NPUGen: NPU as KV Generator for Stable Long-Context Mobile LLM Inference

Kun Wang, Jiani Cao, and Zhenjiang Li
Department of Computer Science, City University of Hong Kong
Hong Kong SAR, China

Abstract

As on-device large language model (LLM) applications like mobile agents become increasingly popular, long-context inference has become a key challenge. The key-value (KV) cache is the foundation of efficient autoregressive decoding, but its memory footprint grows rapidly with context length, often exceeding the memory limits of mobile devices. Existing KV cache management methods, such as compression and storage offloading, often lead to performance degradation or unpredictable stall latency. We find that in the prevalent CPU/GPU-decoding mobile pipelines, the NPU remains largely idle during decoding and can recompute missing KV cache more efficiently than storage offloading. We therefore propose NPUGen, whose KV recomputation engine generates missing KV cache on the idle NPU via a prefill-like forward pass, avoiding unstable storage I/O, CPU intervention, and unnecessary data movement. NPUGen further introduces a hidden-state anchor mechanism that eliminates redundant prefix recomputation, and turns KV recomputation into a schedulable background service to cope with resource contention and guarantee foreground QoS. Experiments on smartphones with Qualcomm SoCs across multiple long-context tasks show that NPUGen significantly reduces decoding latency and fluctuations, providing fast and stable long-context inference on resource-constrained mobile devices.

CCS Concepts

• **Human-centered computing** → **Mobile computing**; • **Computing methodologies** → **Machine learning**.

Keywords

On-device LLMs, On-device KV Cache Management

MobiCom '26, Austin, TX, USA

ACM ISBN 979-8-4007-2505-0/26/10
https://doi.org/10.1145/3795866.3844480

ACM Reference Format:
Kun Wang, Jiani Cao, and Zhenjiang Li. 2026. NPUGen: NPU as KV Generator for Stable Long-Context Mobile LLM Inference. In *The 32nd Annual International Conference on Mobile Computing and Networking (MobiCom '26), October 26–30, 2026, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3795866.3844480

1 Introduction

Mobile devices are rapidly becoming the main platform for intelligent applications, ranging from on-device DNN inference [48] and on-device training [49, 50] to intelligent sensing and interaction [13, 14], and more recently, on-device LLM applications [40, 54, 59]. Smartphones can now run local assistants to summarize long conversations and retrieve evidence from local notes, while also calling tools like calendars and browsers, without sending every request to the cloud [32]. This trend of shifting to on-device computing is very attractive in terms of privacy protection, response delay, and usability [27]. However, this also brings a completely new system level challenge, which is that long context inference must maintain fast response speeds under strict mobile resource limits. In actual agents workloads, prompts are no longer short and single. A single request usually combines multiple rounds of historical conversations, retrieved paragraphs, and tool outputs [43]. Therefore, the quality perceived by the user depends not only on the accuracy of the model inference, but also on whether early contexts can be recovered quickly and predictably during the decoding process.

The core system bottleneck here is the key value cache (KV cache) [36, 57]. KV cache is very important for efficient autoregressive decoding, because it avoids recomputing the complete prefix in each generation step [21, 42]. However, the size of the KV cache grows linearly with the increase of context length. As conversation and retrieval contexts keep accumulating, the KV cache will eventually exceed the fast physical memory capacity on mobile devices [17, 40, 47]. At this point, the runtime system must decide how to handle the large KV caches. Existing systems usually take one of two paths. The first path is to unrecoverably reduce the KV cache, such as using cache compression [37, 38, 51] or aggressive page eviction [12, 33, 52, 55]. The second path maintains recoverable offloading the KV cache to slower storage and

recovering it when needed [25, 26, 41, 44]. Many production systems prefer the second path, because it preserves extremely important semantic fidelity.

However, selecting recoverable offloading methods on mobile devices usually leads to noticeable latency and generation stalls. We find that the root cause of this problem lies in the demand characteristics of certain long text scenarios. For example, agents usually need to read discrete and fragmented segments, which easily causes a serious read amplification phenomenon. Read amplification means that when the attention mechanism only needs to get a tiny part of the KV data that in slow-tier storage, the underlying storage system is forced to read complete large data blocks, as shown in Figure 3. This extremely wasted data movement leads to obvious latency spike during token generation, and these latency spike make the user experience drop sharply. In addition, the frequently data swapping also causes a thermal throttling effect (*e.g.*, DVFS [24, 29]), which further leads to performance degradation.

These observations motivate us to look at the problem from a different perspective, which is that *LLM inference service mechanisms on mobile devices should absolutely not simply reuse server solutions*. The idea of offloading comes from servers using cheap massive storage to save extremely expensive computing resources. The hardware and software ecosystem of mobile devices is completely the opposite. In LLM inference services, the memory budget of mobile devices is extremely tight, but they have idle computing resources that are not fully used. For example, in the prevalent CPU/GPU-decoding pipelines targeted by this work [22, 53], the CPU or GPU is fully loaded during decoding while the NPU is almost completely idle. This is because the decoding phase is not a computation-bounded task, and the NPU architecture design is not suitable for handling the work of this phase. Notably, several mainstream efforts also exploit the NPU during decoding, *e.g.*, full-NPU decoding pipelines [8] or (self-)speculative decoding; NPUGen considers the problem from the same perspective, *i.e.*, harvesting the underutilized NPU compute during decoding, but explores a different design direction. We find that the idle NPU can be effectively used to recompute KV, because recovering the missing KV cache is essentially a recompuation of known tokens, elevating recomputation from an occasional server-side fallback (*e.g.*, upon preemption [25]) to the primary KV recovery path. This recompuation workload is structurally exactly the same as the prefill phase, and is naturally suitable for efficient execution on the NPU.

Based on these insights, we propose **NPUGen**, leveraging the NPU to recompute missing KVs during decoding. Turning this idea into practice, however, is by no means easy. The **first challenge** is that naive prefill-style recomputation drags the CPU back into the critical path: it produces many intermediate tensors irrelevant to KV recovery and requires frequent CPU intervention to splice and convert data formats, offsetting the benefits of NPU computing. We therefore design a dedicated KV recomputation engine that keeps only pure KV generation operations and, through zero-copy direct memory materialization, lets the NPU write generated KV states directly into the final cache blocks, eliminating the CPU data-movement bottleneck.

The **second challenge** is redundant computation. Due to the autoregressive nature of LLMs, the KV state at position t depends on all preceding tokens, so recomputing a cold segment forces replaying the entire prefix even if only a few tokens are ultimately needed. We break this prefix dependency with a hidden-state anchor mechanism: since KV vectors are purely linear functions of the pre-attention hidden states, NPUGen proactively captures these hidden states into a bounded anchor buffer during the initial forward pass, and upon a cold miss regenerates the target segment via just two matrix multiplications and RoPE in a single parallel NPU pass, with no Attention, FFN, or prefix KV required.

The **third challenge** is resource contention. To hide miss latency, recomputation must run proactively in the background, yet uncoordinated background jobs compete with the foreground decoder for shared memory bandwidth and for a fixed memory budget partitioned among the hot KV window, the anchor buffer, and the recomputation output. We therefore build a QoS-aware scheduler: urgent jobs on the critical decoding path preempt proactive jobs at safe scheduling points, and the scheduler continuously monitors foreground pressure, throttling recomputation and retaining more resident KV when pressure rises, while evicting and recomputing more aggressively when resources are abundant.

We implement the prototype system of NPUGen on Android smartphones with Qualcomm SoCs, and conduct a comprehensive evaluation. NPUGen outperforms baselines by 1.86× – 2.57× in terms of throughput. It reduces P99 per-token latency over 4× than baselines, and saving power consumption by 23% – 41% compared to baselines, while keeping accuracy and device temperature are stable throughout continuous long-context tasks. In summary, the main contributions of this paper are as follows.

- First, we deeply reveal that KV cache management based on storage swapping on mobile devices will cause serious read amplification effects, and point out that using heterogeneous computing resources for recompuation is an effective way to break this bottleneck.
- Second, we propose the NPUGen system, including a zero-copy KV recomputation engine, a hidden-state anchor mechanism that eliminates prefix dependency,

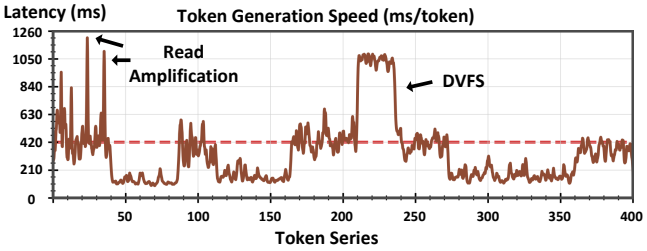

Figure 1: Token generation speed in modern long-context LLM applications with offloading method. Red dot line means the average token generation speed.

