efficient post-training quantization for large language models. In Proceedings of the 40th International Conference on Machine Learning (ICML 2023), Honolulu, HI, USA, 23–29 July 2023; Volume 202, pp. 38087–38099.
9. Frantar, E.; Ashkboos, S.; Hoefler, T.; Alistarh, D. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In Proceedings of the 11th International Conference on Learning Representations (ICLR 2023), Kigali, Rwanda, 1–5 May 2023.
10. Lin, J.; Tang, J.; Tang, H.; Yang, S.; Chen, W.-M.; Wang, W.-C.; Xiao, G.; Dang, X.; Gan, C.; Han, S. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. *Proc. Mach. Learn. Syst.* **2024**, *6*, 87–100.
11. Dettmers, T.; Pagnoni, A.; Holtzman, A.; Zettlemoyer, L. QLoRA: Efficient finetuning of quantized LLMs. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 10088–10115. [CrossRef]
12. Dettmers, T.; Svirschevski, R.; Egiazarian, V.; Kuznedelev, D.; Frantar, E.; Ashkboos, S.; Borzunov, A.; Hoefler, T.; Alistarh, D. SpQR: A sparse-quantized representation for near-lossless LLM weight compression. In Proceedings of the 12th International Conference on Learning Representations (ICLR 2024), Vienna, Austria, 7–11 May 2024.
13. Alizadeh, K.; Mirzadeh, S.I.; Belenko, D.; Khatamifard, S.; Cho, M.; Del Mundo, C.C.; Rastegari, M.; Farajtabar, M. LLM in a flash: Efficient large language model inference with limited memory. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Bangkok, Thailand, 11–16 August 2024; pp. 12562–12584. [CrossRef]
14. Dao, T.; Fu, D.Y.; Ermon, S.; Rudra, A.; Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 16344–16359. [CrossRef]
15. Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C.H.; Gonzalez, J.E.; Zhang, H.; Stoica, I. Efficient memory management for large language model serving with PagedAttention. In Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23), Koblenz, Germany, 23–26 October 2023; pp. 611–626. [CrossRef]

16. Yu, G.-I.; Jeong, J.S.; Kim, G.-W.; Kim, S.; Chun, B.-G. Orca: A distributed serving system for transformer-based generative models. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22), Carlsbad, CA, USA, 11–13 July 2022; pp. 521–538.
17. Wu, B.; Zhong, Y.; Zhang, Z.; Liu, S.; Liu, F.; Sun, Y.; Huang, G.; Liu, X.; Jin, X. Fast distributed inference serving for large language models. *arXiv* **2023**, arXiv:2305.05920. [CrossRef]
18. Sheng, Y.; Zheng, L.; Yuan, B.; Li, Z.; Ryabinin, M.; Chen, B.; Liang, P.; Ré, C.; Stoica, I.; Zhang, C. FlexGen: High-throughput generative inference of large language models with a single GPU. In Proceedings of the 40th International Conference on Machine Learning (ICML 2023), Honolulu, HI, USA, 23–29 July 2023; Volume 202, pp. 31094–31116.
19. Aminabadi, R.Y.; Rajbhandari, S.; Zhang, M.; Awan, A.A.; Li, C.; Li, D.; Zheng, E.; Rasley, J.; Smith, S.; Ruwase, O.; et al. DeepSpeed Inference: Enabling efficient inference of transformer models at unprecedented scale. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '22), Dallas, TX, USA, 13–18 November 2022; pp. 1–15. [CrossRef]
20. Patel, P.; Choukse, E.; Zhang, C.; Shah, A.; Goiri, Í.; Maleki, S.; Bianchini, R. Splitwise: Efficient generative LLM inference using phase splitting. In Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24), Buenos Aires, Argentina, 29 June–3 July 2024; pp. 118–132. [CrossRef]
21. Agrawal, A.; Kedia, N.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B.S.; Tumanov, A.; Ramjee, R. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24), Santa Clara, CA, USA, 10–12 July 2024; pp. 117–134.
22. Arm Developer. Explore Edge AI on Arm. Available online: https://developer.arm.com/edge-ai (accessed on 12 August 2026).
23. Chen, Y.-H.; Krishna, T.; Emer, J.S.; Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circuits* **2017**, *52*, 127–138. [CrossRef]
24. Android Developers. Neural Networks API. Available online: https://developer.android.com/ndk/guides/neuralnetworks/ (accessed on 12 August 2026).
25. Rockchip Linux. RKNN Toolkit2 Operator Support List. Available online: https://github.com/rockchip-linux/rknn-toolkit2/blob/master/doc/RKNN-Toolkit2_OP_Support-1.6.0.md (accessed on 12 August 2026).
26. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An automated end-to-end optimizing compiler for deep learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18), Carlsbad, CA, USA, 8–10 October 2018; pp. 578–594.
27. NVIDIA. TensorRT: High Performance Deep Learning Inference. Available online: https://developer.nvidia.com/tensorrt (accessed on 4 September 2026).
28. MLC Team. MLC-LLM: Universal Deployment of Large Language Models across Hardware Backends. Available online: https://llm.mlc.ai/ (accessed on 4 September 2026).
29. Ruan, C.F.; Qin, Y.; Zhou, X.; Lai, R.; Jin, H.; Dong, Y.; Hou, B.; Yu, M.-S.; Zhai, Y.; Agarwal, S.; et al. WebLLM: A high-performance in-browser LLM inference engine. *arXiv* **2024**, arXiv:2412.15803. [CrossRef]
30. Lai, Y.-H.; Chi, Y.; Hu, Y.; Wang, J.; Yu, C.H.; Zhou, Y.; Cong, J.; Zhang, Z. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), Seaside, CA, USA, 24–26 February 2019; pp. 242–251. [CrossRef]
31. Han, Q. Benchmark Dataset for Prompt-Shape- and Prefix-Cache-Aware CPU/NPU Routing on Edge SoCs. 2026. Available online: https://zenodo.org/records/23117825 (accessed on 3 October 2026). [CrossRef]

**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.