LM-Meter: Unveiling Runtime Inference Latency for On-Device Language Models

Haoxin Wang
haoxinwang@gsu.edu
Georgia State University
Atlanta, GA, USA

Xiaolong Tu
xtu1@student.gsu.edu
Georgia State University
Atlanta, GA, USA

Hongyu Ke
hke3@student.gsu.edu
Georgia State University
Atlanta, GA, USA

Huirong Chai
hchai4@student.gsu.edu
Georgia State University
Atlanta, GA, USA

Dawei Chen
dawei.chen1@toyota.com
Toyota InfoTech Labs
Mountain View, CA, USA

Kyungtae Han
kt.han@toyota.com
Toyota InfoTech Labs
Mountain View, CA, USA

Abstract

Large Language Models (LLMs) are increasingly integrated into everyday applications, but their prevalent cloud-based deployment raises growing concerns around data privacy and long-term sustainability. Running LLMs locally on mobile and edge devices (on-device LLMs) offers the promise of enhanced privacy, reliability, and reduced communication costs. However, realizing this vision remains challenging due to substantial memory and compute demands, as well as limited visibility into performance-efficiency trade-offs on resource-constrained hardware. We propose LM-Meter, the first lightweight, online latency profiler tailored for on-device LLM inference. LM-Meter captures fine-grained, real-time latency at both phase (e.g., embedding, prefill, decode, softmax, sampling) and kernel levels without auxiliary devices. We implement LM-Meter on commercial mobile platforms and demonstrate its high profiling accuracy with minimal system overhead, e.g., only 2.58% throughput reduction in prefill and 0.99% in decode under the most constrained `Powersave` governor. Leveraging LM-Meter, we conduct comprehensive empirical studies revealing phase- and kernel-level bottlenecks in on-device LLM inference, quantifying accuracy-efficiency trade-offs, and identifying systematic optimization opportunities. LM-Meter provides unprecedented visibility into the runtime behavior of LLMs on constrained platforms, laying the foundation for informed optimization and accelerating the democratization of on-device LLM systems. Code and tutorials are available at `github.com/amai-gsu/LM-Meter`.

SEC '25, Arlington, VA, USA

ACM ISBN 979-8-4007-2238-7/2025/12
https://doi.org/10.1145/3769102.3770614

CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Human-centered computing** → **Ubiquitous and mobile computing**.

Keywords

Large Language Models, On-Device AI, Edge Computing

ACM Reference Format:
Haoxin Wang, Xiaolong Tu, Hongyu Ke, Huirong Chai, Dawei Chen, and Kyungtae Han. 2025. LM-Meter: Unveiling Runtime Inference Latency for On-Device Language Models. In *The Tenth ACM/IEEE Symposium on Edge Computing (SEC '25), December 3–6, 2025, Arlington, VA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3769102.3770614

1 Introduction

Large language models (LLMs), such as GPT-series [1, 2], LLaMA [3], and DeepSeek [4, 5], have recently garnered significant attention for their impressive capabilities in natural language understanding, generation, and reasoning across a broad spectrum of tasks. By scaling to billions or even trillions of parameters, these models have achieved state-of-the-art performance in applications including machine translation, dialogue systems, and code generation [6–9]. Today, the predominant approach to deploying LLMs is cloud-based, relying on centralized inference services hosted in large-scale data centers. These services are typically powered by specialized hardware accelerators, custom software stacks, and a suite of system-level optimizations, such as tensor parallelism [10], speculative decoding [11], and memory-efficient KV cache management [12], to support emerging use cases with high-throughput, low-latency inference at scale.

Despite the benefits of deploying LLMs in cloud environments, this paradigm has pushed the computational boundaries of cloud infrastructure and exacerbates concerns over

its long-term sustainability (§2.1). Moreover, issues surrounding data privacy and user data custody have become increasingly prominent. For instance, many private enterprises and government agencies restrict the use of cloud-hosted LLM services for processing sensitive or proprietary information due to potential risks involving data leakage, regulatory noncompliance, and lack of control over data residency. These concerns underscore a growing demand for local LLM accessibility on mobile and edge devices (i.e., on-device LLMs), such as smartphones and Internet-of-Things (IoT) devices, to enable broader and more privacy-preserving integration of LLMs into users' everyday lives. On-device LLMs offer several compelling benefits, including enhanced data privacy, greater reliability under intermittent connectivity, and reduced communication costs. They also enable user-centric personalization and context-aware adaptation, unlocking new opportunities for responsive, interactive applications directly on personal consumer devices [13].

However, deploying LLMs effectively and efficiently on mobile and edge hardware remains largely under-explored. First, the parameter sizes of most state-of-the-art LLMs scale into the billions, demanding substantial memory capacity to accommodate model weights, intermediate activations, and inference computations, often exceeding the limited compute and memory resources available on mobile and edge devices. Second, most existing model architectures and software frameworks are designed and optimized for cloud or high-performance computing environments. Third, there is a limited understanding of the fundamental trade-offs between performance and efficiency in on-device LLM inference. Critical bottlenecks, such as task-dependent throughput limitations and latency in specific inference phases and kernels, remain insufficiently characterized. Existing literature provides minimal insight into opportunities for systematic optimization, adaptive runtime strategies, kernel-level improvements, or efficient scheduling under tight resource budgets. More importantly, the lack of lightweight, real-time profiling tools further exacerbates these challenges, restricting the ability to comprehensively analyze and optimize on-device LLM inference across algorithmic, kernel, and hardware levels.

To this end, we propose LM-METER, the first lightweight online runtime latency profiler for on-device LLM inference. LM-METER enables accurate, real-time latency measurement at both phase (e.g., embedding, prefill, decode, softmax, and sampling) and kernel granularity on consumer-grade mobile and edge platforms, without auxiliary devices. We envision LM-METER as a foundation for studying runtime bottlenecks and deployment constraints, addressing a key barrier to the practical democratization of on-device LLMs. Our main contributions are summarized below:

- We design and implement LM-METER, a lightweight online runtime latency profiler tailored for on-device LLM inference. Through evaluations on multiple consumer-grade mobile devices, we demonstrate that LM-METER accurately captures both phase-level and kernel-level latencies with minimal system overhead, validating its practicality for real-world deployment.
- Leveraging LM-METER, we conduct a phase-oriented empirical study to systematically analyze the Pareto frontier of performance-efficiency trade-offs across diverse benchmarking tasks, popular open-source LLM suites, varying model sizes, and mobile platforms. We further conduct a systematic evaluation to characterize how quantization shifts these trade-offs by proposing a new metric, the Harmonic Quantization score ($\mathcal{HQ}$).
- We perform a kernel-oriented empirical study to uncover fine-grained execution bottlenecks and resource utilization patterns at the kernel level.
- Based on our empirical analyses, we identify critical bottlenecks in on-device LLM inference and distill actionable insights for both system- and model-level optimizations, informing the design of more efficient, scalable, and responsive on-device LLM systems.

2 Background and Motivation

2.1 Why Democratizing On-Device LLMs

Sustainability. Prominent implementations of contemporary LLM products, such as ChatGPT [14] and Google NotebookLM [15], primarily rely on cloud-based infrastructures due to their massive model sizes (ranging from billions to trillions of parameters) and substantial computational demands. As human reliance on LLMs continues to grow, envisioning a future where it becomes deeply integrated into daily life, supporting both front-end interactions, such as personalized companionship [16–19], and back-end applications like recommendation systems, its usage could constitute $\sim$ 5% of an individual's daily time [20]. In such a scenario, operating a *single* LLM model (e.g., GPT-4) at a processing rate of 50 tokens per second to serve the global population would require the deployment of roughly 100 million H100 GPUs (each capable of 60 trillion floating-point operations per second) [21]. This immense computational scale required, excluding network overhead, would be equivalent to the infrastructure of approximately *160 Meta-scale* companies [22].

User privacy. Centralized LLM services require user data, often highly sensitive, to traverse public networks and be processed in remote data centers, creating significant attack surfaces. Privacy risks are most acute in domains that handle regulated or confidential information, including healthcare (e.g., patient health data), finance, and enterprise productivity

workflows containing proprietary IP. On-device LLMs eliminate this exposure by keeping both the model and the data local, dramatically reducing the attack surface and aligning with regulations such as HIPAA [23]. This privacy-by-design paradigm makes local deployment not merely a technical optimization but a prerequisite for user trust. Removing the dependency on continuous connectivity also improves reliability in bandwidth-constrained or offline scenarios.

In summary, transitioning LLM computations to mobile and edge devices, including smartphones, personal computers, and IoT devices, is essential for achieving improved sustainability, enhanced privacy, and robust network independence by bringing computation closer to end users.

2.2 Why a Lightweight Online Profiler Matters

Despite its promising potential, on-device LLM deployment remains largely under-explored and poses formidable research challenges. Bridging the gap between their immense computational demands and the resource constraints of mobile and edge devices requires fundamental breakthroughs. These constraints include restricted memory and compute capacity, limited energy reserves (e.g., battery life or intermittent power), and insufficient support for parallelism [24].

We conduct a pilot benchmark to validate the above concerns using one of the most popular open-source LLM suites, `Qwen` [25]. Three generations, `Qwen-1.5`, `Qwen-2`, and `Qwen-2.5`, are evaluated in their smallest 0.5B parameter configurations to ensure they can run unmodified (no quantization or fine-tuning) on contemporary mobile hardware. Table 1 presents accuracy on three benchmark tasks (ARC-Challenge, HellaSwag, GSM8K) and token-generation throughput measured on a Google Pixel 8 Pro. Interestingly, the results reveal a widening accuracy-efficiency gap. While each successive generation improves accuracy (e.g., +4.4 percentage point (pp) on HellaSwag and +24.5 pp on GSM8K from `Qwen-1.5` to `Qwen-2.5`), on-device decode throughput drops by 24%, from 22.76 tokens/s to 17.29 tokens/s. This trend underscores a prevailing research emphasis on accuracy gains, often at the expense of runtime efficiency, and directly conflicts with the constraints of on-device LLMs.