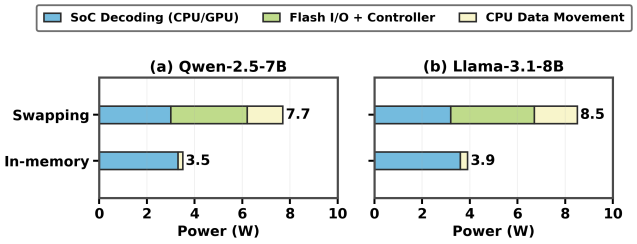

Figure 2: Power consumption breakdown on smartphone SoCs under in-memory decoding and swapping-based decoding.

and a QoS-aware dynamic scheduler, achieving a paradigm shift from storage-driven to computation-driven KV recovery.

- Third, we verify on real mobile hardware that the system can provide a stable long context inference experience under strict resource limits.

2 Background & Motivation

2.1 Primer of LLM Inference and KV Cache

LLM Inference. Currently, LLM inference follows an autoregressive process, sequentially generating one token at a time. The inference process can be divided into two distinct phases, prefilling and decoding. In the prefilling phase, the model takes all input tokens, computes their representation, and predicts the first output token. In the decoding phase, the model takes the previously generated tokens as input to predict the next token. Both phases incur heavy computational costs and involve significant memory burden, especially in the memory-constrained environment of mobile devices [17, 40, 47].

KV Cache. During decoding phase, since each new token generation rely on attention with all preceding tokens, which requires all KV vectors of preceding tokens. Current open-source LLMs models [5, 9] and commercial LLM company [1, 3, 4] usually caching preceding KVs as KV cache to avoid recomputation of KVs during decoding steps. However, as the context length increases, the storage required for KV cache also grows linearly, eventually consuming large amount of memory, even larger than the model size. Given the limited memory on mobile devices, optimizing KV cache management is critical for enabling long-context applications.

2.2 Existing KV Cache Optimizations

Current KV cache optimizations on mobile devices primary fall into two categories:

- **Unrecoverable KV Cache Reduction.** This type of methods including KV compression [37, 38, 51], aggressive KV eviction [12, 33, 52, 55], or alternative attention mechanisms [39, 45, 56]. These methods can save memory but may introduce irrecoverable quality loss, which is risky for tool-augmented and evidence-driven applications.
- **Recoverable KV Cache Offloading.** This type of method evicts unimportant KVs to a slower-tier storage and fetches it back when needed, preserving model performance. This type of approaches [25] retain frequently accessed KV as Hot KV in memory and evict the rest as Cold KV into slower-tier storage. Recent methods [26, 41, 44] further optimize this by using intelligent prefetching techniques to reduce the offloading overhead.

In mobile environments, to ensure the the accuracy of long-context tasks (*i.e.*, Mobile Agents), the recoverable solution is usually preferred.

2.3 Inefficiencies of Offloading in Modern Long-Context Applications

Power Consumption. KV cache offloading to flash incurs substantial additional power consumption compared to keeping all KVs in memory. We conduct the measurements on real mobile device (Redmi K70 Pro with Snapdragon 8Gen3). The results show that the swapping path introduces visible extra power on Flash I/O and CPU data movement, while in-memory decoding is dominated by SoC decoding compute. Specifically, swapping increases total decoding power from 3.5W (Qwen-2.5-7B) and 3.9W (Llama-3.1-8B) to 7.7–8.5 W (around 2.2 × higher), as shown in Figure 2. This high power consumption can quickly push the SoC toward thermal throttling (*i.e.*, DVFS [24, 29]), further slowing token generation speed on mobile devices, as shown in mid part of the Figure 1 .

Token Generation Speed. Besides large power consumption and thus triggered DVFS, we also discover huge and unstable latency overhead when using offloading methods in modern long context applications, as shown in beginning part of Figure 1. Specifically, this is caused by frequently triggering the read amplification. As shown in Figure 3, read amplification means a standard KV block (*i.e.*, the per-head

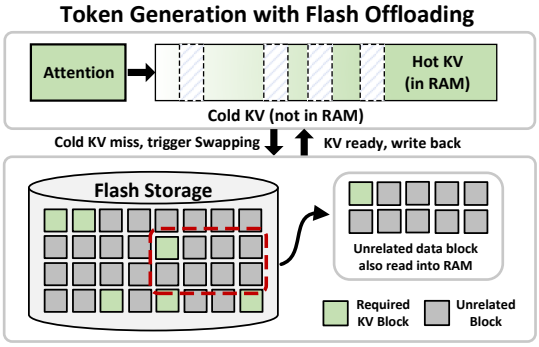

Figure 3: Detailed workflow of modern long-context LLM applications with offloading method.

K/V of one token, 2 × 128 FP16 values; head granularity is the retrieval and layout unit under head-wise selective policies [26]) may only take up 512 bytes, but the minimum page size managed by the flash controller is 16 KB. Therefore, when cache miss occurs and cold KV block needs to be read back from flash storage, the controller must read the entire large physical page including a massive amount of irrelevant data blocks. This problem is especially obvious in long context applications (*e.g.*, agents) because they have highly scattered history KV Cache requirements. These requirements include highly discrete fragments like system prompts, chat history of different rounds, and retrieved information. This highly discrete KV block access requirement frequently triggers read amplification and leads to huge and unstable latency overhead.

We also measured read amplification factor in different context length. As shown in Table 1, as total context grows, required KV blocks become more scattered in flash storage, which rapidly increases amplification factor and reduces useful read operation ratio.

This read amplification is also insensitive to the write-time layout: the fragments needed by future rounds are unknown when KV is written, so even write-side optimizations such as buffered flushes (aggregating KV in RAM and flushing large sequential writes) cannot make the discrete reads contiguous, though they remain useful when a storage fallback is kept.

Overall, these results highlight that simply utilizing server-side offloading solutions to mobile is fundamentally inefficient. These inefficiencies motivate us to find a different approach in mobile long-context inference.

2.4 Utilizing NPU for Stable Long-Context Decoding on Mobile Devices

Architecture Difference. Unlike cloud servers, which maintain massive KV Cache in GPU memory (*i.e.*, HBM) and can offload to high speed CPU memory, mobile devices rely on a resource-constrained unified memory architecture (*i.e.*, processors share same memory). They can only offload KV Cache to slower flash storage, and suffer from significantly lower bandwidth.

Context Setting	Amplification Factor	Useful Ratio
Ideal Case	1×	100%
4K context	4×	25.0%
8K context	12×	8.3%
16K context	22×	4.5%

Table 1: Read amplification of swapping-based method on smartphones under fixed KV demand.

However, as we illustrated before, applying flash-based offloading methods on mobile will introduce thermal throttling and read amplification thus cause unstable and huge I/O latency in long-context applications, fatally degrading the user experience. This necessitates a stable approach that replaces unpredictable storage I/O in mobile devices.

Observations. We observe a unique opportunity arising from mobile-side specific hardware characteristics: the idle NPU during decoding phase. In typical mobile LLM inference workflow, the NPU is commonly used for prefilling phase, but the decoding phase is often delegated to the GPU or CPU. This is because decoding is an I/O-bounded and serial process that cannot effectively saturate the high throughput design of NPUs.

Consequently, during long decoding phase, the mobile NPU often falls into an idle state. Existing works, such as [53], have demonstrated that mobile NPUs can achieve prefill throughputs exceeding 1000 tokens/s, showcasing NPU potential for rapidly processing large token chunks, which provides a crucial opportunity. We can leverage this immense but wasted computational resource to quickly recompute a KV segment in the background, rather than suffering from slow and unpredictable I/O access to flash. This opportunity has also been recognized by prior efforts, which mainly utilize the NPU to accelerate token generation, *e.g.*, speculative decoding or full-NPU decoding pipelines [8]. NPUGen instead exploits the idle NPU for KV cache management, exploring a different design direction over the same available resource: the two target different bottlenecks and are preferable in different operating regimes (see results in § 4.2.5).

Key Idea. Based on these observations, we propose the concept of **NPU as KV Generator (NPUGen)**, *transforming the role of the mobile NPU from a one-time prefiller to an opportunistic KV generator during the long decoding phase.* KV recomputation is a playback of known history tokens, which is inherently parallel and batchable, similar to prefill. This parallel characteristic makes KV recomputation perfectly suited for mobile NPUs. By leveraging effective NPUs, we can replace unpredictable and bandwidth-limited storage I/O

with high-throughput compute, thereby stabilizing the KV recovery during long-context decoding on mobile devices.

Overall, our idea can be summarized as follows: Server-side prioritizes using cheap storage to trade for expensive compute. Conversely, the severely constrained memory and unstable flash access on mobile devices motivate that we should instead leverage the idle compute resource (*i.e.*, NPU) to trade for constrained storage. In the following sections, we will detail the design of our idea.

