PROFINFER: AN EBPF-BASED FINE-GRAINED LLM INFERENCE PROFILER

Bohua Zou [1 2] **Debayan Roy** [1] **Dhimankumar Yogesh Airao** [1] **Weihao Xu** [2] **Binqi Sun** [2] **Yutao Liu** [1]
Haibo Chen [3 4]

ABSTRACT

As large language models (LLMs) move from research to production, understanding how inference engines behave in real time has become both essential and elusive. Unlike general-purpose engines such as ONNX Runtime, today's LLM inference systems offer little operator-level visibility, leaving developers blind to where time and resources go. Even basic questions—is this workload memory-bound or compute-bound?—often remain unanswered. To close this gap, we develop a fine-grained, non-intrusive profiling framework for modern LLM inference engines, with a specific focus on resource-constrained edge devices, exemplified by llama.cpp, but applicable to similar runtime architectures. Built on extended Berkeley Packet Filter (eBPF) technology, our system dynamically attaches probes to runtime functions across multiple layers—without modifying or recompiling the source. It transforms collected traces into rich visualizations of operators, graphs, timelines, and hardware counter trends, exposing how dense inference, Mixture-of-Experts routing, and operator offloading behave in practice. With less than 5% runtime overhead and high profiling fidelity, our framework makes LLM inference both transparent and diagnosable, turning performance profiling into a practical tool for optimization, scheduling, and resource-aware deployment.

1 INTRODUCTION

Large language models (LLMs) are reshaping how we interact with computers by powering conversational agents, writing assistants, multilingual translation services, travel-planning aids, and personalized tutors. Underpinned by transformer architectures that process tokens while computing inter-token attention relationships, LLMs deliver strong natural-language understanding and generation capabilities. Deploying LLM inference on mobile and edge devices rather than relying exclusively on cloud infrastructure opens important opportunities, including offline operation, lower latency, enhanced user privacy, and reduced network dependence (Wang et al., 2023; Apple, 2024).

Enabling on-device LLM inference is far from trivial. Even when employing highly optimized inference engines, the computation and memory demands of LLMs are orders of magnitude greater than those of traditional mobile neural networks. A model may generate text token-by-token, rather than in a single forward pass, and its inference proceeds through distinct phases: a *prefill* phase that is typically compute-bound, and a *decode* phase that is often memory- or bandwidth-bound. Meanwhile, mobile devices impose stringent limits on processor power, memory size, caching behavior, thermal dissipation, and energy budgets. As recent studies highlight (Chen et al., 2025; Li et al., 2025), deploying LLMs on mobile hardware requires careful balancing of model size, memory management, and accelerator usage. In such constrained and heterogeneous environments, accurate and low-overhead performance visibility becomes essential for identifying inefficiencies and guiding optimizations.

Profiling tools are crucial for understanding and optimizing ML workloads: by exposing operator-level timings, memory usage, and threading behavior, they enable developers to pinpoint performance bottlenecks. General-purpose ML inference engines such as ONNX Runtime (Microsoft, 2025) and TensorRT (NVIDIA Corporation, 2024) provide built-in profilers for operator execution time, memory allocation, and thread utilization, aiding optimization in latency- and resource-sensitive settings. Yet for LLM inference engines, especially those targeting on-device and edge scenarios, we observe a striking lack of *fine-grained* and *non-intrusive* profiling support. Existing profilers often require recompilation or runtime instrumentation, offer only coarse-grained metrics such as throughput or token rate, and fail to expose critical dimensions of execution such as dynamic operator graphs, per-thread scheduling, hardware-counter events, and phase-specific behaviors. Besides, some hardware-based profilers typically rely on support from the underlying hard-

[1]Huawei Hilbert Research Center (Dresden), Dresden, Germany [2]Technical University of Munich, Munich, Germany [3]Huawei Central Software Institute [4]Shanghai Jiao Tong University, Shanghai, China. Correspondence to: Debayan Roy <debayan.roy6@huawei.com>.

Proceedings of the 9^{th} MLSys Conference (Industry Track), Bellevue, WA, USA, 2026.

ware and often introduce non-negligible overhead.

While extended Berkeley Packet Filter (eBPF) has recently been used for tracing deep learning performance (Chu et al., 2025; Yang et al., 2025; Craun et al., 2024), it remains largely decoupled from LLM execution semantics. Existing work rarely correlates low-level hardware metrics with high-level operator semantics, nor has it been adapted to mobile or edge environments where runtime overhead and observability constraints are more stringent. Consequently, developers lack the necessary observability to understand how quantization, KV-cache reuse, accelerator offloading, or memory bandwidth constraints affect end-to-end latency and efficiency on-device.

To address these limitations, we present *ProfInfer*, a profiling framework tailored to on-device LLM inference. The design of *ProfInfer* emphasizes three key attributes. First, it offers *fine-grained observability* by capturing every forward pass and operator invocation (e.g., matrix multiplication, attention, and softmax) along with tensor dimensions, operator types, and the dynamically constructed computational DAG. Second, it ensures *non-intrusive instrumentation* through the use of eBPF probes that attach to runtime functions without requiring source modification, enabling easy deployment on Linux-based mobile and edge operating systems. Third, it supports *hardware-counter integration and visual analytics*, collecting per-operator performance monitoring counter (PMC) data, such as cache misses, memory accesses, and thread execution, and presenting them through timeline views, DAG visualizations, and per-operator plots.