Consequently, a comprehensive and systematic understanding of on-device LLM bottlenecks is now more urgent than ever. To this end, we advocate for a lightweight, online profiler tailored to on-device LLMs. Unlike traditional offline profilers, an online profiler enables real-time collection of fine-grained performance metrics (e.g., latency, memory usage) during live model execution. These data can be immediately accessed by the application or operating system, enabling dynamic optimizations, adaptive scheduling, and

Table 1: Comparison of accuracy and on-device decode throughput of `Qwen` model generations.

Models	Accuracy ↑			Throughput (tokens/s) ↑
	ARC-Challenge	HellaSwag	GSM8K	
`Qwen-1.5-0.5B`	0.293	0.491	0.171	**22.76**
`Qwen-2-0.5B`	0.310	0.491	0.364	17.33
`Qwen-2.5-0.5B`	**0.356**	**0.522**	**0.416**	17.29

* Throughput data are collected on Google Pixel 8 Pro. Model accuracy on individual tasks is evaluated using the *lm-evaluation-harness* [32].

informed system-level decisions, key steps toward the practical democratization of LLMs at the edge. Below, we highlight two key benefits of a lightweight, online profiler:

Empirical study on on-device LLMs. A lightweight profiler is essential for enabling accurate, empirical analysis of LLM behavior on mobile and edge devices. Unlike cloud-based profiling tools that can tolerate additional system overhead, on-device studies demand minimal interference with the model's runtime. A profiler that introduces significant overhead distorts key performance signals such as latency, memory usage, and energy consumption, leading to misleading conclusions about real-world performance. By keeping overhead low and integrating seamlessly with existing runtime pipelines, a lightweight online profiler allows researchers to faithfully observe the system under typical usage conditions.

Online runtime performance prediction. LLMs generate output in an autoregressive manner, producing a token at a time during decoding. For each token, the model embeds the input, attends over all previously generated tokens stored in a growing key-value (KV) cache, processes the result through transformer layers, samples the next token, and updates the KV cache. The computational cost of this process increases approximately linearly with sequence length, while hardware-level factors on mobile and edge devices, such as dynamic frequency scaling and thermal throttling, tend to evolve gradually. These properties make throughput prediction for on-device LLM both feasible and valuable. A lightweight profiler that measures per-token latency in real time can exploit these trends to predict future throughput over short horizons. Such predictions are critical for enabling dynamic scheduling decisions and system adaptation.

In addition, recent studies have shown that kernel-level prediction mechanisms can achieve state-of-the-art accuracy in predicting both latency [26, 27] and energy consumption [28–31] for conventional deep learning models. These approaches benefit from the fact that kernels serve as fundamental scheduling units and encapsulate many framework-level optimizations. On-device LLM systems can adopt similar strategies by accurately profiling fine-grained, kernel-level execution data.

2.3 Limitations of Existing Profilers

As shown in Table 2, most existing profilers for mobile and edge platforms were not designed with LLM inference in mind, and therefore fall short of meeting the requirements outlined in Section 2.2. Specifically, offline profilers such as Android GPU Inspector (AGI) [33] and Perfetto [34] collect raw execution traces during runtime but rely on extensive post-processing, often on a host machine, to extract meaningful insights. This delayed analysis pipeline, which involves exporting logs, synchronizing metadata, and using external visualization tools, makes them unsuitable for real-time adaptation and on-device scheduling. Furthermore, these tools lack kernel-level visibility, which is essential for capturing the fine-grained behaviors of LLM execution.

Recently, several online profilers have been developed for on-device learning scenarios, including MELTing Point [35] and nnPerf [36]. However, both exhibit several limitations. MELTing Point reports only the average duration of each GPU kernel in isolation, while capturing cumulative runtime, it lacks timeline visibility and does not provide fine-grained start/end timestamps or show how kernels are sequenced and interleaved. As a result, it fails to reveal runtime behaviors such as kernel launch delays, inter-kernel gaps, or scheduling contention. More importantly, enabling MELTing Point's kernel-level tracing introduces substantial latency overhead (see Fig. 4), making it unsuitable for accurate kernel-granular analysis of on-device LLM inference. nnPerf [36], on the other hand, was primarily designed for smaller convolutional or feedforward models, such as MobileNet, EfficientNet, and ResNet. While effective for traditional DNNs, it lacks support for the unique computational characteristics of LLMs, which involves dynamic input lengths, token-level scheduling bottlenecks, and interleaved execution phases, e.g., embedding, prefill, autoregressive decoding, softmax, and token sampling.

To this end, we propose LM-Meter, a lightweight, online profiler that enables LLM phase-aware and fine-grained kernel-level latency tracing, specifically designed for on-device LLM inference. LM-Meter equips researchers and system designers with transparent and deeper visibility into runtime behavior of LLMs on resource-constrained platforms, facilitating informed optimization, adaptive scheduling, and accelerating the democratization of intelligent on-device systems.

Table 2: LM-Meter vs. existing profilers for mobile and edge hardware.

Profilers	Support LLMs	Phase awareness	Kernel-level	Timeline visibility	Low overhead	Real-time output	No-host machine
AGI [33]	✓	✓	✗	✓	✓	✗	✗
Perffeto [34]	✓	✓	✗	✓	✓	✗	✗
TFLite benchmark [37]	✗	✗	✓	✗	✓	✗	✗
nnPerf [36]	✗	✗	✓	✓	✓	✓	✓
MELT [35]	✓	✓	✓	✗	✗	✓	✓
LM-Meter (Ours)	✓	✓	✓	✓	✓	✓	✓

* Timeline visibility: whether the profiler can log and reconstruct an execution timeline with start/end timestamps for each kernel, data transfer, and GPU idle period.
† Phase awareness: whether the profiler can distinguish major LLM inference phases and report per-phase latency attribution and bottleneck analysis.
★ No-host machine: whether the profiler operates independently on-device without relying on an auxiliary machine to run or process measurements.

3 LM-Meter Design

3.1 Phase-level Latency Profiling

Phase-level inference latency refers to the runtime spent in distinct functional stages of LLM execution, such as embedding, prefill, decode, softmax, and token sampling. Unlike end-to-end (E2E) latency, which offers a coarse aggregate metric, phase-level profiling provides a semantically meaningful decomposition of the inference pipeline. This finer granularity brings several advantages. First, it enables precise bottleneck diagnosis by identifying which stages dominate latency, thereby guiding targeted optimizations. Second, it uncovers how different hardware architectures interact with individual phases, supporting platform-aware deployment and tuning. Third, it prevents misleading conclusions that may arise from aggregated E2E metrics by revealing heterogeneous performance characteristics across phases. Lastly, phase-level latency traces lay the groundwork for accurate runtime modeling, an essential capability for dynamic scheduling and improving interactive responsiveness in on-device LLM systems.

Unfortunately, directly profiling phase-level latency for on-device LLMs via application-level APIs is infeasible. First, application-level timing is inherently imprecise. User-facing apps typically interact with inference runtimes through high-level wrappers (e.g., Java/Kotlin on Android or Swift on iOS), which expose only coarse-grained timing signals, such as those from `SystemClock`, spanning the entire model dispatch. These measurements also include overheads from library calls, runtime transitions, and language bindings (e.g., JNI bridges to native C++), obscuring the true execution time. Second, application-level instrumentation lacks visibility into internal phase boundaries of LLM inference, which are handled within the native runtime.

To enable accurate phase-level latency measurements, LM-Meter instruments the native inference engine, such as MLC [38], at the runtime level. Specifically, we insert lightweight trace timers into the C++ backend, placed immediately before and after each semantic phase. Each timer is implemented using `std::chrono::steady_clock`, a monotonic clock provided by the C++ `chrono` library, wrapped in a custom time utility to ensure consistent collection. We select `steady_clock` after systematic evaluation, based on the following practical advantages: (i) It is monotonic

and immune to wall-clock adjustments, ensuring timestamp differences are reliable. (ii) It is supported across all major platforms (Linux, Android, iOS, macOS, and Windows), as required by the C++ standard, enabling portable instrumentation without platform-specific code. (iii) On modern systems, `steady_clock` maps to high-resolution hardware clocks, typically offering sub-microsecond precision, sufficient for capturing LLM phase durations ranging from microseconds to hundreds of milliseconds. (iv) The call to `steady_clock::now()` incurs minimal overhead. Consequently, `steady_clock` strikes an effective balance between portability, accuracy, and low instrumentation overhead, making it a robust choice for fine-grained latency profiling in resource-constrained environments. Additionally, LM-METER tags each timing record with its corresponding phase name and sequence index, enabling alignment with the input prompt sequence and per-token generation steps. This facilitates per-token phase-level latency analysis, providing fine-grained insights into the temporal behavior of LLM inference across the generation sequence.

3.2 Kernel-level Latency Profiling

A kernel typically serves as the fundamental unit of scheduling and execution in modern machine learning (ML) frameworks, especially on mobile and edge platforms [26]. The execution latency of individual kernels is highly sensitive to hardware characteristics, including compute capacity, memory bandwidth, and cache hierarchy. As a result, kernel performance can vary significantly across devices and fluctuate even on the same device due to dynamic resource contention from background applications. Hence, accurate, online profiling of kernel-level latency is essential for identifying performance bottlenecks and enabling hardware-aware optimizations, which includes device-specific kernel tuning, dynamic scheduling strategies, all of which are crucial for maximizing efficiency in on-device LLM systems.