3 System Design

3.1 System Overview

This section presents the design of NPUGen. Figure 4 illustrates the system overview. NPUGen is designed for long context inference tasks on mobile devices equipped with NPUs, including complex applications such as multi-round conversations and Agents. Our primary goal is to provide stable LLM inference on mobile devices.

As shown in the upper part of the Figure 4, in actual long context applications, the complete prompt word is built by combining various structured fragments, like system instructions, conversation rounds, and retrieved evidence. This concatenated long prompt word inevitably involves frequent recomputation of old KV states. Just like the common practice adopted by previous work [25, 30], NPUGen also maintain an active hot KV window in physical memory, to ensure smooth decoding of the frequently accessed context. Cold KVs outside the window are considered released, and do not exist in physical memory. Attention at each step covers the hot window plus the cold segments selected by importance policies reused from prior offloading work [35]. When the decoder needs to access these old contexts again, NPUGen will not read the disk, but provide the missing KV data on demand through recomputation tasks on the NPU.

As shown in the lower part of the Figure 4, the actual design of NPUGen consists of three interlocking components:

- **KV Recomputation Engine (§ 3.2).** The recomputation engine is responsible for efficient on-demand KV generation for cold misses, replacing the traditional storage read path.
- **Hidden State Anchor (§ 3.3).** The hidden state anchor mechanism is responsible for limiting recomputation range and reducing redundant recomputation.
- **QoS-aware Recomputation Scheduler (§ 3.4).** The scheduler is responsible for prioritizing urgent recomputation, adjusting proactive background recomputation, and preserving stable foreground latency under shared resource constraints.

These three components work closely together to intercept all access miss events for cold KVs at the bottom layer, and

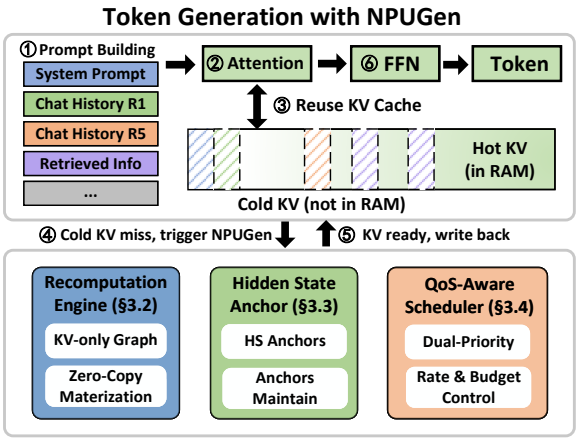

Figure 4: System Overview of NPUGen.

stably bridge the recomputation results back to main memory. Through this deep system level encapsulation design, NPUGen gives the upper inference engine an imperceptible illusion, that the engine will mistakenly believe that all KV states always fully reside in physical memory. In the following content, we will explain the detailed design.

3.2 KV Recomputation Engine

This section builds the KV recomputation engine in NPUGen. When the decoder needs cold KV, the engine regenerates it on the NPU instead of fetching from storage, focusing on reducing CPU overhead and data movement during recomputation.

3.2.1 *Why Naive Prefill-style Recomputation Is Inefficient.*

A KV recomputation job takes a segment of known tokens with their positions and runs a prefill-like forward pass to regenerate the corresponding KV vectors. A naive implementation of this path is inefficient for two reasons. First, it pulls the CPU back into the critical path: for every job, the CPU, which is simultaneously running the inference runtime and decoder [53], must launch and synchronize NPU graphs, prepare inputs (token IDs, positions, masks, and padding), and post-process outputs. Second, there is a fundamental memory-layout mismatch: standard prefill emits dense, contiguous outputs (including logits and other tensors irrelevant to KV recovery), while long-history KV caches are managed as discrete blocks, so the CPU must splice temporary tensors into KV blocks layer by layer, becoming a data mover that competes with the decoder for shared memory bandwidth and causes latency spikes.

Therefore, the design principles in this section can be concluded as: (1) Compute only what KV restoration needs, (2) keep CPU off the data movement path, and (3) keep NPU execution stable through fixed-shape graph reuse.

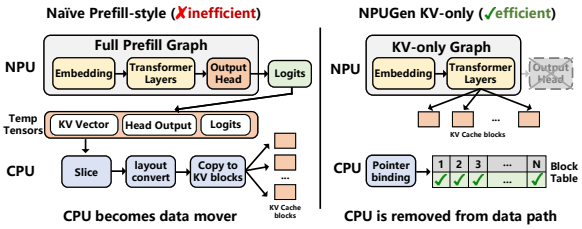

Figure 5: KV recomputation engine: a pure KV graph with zero-copy output rebinding materializes KV directly into target cache blocks.

3.2.2 KV-only Recomputation with Direct KV Materialization. Following the principles above, NPUGen promotes the KV cache as the priority output target of recomputation, which specifically includes the following three core mechanisms, as shown in Figure 5.

- **Pure KV computation graph.** This graph runs only the embedding and Transformer layers and terminates right after generating KV vectors, bypassing the output head and logits processing together with their tensor management overhead.
- **Zero copy direct memory materialization.** The runtime reserves the target KV blocks in advance, computes the precise pointer positions inside them, and binds these pointers as the output buffers of the pure KV graph, so the NPU writes generated KV directly into the final blocks without any intermediate buffer or CPU-side copy.
- **Block table and ready state bit.** The system uses a lightweight block table to map logical spans to physical blocks, and each block is equipped with a ready state bit. When the NPU task is completed, the runtime marks the written blocks as ready, and the decoder subsequently utilizes only these ready blocks.

To keep NPU execution stable, arbitrary spans are further split into fixed-shape segments ({128, 256, 512} tokens) and served by a small number of precompiled graphs, which also simplifies runtime scheduling and resource management.

On the QNN runtime, NPUGen maintains two pre-compiled graph families: the token-to-KV *fallback* graph executes the Transformer stack while omitting the LM head (obtained by truncating the model graph right after the per-layer K/V projections), whereas the anchor-to-KV graph (§ 3.3) consumes saved per-layer hidden states and executes only K/V projections and RoPE. Both are compiled once at initialization and reused across jobs. KV blocks are allocated as shared-memory QNN handles (over ION/DMA-BUF) accessible to both CPU and NPU; dispatching a job merely rebinds the graph's output tensors to offsets inside the reserved blocks, a 0.7 ms pointer update without recompilation. The ready flag is an atomic bit set in the completion callback and checked by the decoder (0.3 μs per token); accessing a non-ready block escalates its job to the urgent queue (§ 3.4).

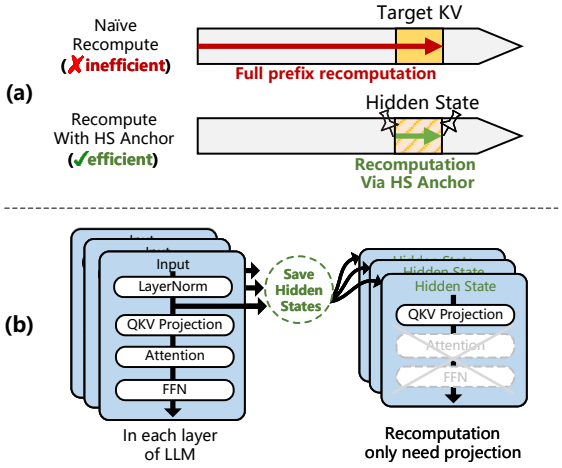

Figure 6: Hidden-state anchors. (a) Anchors confine recomputation to the target segment instead of the full prefix. (b) KV regeneration needs only a single projection pass, skipping Attention and FFN.

3.3 Hidden-State Anchors for Efficient Recomputation

While the recomputation engine efficiently regenerates cold KV, long-context applications like mobile agents often access many discrete cold segments (*e.g.*, chat history from different rounds), motivating the second component that minimizes redundant recomputation.

3.3.1 Redundant Recomputation. Cold KV accesses are sparse and discontinuous: a single request may need fragments from the conversation beginning and fact paragraphs in the middle, separated by thousands of unrelated tokens. The fundamental difficulty is strict prefix dependency: computing the KV state at position t requires all preceding tokens, so recomputing a cold segment forces replaying the entire prefix even if only a few dozen tokens are needed (Figure 6(a)), multiplying restoration latency and stalling the decoder.