We implement *ProfInfer* over the llama.cpp (Gerganov & contributors, 2025) inference engine, with a high-level overview illustrated in Figure 1. Before starting the workload, the *tracer* attaches several probes to the runtime libraries and enables relevant system tracing events. Online, the *tracer* continuously gathers and logs the submitted buffer from each probe handler in the kernel space and conditionally switches off the unnecessary probes. Offline, the *analyzer* parses the collected traces, identifies the computation and the backend types, and then provides three representations of structural and performance metrics at the token, graph, and operator levels. We released the project of *ProfInfer* publicly [1].

The main contributions of this paper are as follows:

- We propose *ProfInfer*, an eBPF-based profiling framework for LLM inference that is fine-grained, non-intrusive, and lightweight, making it suitable for both mobile and edge environments.

[1]https://gitcode.com/openharmony-robot/oh-llama.cpp/tree/main/profinfer

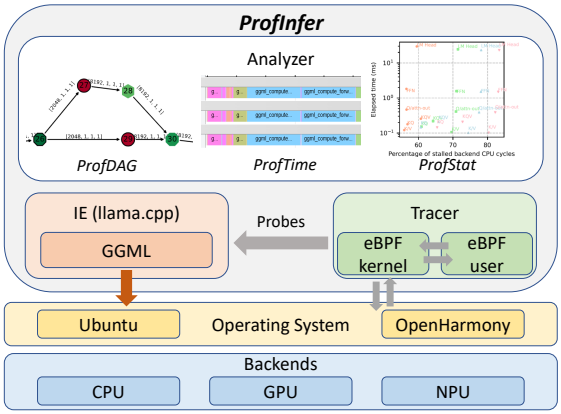


Figure 1. High-level design of *ProfInfer* comprising eBPF-based tracers followed by trace analyzers.

- *ProfInfer* enables comprehensive visibility across the entire inference pipeline—covering each forward pass, computation graph, operator execution, and processor thread, while collecting a rich set of performance metrics through hardware counters.
- It offers intuitive performance analytics, including timeline views, DAG visualizations, and op-level plots that correlate model structure with hardware behavior.
- We demonstrate how *ProfInfer* facilitates analysis of compute and memory bottlenecks, KV-cache effects, workload interference, backend performance divergence, and the dynamic characteristics of MoE models.

The rest of the paper is organized as follows. Section 2 provides the necessary background on LLM inferences and eBPF. Sections 3 and 4 discuss the tracers and analyzers in *ProfInfer*, respectively. Section 5 evaluates the overhead and fidelity of *ProfInfer*, as well as presents the experimental results with valuable insights. Section 6 reviews relevant existing works. Section 7 discusses the portability of *ProfInfer*, its potential application scenarios, and the future directions of this work.

2 Background

2.1 Large language model (LLM)

LLMs are mostly transformer-based models leveraging the self-attention mechanism to capture contextual information (Vaswani et al., 2017). After GPT (Radford et al., 2018) was released, the decoder-only model became the de facto standard architecture for generative tasks and demonstrated superior general intelligence. Recently, an increasing number of LLMs have been open-sourced, and their performance now rivals that of many closed-sourced LLMs.

When examining the architecture of a decoder-only LLM of

the LLaMA family (Grattafiori et al., 2024), we see that it inherits the basic design of GPT, stacking multiple transformer layers with identical structure. Each layer is composed of a self-attention module and a feed-forward network (FFN). In particular, LLaMA differs by adopting the grouped-query attention (GQA) (Ainslie et al., 2023), rotary positional embedding, root-mean-squared layer-normalization (RMS Norm), and SwiGLU activation function. Other families of LLMs, such as Qwen (Bai et al., 2023), Phi (Abdin et al., 2024), and Gemma (Team, 2024), each adopt their own unique architectural design. Apart from the training dataset and maximum context length, the primary distinctions stem from architectural hyperparameters, including the number of layers, hidden dimension, and attention-head count.

	BPF APIs	Description
Kernel space	`bpf_ktime_get_ns`	Get the timestamp
	`bpf_get_current_pid_tgid`	Get pid and tid
	`bpf_get_smp_processor_id`	Get cpu id
	`bpf_probe_read_user(kernel)`	Read user/kernel address space
	`map.lookup`	Query a BPF map
	`map.update`	Update a BPF map
	`map.perf_read`	Read the value of a PMC
	`ringbuf(perf)_submit`	Submit a ring(perf) buffer
User space	`attach_u(ret)probe`	Attach probes when a user-space function is invoked or returned
	`attach_tracepoint`	Attach a kernel trace point
	`open_perf_event`	Create a fd to monitor PMC
	`perf(ring)_buffer_poll`	Poll the submitted perf(ring) buffer

Table 1. A list of `BCC`-offered APIs used in our `BPF C` and `Python` programs. `libbpf` also offers equivalent ones.

2.2 LLM inference and inference engine

2.2.1 *LLM inference*

Most decoder-only LLMs perform inference in an auto-regressive fashion. After processing the prompt in the prefill stage, the LLM generates one token per decoding step (or forward pass) to ensure causality, until it reaches a preset maximum length or a termination token such as `<eos>`. The (real-time) performance of an LLM is commonly evaluated using two key metrics: the time to the first token (TTFT) in the prefill stage, and the decoding speed or period, i.e., tokens per second (TPS) or time per output token (TPOT), during the decoding stage. Both metrics are influenced by the model architecture and the hardware configuration.