However, accurate kernel-level runtime latency profiling for on-device LLMs faces two challenges, recognized in the broader systems research community. First, inserting trace timers directly into individual GPU kernel source code is infeasible because most mobile GPU drivers and runtime libraries are proprietary and closed-source. Second, modern ML compilers, such as TVM [39] and LiteRT [40], translate a graph of high-level operators (e.g., `Dequantize` → `MatMul` → `Add` → `RMSNorm`) into fused GPU kernels. During this process they apply kernel-fusion passes that merge multiple adjacent operators into a single launch to minimize memory traffic and kernel-launch overhead. These fusion passes are driven by heuristic cost models, autotuning, or backend-specific templates but the resulting fusion strategies and kernel boundaries are opaque to users.

```
OPENCL_CALL(clGetEventProfilingInfo(
    ev, CL_PROFILING_COMMAND_QUEUED, sizeof(r->t_queued_ns),
    &r->t_queued_ns, nullptr));
OPENCL_CALL(clGetEventProfilingInfo(
    ev, CL_PROFILING_COMMAND_SUBMIT, sizeof(r->t_submit_ns),
    &r->t_submit_ns, nullptr));
OPENCL_CALL(clGetEventProfilingInfo(
    ev, CL_PROFILING_COMMAND_START, sizeof(r->t_start_ns),
    &r->t_start_ns, nullptr));
OPENCL_CALL(clGetEventProfilingInfo(
    ev, CL_PROFILING_COMMAND_END, sizeof(r->t_end_ns),
    &r->t_end_ns, nullptr));
```

Figure 1: Code snippet for querying OpenCL kernel-execution timestamps.

To address these challenges, LM-METER leverages the event callback mechanisms defined in GPU libraries, such as OpenCL, VulKan, and Metal, to capture the complete lifecycle of every kernel during model execution, without requiring access to kernel source code. Inspired by nnPerf [36], this approach enables fine-grained kernel-level profiling even in closed-source environments. Using OpenCL as an example, the runtime maintains three queues, the *command queue*, *host queue*, and *device queue*. After model compilation, the interpreter constructs a command queue, where each entry represents either a GPU kernel invocation or a data movement operation. During inference, each entry from the command queue are copied to the host queue (CPU) and finally enqueued into the device queue (GPU) chronologically for execution. LM-METER instruments this process by attaching profiling callbacks to each OpenCL command via the native `cl_event` interface. Specifically, it queries the `cl_event` object associated with each entry using `clGetEventProfilingInfo`, which exposes four standardized profiling nanosecond-resolution timestamps:

- `CL_PROFILING_COMMAND_QUEUED`: when the command (e.g., a kernel launch or data movement) is enqueued into the command queue;
- `CL_PROFILING_COMMAND_SUBMIT`: when the command is submitted by the host to the device queue;
- `CL_PROFILING_COMMAND_START`: when the device actually begin execution of the command;
- `CL_PROFILING_COMMAND_END`: when the device complete execution of the command.

A code snippet illustrating this is shown in Fig. 1. Additionally, LM-METER maintains a lightweight per-kernel data structure containing: the kernel name, a pointer to its associated command queue, a host-side enqueue timestamp `t_cpu_enqueue_ns`, and the four device-side timestamps (`t_queued_ns`, `t_submit_ns`, `t_start_ns`, and `t_end_ns`). These measurements allow us to reconstruct the full execution timeline for each kernel launch during inference. They also enable precise attribution of latency across three key

stages: (i) host-to-device queuing, (ii) scheduling and dispatch delays within the device runtime, and (iii) device-side kernel execution.

3.3 From Latency to Energy and Adaptation

LM-METER is designed with extensibility in mind and provides the infrastructure hooks to enable future work.

Energy profiling. While LM-METER currently targets latency analysis, its modular architecture readily supports integration with energy and power measurement tools, such as built-in current sensor on mobile devices or external power monitor like Monsoon [41]. Time-aligned phase- and kernel-level traces make it a solid foundation for profiling on-device LLM energy and for joint latency–energy optimization.

Runtime system adaptation. Although adaptation mechanisms are not implemented in this study, LM-METER can serve as a telemetry backend for adaptive LLM systems. Real-time profiling signals can inform an *Adaptation Engine* to dynamically adjust model selection, resource allocation, or execution strategies (e.g., model switching, frequency scaling, or phase skipping) in response to runtime conditions. By delivering structured latency feedback during inference, LM-METER opens new opportunities for fine-grained, performance-aware adaptation in on-device LLMs.

4 Evaluation

In this section, we present a comprehensive evaluation of LM-METER's profiling accuracy and system overhead on commercial mobile devices. We begin by detailing the implementation of LM-METER, followed by an in-depth analysis of its phase- and kernel-level profiling performance. We conclude with an assessment of the system overhead incurred. This evaluation is essential to demonstrate that LM-METER provides reliable, high-fidelity measurements while remaining lightweight enough for real-time on-device LLM profiling.

4.1 LM-METER Implementation

We implement LM-METER on top of the MLC framework [38] and TVM [39], comprising approximately 3,500 lines of C++ (e.g., instrumentation and profiler runtime), Python (e.g., model compilation, build tooling, and data post-processing), and Java/Kotlin (e.g., Android app) code to support the cross-stack nature of on-device LLM deployment, which span both back-end systems and front-end interfaces. Since OpenCL-based GPU acceleration is widely supported by mainstream ML frameworks, such as LiteRT (formerly known as TensorFlow Lite) [40], ONNX [42], OpenVINO [43] and others, LM-METER is designed to be readily portable to alternative frameworks. While implementation details may differ across frameworks, the core profiling and instrumentation logic can be adapted with minimal effort.

Table 3: Mobile devices used in our experiments.

Devices	SoCs	RAM	Year	Tier
Google Pixel 8 Pro	Google Tensor G3	12GB	2024	High
Google Pixel 7	Google Tensor G2	8GB	2023	Mid
Google Pixel 6	Google Tensor	8GB	2022	Low

To ensure experimental repeatability, we fix the input prompts and set the maximum number of generation tokens. We configure the sampling parameters to `temperature = 0.0` and `top_p = 1.0`, and disable stop strings to enforce deterministic token generation during profiling. Unless otherwise specified, all experiments are repeated three times, and we report the average results. All experimental results presented in Sections 4 and 5 are obtained using three commercial mobile platforms, summarized in Table 3. We present selected device combinations in each section to balance hardware diversity within space constraints.

Proposed profiling evaluation metrics. To rigorously assess the profiling accuracy, we define two complementary metrics for each phase or kernel k:

(1) *Accuracy* α_k (%):

$$\alpha_k = \left(1 - \frac{\left|t_k^{\mathrm{LM}} - t_k^{\mathrm{GT}}\right|}{t_k^{\mathrm{GT}}}\right) \times 100, \tag{1}$$

where t_k^{LM} and t_k^{GT} denote the latency measured by LM-METER and the ground-truth latency, respectively, in milliseconds. α_k quantifies the proportion of the true execution time correctly captured by LM-METER.

(2) *Scaled error rate* $\varepsilon_k^{\star}$ (μs/ms):

$$\varepsilon_k^{\star} = \frac{10^3\,\Delta_k\ [\mu\mathrm{s}]}{t_k^{\mathrm{GT}}\ [\mathrm{ms}]} \quad (\mu\mathrm{s\ per\ ms}), \tag{2}$$

where $\Delta_k = \left|t_k^{\mathrm{LM}} - t_k^{\mathrm{GT}}\right|$. By normalizing the absolute error by the ground-truth runtime (in milliseconds), $\varepsilon_k^{\star}$ measures the number of microseconds of error incurred per millisecond of actual execution time. This unitless quantity is typically on the order of $O(1)$, offering an intuitive sense of relative error magnitude.

In summary, α_k provides a direct measure of ground-truth coverage as a percentage (higher is better), while $\varepsilon_k^{\star}$ quantifies the absolute timing error normalized by runtime (lower is better). Together, these metrics offer complementary relative and absolute perspectives on the fidelity of the profiler.

4.2 Phase-Level Profiling Performance

Phase-level latency ground-truth. To evaluate the accuracy of phase-level profiling, we first establish a trusted baseline reference using AGI, which does not natively support phase-aware latency measurement for LLM inference.

Table 4: Phase-level profiling accuracy on Pixel 8 Pro.

Models	Phases	Profiled latency (ms)		α (%)	$\varepsilon^{\star}$(μs/ms)
		LM-METER	AGI		
Llama-3.2-3B-Instruct	Embedding	0.8038	0.7763	96.46	35.412
	Prefill	3433.8628	3433.8142	99.99	0.014
	Decode	62.5669	62.5303	99.94	0.585
	Softmax	142.6166	142.6542	99.97	0.264
	CopyProbsToCPU	0.4929	0.4616	93.22	67.718
	Sampling	0.0675	0.0824	81.86	181.439
	End-to-end	3640.4104	3640.3191	**99.99**	**0.025**
Gemma-2-2B-it	Embedding	0.7659	0.7398	96.48	35.226
	Prefill	9301.1318	9301.0589	99.99	0.008
	Decode	54.5909	54.5557	99.94	0.646
	Softmax	502.3319	502.3698	99.99	0.076
	CopyProbsToCPU	0.5570	0.5255	94.02	59.829
	Sampling	0.1698	0.1830	92.76	72.365
	End-to-end	9859.5473	9859.4329	**99.99**	**0.012**
TinyLlama-1.1B-Chat-v1.0	Embedding	0.6858	0.6636	96.66	33.428
	Prefill	5023.1182	5023.0765	99.99	0.008
	Decode	34.6845	34.6534	99.91	0.897
	Softmax	188.1463	188.1779	99.98	0.168
	CopyProbsToCPU	0.3887	0.3602	92.10	78.980
	Sampling	0.0504	0.0669	75.36	246.372
	End-to-end	5247.0738	5246.9985	**99.99**	**0.014**
DeepSeek-R1-Distill-Qwen-1.5B	Embedding	0.6753	0.6531	96.60	33.975
	Prefill	7630.7553	7630.6979	99.99	0.008
	Decode	49.3840	49.3499	99.93	0.690
	Softmax	437.0047	437.0459	99.99	0.094
	CopyProbsToCPU	0.4206	0.3888	91.84	81.565
	Sampling	0.0671	0.0786	85.41	145.949
	End-to-end	8118.3069	8118.2143	**99.99**	**0.011**
SmolLM2-360M-Instruct	Embedding	0.9844	0.9671	98.21	17.909
	Prefill	2399.3860	2399.3246	99.99	0.026
	Decode	44.7612	44.7377	99.95	0.527
	Softmax	221.8416	221.8672	99.99	0.116
	CopyProbsToCPU	0.3081	0.2854	92.05	79.532
	Sampling	0.0300	0.0394	76.06	239.395
	End-to-end	2667.3113	2667.2214	**99.99**	**0.034**