3.3.2 Prefix-Free KV Regeneration via Hidden States. To address the strict prefix dependency problem, NPUGen introduces a *hidden-state anchor* mechanism, as shown in Figure 6. The key insight is that KV vectors are linear projection followed by a position-dependent rotation. In a standard Transformer decoder layer L:

$$K_i^L = \mathrm{RoPE}(W_k^L \cdot H_i^L), \quad V_i^L = W_v^L \cdot H_i^L,$$

where H_i^L is the residual stream after LayerNorm, immediately before the QKV projection. Because H_i^L already encodes

the full prefix context through all preceding layers, this projection is independent of any earlier KV cache. Given H_i^L, we can regenerate K_i^L and V_i^L with two matrix multiplications and RoPE, with no Attention or FFN and no prefix KV required, as shown in Figure 6(b). NPUGen therefore stores H_i^L as a compact anchor and uses it as a direct springboard for KV recomputation via three mechanisms.

- **4-bit quantized hidden-state storage.** NPUGen applies per-token group-wise 4-bit quantization to H_i^L before persisting it, reducing the anchor footprint to substantially smaller than storing full KV. The restoration path introduces no additional quantization stages or autoregressive error accumulation; the restored KV subsequently participates in attention, and its end-to-end impact is measured empirically and stays negligible, consistent with prior work [19] on per-token group-wise INT4 quantization. On Qualcomm NPUs, dequantization is fused directly into the GEMM kernel via QNN's native INT4×FP16 matmul, adding negligible latency.
- **Proactive capture during forward pass.** Hidden states are transient: they exist only during the forward pass and are released immediately after KV projection, making retroactive recovery at eviction time impossible. NPUGen therefore inserts lightweight snapshot hooks, implemented as zero-copy tensor views in the QNN graph, that capture and immediately quantize H_i^L for every layer and token during the initial prefill or decoding pass. The quantized anchors are written into a dedicated *anchor buffer* of size $\mathcal{B}_{anc}$ (defined in Section 3.4).
- **Parallel NPU execution.** The hidden-to-KV projection is independent across all tokens, heads, and layers, yielding a pure batch-GEMM workload that saturates the NPU in a single pass with no autoregressive loop and no CPU involvement.

By storing 4-bit hidden-state anchors instead of full KV blocks, NPUGen confines recomputation strictly to the target segment and substantially reduces prefix-dependent recomputation. Under grouped-query attention (GQA), an INT4 anchor takes $0.5 \times d_{model}$ bytes per token per layer versus $4 \times n_{kv} \times d_{head}$ bytes for the FP16 KV: it stays smaller whenever $d_{model} < 8 \cdot n_{kv} \cdot d_{head}$ (0.52× for Llama-3.1-8B and 0.90× for Qwen-2.5-7B), and the ratio inverts only under extreme MQA configurations. In any case, the anchor is designed for *prefix-freedom* rather than compression, and its footprint is bounded by the fixed-budget pool $\mathcal{B}_{anc}$: a larger hidden dimension only reduces how much history the budget covers, never the memory cost.

3.3.3 Anchor Buffer Management. The anchor buffer has a fixed memory budget. The system prompt and initial tokens (*i.e.*, attention sinks [52]) are pinned in the hot KV window and never evicted, so the eviction policy governs only the anchors of non-pinned history. When the buffer reaches capacity, NPUGen evicts anchors in order of increasing sequence position (earliest first), a cost-driven policy: an evicted shallow-position anchor only falls back to a cheap short-prefix recomputation, while deep-position anchors, whose fallback would be prohibitively expensive, are naturally the last to be evicted, so eviction always sacrifices the cheapest fallbacks first and the worst-case fallback is determined by the position of the earliest retained anchor. We choose this position-based policy over frequency-based (LFU) or attention-score-based alternatives: the latter track reuse frequency better, but require per-access bookkeeping on the critical path and may evict deep-position anchors, re-introducing expensive deep-prefix recomputation stalls.

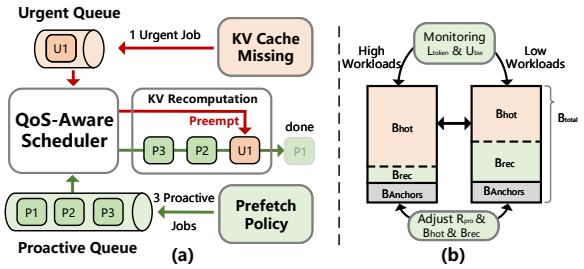

Figure 7: QoS-aware scheduler. (a) Urgent jobs preempt proactive jobs in a two-level queue. (b) R_{pro}, $\mathcal{B}_{hot}$, and $\mathcal{B}_{rec}$ are adapted from monitored L_{token} and U_{bw}.

3.4 QoS-aware Recomputation Scheduler

With the KV recomputation engine and anchor mechanisms in place, NPUGen can recover missing KV efficiently. However, if recomputation starts only after a cache miss, visible stalls still remain. To hide this latency, NPUGen uses proactive recomputation in the background (e.g., like prefetch). This creates a new systems problem: proactive recomputation competes with the foreground path for shared resources. The third component therefore turns KV recomputation into a QoS-aware proactive recomputation service.

3.4.1 Recomputation Pressure and Bandwidth Contention. NPUGen prepares likely-needed fragments before misses occur via proactive recomputation. The prediction itself reuses established prefetch strategies from prior offloading methods [35]; the difference is that predicted fragments are prepared through NPU recomputation rather than storage I/O. The resulting bottleneck is contention on shared resources, especially memory bandwidth: unthrottled proactive jobs make foreground token generation jitter, so the scheduler must control their timing and rate.

3.4.2 Urgent and Proactive Recomputation Jobs. To resolve this contention, we build a two-level preemptive scheduling framework in NPUGen, as shown in Figure 7(a). The runtime

divides all recomputation tasks into two categories: urgent recomputation jobs and proactive recomputation jobs.

Proactive recomputation is speculative and best-effort: a fragment may still become urgent if it was not predicted, admitted, or finished before the access reaches the foreground decoding path.

- **Urgent Jobs.** Urgent jobs correspond to cold KV misses that already lie on the current decoding critical path. They are inserted into the highest-priority queue and executed immediately.
- **Proactive Jobs.** Proactive jobs are opportunistic jobs triggered by proactive recomputation. They warm up fragments that are predicted to be needed soon, so that future accesses can be served without stalling the foreground path.

Crucially, NPUGen treats KV recomputation as a schedulable service rather than a monolithic black-box execution. Urgent jobs are always scheduled ahead of proactive jobs and may preempt or pause proactive execution at safe scheduling points (e.g., segment boundaries). This design bounds the interference of background recomputation on the foreground decoding path. In our implementation, preemption is realized at segment granularity rather than arbitrary instruction boundaries. This is sufficient in practice because recomputation spans are already partitioned into fixed-shape segments for graph reuse. Since commercial NPU schedulers are proprietary black boxes, NPUGen never preempts an in-flight graph; it simply stops enqueuing the next proactive segment and inserts the urgent job at the queue head, bounding the urgent-job wait by one in-flight segment (§ 4.5). This priority order extends naturally to deployments where the NPU also serves foreground decoding (*e.g.*, (self-)speculative decoding): such foreground consumers hold the highest priority, and recomputation jobs are admitted only behind them as a background service. In addition, the runtime enforces admission control on the proactive queue and limits its submission rate, so the memory traffic generated by proactive recomputation always stays within a safe range.

3.4.3 Rate Adaptation under Fixed Budgets. To keep latency hiding from degrading the foreground Quality of Service (QoS), NPUGen adapts its scheduling to real-time system pressure, quantified by two continuously monitored metrics: the average token generation latency (L_{token}) and the memory bandwidth utilization (U_{bw}, read from the SoC DDR bandwidth counters exposed via Snapdragon Profiler interfaces). This feedback is agnostic to the contention *source*: pressure from UI rendering or co-running services manifests in the same two signals and triggers the same throttling response (§ 4.4.6).

This scheduling operates under a strict unified memory budget: reusing the memory management of existing works [25], the total KV memory $\mathcal{B}_{total}$ is partitioned into the resident hot KV budget ($\mathcal{B}_{hot}$), the anchor budget ($\mathcal{B}_{anc}$), and a recomputation buffer ($\mathcal{B}_{rec}$) for newly generated KV, with $\mathcal{B}_{total} = \mathcal{B}_{hot} + \mathcal{B}_{anc} + \mathcal{B}_{rec}$.

Since $\mathcal{B}_{total}$ is fixed, the trade-off is zero-sum: the scheduler jointly adjusts the proactive recomputation rate R_{pro} (via admission control on the background queue) and the eviction threshold of $\mathcal{B}_{hot}$. When foreground pressure is high (*e.g.*, L_{token} exceeds the QoS target, U_{bw} saturates, or the device heats up), it throttles R_{pro} and keeps more historical KV resident (expanding $\mathcal{B}_{hot}$, shrinking $\mathcal{B}_{rec}$), yielding bandwidth to the foreground and minimizing new recomputation triggers; when resources are abundant, it relaxes admission control and evicts $\mathcal{B}_{hot}$ more aggressively, providing the space ($\mathcal{B}_{rec}$) for the NPU to materialize predicted future KV.