By leveraging the sparsity inside LLMs, architectural innovations such as dynamic pruning (PowerInfer (Song et al., 2024)) and mixture-of-experts (MoE) (Lepikhin et al., 2021) can further accelerate on-device inference by activating only a subset of weights at runtime. The variability introduced by these dynamic workloads can also lead to performance fluctuations. For example, in MoE models, the frequency with which the same experts are activated affects runtime memory usage and disk I/O overhead, which in turn impacts inference performance.

2.2.2 *LLM inference engine*

Unlike conventional neural networks that only perform a forward pass, LLM inference also involves tokenization, sampling, and KV cache management. As LLM inference typically requires significantly more compute and memory resources, lightweight designs are often adopted by LLM inference engines (IEs). vLLM (Kwon et al., 2023) utilizes paged attention to optimize KV-cache memory allocation across batches, but it is not suitable for mobile scenarios where the batch size is typically 1. MNN-LLM (Wang et al., 2024) and mlc-llm (MLC team, 2023-2025) are two representative IEs for mobile devices, supporting different model types and hardware backends. The former is a C++-based library that optimizes memory layout and access patterns and also employs mixed-precision quantization. The latter is built on Apache TVM (Chen et al., 2018) and applies operator- and graph-level optimizations during compilation.

ProfInfer mainly focuses on llama.cpp (Gerganov & contributors, 2025), one of the most widely used open-source IEs, which has a bunch of well-known front-ends, like Ollama (Team, 2023), many bindings in other programming languages, and also many derivatives, such as PowerInfer (Song et al., 2024). As a C/C++ based IE, llama.cpp provides versatile functionalities, such as *llama-server*, speculative decoding, and multimodal model inference. Moreover, llama.cpp is built on GGML, a machine learning runtime library that executes graph and operator computations and supports deployment across heterogeneous backends, including CPUs, GPUs, and certain types of NPUs.

2.3 Extended Berkeley Packet Filter (eBPF)

eBPF is a powerful technology that enables running sandboxed programs in the kernel space, thereby extending kernel functionality without requiring modification and recompilation of the kernel source code (Rice, 2023). It was first supported by Linux—introduced in 2014—but later expanded to macOS and Windows. Since it offers avenues to hook into kernel events and system calls, and to attach probes to user-space function calls and returns, it can be used to develop sophisticated system profiling tools (Cassagnes et al., 2020).

In this work, we use two popular frameworks, `libbpf` and `BCC` (BPF Compiler Collection), to implement eBPF programs. `BCC` provides Python bindings and is ideal for rapid prototyping; hence, we use it to implement *ProfInfer* over Ubuntu. However, since OpenHarmony does not support a Python environment by default, we implement *ProfInfer* over it using a C library `libbpf`. In Table 1, we summarize the BPF APIs used in our `BCC`-based implementation of *ProfInfer*'s tracer, as detailed in Section 3.

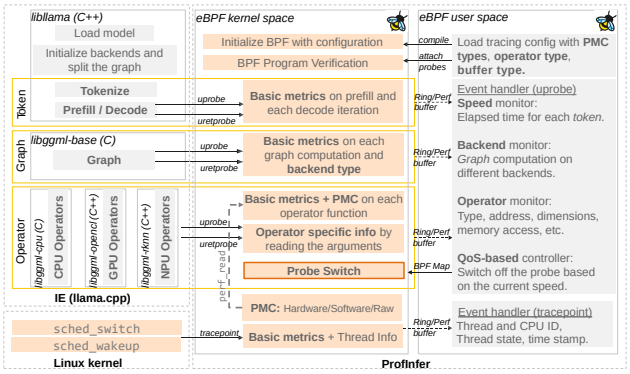


Figure 2. An operational overview of *ProfInfer*'s tracer for LLM inferences over llama.cpp. Basic metrics refer to type of the probe, time stamp, thread ID and CPU ID.

Probed functions	Type	Level	Traced information
`llama_decode`	u(ret)probe	Token	batch size
`ggml_backend_graph_compute_async`	u(ret)probe	Graph	backend type
`ggml_compute_forward`	u(ret)probe	OP(CPU)	tensor info, PMC
`ggml_cl_compute_forward`	u(ret)probe	OP(GPU)	tensor info
`ggml_rk_compute_forward`	u(ret)probe	OP(NPU)	tensor info
`sched_switch`	tracepoint	kernel	thread IDs
`sched_wakeup`	tracepoint	kernel	thread IDs

Table 2. Probes in *ProfInfer*. The timestamp, thread ID, and CPU ID are logged by all probe handlers.

3 LLM TRACER IN PROFINFER

Figure 2 gives an overview of *ProfInfer*'s tracer. A part of the tracer operates in user space while the other part runs in kernel space. Further, it attaches probes to several functions of llama.cpp to enable tracing LLM inferences at different granularities. The probe handlers are implemented to collect runtime information about an inference facilitating both architectural and performance analyses. The remainder of this section will explore the implementation and various features of the tracer in greater detail.