To enable AGI-based tracing for on-device LLM inference, we extend the MLC runtime with instrumentation support via the Perfetto Tracing SDK. Specifically, we define and insert phase-level annotations using the `TRACE_EVENT()` macro [34] to mark key inference phases, such as embedding, prefill, decoding, softmax, sampling, and probability transfer to CPU. All events are grouped under a unified tracing category (e.g., "llm"), which is statically registered during runtime initialization. During profiling, the target mobile device is connected via USB to a host machine running AGI. Upon completion of inference tasks, AGI generates a binary `.perfetto` trace file, which we post-process to extract timestamped phase events. In parallel, LM-METER captures phase-level timing data in real time, enabling a direct comparison against the AGI measurements.

Evaluation of phase-level profiling accuracy. Table 4 presents a comparison of phase-level runtime latencies measured by LM-METER against baseline values obtained using AGI across five LLMs. These models span a wide range of scales, from 360 million to 3 billion parameters, and represent a diverse set of architectural designs. Among them, `SmolLM2-360M-Instruct` operates entirely in FP16 precision (unquantized), while the remaining models employ group-wise 4-bit weight quantization with 16-bit activations. Each experiment is repeated three times, with ten prompt-response turns per run. We report the mean latency in Table 4. Across all models and phases, LM-METER consistently demonstrates high profiling accuracy, achieving end-to-end latency accuracy[1] of $\alpha \geq 99.99\%$ and scaled error $\varepsilon_k^{\star} \leq 0.034$ μs/ms, indicating that LM-METER introduces only negligible deviation from AGI measurements.

Additionally, inference phases with the greatest impact on end-to-end latency, including prefill, decode, and softmax, are profiled with exceptional accuracy. These phases account for over 99% of total inference time, as shown in Fig. 2, and exhibit profiling accuracy of $\alpha \geq 99.91\%$ and $\varepsilon_k^{\star} \leq 0.897$ μs/ms. The only notable outlier is the sampling phase. On the Pixel 8 Pro, a single sampling phase typically completes within just 50–80μs, making it highly sensitive to profiling noise. Consequently, the inherent ± 1–3 μs timing variability of LM-METER constitutes a substantial fraction of the total phase duration. This results in noticeably reduced profiling accuracy ($\alpha = 75$–92%) and elevated scaled error ($\varepsilon_k^{\star} \leq 246$ μs/ms). Note that this reduced accuracy is not caused by limitations in the profiler's granularity. Rather, it likely result from the inherent difficulty of measuring an *extremely short*, partially host-bound routine whose execution time is comparable to system-level noise, such as OS scheduling jitter and CPU-GPU synchronization delays.

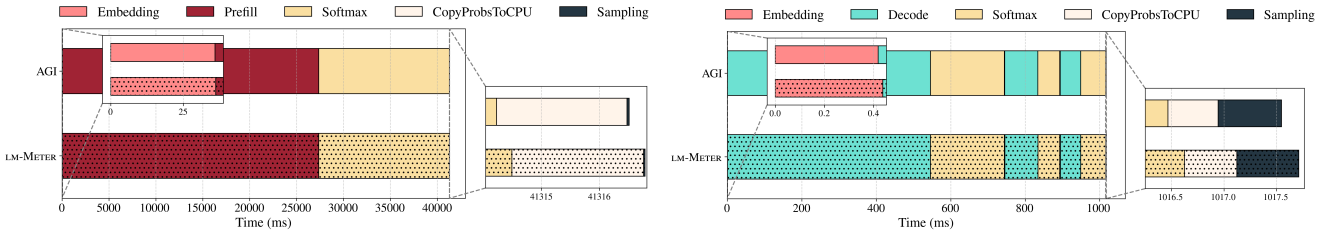
In summary, these results collectively demonstrate that LM-METER delivers accurate and reliable phase-level profiling across a wide range of model scales and architectures.

4.3 Kernel-Level Profiling Performance

Kernel-level latency groundtruth. Due to the lack of existing profilers capable of accurately capturing kernel-level latency for on-device LLMs (§2.3), we estimate ground-truth latencies using a custom *kernel duplication* technique. Specifically, for each target kernel within an LLM inference phase, we perform the following two-step profiling:

- Baseline profiling: We first profile the *unmodified* model using AGI, recording the latency of each inference phase as t_i, where i indexes the phases (e.g., embedding, prefill, decode, softmax).
- Kernel-duplicated profiling: We compile a modified model where the target kernel is duplicated n times and inserted immediately after its original invocation within the same phase. Profiling this modified model yields updated phase latencies t_i^{dup}.

[1] End-to-end latency is computed as the sum of all phase latencies.



(a) Single *prefill* step. (Total absolute error = 0.2739ms)

(b) Three consecutive *decode* steps. (Total absolute error = 0.1643ms)

Figure 2: Phase-level latency comparison between LM-METER and AGI on Pixel 8 Pro running a quantized `gemma-2-2b-it` (4-bit weights, 16-bit activations).

Table 5: Kernel-level runtime latency profiling results on Google Pixel 8 Pro and Pixel 7.

Kernels	Phases	Google Pixel 8 Pro				Google Pixel 7	
		Profiled latency (ms)		α (%)	$\varepsilon^\star$(μs/ms)	α (%)	$\varepsilon^\star$(μs/ms)
		LM-METER	GT				
`dequantize1_NT_matmul5`	Prefill	81.1899	82.1329	98.85	11.481	98.88	11.212
`dequantize2_NT_matmul6`		31.3407	31.7568	98.69	13.103	95.18	48.209
`dequantize3_NT_matmul7`		330.3757	332.7218	99.29	7.051	98.87	11.328
`dequantize4_NT_matmul8`		367.5603	367.0284	**99.86 (highest)**	1.449	99.11	8.896
`dequantize1_NT_matmul10`	Decode	0.3643	0.3737	97.46	25.391	97.19	28.145
`dequantize2_NT_matmul11`		0.2062	0.2006	97.23	27.706	98.14	18.587
`dequantize3_NT_matmul12`		1.3813	1.3601	98.44	15.587	98.17	18.267
`dequantize4_NT_matmul13`		0.6862	0.6586	95.81	41.921	97.50	25.044
`dequantize_NT_matmul14_divide2_tir_tanh2_multiply8`		18.4379	18.3619	99.59	4.147	98.13	18.705
`add_norm_prefill`		0.1149	0.1059	**91.51(lowest)**	84.891	93.29	67.080
`rms_norm2`		0.1037	0.1092	94.93	50.641	92.65	73.531
`split2_gelu_tanh2_multiply7`		0.0952	0.0939	98.62	13.727	93.75	62.517
`multiply6`		0.1061	0.1005	94.35	56.546	**90.31**	96.934
`chunk_lse`	Softmax	0.2718	0.2839	95.53	44.735	99.39	6.026
`softmax_with_chunked_sum`		0.2376	0.2392	99.33	6.689	**99.40**	5.992
`dequantize_take1`	Embedding	0.1034	0.1097	94.26	57.429	95.73	42.676

We estimate the ground-truth latency of the target kernel as: $\frac{t_i^{dup} - t_i}{n}$. We set $n = 50$ for kernels with expected execution latency exceeding 1ms, and $n = 1000$ for shorter kernels. This procedure is repeated independently for each kernel of interest to obtain reliable per-kernel latency estimates.

To implement the kernel duplication technique without introducing extra overhead or altering the semantics of the model, we modify the model compilation pipeline in MLC, which includes the code generation (codegen) path for kernel emission. Specifically, (1) the kernel generation is extended to emit n back-to-back copies of target kernel, preserving the original data dependencies; (2) optimization passes that may eliminate unused or redundant code, such as `remove_all_unused` and `DeadCodeElimination` are explicitly disabled to retain the duplicates; and (3) CPU and GPU frequencies are fixed to eliminate variability from dynamic voltage and frequency scaling (DVFS), and all non-essential foreground/background applications are terminated to minimize system-level noise. Fig. 3 shows kernel duplication for `fused_dequantize2_NT_matmul11` in decode phase.