4 Evaluation

We implement a prototype of NPUGen with a CPU decoding pipeline and an NPU backend for KV recomputation, using Qualcomm's QNN runtime to execute the recomputation path; the design itself is backend-agnostic. It requires only the ability to run a prefill-like forward pass and materialize per-layer K/V states into pre-allocated buffers under CPU-shared memory; mainstream mobile NPU stacks such as MediaTek NeuroPilot provide the same capabilities, and backends without native low-bit GEMM can dequantize anchors in a separate lightweight pass.

4.1 Experimental Setup

4.1.1 Testbed. We run experiments on RedMi K70 Pro and Xiaomi 15, equipped with Qualcomm NPUs including Snapdragon 8Gen3 and Snapdragon 8Gen4 (also called Snapdragon 8 Elite). QNN is used as the NPU backend. Unless otherwise noted, we keep the screen on, fix the performance mode, and report average results over multiple trials to capture variability. Background apps are killed. All experiments use a batch size of one with no concurrent requests, reflecting single-user mobile serving. Power is sampled from the battery fuel gauge (Android `power_supply` sysfs) during active decoding, with the screen-on idle baseline subtracted and averaged over 5 runs; per-component breakdowns come from Snapdragon Profiler power rails.

4.1.2 Tasks and Models. We evaluate NPUGen on long-context tasks including the LongBench-V2 [11] and RULER suite [20]. We use decoder-only LLMs representative of on device deployments including Llama-3.1-8B [5] and Qwen-2.5-7B [46] with context lengths from 4K to 32K tokens.

4.1.3 Baselines. We compare NPUGen against the following baselines. All methods run the same W4A16-quantized

model weights with identical tokenizer and greedy decoding parameters; only the KV cache handling differs.

- **In memory KV.** This baseline represents the upper bound which keeps all KV resident when memory is enough without swapping or recomputation.
- **vLLM [25].** This baseline is a standard recoverable eviction to flash with on demand restore utilizing block based swap similar to vLLM paging adapted for mobile devices.
- **InfiniGen [26].** This baseline manages KV states across GPU and CPU memory with a fine grained importance policy. It estimates which KV entries are important at the level of attention heads and token positions, and retains or restores them accordingly.
- **FlexGen [41].** This baseline is designed around aggressive KV offloading, where the cache can be placed on secondary storage. During decoding, it streams the KV states of each layer back into memory before executing the full attention computation.
- **ShadowKV [44].** This baseline uses an asymmetric KV design that keeps a compact low rank representation of K on the GPU while placing V in CPU memory. At generation time, it fetches only the V segments predicted to be important and reconstructs the K states on the fly.
- **NPU-only (GENIE [8]).** The vendor-optimized full-NPU pipeline for Snapdragon, running prefill and decoding entirely on the NPU, with and without (self-)speculative decoding (SSD).

These baselines were originally designed for servers. We modified their source code to make them compatible with the unified memory architecture on mobile devices and offloading KV to the flash storage for a fair comparison.

4.1.4 Evaluation Metrics. Due to the compared baselines only change KV cache management rather than model weights or decoding rules, we therefore focus our quantitative evaluation on metrics that reflect user QoS.

- **Throughput.** It measures the average number of tokens generated per second during the decoding phase.
- **Inference Stability.** It is measured by the P99 per token latency to capture sudden latency spikes and user visible stall events during text generation.
- **Power Consumption.** It measures the average device power draw during active decoding to evaluate the energy efficiency of NPUGen on battery powered smartphones.
- **Accuracy.** It measures the model's ability to maintain performance across different tasks and datasets.

4.2 Overall Performance

This section presents the end-to-end results of NPUGen on the representative edge workloads. We focus on four core metrics: Throughput (tokens/second), Inference Stability (P99 per-token latency), Power Consumption (W), and Accuracy Drop (%). The results are evaluated on the Snapdragon 8Gen3 with Llama-3.1-8B.

4.2.1 Throughput. The superiority of NPUGen is highly pronounced in generation speed. As shown in Figure 8(a), NPUGen maintains a high throughput of 10.8 tok/s, which is 157% higher than vLLM and 86% higher than InfiniGen. The traditional offloading methods suffer from severe read amplification: when the attention mechanism accesses discrete historical fragments, the flash controller is forced to read 16 KB pages for only 512-byte KV blocks, congesting the storage channel. In contrast, NPUGen replaces unpredictable random Flash I/O with highly parallel NPU recomputation, ensuring a fast and steady token decoding stream.

4.2.2 Inference Stability. We evaluate P99 per-token latency to capture user-visible stall events. As shown in Figure 8(b), swapping-based methods exhibit massive latency spikes, frequently freezing generation for over 982 ms. This is because heavy CPU involvement in block lookup and memory copying directly competes with the foreground decoder. In contrast, NPUGen keeps P99 latency securely within 216 ms, only 154% higher than the ideal in-memory case. By utilizing the KV recomputation engine and hidden state anchors, NPUGen completely removes the CPU from the data-movement path and skip the unnecessary work. The generation process remains extremely smooth even when the model processes highly discontinuous context jumps.

4.2.3 Power Consumption. Low power consumption is critical for battery-operated devices. As shown in Figure 8(c), NPUGen only consumes an average of 6.2 W during active decoding, and the additional power is consumed by the NPU. Compared with swapping baselines that incur large Flash I/O and CPU data-movement overheads (8.1–10.4 W), NPUGen maintains clearly lower power draw by replacing random storage access with stable NPU computation. These results confirm that NPUGen improves overall performance without sacrificing energy efficiency on smartphones.

4.2.4 Accuracy. As shown in Figure 8(d), InMem, vLLM, and FlexGen perform no KV compression and remain fully lossless. InfiniGen and ShadowKV store a compressed K cache (similar in ideas to our hidden-state anchors) to accelerate cold recovery, but their methods are more aggressive, incurs −13% and −6% accuracy drops respectively. NPUGen applies 4-bit group-wise quantization only to hidden states, not KV itself, will dequantized during KV recomputation which

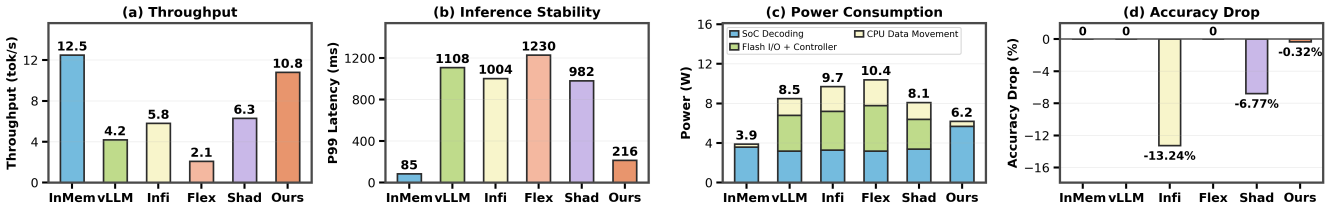

Figure 8: Overall performance on Llama-3.1-8B (8Gen3, 32K): (a) throughput, (b) P99 per-token latency, (c) power breakdown, (d) accuracy drop.

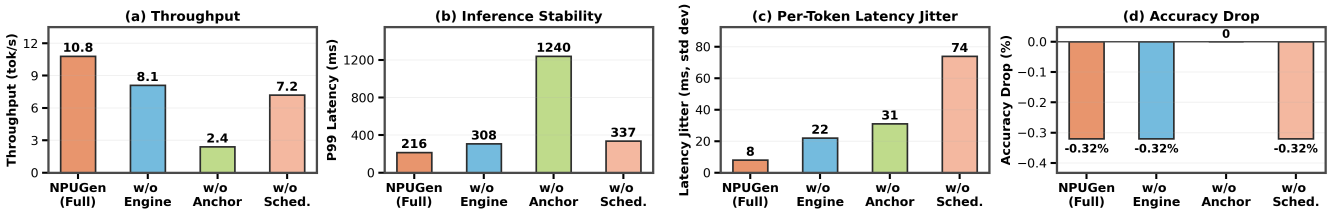

Figure 9: Ablation study: (a) throughput, (b) P99 latency, (c) latency jitter (std dev), (d) accuracy drop.

introduces a negligible −0.32% accuracy drop on LongBench-V2 while greatly accelerating recomputation by eliminating prefix dependency. Across eight LongBench-V2 and RULER task categories (retrieval, aggregation, multi-hop, dialogue), unquantized FP16 anchors are nearly lossless (−0.03% on average), confirming the restoration path itself is accuracy-neutral, while INT4 quantization adds only −0.33% on average (−0.52% worst case on multi-hop tracing), since the restoration path introduces no additional quantization stages or autoregressive error accumulation, and the end-to-end impact of the restored KV on subsequent attention is measured empirically; FP16 anchors are also ∼3.7× larger, covering less history under the same budget.