3.1 *ProfInfer* in user space

ProfInfer implements a user-space application that is primarily responsible for compiling, launching, and verifying the BPF program. It specifies the compilation flags—according to a user-defined configuration file—for the BPF program, thereby filtering out unnecessary information to reduce tracing overheads. For example, a flag can be set to retrieve tensor dimensions when needed for an analysis. The user-space application further attaches probes (uprobes, uretprobes, and tracepoints) to target functions. It subsequently polls for the buffers submitted by the probe handlers in the kernel space. When it receives a buffer, it will asynchronously process the data based on the probe type in the event handler. It also conditionally writes back into the BPF maps to control the probes and the information to be traced. When it detects a degradation in the inference performance, i.e., the overall decoding speed is under the Quality of Service (QoS) requirement, it can disable certain tracing features to reduce the overhead and improve the speed.

3.2 *ProfInfer* in kernel space

ProfInfer implements several probe handlers that run in the kernel space. Each of them traces the PID of the thread executing the probed function, the CPU where the thread is running, and the timestamp when the handler is invoked. Further, some of the handlers examine the arguments of the probed function, and by parsing them, they can infer the state of the inference. These arguments are typically C structs or pointers to C structs containing information about the current operator, including its type, input and output tensors, among others. To log the traced data, a handler submits a *ring buffer* or a *perf buffer* to *ProfInfer*'s user-space application, where the former can enhance efficiency but at the cost of unreported missing events.

3.3 Multi-granularity eBPF tracing

Considering that llama.cpp is an open-source IE, we traverse its logic to identify specific functions to probe based on our requirements of examining its performance at different granularities, i.e., with respect to each token, computational graph, and operator. Nevertheless, we note that it is a vast and complex project, which makes this non-trivial. Besides the IE, we trace a few kernel events that are relevant to thread scheduling. Table 2 summarizes the important details of the probes we have designed and implemented.

3.3.1 Token-level tracing

Profiling an inference running over llama.cpp at the token level covers the prefill stage and each decoding iteration. In libllama, `llama_decode` is the generic function that is invoked to generate the next token both during prefill and decode stages. Embeddings of the tokenized input are passed to it. By attaching uprobe and uretprobe to this function, we obtain the timestamps of its invocation and the corresponding return; the difference between the timestamps gives TTFT during prefill and TPOT during decoding. In this work, we use these values during runtime to dynamically adjust the tracing overheads based on the decoding speed QoS requirement (e.g., 5 tokens/s) by switching the probes on or off. Besides, future work is to use them to adjust resource allocations to inferences based on timing requirements, especially when there are parallel workloads.

3.3.2 Graph-level tracing

To manage heterogeneous backends, llama.cpp adopts graph partitioning and a backend scheduling strategy within the GGML library. That is, if multiple backends are selected to run an LLM inference, several computational graphs are constructed during the initialization phase such that all operators in a graph run using the same backend. For example, if a certain number of transformer layers are to be offloaded to a GPU (e.g., using a CUDA backend) due to its restricted memory capacity, two or more graphs are created, depending on the selected layers for offloading. To compute the execution time of each graph in a forward pass, we attach uprobe and uretprobe to the function `ggml_backend_graph_compute_async` and collect timestamps from their handlers. Such information can be useful to discern as well as optimize the graph partitioning and scheduling heuristic in llama.cpp. Further, the uprobe handler extracts the value of `guid`, a 16-digit backend identifier, from a function argument.

3.3.3 Operator-level tracing

For a given backend, we identify the function that executes the logic to identify the type of an operator and, correspondingly, call its low-level implementation. Currently, we support CPU and OpenCL backends, along with a self-developed Rockchip NPU backend, while extending to other backends remains future work. The identified function in each of our target backends is listed in Table 2. We attach probes to the function to collect timestamps of its invocation and return, thereby obtaining the execution time of each tensor operation. We also parse `ggml_tensor`, the second function argument and a C struct. By traversing this data structure using eBPF APIs, we can extract information about the name and type of an operation, e.g., `ffn_out-0` and `GGML_OP_MUL_MAT` (the output matrix multiplication of the feedforward network in the first transformer layer), as well as the pointer address and dimension of source and target tensors. Such trace data enable to construct the computational graph and operator schedules as well as analyze performance of each operator during an LLM inference.

3.3.4 Scheduler tracing

Besides user-space tracing, we hook into two tracepoints in Linux scheduler, i.e., `sched_switch` and `sched_wakeup`. The handlers of these tracepoints can read the state of a thread, i.e., running, runnable, or idle. We apply filters to only trace the events related to the inference threads otherwise the overheads might become unacceptable as there can be numerous such events. This tracing feature helps to identify interfering parallel workloads as well as to study their impact on an inference.

PMC	Type	Description	Unit
l3d_cache_refill	per-core	Read from memory (A76)	64B
mem_access_wr	per-core	Write into memory(A76)	16B
major-faults	software	Major page faults	1 Page
cycles	hardware	CPU cycles	1 Cycle
idle-backend-cycles	hardware	stalled CPU cycles at backend	1 Cycle

Table 3. A summary of the PMCs to reflect memory access, computational efficiency, and disk IO efficiency.

3.4 Enabling PMCs

Considering that PMCs offer low-level insights into CPU behavior with respect to computations and memory accesses, they are essential for profiling and optimizing software performance. Hence, in *ProfInfer*, we offer the possibility for opening and reading a PMC at the operator level. In Table 3, we list a few meaningful PMCs that we can trace.