Evaluation of kernel-level profiling accuracy. Table 5 presents kernel-level latency measurements captured by LM-METER on both the Google Pixel 8 Pro and Pixel 7, compared against ground-truth (GT) values collected via AGI with kernel duplication. Across both platforms, LM-METER consistently delivers high profiling accuracy at kernel granularity. Specifically, on the Pixel 8 Pro, the mean and median accuracy across all 16 evaluated kernels are 96.82% and 97.35%, respectively. On the Pixel 7, the corresponding values are 96.61% and 97.81%. Even in the worst-case scenario, `multiply6`, LM-METER achieves 90.31% accuracy. The four GEMM kernels within the prefill phase, which dominate the total inference latency, are profiled with near-perfect accuracy. On the Pixel 8 Pro, these kernels achieve 98.69 % to 99.86 % accuracy with scaled error rates $\varepsilon^\star < 13.1\,\mu s/ms$; they reach 95.18 % to 99.11 % accuracy with $\varepsilon^\star < 48.2\,\mu s/ms$

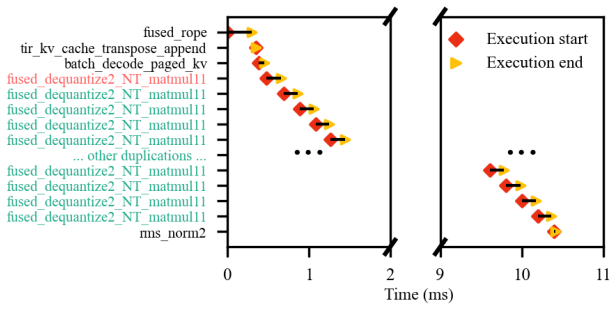


Figure 3: Illustration of kernel duplication for `fused_dequantize2_NT_matmul11`, showing the original kernel (red) and its 50 duplicated instances (green). A broken x-axis with omitted rows is used to save space.

on the Pixel 7. Moreover, LM-METER also demonstrates high accuracy on *micro-kernels* with true runtimes under 1 ms. On the Pixel 8 Pro, these kernels achieve a mean accuracy of 95.44 % and a median of 95.23 %; on the Pixel 7, the mean and median accuracy are 95.73 % and 96.46 %, respectively.

In summary, these results validate LM-METER's reliability in accurately capturing the latency of both latency-dominant kernels and short-duration micro-kernels.

4.4 System Overhead

Since LM-METER performs online profiling directly on mobile devices, its practicality depends critically on introducing minimal latency overhead. To quantify this overhead, we measure the throughput (in tokens per second) of the two primary inference phases, prefill and decode, under three CPU governor configurations that emulate varying levels of on-device resource availability: (1) `Performance`: All CPU cores are prioritized to operate at their peak frequencies; (2) `Conservative`: Dynamic DVFS with a bias toward lower frequencies; (3) `Powersave`: All CPU cores are prioritized to run at their minimum frequencies.

We compare LM-METER against two baselines: (1) *No profiling* and (2) MELTing Point [35], a state-of-the-art profiler for on-device LLMs. Fig. 4 summarizes the results. Across all CPU governors, LM-METER consistently imposes negligible throughput degradation. Under the `Performance` and `Conservative` governors, its throughput closely matches that of the no-profiling baseline for both phases. Even under the `Powersave` setting, where system resources are most constrained, LM-METER exhibits only a modest throughput reduction of 2.58% for prefill and 0.99% throughput for decode. In contrast, MELTing Point incurs substantial profiling overhead in all three settings, especially under *Powersave*, where its throughput drops by 22% for prefill and by more than 93% for decode. These results underscore the efficiency of LM-METER and affirm its suitability for real-time, online profiling in resource-constrained mobile environments.

Table 6: Model suites for phase-oriented study.

Model suites	Model scales (parameters)	Precision	HuggingFace collections
`Pythia` [44]	{70M, 160M, 410M, 1B, 1.4B}	float16	EleutherAI/pythia
`SmolLM2` [45]	{135M, 360M, 1.7B}	bfloat16	HuggingFaceTB/SmolLM2
`Qwen1.5` [25]	{0.5B, 1.8B}	bfloat16	Qwen/Qwen1.5

Table 7: Evaluation tasks for phase-oriented study.

Categories	Tasks	Full datasets	Sampled subsets
Knowledge ability	ARC-Easy [46]	2.38k prompts	238 prompts
	ARC-Challenge [46]	1.12k prompts	112 prompts
Inference ability	HellaSwag [47]	10k prompts	100 prompts
	GSM8K [48]	1.32k prompts	131 prompts

5 On-Device LLM Empirical Study

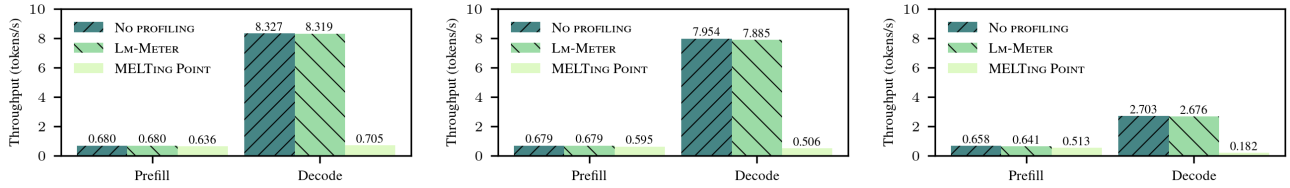
Through LM-METER, we conduct two empirical studies: *phase-oriented* and *kernel-oriented*, to illuminate realistic performance behavior of on-device LLMs.

5.1 Phase-oriented Empirical Study

In this section, we present a phase-oriented empirical study of on-device LLMs and an in-depth analysis leveraging our LM-METER. The goal is to characterize the trade-offs between model accuracy and efficiency during the *prefill* and *decode* phases across diverse tasks, model suites, and parameter scales. To our knowledge, this is among the first systematic explorations of phase-level performance-efficiency trade-offs in LLM inference for on-device environments, which can contribute insightful suggestions for optimizing on-device LLM deployment and lay a foundation for future research.

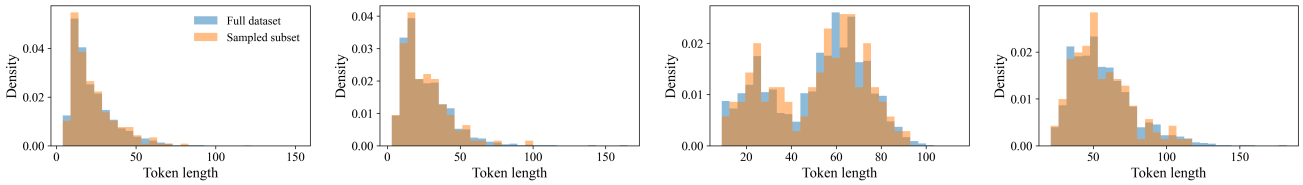
Model suite selection. We evaluate three representative model suites: `Pythia` [44], `SmolLM2` [45], and `Qwen1.5` [25], each offering a range of model sizes suited for affordable on-device deployment. Details are summarized in Table 6.

Task selection. To capture a broad spectrum of model capabilities, we categorize mainstream evaluation tasks into two primary skill domains: *knowledge ability* and *reasoning ability*. Knowledge ability measures factual understanding without additional reasoning, while reasoning ability reflects multi-step inference grounded in prior knowledge. In this study, we select two representative datasets for each category, listed in Table 7. ARC-Easy and ARC-Challenge [46] assess factual recall and basic comprehension, while HellaSwag [47] and GSM8K [48] focus on reasoning. Specifically, HellaSwag evaluates commonsense reasoning in everyday contexts, and GSM8K targets multi-step mathematical problem solving. These tasks vary in complexity and input length, offering a diverse view of inference behavior across different demands.



(a) CPU governor set to `Performance` (b) CPU governor set to `Conservative` (c) CPU governor set to `Powersave`

Figure 4: Comparison of profiling-induced latency overhead, measured as throughput (tokens/s) during prefill and decode, under three CPU governors. Results compare LM-METER, MELTing Point [35], and a no-profiling baseline.



(a) ARC-Easy (KL = 0.0195) (b) ARC-Challenge (KL = 0.0311) (c) HellaSwag (KL = 0.0433) (d) GSM8K (KL = 0.0330)

Figure 5: Token-length distributions of each full dataset (blue) vs. KL-matched subset (orange). We sample 10% of ARC-Easy, ARC-Challenge, and GSM8K (238, 112, and 131 prompts) and 1% of HellaSwag (100 prompts). Reported KL divergences measure how closely each subset's token-length histogram matches the full distribution.

Subset sampling and evaluation. We find that full-benchmark evaluation of LLMs is often impractical under constrained resources. First, inference on mobile and edge hardware is significantly slower than on server-scale accelerators. For instance, running the full HellaSwag test set (10k prompts) can take hours or even days per model. As the number of models and tasks grows, evaluation cost scales poorly: Cost = #models × #tasks × #prompts × inference latency × #devices. Second, prolonged evaluation risks triggering thermal throttling or battery degradation on mobile devices, incurring inconsistent or unreliable measurements.

To address the challenges of full-scale on-device evaluation, we develop a sampling-based approach. Instead of randomly selecting prompts, which may introduce bias due to skewed token-length distributions, we apply a Kullback-Leibler (KL) divergence-based sampling algorithm. This method selects a representative subset whose token-length distribution closely matches that of the full dataset, thereby preserving task complexity and linguistic diversity. To balance scalability and fidelity, we retain 10% of ARC-Easy, ARC-Challenge, and GSM8K (238, 112, and 131 prompts, respectively), and 1% subset of HellaSwag (100 prompts), due to its larger size. Fig. 5 illustrates the token-length distributions of both the full datasets and our sampled subsets. Notably, in all cases, the KL divergence remains below 0.0433, demonstrating high representativeness. Additionally, to streamline benchmarking across different models and tasks, LM-METER automates prompt injection from the KL-matched subsets directly into the LLM inference pipeline. This removes manual interaction and eliminates human-induced variability, e.g., typing, ensuring consistent, repeatable, and unbiased runtime measurements.

Accuracy vs. decode/prefill latency. We evaluate phase-specific efficiency using two metrics: *decode latency per output token*[2] and *prefill latency per input token*[3]. Model accuracy on individual tasks is evaluated using the *lm-evaluation-harness* [32], while latency profiling is conducted with LM-METER. To enable a fair and consistent comparison across tasks, the accuracy of each model is normalized to the performance of LLaMA-33B [3] on the corresponding benchmark. We perform the same experiments on both Pixel 8Pro and 6. The results are shown in Figs. 6, 7, and 8, from which we derive the following key observations and implications.