4.2.5 Comparison with NPU-only Inference Pipelines. We compare NPUGen against Qualcomm GENIE (Llama-3.1-8B W4A16, Snapdragon 8Gen3, LongBench-V2) with and without SSD, as shown in Table 2. At 4K context, where the KV cache fits in memory, the NPU-only pipeline is the fastest and most efficient option (14.6 tok/s at 4.3 W; 22.4 tok/s with SSD), since no KV management is needed at all. At 32K, however, it faces a dilemma: *Evict* stays fast (13.8 tok/s) but loses −19.4% accuracy as discarded evidence is unrecoverable, while *Offload* preserves accuracy but collapses to 4.8 tok/s with 976 ms P99 stalls under the same read amplification as other offloading baselines. NPUGen is the only configuration sustaining high throughput (10.8 tok/s), stable latency (216 ms P99), moderate power (6.2 W), and semantic fidelity (−0.32%) simultaneously. SSD accelerates the in-memory configurations by 1.53× but only 1.10× on the offloading path, where the bottleneck shifts from generation to KV recovery, the exact problem NPUGen solves. These results delineate

Method	4K (in-mem.)		32K (beyond budget)			
	Thpt.	P99	Thpt.	P99	Power	Acc. drop
GENIE (Evict)	14.6	71	13.8	78	4.5	−19.4%
GENIE (Offload)	14.6	71	4.8	976	8.9	0%
GENIE+SSD (Evict)	22.4	83	20.1	94	5.2	−19.4%
GENIE+SSD (Offload)	22.4	83	5.3	991	9.2	0%
NPUGen	12.9	86	10.8	216	6.2	−0.32%

Table 2: Comparison with NPU-only pipelines (tok/s, ms, W).

the two regimes: NPU-only pipelines suffice for short contexts, while NPUGen pays off once the context outgrows the memory budget.

4.3 Ablation Study

To isolate and verify the contribution of each core design in NPUGen, we conduct a thorough ablation study on the following three system variants.

- **NPUGen w/o Recomputation Engine.** In this variant, the system falls back to a naive prefill-style recomputation that requires full CPU data movement and temporary tensor generation.
- **NPUGen w/o Hidden State Anchor.** In this variant, the system disables the hidden-state anchor mechanism. It recomputes any cold segment with a full prefix computation.
- **NPUGen w/o QoS Scheduler.** In this variant, the system greedily executes all background regeneration jobs without any thermal control or admission limits.

Figure 9 presents the ablation results across throughput, P99 latency, per-token latency jitter and accuracy drop.

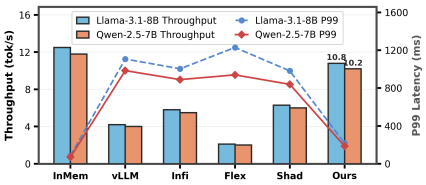

Figure 10: Impact of models: throughput (bars) and P99 latency (lines).

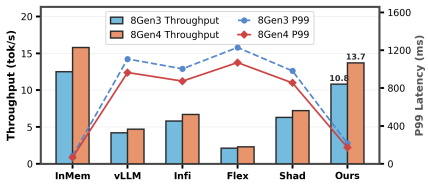

Figure 11: Impact of SoC generation: throughput (bars) and P99 latency (lines).

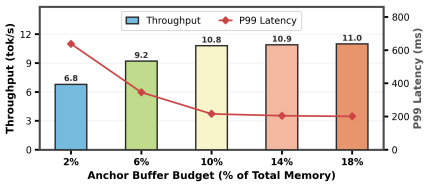

Figure 12: Impact of anchor budget: throughput (bars) and P99 latency (line).

Contribution of Recomputation Engine. The variant w/o Recomputation Engine shows a notable drop in inference stability. Its P99 per-token latency jumps from 216 ms to 308 ms (43% increase) because the per-recompute CPU bandwidth consumption rises to 12 GB/s, causing 35% extra contention with the foreground decoder. Without the pure-KV graph and zero-copy direct materialization, standard prefill operations create a fundamental memory-layout mismatch with the discrete KV cache blocks. This reveals that our dedicated KV-only engine and direct memory binding are strictly necessary for latency stability.

Contribution of Hidden State Anchor. The variant w/o Hidden State Anchor experiences the most dramatic performance collapse. Without stored hidden-state snapshots, the system must recompute from the sequence start for every cold miss, regardless of how far the target segment is from the hot window boundary. When handling 20 discrete segments, the average recompute distance explodes to 5200 tokens, increasing stall frequency to 1240ms and dropping throughput by 78% (from 10.8 to 2.4 tok/s). This powerfully validates that storing hidden state anchors successfully eliminates prefix dependency and bounds recomputation strictly to the target segment. This variant also eliminates the −0.32% accuracy drop of the full system, since no quantized hidden-state anchors are used.

Contribution of QoS Scheduler. The variant w/o QoS Scheduler exhibits a dramatic increase in per-token latency jitter, rising from 8 ms to 74 ms (9.3×), even though its median throughput (7.2 tok/s) and P99 latency (337 ms) appear only moderately degraded. This reveals the core problem: without admission control on the proactive recomputation queue, background jobs compete unpredictably with the foreground decoder for shared memory bandwidth (U_{bw}). The resulting bandwidth contention is bursty and irregular. In contrast, NPUGen's rate adaptation mechanism continuously monitors L_{token} and U_{bw}, throttling R_{pro} whenever foreground pressure rises. This confirms that the scheduler's contribution is orthogonal to recomputation efficiency, it is responsible for isolating the foreground decoding path from background interference.

4.4 Micro benchmarks

4.4.1 Impact of Different Models. To evaluate generalizability, we replace Llama-3.1-8B with Qwen-2.5-7B and re-run the baseline comparison on Snapdragon 8Gen3 under a 32K context. As shown in Figure 10, NPUGen achieves 10.2 tok/s, outperforming vLLM by 155% and InfiniGen by 85%, with P99 per-token latency at 185 ms, well below the swapping baselines (840–1085 ms). FlexGen and ShadowKV show lower P99 than on Llama-3.1-8B, since their bottleneck is KV transfer volume, which shrinks with Qwen's smaller KV footprint. In contrast, vLLM and InfiniGen show similar P99 across both models, as their worst-case stalls stem from random flash I/O seek latency, not transfer size. NPUGen and InMem also benefit from fewer layers and attention heads, reducing per-recompute and per-decode cost. Despite Qwen-2.5-7B's different attention configuration, the NPU recomputation path adapts transparently, as NPUGen operates at the architecture-independent KV-block level. These results confirm that NPUGen's performance advantage transfers across on-device LLMs. The power and accuracy advantages also transfer: 5.8 W and −0.29% on Qwen-2.5-7B, versus 7.7–9.8 W and up to −12.87% for the baselines.

4.4.2 Impact of Different SoCs. We evaluate NPUGen on Llama-3.1-8B across two SoC generations: Snapdragon 8Gen3 and the newer Snapdragon 8Gen4, which delivers approximately 40% higher CPU and NPU peak throughput. As shown in Figure 11, upgrading to 8Gen4 lifts NPUGen's throughput from 10.8 to 13.7 tok/s (a 27% gain), and reduces its P99 latency from 216 ms to 197 ms. In contrast, the baselines benefit only marginally (10–13%), since their bottleneck lies in storage I/O rather than compute. This asymmetric scaling further widens NPUGen's lead on newer hardware, demonstrating that NPUGen is well-positioned to exploit the rapidly improving NPU capabilities of future mobile SoCs. On 8Gen4, NPUGen draws 6.0 W (vs. 6.2 W on 8Gen3) while running 27% faster, improving energy per token by 24% (0.57→0.44 J); baseline power stays essentially unchanged, and accuracy is unaffected by SoC generation.

4.4.3 Impact of Anchor Buffer Budget. The anchor buffer $\mathcal{B}_{anc}$ controls how many hidden-state snapshots are retained in DRAM, directly bounding the worst-case recomputation

Context length	2K	4K	8K	16K	32K
InMem (tok/s)	13.4	13.1	12.9	12.7	12.5
Best offloading (tok/s)	—	—	—	8.1	5.8
NPUGen (tok/s)	13.2	12.9	12.6	11.8	10.8
vs. InMem	−1.5%	−1.5%	−2.3%	—	—
vs. best offloading	—	—	—	+46%	+86%

Table 3: Throughput across context lengths.

distance on a cold miss. As shown in Figure 12, increasing the budget from 2% to 18% of total memory yields substantial gains (6.8⟶11.0 tok/s, P99 638⟶201 ms). Beyond 10% the improvement saturates, with throughput and P99 changing by only 2% and 7% even when the budget nearly doubles. NPUGen therefore adopts 10% as its default allocation (10.8 tok/s, 216 ms), matching the numbers reported in the overall evaluation, and leaving the remaining memory headroom for co-running applications. Power and accuracy saturate similarly: below 10%, power *rises* (7.4 W at 2%) as evicted anchors force longer full-prefix recomputation, while accuracy improves only marginally (−0.19% at 2%); beyond 10% both saturate.