Let us consider an example where we want to determine the amount of data fetched from DRAM for an operator. Per-core `l3d_cache_refill` is a useful PMC for that (Pradhan et al., 2025). We use `open_perf_event` in the user-space program to open and configure a file descriptor for each thread and via it we can read `l3d_cache_refill` counter per thread. Now, in the handlers of uprobe and uretprobe to `ggml_compute_forward`, we read the counter using `perf_read`. The difference between the two values gives how many times L3 data cache was refilled with 64 B of data by an inference thread for the operator. Summing these values over all inference threads can indicate the total amount of data fetched from DRAM for the operator.

We note that such a PMC-based analysis can assist in identifying resource under-utilization or boundedness and other performance bottlenecks for each operator.

3.5 Tracing expert activations in MoE models

When running an inference based on an MoE model, different experts can be activated in the FFN of a transformer layer during each decoding iteration. Expert activations are determined by a gating mechanism, which is usually implemented by a *softmax* or a *sigmoid* operator followed by a *top_k* operator. The result is a tensor of length k comprising the IDs of k activated experts. Based on this understanding, we have attached a uprobe to the function `ggml_compute_forward_mul_mat_id` where the probe handler reads the k expert IDs by traversing the third source `ggml_tensor` using two-level pointer dereferencing. We note that the DRAM of a mobile device is typically insufficient to accommodate all available experts. Hence, when an activated expert is not in the DRAM, it has to be fetched from the storage, which increases operator's execution time. That is, traced information related to expert activations enables accurate analysis of performance data at the operator level.

4 LLM TRACE ANALYZERS IN PROFINFER

Taking an LLM and the hardware configuration as input, *ProfInfer* generates the raw tracing data and then, through further analysis, produces the following three types of views.

4.1 Profiled DAG (ProfDAG) view

To reduce the model's memory footprint, llama.cpp compresses it based on the `GGUF` format, which excludes certain structural information. However, after analyzing the trace data, ProfDAG view can illustrate the topology of the inference workload in a DAG along with the profiling results for each operator. The detailed process of ProfDAG retrieval is presented in Algorithm 1.

The algorithm first groups the raw data by thread ID, then isolates the target decoding iteration and retains events only concerning operator-level function calls and returns (lines 1-4). Subsequently, it iterates over each operator-start or function-call event (i.e., with a step of 2), computes different metrics and assigns them to a node in the DAG together with other information about the operator (lines 5-14). The execution time of an operator is calculated as the difference between the first invocation and the last return of the operator functions across all threads. Further, the PMC difference is calculated on each thread and summed up. The type, name, and dimension of the operator inside the $'info'$ help identify each operator, while the execution order reflects the topological sort generated by the llama.cpp.

The structural information is obtained by iteratively querying the source tensor addresses for each operator (lines 15-20) and matching them with the addresses of the outputs of previously added operator nodes in the graph. If a match is not found, the source is a constant tensor—typically a weight matrix—that is not an output of any operator. For each identified dependency between two nodes, we add a directed edge to the graph. A graph processing package can help generate and plot the graph, e.g., networkx (Hagberg et al., 2008). Figure 3 depicts a ProfDAG of LLaMA3.2-1B-F16 running with two Cortex-A76 CPU cores on Orange Pi 5 Ultra where the memory bandwidth consumption of each operator is indicated.

4.2 Timeline (ProfTime) view

To enable visualization of the temporal dynamics of an LLM inference, *ProfInfer* can generate a timeline view—ProfTime—showing token-level, graph-level, and operator-level activities, as well as their scheduling across different processor threads. For instance, ProfTime can show how the computational graph is partitioned and scheduled over different backends.

To reveal the underlying scheduler behavior, *ProfInfer* captures all thread IDs concerning the inference. Analyzing `sched_switch` and `sched_wakeup` events, we identify the next state of a working thread (Abaza et al., 2024). If a working thread is switched out with `prev_state` being 0, it turns "idle". However, if a working thread is woken up or it is switched out with `prev_state` being 1, the thread is preempted and "runnable". A working thread that is switched in is considered "running". Also, the CPU ID is appended to each event's name. By meaningfully converting the raw trace data into the Chrome Trace Event Format (tra, 2025), ProfTime can visualize the timeline with the help of Perfetto (Google LLC, 2025).

Algorithm 1 ProfDAG Retrieving.

Input: df_{raw}: a table with all raw buffer entries
n_{tar}: target decoding iteration
Output: $graph$: a ProfDAG given of LLM

1 $dfs\_ops \leftarrow \{\}$;
foreach df_{tid} *in* $df_{raw}.groupby(\texttt{tid})$ **do**
2 $df_{tid} \leftarrow df_{tid}.sorted(by = timestamp)$;
$df\_ops^{iter}_{tid} \leftarrow df_{tid}[func == op \wedge n_{iter} == n_{tar}]$
$dfs\_ops \leftarrow dfs\_ops \cup \{df\_ops^{iter}_{tid}\}$;
3 $graph \leftarrow DAG()$
$df\_ops \leftarrow dfs\_ops[0]$; `// One thread's result`
4 **for** $i \leftarrow 1$ **to** $len(df\_ops)$ ***step*** *2;* `// At start`
5 **do**
6 $op_{cur} \leftarrow df\_ops['addr']$
$op_{cur}['ts'] \leftarrow get\_end\_to\_end\_time(dfs\_ops, i)$;
$op_{cur}['pmc'] \leftarrow sum\_pmc\_diff(dfs\_ops, i)$
$op_{cur}['info'] \leftarrow get\_op\_info(dfs\_ops, i)$
$op_{cur}['order'] \leftarrow i$;
$graph.add\_node(op_{cur})$;
foreach op_{pre} *in* $dfs\_ops[0]['src']$ **do**
7 **if** op_{pre} *not in* $graph.nodes$ **then**
8 $graph.add\_node(op_{pre})$;
9 $graph.add\_edge(op_{pre}, op_{cur})$;
10 **return** $graph$;