First, *prefill emerges as the dominant throughput bottleneck in on-device LLM inference, in contrast to trends observed in server-scale deployments.* For latency-sensitive, small-batch inference workloads common in interactive LLM applications, the autoregressive decode phase is widely recognized

[2]Decode latency per output token (s/token) measures how quickly the model generates tokens autoregressively during the decode phase.
[3]Prefill latency per input token (s/token) quantifies how efficiently the model processes the input context during the prefill phase.

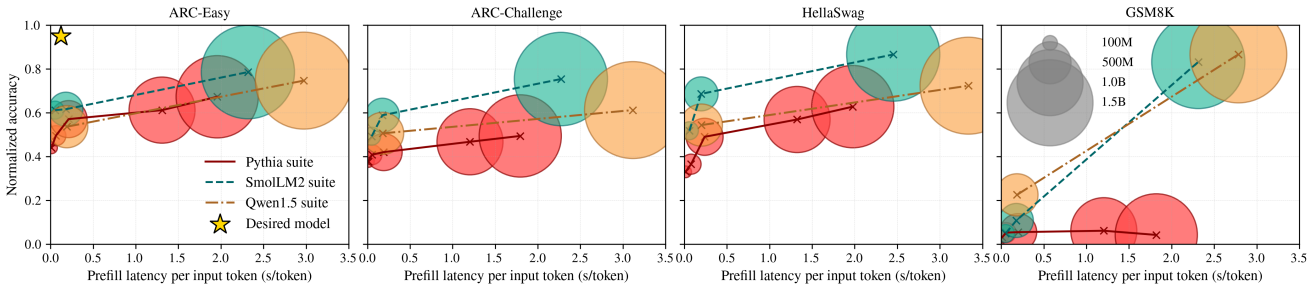


Figure 6: Prefill latency per input token vs. accuracy trade-off across LLM suites and tasks on Pixel 8Pro. Bubble area ∝ model size; accuracy is normalized to LLaMA-33B [3] on each task for fair cross-benchmark comparison.

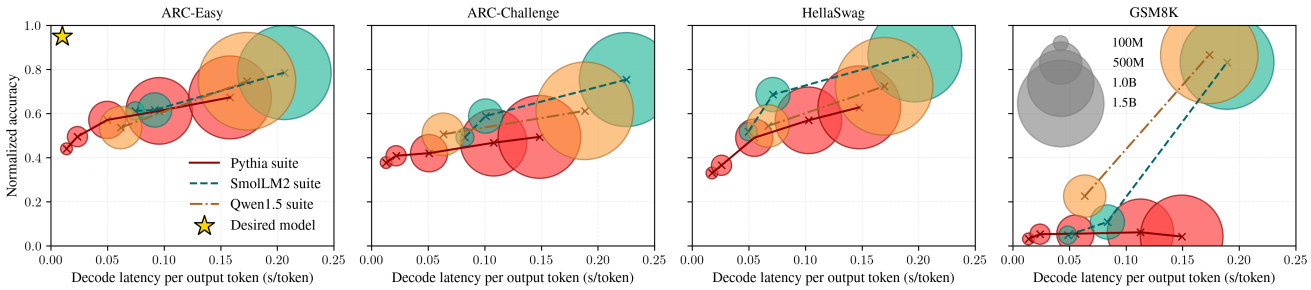


Figure 7: Decode latency per output token vs. accuracy trade-off across LLM suites and tasks on Pixel 8Pro.

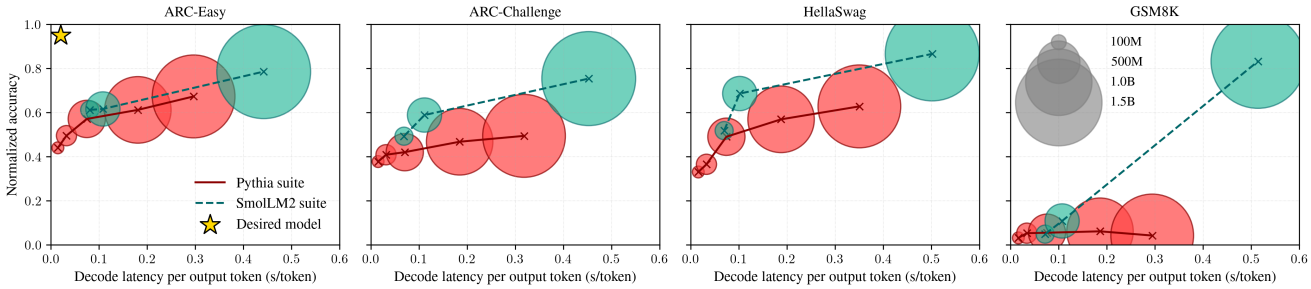


Figure 8: Decode latency per output token vs. accuracy trade-off across LLM suites and tasks on Pixel 6. `Qwen1.5-1.8B` is not evaluated due to insufficient memory on the device.

as the primary performance bottleneck on server-scale GPUs. Its inherently sequential execution and limited batching efficiency lead to significantly lower hardware utilization compared to the highly parallelizable prefill phase. Prior work has demonstrated that prefill can be up to 100× faster than decode on a single A100 GPU at batch size 1 [49]. However, our findings reveal the opposite trend on mobile and edge devices. For example, when scaling the `Pythia` from 70M to 1.4B parameters (at batch size 1), the average prefill latency per input token across four tasks increases dramatically from 0.012 s/token to 1.9 s/token, a 158× slowdown. In contrast, decode latency per output token grows more moderately, from 0.015 s/token to 0.15 s/token, representing only a 10× slowdown. This inversion arises because the prefill requires processing the entire input context in a single forward pass, which becomes increasingly expensive on mobile and edge hardware with constrained compute resources, limited memory bandwidth, and the lack of support for parallelized prefill.

Implication 1: *These observations underscore the need for phase-specific optimizations tailored to the unique constraints of on-device LLMs, particularly prefill. While decode-phase optimizations have been the primary focus in server-scale LLM inference, such as speculative decoding [11, 50], paged*

KV cache [51], and tensor parallelism [52], their effectiveness in on-device settings remains under-explored.

Second, *accuracy-latency trade-offs in on-device LLMs are highly task-dependent.* We observe that the extent to which a model can balance inference quality (accuracy) and efficiency (latency) varies significantly across tasks, likely due to differences in input complexity, prompt structure, and reasoning requirements. As illustrated in Fig. 7 and 8, for instance, ARC-Easy comprises relatively simple multiple-choice questions with short, factual prompts. In this case, reducing model size significantly improves latency without incurring a steep accuracy penalty. Compared to `Pythia-1.4B`, the average normalized accuracies of `Pythia-1B`, `410M`, `160M`, and `70M` degrade by only 9%, 15%, 27%, and 34%, respectively, while their decode latency per output token improves by 0.6×, 2.2×, 5.1×, and 10.4× on Pixel 8Pro. This yields a favorable accuracy-latency Pareto frontier for on-device LLM inference in resource-constrained environments. In contrast, GSM8K, a math-intensive benchmark requiring multi-step reasoning, exhibits the most polarized trade-off. Smaller models yield high decoding efficiency but produce unusably low accuracy, while larger models offer acceptable accuracy only at the cost of prohibitively high latency. This sharp dichotomy underscores the difficulty in supporting GSM8K efficiently on-device without task-specific optimizations. HellaSwag, which involves longer, narrative-style prompts and commonsense reasoning, falls between these two extremes. Notably, a clear "most-balanced" model, `SmolLM2-360M`, emerges near the knee of the Pareto frontier. Finally, even within the same task category, task difficulty can significantly shape the trade-off. Compared to ARC-Easy, ARC-Challenge exhibits a flatter accuracy-latency frontier, where accuracy improves more slowly with increased latency, and smaller models rapidly encounter diminishing returns.

Implication 2: *These observations suggest that task- and difficulty-aware adaptive model selection might be crucial for achieving an optimal balance between performance and efficiency in on-device LLM deployments. Moreover, not all task categories are suitable for local processing on mobile and edge devices. Awareness of task type and difficulty can also inform decisions about when and how to offload computation to edge or cloud infrastructures, enabling effective device-server collaboration for LLM inference.*

Third, *cross-suite differences hint at model architectural effects, beyond mere parameter scaling.* While scaling laws suggest that increasing model parameter size is the primary driver of LLM performance [53], our results show that architectural design is equally critical, especially for compact models in on-device settings, in shaping the accuracy-latency trade-off. For instance, `SmolLM2-360M` consistently matches or outperforms `Pythia-1B`, a model nearly three times larger, across all tasks. Notably, `SmolLM2` suite advances the Pareto frontier on benchmarks, such as ARC-Challenge and HellaSwag, achieving higher accuracy with lower decode latency. This is primarily enabled by `SmolLM2`'s architecture and pretraining data curation, both carefully optimized for resource-constrained, on-device inference [20, 54].