4.4.4 Break-even Context Length. We further measure NPUGen from 2K to 32K under the default memory budget, as shown in Table 3. The hot KV window is exhausted at ∼**9K tokens**, the break-even point. Below it, no recomputation is triggered and NPUGen behaves as in-memory decoding, with residual overhead within 2.3% end-to-end (§ 4.5). Beyond it, NPUGen's advantage over the best offloading baseline grows from +46% (16K) to +86% (32K).

4.4.5 Effectiveness of Proactive Recomputation. We quantify the prefetch policy under three access patterns of increasing fragmentation (Llama-3.1-8B, 8Gen3, 32K context), as shown in Table 4. The hit rate stays between 85.6% and 72.8%: context reuse, although discrete in position, concentrates on semantic boundaries (system prompts, whole rounds, retrieved chunks), the same granularity the predictor operates at. As fragmentation grows, the urgent-vs-proactive ratio shifts from 1:6.4 to 1:3.3, and the share of mispredicted proactive jobs (regenerated but never accessed) rises from ∼7% to ∼19%; these wasted jobs are admission-controlled and consume only otherwise-idle NPU cycles. Meanwhile, the preemptive queue keeps P99 within 204–241 ms: unlike storage offloading, where a prefetch miss falls back to slow flash I/O, a miss here falls back to the anchor-bounded urgent recomputation path, so P99 stays far below the ∼1000 ms of offloading baselines even at the lowest hit rate.

4.4.6 Adaptation under Dynamic Memory Fluctuations. We further evaluate NPUGen under dynamic memory fluctuations, as shown in Figure 13: a 10-minute decoding task

Access pattern	Hit rate	Urg. : Pro.	Thpt.	P99
Multi-round conv. (8–12 rounds)	85.6%	1 : 6.4	11.2	204
Multi-doc QA / RAG (12–16 seg.)	79.3%	1 : 4.4	10.7	219
Agent traces (20+ seg.)	72.8%	1 : 3.3	10.3	241

Table 4: Prefetch hit rate and urgent-vs-proactive job ratio.

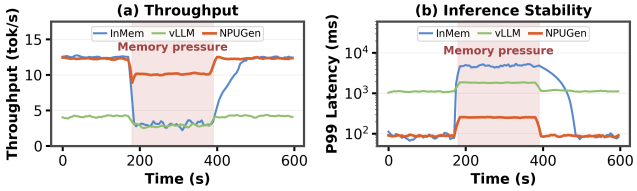

Figure 13: Adaptation under dynamic memory pressure.

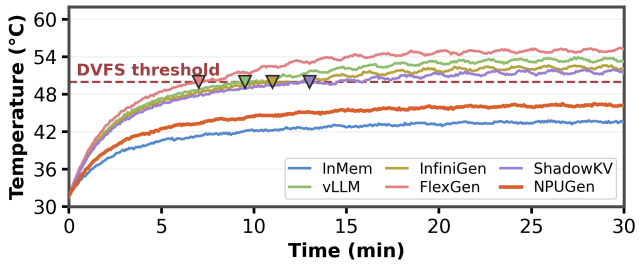

Figure 14: Device temperature traces. Triangles mark DVFS throttling onset.

(32K context) starts with all KV resident, so NPUGen effectively operates atop InMem (12.3 vs. 12.5 tok/s, a ∼2% gap from anchor capture and bookkeeping, consistent with the break-even overhead in Table 3); at t=180 s, co-running background workloads plus a controlled ballast shrink the available memory by 40% (shaded window), released at t=390 s. Under pressure, the scheduler rebalances $\mathcal{B}_{hot}/\mathcal{B}_{anc}/\mathcal{B}_{rec}$ within ∼1.6 s, stabilizing at 10.1 tok/s with P99 rising only from 90 to 255 ms, and returns to the fully resident mode within ∼2 s after release. In contrast, InMem collapses to 3.1 tok/s under OS paging (P99 ≈ 4.8 s, ∼90 s recovery) and vLLM degrades to 2.8 tok/s (P99 1.8 s).

4.5 System Overhead

This section analyzes the thermal stability and the intrinsic overhead of our design. Thermal stability is paramount for mobile deployment. We trace the device temperature over a continuous 30 minutes agent task. While traditional offloading methods and greedy recomputation baselines quickly trigger DVFS throttling due to massive IO and CPU bursts, the thermal trace of NPUGen remains perfectly stable below 47 degrees Celsius. As shown in Figure 14, all swapping baselines cross the 50 °C DVFS threshold (battery-adjacent

skin sensor; 50 °C is the vendor's skin-temperature throttle trigger) between ∼7 min (FlexGen) and ∼13 min (ShadowKV), suffering −12% to −23% post-throttle throughput loss as the big cores are capped, driven by sustained flash I/O and CPU data movement; NPUGen plateaus at 46.5 °C (InMem: 43.8 °C) without ever throttling. This demonstrates that utilizing the idle NPU cycles for KV generation successfully respects the strict thermal envelope of mobile platforms without causing overheating.

NPUGen's own overhead is small: anchor capture adds +1.1% prefill latency (zero-copy hooks with fused INT4 quantization), block-table lookup costs 0.3 μs per token, the largest 512-token segment executes in a measured ∼25 ms, which together with the measured 0.7 ms dispatch cost bounds the worst-case urgent-job wait (∼26 ms), and the scheduler thread takes 0.9% of one core. Memory-wise, the anchor buffer occupies 800 MB (10% of the budget, Figure 12), metadata 6.4 MB, and pre-compiled graph contexts 54 MB, with model weights shared rather than duplicated.

5 Related Work

On-device LLM Inference. Recent years have witnessed rapid progress in deploying LLMs directly on mobile and edge devices. Frameworks such as MLC-LLM [6] and ExecuTorch [2] have enabled cross-platform LLM inference by compiling models for heterogeneous hardware backends including CPU, GPU, and NPU. Systems like mllm [7] leverage on-device NPU to accelerate the prefill phase, achieving substantial improvements in latency and energy efficiency on Qualcomm Snapdragon platforms. Other efforts [15, 32, 54] focus on model quantization, kernel fusion, and compact architectures to fit models within tight mobile memory and power budgets. These works primarily target efficient single-phase inference under the assumption that the entire KV cache fits comfortably in fast memory.

NPUGen builds directly on this heterogeneous acceleration foundation but addresses a complementary system-level challenge, *i.e.*, providing stable long-context KV cache management. It can be seamlessly integrated into existing on-device inference engines [22, 53] without modifying their core model execution, thereby extending their capabilities to real-world long-context workloads.

KV Cache Management. The $O(L^2)$ attention complexity and $O(L)$ KV memory growth are primary bottlenecks for long-context inference, motivating a substantial body of work [10, 19, 28, 31, 34, 52, 58]. Eviction methods keep only recent or important tokens: StreamingLLM [52] maintains a fixed-size sliding window anchored on attention sinks, while SnapKV [28] and H2O [58] selectively retain heavy-hitter entries via data-driven policies. Quantization, exemplified by KIVI [34], compresses the cache to lower bit-widths (*e.g.*, 2-bit) with minimal performance loss.

Another line of research preserves model accuracy through offloading. InfiniGen [26] and ShadowKV [44] move KV entries between GPU and CPU with intelligent prefetching and low-rank compression; SpeCache [23] combines low-bit compression with speculative prefetching, and page-based managers [16] recall evicted tokens on demand. These methods assume high-bandwidth, low-latency connections such as PCIe, whereas mobile devices rely on much slower storage hierarchies (UFS, eMMC, or flash) that suffer severe bandwidth limitations and read amplification. Recomputation has likewise served only as an occasional server-side fallback (*e.g.*, vLLM's recompute-on-preemption [25]); HCache [18] restores KV from intermediate activations, but targets multi-request serving, restoring whole contexts by overlapping storage I/O with restoration compute. The inverse resource profile of mobile devices leads NPUGen to instead make recomputation the *primary* recovery path: prefix-free and segment-level via DRAM-resident INT4 anchors, free of storage I/O, and scheduled as a preemptible, rate-controlled service interleaved with foreground decoding.