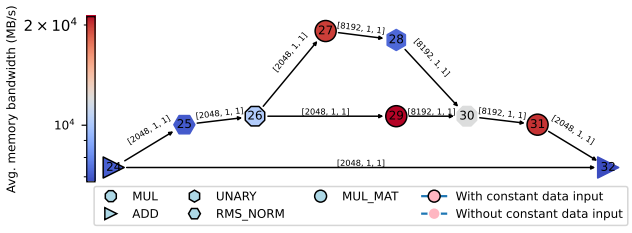


Figure 3. Graphical view of an FFN for LLaMA-3.2-1B, in terms of memory bandwidth.

Device	SoC	RAM	OS
Orange Pi 5 Plus	RK3588	32G LPDDR4X	OpenHarmony 5.1.0
Orange Pi 5 Ultra	RK3588	8G LPDDR5	Ubuntu 22.04
RUBIK Pi 3	QCS6490	8G LPDDR4X	Ubuntu 24.04

Table 4. List of test devices and their configurations.

Backend	Version
CPU	b4743
OpenCL (Adreno GPU)	b6362
CLBlast (ARM Mali GPU)	b3065
Rockchip NPU	forked from b4984

Table 5. List of the test versions of llama.cpp.

4.3 Statistical (ProfStat) view

To analyze the key characteristics that affect inference performance and to uncover the correlation among them, we propose ProfStat view in three dimensions.

- *Across tokens:* By varying the number of tokens in the prompt and the generated text, we analyze the variation in TTFT (prefill time) and TPOT (i.e., how it evolves with decoding iterations). ProfStat also helps to pinpoint operators primarily contributing to the variations.
- *Per operator type:* Fixing an operator type, e.g., matrix multiplication (MatMul), the performance also depends on the dimensions of the input and output tensors, which are determined by the model and the prompts. ProfStat's depiction of the PMC values and other profiled information can help identify the performance bottlenecks of a certain type of operator.
- *Across experts:* For MoE models, ProfStat plots the activated experts across decoding iterations. This can help to derive meaningful insights with respect to expert reuse.

5 Evaluations

We evaluate *ProfInfer* on three devices listed in Table 4. While *ProfInfer* is developed using `libbpf` on the Orange Pi 5 Plus due to a lightweight design of OpenHarmony, `BCC` is used on other devices that have more features. Table 5 further lists the different versions of llama.cpp that we have run to evaluate different backend.

5.1 Overhead analysis

Overhead of *ProfInfer*. For a reliable overhead analysis, we select three popular LLMs with *FP16* precision, namely LLaMA3.2-1B, Qwen2.5-1.5B, and Gemma2-2B. We consider three different lengths of generated text, i.e., 50, 100, and 200 tokens. We run inferences over each model five times for a given configuration. To ensure the reproducibility of the results, a sufficient sleep period is introduced between two consecutive experiments. Additionally, we use *bpftool* to measure the cost of each probe. Table 6 summarizes the relative decrease in the decoding speed due to *ProfInfer*. Further, it lists the average CPU load generated by the probes with respect to the runtime. We observed that the load is always 100% on each CPU during an inference; hence, when using n inference threads, the average CPU load of the probes on each core is calculated as $\sum(t_{probe})/(t_{runtime} \times n)$. Also, we measure temperature and power using the internal temperature sensor and an FNB58 USB power meter (FNIRSI, 2026), respectively, and report the *ProfInfer*-induced power, energy, and thermal overheads in Table 6. We note that the impact of *ProfInfer* on the prefill speed is negligible.

As shown, *ProfInfer* supports three major control flags that are configured in the initialization phase. *Str.* refers to tracing the address, dimensions, and also source tensors of each operator. *PMC* refers to reading the PMC value at each probe handler. *P-buf.* means using the perf buffer to be aware of the missing events, while the ring buffer is chosen for a lower memory overhead. The average CPU load of the probes is basically negligible. In BCC, the average degradation in the decoding speed can vary from 1.63% to 3.67%. As a C-based library, libbpf has the lowest speed drop of 1.19%, while certain features are still not supported. Besides, the results are collected without switching off any type of operator. When tracing only at the token and graph level, the speed decrease is only 0.1%. The average power has minimal overhead, while the additional energy consumption is mainly due to post-processing. Thermal overhead is always within 2 °C. In terms of memory footprint, we use only 9 to 11 BPF maps, each with 4KB of blocked memory. The peak Resident Set Size (RSS) ranges from 102 MB to 172 MB, while the trace throughput varies from 0.784 MB/s to 2.16 MB/s.

Overhead of existing ML profilers. llama.cpp has a built-in function called `ggml_graph_dump_dot` that writes the computational graph representing an LLM into a *.dot* format file. The output contains only the architectural information of a model without any performance data. This functionality involves making one forward pass through the model; however, the time for that pass is *13%* longer than when it is not used.