Implication 3: *These observations suggest that model architecture, beyond parameter scaling, is a key factor shaping performance-efficiency trade-offs for compact on-device language models (i.e., sub-billion to billion-scale). Targeted architectural design or search prior to pre-training may shift the Pareto frontier as effectively as scaling model size.*

Quantization impact on accuracy-latency trade-offs. Model quantization is a widely adopted technique that reduces the precision of weights and activations in LLMs, thereby lowering memory footprint and computational cost. However, its practical impact on on-device LLMs remains underexplored, especially in terms of how quantization shifts the trade-off between accuracy and runtime latency. Prior studies have largely focused on theoretical metrics, such as FLOPs or parameters, while overlooking systematic empirical evaluation on this perspective. To address this gap, we propose a new evaluation metric, the *Harmonic Quantization score* ($\mathcal{HQ}$), and conduct a benchmarking study across the aforementioned model suites, tasks, and mobile platforms. Our goal is to systematically quantify the impact of quantization on both accuracy and runtime latency. We define the $\mathcal{HQ}$ for task i:

$$\mathcal{HQ}_i = \left(\frac{1}{3} \left(\frac{1}{\mathcal{M}_{a,i}} + \frac{1}{\mathcal{M}_{l,i}^{\text{prefill}}} + \frac{1}{\mathcal{M}_{l,i}^{\text{decode}}} \right) \right)^{-1}, \tag{3}$$

where $\mathcal{M}_{a,i} = \frac{A_i^c}{A_i}$, with A_i^c and A_i representing the normalized accuracy of the quantized and original models on task i, respectively. Similarly, $\mathcal{M}_{l,i}^p = \frac{L_{i,p}}{L_{i,p}^c}$, where $L_{i,p}^c$ and $L_{i,p}$ are the runtime latencies of the quantized and original model during inference phase $p \in \{\text{prefill}, \text{decode}\}$. The $\mathcal{HQ}$ rewards quantized models that achieve balanced accuracy and latency. Its harmonic mean formulation penalizes any severe degradation in individual components, thus discouraging improvements that come at the expense of others.

We evaluate 4-bit quantized models from the `Pythia` and `SmoLM2` suites across multiple tasks. Fig. 9 compares their $\mathcal{HQ}$ scores. First, *quantization-induced efficiency gains vary across model architectures.* For `Pythia`, $\mathcal{HQ}$ increases nearly monotonically with model size, peaking at 1.4B for the more challenging tasks (ARC-Challenge and GSM8K), while saturating for ARC-Easy beyond 1B. In contrast, `SmoLM2` achieves its best accuracy-latency trade-off at smaller scales (135M-360M). Second, *quantization benefits are task-dependent and appear correlated with task difficulty.* The optimal model size

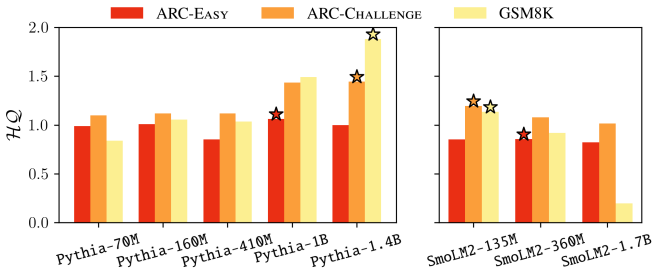


Figure 9: Task-wise Harmonic Quantization ($\mathcal{HQ}$) scores (higher is better) across model sizes for `Pythia` and `SmoLM2`. Stars mark the best-balanced model per suite. `Pythia` benefits from scaling up to 1.4B, while `SmoLM2` peaks at 135M-360M.

for quantization varies not only across architectures, but also across tasks within the same suite. For `Pythia`, the average $\mathcal{HQ}$ rises from 0.982 on ARC-Easy to 1.243 on ARC-Challenge and 1.260 on GSM8K, indicating that 4-bit quantization yields limited benefit on simpler tasks, but delivers clear gains on demanding ones. One anomalous case is `SmoLM2-1.7B` on GSM8K, whose $\mathcal{HQ}$ collapses due to near-zero accuracy reported by the *lm-evaluation-harness*. We have evaluated three different 4-bit quantization schemes for this model, all yielding similarly poor accuracy. A plausible explanation is that aggressive 4-bit quantization introduces significant noise into numerically sensitive layers, such as those performing multi-step arithmetic.

Implication 4: *These results underscore the value of our proposed $\mathcal{HQ}$ metric for benchmarking quantization techniques in on-device settings. Relying solely on accuracy or latency may hide critical efficiency regressions that impact on-device deployment. We consider evaluating diverse quantization techniques using $\mathcal{HQ}$ as our future work.*

Implication 5: *The presence of architecture- and task-specific sweet spots implies that quantization is not universally beneficial for on-device LLMs. The choice between quantized and full-precision models should jointly consider model size, task type, and hardware constraints.*

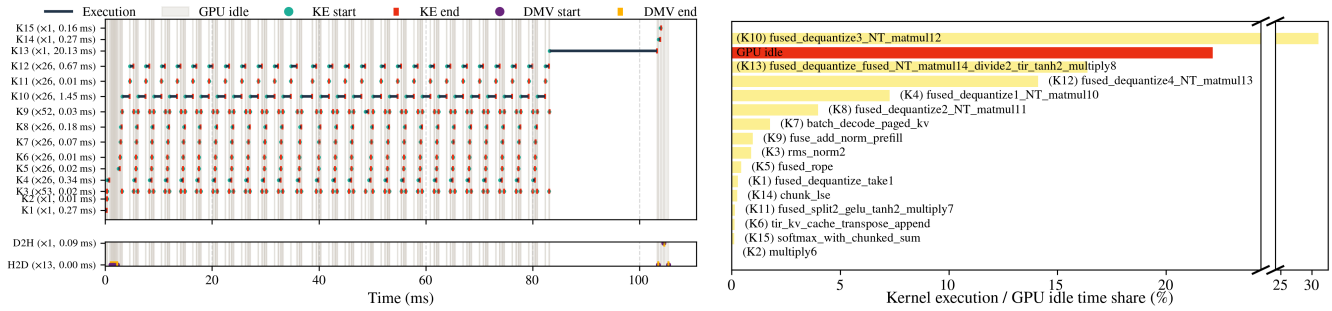
5.2 Kernel-oriented Empirical Study

We then employ LM-METER for a kernel-level analysis of on-device LLM execution. Since kernels serve as the fundamental scheduling units and embed numerous framework-level optimizations, profiling them provides deep visibility into the true performance landscape and enables characterization of fine-grained runtime behavior. Our empirical study is guided by the following research questions: (i) Which kernels dominate runtime latency? (ii) Do these dominant kernels remain consistent bottlenecks throughout the decode sequence, or do new ones emerge as the KV cache grows? (iii) How much of the decode timeline is spent in GPU-idle states? (iv) Can the observed kernel behaviors be leveraged to predict per-step decode latency for scheduling and runtime adaptation? Due to the page limit, we present representative results from running a 4-bit quantized `Gemma-2-2B-it` on a Pixel 8 Pro. These insights expose several system-level inefficiencies and optimization opportunities, many of which generalize to other models and platforms.

Kernel runtime performance. Fig. 10(a) depicts fine-grained start/end timestamps of GPU kernel executions within a single decode step, illustrating how kernels are sequenced and interleaved. Notably, frequent gaps between successive kernel launches (gray regions) lead to over 21% GPU idle time, making it the second-largest contributor to total latency. These idle periods are primarily attributed to host-side data preparation delays and I/O stalls. Additionally, multiple memory-bound short-duration kernels, including `fused_rope` (K5), `tir_kv_cache_transpose_append` (K6), `fuse_add_norm_prefill` (K9), and `rms_norms` (K3), are invoked frequently yet contribute less than 10% of total execution time. Their high invocation frequency, however, increases kernel launch overhead and widens idle gaps, exposing inefficiencies in kernel scheduling and dispatch. Moreover, three fused GEMM kernels (K10, K12, K13) dominate GPU execution, contributing over 60% of total latency. This confirms that MLP projections are primary compute bottleneck during decoding, especially for short sequences, surpassing attention-related kernels (K4, K8).

Implication 6: *Frequent GPU idle periods reveal opportunities to enhance system performance and efficiency. Runtime systems can downscale GPU frequency during idle windows to save power or overlap lightweight computations to boost utilization. Moreover, the abundance of short, fragmented kernels underscores the need for improved kernel fusion strategies tailored to on-device LLMs.*

Paged-attention kernel latency. Fig. 11 depicts the average latency of the `batch_decode_paged_kv` kernel over decode steps for four target output token sequence lengths. This kernel is the core operator for paged attention [51]. We observe that its latency grows nearly linearly with decode steps, as each invocation must scan an increasingly large KV cache. The growth is steeper for longer outputs, reaching up to 0.9 ms per token by step 250 in the 256-token case. The right panel compares total GPU idle time for 16- and 256-token generations. Interesting, generating longer sequences reduces idle time from 21% to 12%. These results indicate that long-sequence generation may be ill-suited



(a) Timeline view of kernel executions and data movements within a decode step.

(b) Breakdown of kernel execution time and GPU idle time.

Figure 10: Visualization of kernel execution (KE), data movement (DMV), and GPU utilization in a single decode step. (a) Each kernel (y-axis) is labeled by index, invocation count, and average runtime; gray regions indicate GPU idle periods. (b) Kernel names are annotated with indices; red bar shows the GPU idle proportion during decoding.

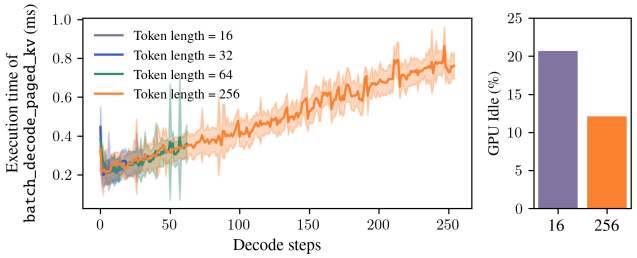


Figure 11: Paged-attention kernel latency grows with decode steps, while GPU idle time drops with longer outputs. Each step generates one token.

for resource-constrained devices due to rising per-step latency. Conversely, short-sequence generations leave significant headroom for optimizing GPU utilization efficiency.