Unlike these prior approaches, NPUGen avoids irrecoverable aggressive eviction (risky for evidence-driven RAG and agent tasks) and model retraining, and replaces unpredictable, power-hungry storage I/O with idle NPU computation for lower power and more predictable latency. It also remains orthogonal to existing KV management strategies, which can be combined as controlled fallbacks (*e.g.*, compression or page-based offloading) on resource-constrained devices. Indeed, NPUGen reuses their importance-selection and prefetch policies [26, 35], replacing only the storage-backed recovery path, and complements self-speculative decoding [8].

6 Conclusion

We present NPUGen, an NPU-driven KV cache management system for stable long-context LLM inference on mobile devices. Instead of unstable storage offloading, NPUGen recomputes missing KV states using the idle NPU. Evaluations on Qualcomm SoCs show it significantly reduces latency fluctuations. This shifts KV cache management from storage-centric to computation-centric, enabling practical on-device LLM applications.

Acknowledgments

We thank all the reviewers. This work is supported by the General Research Fund (GRF) grants (CityU 11202623 and CityU 11211326) and the Collaborative Research Fund (CRF) grant (C5085-25G) from Hong Kong Research Grants Council. The corresponding author is Zhenjiang Li.

References

[1] ChatGPT. https://chat.openai.com/.
[2] Executorch. https://github.com/pytorch/executorch.
[3] Gemini. https://gemini.google.com/.
[4] Grok. https://www.grok.ai/.
[5] Llama3. https://llama.meta.com/llama-3/.
[6] Mlc-llm. https://github.com/mlc-ai/mlc-llm.
[7] mllm. https://github.com/UbiquitousLearning/mllm.
[8] Qualcomm ai engine direct sdk: Generative ai inference extensions (genie). https://www.qualcomm.com/developer/software/qualcomm-ai-engine-direct-sdk.
[9] Qwen. https://qwenlm.github.io/.
[10] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proc. of EMNLP*, 2023.
[11] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, et al. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In *Proc. of ACL*, 2025.
[12] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.
[13] Jiani Cao, Jiesong Chen, Chengdong Lin, Yang Liu, Kun Wang, and Zhenjiang Li. Practical gaze tracking on any surface with your phone. *IEEE Transactions on Mobile Computing*, 2024.
[14] Jiani Cao, Kun Wang, Yang Liu, and Zhenjiang Li. Neuropath: Practically adopting motor imagery decoding through eeg signals. In *Proc. of SenSys*, 2026.
[15] Le Chen, Dahu Feng, Erhu Feng, Rong Zhao, Yingrui Wang, Yubin Xia, Haibo Chen, and Pinjie Xu. Heterollm: Accelerating large language model inference on mobile socs platform with heterogeneous ai accelerators. In *Proc. of SOSP*, 2025.
[16] Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. Arkvale: Efficient generative llm inference with recallable key-value eviction. In *Proc. of NeurIPS*, 2024.
[17] Othmane Friha, Mohamed Amine Ferrag, Burak Kantarci, Burak Cakmak, Arda Ozgun, and Nassira Ghoualmi-Zine. Llm-based edge intelligence: A comprehensive survey on architectures, applications, security and trustworthiness. *IEEE Open Journal of the Communications Society*, 2024.
[18] Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in llm serving with hcache. In *Proc. of EuroSys*, 2025.
[19] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. In *Proc. of NeurIPS*, 2024.
[20] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? In *Proc. of COLM*, 2024.
[21] Yunpeng Huang, Jingwei Xu, Junyu Lai, Zixu Jiang, Taolue Chen, Zenan Li, Yuan Yao, Xiaoxing Ma, Lijuan Yang, Hao Chen, et al. Advancing transformer architecture in long-context large language models: A comprehensive survey. *arXiv preprint arXiv:2311.12351*, 2023.
[22] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. In *Proc. of MLSys*, 2020.
[23] Shibo Jie, Yehui Tang, Kai Han, Zhi-Hong Deng, and Jing Han. Specache: Speculative key-value caching for efficient generation of llms. In *Proc. of ICML*, 2025.
[24] Seyeon Kim, Kyungmin Bin, Sangtae Ha, Kyunghan Lee, and Song Chong. ztt: Learning-based dvfs with zero thermal throttling for mobile devices. In *Proc. of MobiSys*, 2021.
[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proc. of OSDI*, 2023.
[26] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *Proc. of OSDI*, 2024.
[27] Xiang Li, Zhenyan Lu, Dongqi Cai, Xiao Ma, and Mengwei Xu. Large language models on mobile devices: Measurements, analysis, and insights. In *Proc. of EdgeFM*, 2024.
[28] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. In *Proc. of NeurIPS*, 2024.
[29] Chengdong Lin, Kun Wang, Zhenjiang Li, and Yu Pu. A workload-aware dvfs robust to concurrent tasks for mobile devices. In *Proc. of MobiCom*, 2023.
[30] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.
[31] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proc. of SIGCOMM*, pages 38–56, 2024.
[32] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases. In *Proc. of ICML*, 2024.
[33] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. In *Proc. of NeurIPS*, 2023.
[34] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. In *Proc. of ICML*, 2024.
[35] Lingkun Long, Rubing Yang, Yushi Huang, Desheng Hui, Ao Zhou, and Jianlei Yang. Sliminfer: Accelerating long-context llm inference via dynamic token pruning. In *Proc. of AAAI*, 2026.
[36] Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D Lane, and Mengwei Xu. Small language models: Survey, measurements, and insights. *arXiv preprint arXiv:2409.15790*, 2024.
[37] Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo M Ponti. Dynamic memory compression: Retrofitting llms for accelerated inference. In *Proc. of ICML*, 2024.
[38] Jianhui Pang, Fanghua Ye, Derek Wong, Xin He, Wanshun Chen, and Longyue Wang. Anchor-based large language models. In *Proc. of ACL*, 2024.
[39] Zhen Qin, Weigao Sun, Dong Li, Xuyang Shen, Weixuan Sun, and Yiran Zhong. Various lengths, constant speed: Efficient language modeling with lightning attention. In *Proc. of ICML*, 2024.
[40] Guanqiao Qu, Qiyuan Chen, Wei Wei, Zheng Lin, Xianhao Chen, and Kaibin Huang. Mobile edge intelligence for large language models: A contemporary survey. *IEEE Communications Surveys & Tutorials*, 2025.
[41] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang.

Flexgen: High-throughput generative inference of large language models with a single gpu. In *Proc. of ICML*, 2023.

[42] Luohe Shi, Hongyi Zhang, Yao Yao, Zuchao Li, and Hai Zhao. Keep the cost down: A review on methods to optimize llm's kv-cache consumption. In *Proc. of COLM*, 2024.

[43] Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. Agentic retrieval-augmented generation: A survey on agentic rag. *arXiv preprint arXiv:2501.09136*, 2025.

[44] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference. In *Proc. of ICML*, 2025.

[45] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

[46] Qwen Team. Qwen2.5: A party of foundation models, September 2024.

[47] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, et al. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness. *ACM Transactions on Intelligent Systems and Technology*, 2025.

[48] Kun Wang, Jiani Cao, Zimu Zhou, and Zhenjiang Li. Swapnet: Efficient swapping for dnn inference on edge ai devices beyond the memory budget. *IEEE Transactions on Mobile Computing*, 2024.

[49] Kun Wang, Zimu Zhou, and Zhenjiang Li. Latte: Layer algorithm-aware training time estimation for heterogeneous federated learning. In *Proc. of MobiCom*, 2024.

[50] Kun Wang, Zimu Zhou, and Zhenjiang Li. Towards accurate training time estimation for on-device heterogeneous federated learning. *IEEE Transactions on Mobile Computing*, 2025.

[51] Zheng Wang, Boxiao Jin, Zhongzhi Yu, and Minjia Zhang. Model tells you where to merge: Adaptive kv cache merging for llms on long-context tasks. *arXiv preprint arXiv:2407.08454*, 2024.

[52] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *Proc. of ICLR*, 2024.

[53] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Fast on-device llm inference with npus. In *Proc. of ASPLOS*, 2025.

[54] 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.

[55] Dongjie Yang, XiaoDong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. Pyramidinfer: Pyramid kv cache compression for high-throughput llm inference. In *Proc. of ACL*, 2024.

[56] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. In *Proc. of ICML*, 2024.

[57] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. Llm as a system service on mobile devices. *arXiv preprint arXiv:2403.11805*, 2024.

[58] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *Proc. of NeurIPS*, 2023.

[59] Yue Zheng, Yuhao Chen, Bin Qian, Xiufang Shi, Yuanchao Shu, and Jiming Chen. A review on edge large language models: Design, execution, and applications. *ACM Computing Surveys*, 57(8):1–35, 2025.