ONNX Runtime (ONNX Runtime developers, 2021), a popular ML library, also provides a profiling tool that produces a performance overhead of 8% with respect to the model's execution time for a typical convolutional neural network. This tool has also been recently upstreamed to ONNX Runtime-GenAI, which is a dedicated software stack for LLM inferences. Since *FP16* models are not supported by the stack with the CPU backend, we use the same LLMs as reported

Plat.	Str.	PMC	P-buf.	Rel. Dec. (%)	CPU load P. (%)	Power. Inc. (W)	Energy Inc. (J)	Temp. Inc. (°C)
BCC	✓	✓	✓	3.67 ± 1.21	0.70 ± 0.01	0.06 ± 0.29	19.96 ± 20.15	1.22 ± 0.62
	✓	✗	✓	3.47 ± 1.37	0.69 ± 0.01	0.03 ± 0.29	18.69 ± 19.77	0.91 ± 0.77
	✗	✓	✓	3.30 ± 1.27	0.66 ± 0.00	0.01 ± 0.30	17.82 ± 19.53	0.66 ± 0.75
	✓	✓	✗	2.71 ± 1.09	0.62 ± 0.01	0.05 ± 0.28	18.22 ± 19.90	1.28 ± 0.72
	✓	✗	✗	2.42 ± 0.95	0.62 ± 0.00	0.03 ± 0.30	17.12 ± 19.47	0.77 ± 0.60
	✗	✓	✗	2.31 ± 0.92	0.63 ± 0.00	0.04 ± 0.29	17.27 ± 19.31	0.95 ± 0.65
	✗	✗	✓	2.26 ± 0.46	0.63 ± 0.00	0.00 ± 0.35	18.18 ± 22.59	0.83 ± 0.70
	✗	✗	✗	1.63 ± 0.66	0.61 ± 0.00	0.00 ± 0.34	17.79 ± 22.41	0.85 ± 0.76
libbpf	✓	✗	✗	1.46 ± 0.3	0.41 ± 0.00	0.05 ± 0.08	2.18 ± 8.10	0.86 ± 0.65
	✗	✗	✗	1.19 ± 0.18	0.40 ± 0.00	0.03 ± 0.09	2.85 ± 7.31	0.31 ± 0.90

Table 6. Overhead analysis with 2 and 4 Cortex-A76 cores on Orangepi 5 Pro (BCC) and Orangepi 5 Plus (libbpf) on 3 *FP16* LLMs. *Str.*: parsing the operator structure. *PMC*: enabling PMC reading. *P-buf.*: using perf buffer. *Rel. Dec.*: the relative decrease of the decoding speed in percentage. *CPU load P.*: the average CPU load of the probe handlers in percentage. *Power Inc.*, *Energy Inc.*, and *Temp. Inc.* correspond to the overhead of average power, end-to-end energy, and the temperature rise accordingly.

IE	Overhead		Supported features		
	Prefill (%)	Decode (%)	Str.	PMC	Thread
ProfInfer	0.25 ± 0.62	3.60 ± 0.80	✓	✓	✓
ONNX	0.46 ± 0.70	8.91 ± 1.94	✗	✗	✗

Table 7. Comparison between *ProfInfer* and ONNX Runtime in terms of performance overhead and supported features. While the flags, Str. and PMC, are defined in Table 6, Thread indicates if the profiler traces all inference threads.

before but with 4-bit quantization. ONNX profiler only supports operator-level tracing on the main thread, i.e., it is only possible to obtain the runtime of each operator. Accordingly, we disabled the *Str.* and *PMC* flags in *ProfInfer* for a fair comparison. As shown in Table 7, *ProfInfer* incurs a slightly higher overhead for 4-bit quantized LLMs than for *FP16* ones, because the trace volume per token remains unchanged despite the shorter inference time. Nevertheless, it is still much lighter than the ONNX profiler.

5.2 Fidelity assessment.

The accuracy of timestamps and PMC values can be affected by the latency experienced by uprobes and uretprobes during transitions between user space and kernel space. However, such microsecond-scale latency typically has a limited impact on the fidelity of operator-level tracing. *bpftime* adopts a user-space-only uprobe implementation for higher timestamp accuracy. Migrating *ProfInfer* to *bpftime* is left as future work. Besides, when *Str.* is enabled, the intensity of user-space buffer processing increases, which may lead to a small fraction of events—about 0.4%˜0.6%—getting lost. Since the structural information of an LLM is deterministic, we only need to enable the *Str.* flag offline, use as few threads as possible together with the perf buffer, and repeatedly run *ProfInfer* in a relatively idle environment until no samples are lost.