Implication 7: *Among all decode-phase kernels, we observe that* `batch_decode_paged_kv` *is the only one whose latency scales with decode steps. Modeling this trend enables lightweight online prediction of per-step latency, supporting efficient scheduling and improved responsiveness in on-device LLM systems.*

5.3 Limitations

This work offers a first step in characterizing on-device LLM runtime behavior but is limited to a subset of mobile SoCs. Other edge platforms (Jetsons and Intel NPUs) may exhibit different bottlenecks and optimization opportunities. Our compression analysis focuses on post-training 4-bit quantization, a popular yet narrow slice of the broader compression landscape. Techniques like structured pruning, activation sparsity, and hybrid-precision quantization may yield different trade-offs depending on hardware and workload.

6 Related Work

Recent advances in model compression and compiler optimizations have made on-device LLM deployment increasingly feasible [55–57]. Techniques such as quantization [58–61], knowledge distillation [62], and sparsification [63–65] reduce model size and computation while preserving accuracy. ML compilers like MLC [38], LLMFarm [66], TensorRT-LLM [67], and llama.cpp [68] unify LLM inference across heterogeneous edge platforms, via kernel fusion, quantization-aware compilation, and hardware-specific tuning. However, existing evaluations of on-device LLM systems largely focus on end-to-end latency or task-level accuracy, offering limited insight into phase- or kernel-level bottlenecks. Our work complements this direction by introducing a lightweight, online profiler that enables fine-grained decomposition of inference latency, providing deeper visibility into runtime behavior and guiding system-level optimization strategies.

7 Conclusion

In this paper, we present LM-Meter, a lightweight online profiler that provides accurate phase- and kernel-level latency measurements. Using lm-Meter, we conduct empirical studies and uncover actionable insights: prefill is the dominant on-device bottleneck, accuracy-latency trade-offs are highly task- and architecture-dependent, and our Harmonic Quantization score offers a holistic view of quantization impact. Kernel-level profiling also reveals substantial GPU idle time and highlights opportunities for optimization.

Acknowledgments

We thank the reviewers and our shepherd for their insightful comments. This work was supported by funds from Toyota Motor North America.

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[4] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[5] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proc. the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186, 2019.

[7] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[10] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[11] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[13] Stefanos Laskaridis, Stylianos I Venieris, Alexandros Kouris, Rui Li, and Nicholas D Lane. The future of consumer edge-ai computing. *IEEE Pervasive Computing*, 2024.

[14] OpenAI ChatGPT. https://openai.com/index/chatgpt/. Accessed on June 2025.

[15] Google NotebookLM. https://notebooklm.google.com/. Accessed on June 2025.

[16] Haoxin Wang, BaekGyu Kim, Jiang Xie, and Zhu Han. LEAF+ AIO: Edge-assisted energy-aware object detection for mobile augmented reality. *IEEE Transactions on Mobile Computing*, 22(10):5933–5948, 2022.

[17] Haoxin Wang and Jiang Xie. User preference based energy-aware mobile AR system with edge computing. In *Proc. IEEE INFOCOM*, pages 1379–1388, 2020.

[18] Haoxin Wang, Jiang Xie, and Tao Han. V-handoff: A practical energy efficient handoff for 802.11 infrastructure networks. In *Proc. IEEE ICC*, pages 1–6, 2017.

[19] Hongyu Ke, Wanxin Jin, and Haoxin Wang. Carboncp: Carbon-aware dnn partitioning with conformal prediction for sustainable edge intelligence. *arXiv preprint arXiv:2404.16970*, 2024.

[20] 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 *Forty-first International Conference on Machine Learning*, 2024.

[21] NVIDIA Corporation. Nvidia H100 tensor core gpu. https://www.nvidia.com/en-us/data-center/h100/, 2025. Accessed: Feb. 2025.

[22] Soumith Chintala. Meta will have 600k h100-equivalent gpus. https://x.com/soumithchintala/status/1748074223187173724, 2024. Accessed: 2025-02-01.

[23] Health Insurance Portability and Accountability Act of 1996 (HIPAA). https://www.cdc.gov/phlp/php/resources/health-insurance-portability-and-accountability-act-of-1996-hipaa.html#:~:text=At%20a%20glance,Rule%20to%20implement%20HIPAA%20requirements. Accessed on June 2025.

[24] Cong Guo, Feng Cheng, Zhixu Du, James Kiessling, Jonathan Ku, Shiyu Li, Ziru Li, Mingyuan Ma, Tergel Molom-Ochir, Benjamin Morris, Haoxuan Shan, Jingwei Sun, Yitu Wang, Chiyue Wei, Xueying Wu, Yuhao Wu, Hao Frank Yang, Jingyang Zhang, Junyao Zhang, Qilin Zheng, Guanglei Zhou, Hai Li, and Yiran Chen. A survey: Collaborative hardware and software design in the era of large language models. *IEEE Circuits and Systems Magazine*, 25(1):35–57, 2025.

[25] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[26] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proc. the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 81–93, 2021.

[27] Chengquan Feng, Li Lyna Zhang, Yuanchi Liu, Jiahang Xu, Chengruidong Zhang, Zhiyuan Wang, Ting Cao, Mao Yang, and Haisheng Tan. LitePred: Transferable and scalable latency prediction for hardware-aware neural architecture search. In *Proc. 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1463–1477, 2024.

[28] Xiaolong Tu, Anik Mallik, Dawei Chen, Kyungtae Han, Onur Altintas, Haoxin Wang, and Jiang Xie. Unveiling energy efficiency in deep learning: Measurement, prediction, and scoring across edge devices. In *Proc. the Eighth ACM/IEEE Symposium on Edge Computing*, pages 80–93, 2023.

[29] Anik Mallik, Haoxin Wang, Jiang Xie, Dawei Chen, and Kyungtae Han. EPAM: A predictive energy model for mobile AI. In *Proc. IEEE ICC*, pages 954–959, 2023.

[30] Xiaolong Tu, Anik Mallik, Haoxin Wang, and Jiang Xie. Deepen2023: Energy datasets for edge artificial intelligence. In *Proc. Tackling Climate*

Change with Machine Learning: Workshop at NeurIPS 2023, 2023.

[31] Xiaolong Tu, Dawei Chen, Kyungtae Han, Onur Altintas, and Haoxin Wang. Greenauto: An automated platform for sustainable AI model design on edge devices. In *Proc. the 26th International Workshop on Mobile Computing Systems and Applications*, pages 7–12, 2025.

[32] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024.

[33] Android GPU Inspector (AGI). https://developer.android.com/agi. Accessed on June 2025.

[34] Tracing SDK. https://perfetto.dev/docs/instrumentation/tracing-sdk. Accessed on June 2025.

[35] Stefanos Laskaridis, Kleomenis Katevas, Lorenzo Minto, and Hamed Haddadi. Melting point: Mobile evaluation of language transformers. In *Proc. the 30th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 890–907, 2024.

[36] Haolin Chu, Xiaolong Zheng, Liang Liu, and Huadong Ma. nnPerf: Demystifying dnn runtime inference latency on mobile platforms. In *Proc. the 21st ACM Conference on Embedded Networked Sensor Systems*, page 125–137, 2023.

[37] TFLite Model Benchmark Tool. https://github.com/sourcecode369/tensorflow-1/blob/master/tensorflow/lite/tools/benchmark/README.md. Accessed on June 2025.

[38] MLC LLM: Universal LLM Deployment Engine With ML Compilation. https://llm.mlc.ai/. Accessed on June 2025.

[39] TVM: open deep learning compiler stack. https://github.com/apache/tvm. Accessed on June 2025.

[40] LiteRT overview. https://ai.google.dev/edge/litert. Accessed on June 2025.

[41] Monsoon Power Monitor. https://www.msoon.com/specifications. Accessed on June 2025.

[42] Open Neural Network Exchange. https://onnx.ai/. Accessed on June 2025.

[43] Open-source software toolkit for optimizing and deploying deep learning models. https://github.com/openvinotoolkit/openvino. Accessed on June 2025.

[44] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *Proc. International Conference on Machine Learning*, pages 2397–2430. PMLR, 2023.

[45] Smol Models. https://github.com/huggingface/smollm. Accessed on June 2025.

[46] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *arXiv:1803.05457v1*, 2018.

[47] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proc. the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[48] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[49] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *Proc. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[50] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proc. International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[51] 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. the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[52] Jan Hansen-Palmus, Michael Truong Le, Oliver Hausdörfer, and Alok Verma. Communication compression for tensor parallel LLM inference. *arXiv preprint arXiv:2411.09510*, 2024.

[53] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[54] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, et al. SmolLM2: When smol goes big–data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*, 2025.

[55] Xiang Li, Zhenyan Lu, Dongqi Cai, Xiao Ma, and Mengwei Xu. Large language models on mobile devices: Measurements, analysis, and insights. In *Proc. the Workshop on Edge and Mobile Foundation Models*, pages 1–6, 2024.

[56] Daliang Xu, Wangsong Yin, Hao Zhang, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. Edgellm: Fast on-device llm inference with speculative decoding. *IEEE Transactions on Mobile Computing*, 2024.

[57] Jie Xiao, Qianyi Huang, Xu Chen, and Chen Tian. Understanding large language models in your pockets: Performance study on cots mobile devices. *arXiv preprint arXiv:2410.03613*, 2024.

[58] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*, 2023.

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

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

[61] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proc. Machine Learning and Systems*, 6:87–100, 2024.

[62] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. MiniLLM: Knowledge distillation of large language models. *arXiv preprint arXiv:2306.08543*, 2023.

[63] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023.

[64] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.

[65] Xinyin Ma, Gongfan Fang, and Xinchao Wang. LLM-pruner: On the structural pruning of large language models. *Advances in Neural Information Processing Systems*, 36:21702–21720, 2023.

[66] LLMFarm. https://github.com/guinmoon/LLMFarm. Accessed on June 2025.

[67] TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM. Accessed on June 2025.

[68] llama.cpp. https://github.com/ggml-org/llama.cpp. Accessed on June 2025.