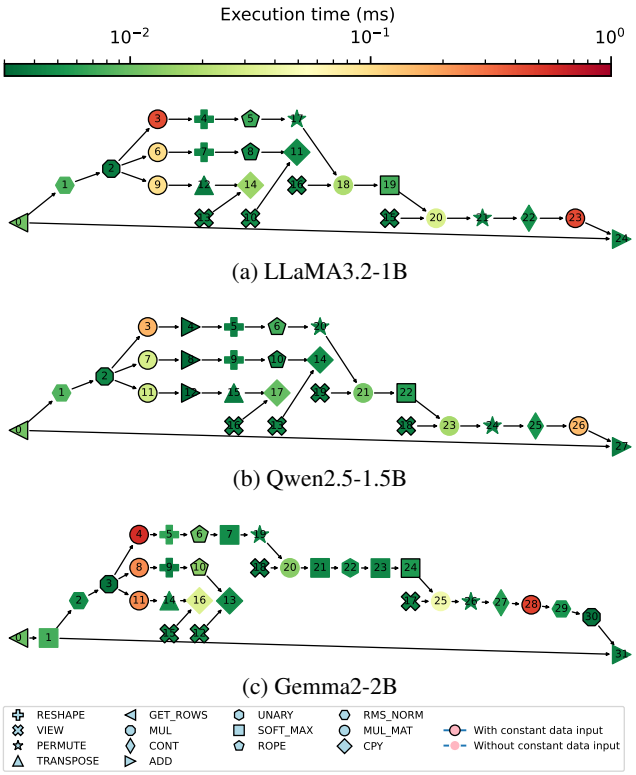
5.3 Profile-driven insights

5.3.1 ProfDAG visualization

We apply *ProfInfer* to obtain ProfDAG of the aforementioned three LLMs. Using them, we infer the following: (i) The differences in the self-attention implementation of them can be observed in Figure 4. Compared to LLaMA3.2-1B, we see that inside one transformer layer, Qwen2.5-1.5B has an ADD (e.g., 4, 8, or 12 denoted by ▷) after three MUL_MAT (i.e., 3, 7, and 11 denoted by ○). Further, Gemma2-2B has six more operators compared to LLaMA3.2-1B, including three SOFT_MAX (□) and one each of UNARY (⬡), RMS_NORM (⬡), and MUL (○). (ii) ProfDAG shows the execution order of tensor operators during an LLM inference. Here, we see that MUL_MAT 3, 6, and 9 in LLaMA3.2-1B do not execute consecutively despite having the same intermediate tensor as input. A similar observation can be made for Qwen2.5-1.5B and Gemma. (iii) In all three ProfDAGs, we see that two MatMuls (e.g., MUL_MAT 4 and 28 in Gemma2-2B) are the heaviest operations in self-attention. Also, the same operation takes longer in LLaMA3.2-1B (MUL_MAT 3) and Gemma2-2B (MUL_MAT 4) compared to Qwen2.5-1.5B (MUL_MAT3). In Figure 4, the color of a node conveys the execution time; however, any other metric (e.g., Figure 3 shows average memory bandwidth utilization) can be visualized as well.

5.3.2 ProfTime visualization

We run an inference based on LLaMA3.2-1B-F16 over llama.cpp with 4 CPU threads on Orange Pi 5 Ultra. *ProfInfer* generates a ProfTime (timeline) visualization of it, using which we can infer the following: (i) Figure 5 shows that llama.cpp runs tensor operations one after the other. However, one tensor operation, e.g., MatMul, can be run by multiple threads. That is, it exploits intra-operator parallelism, but there is no inter-operator parallelization. (ii) It



(a) LLaMA3.2-1B

(b) Qwen2.5-1.5B

(c) Gemma2-2B

Figure 4. Architecture differences in the self-attention of different LLMs.

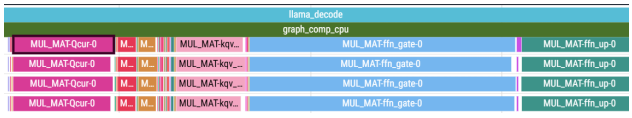
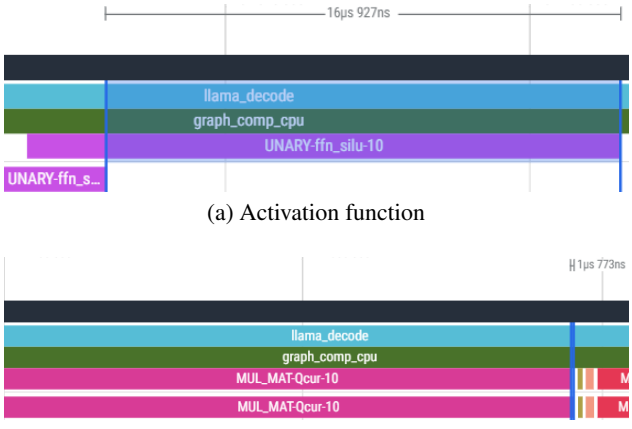


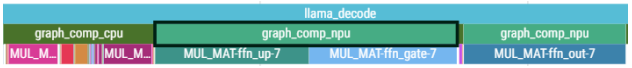
Figure 5. Timeline view in Perfetto. Only intra-operator parallelism is observed. (LLaMA3.2-1B-F16 with 4 threads.)



(a) Activation function

(b) Matrix multiplication

Figure 6. (Un)balanced operator load across threads.



(a) Rockchip NPU backend (graph-level).

(b) CLBlast backend for ARM Mali GPU (operator-level).

(c) OpenCL backend for Adreno GPU (graph-level and kernel-level observability combined with built-in OpenCL profiling).

Figure 7. Timeline view of graph and operator execution on different backends.

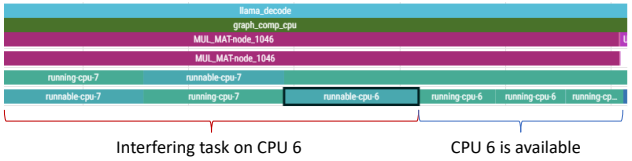


Figure 8. Timeline view of the thread and operator tracing of running LLaMA3.2-1B on 2 Cortex-A76 cores with a interfering task on one core.

can be further observed in the extracted segment that MatMuls dominate other operations, which holds for the entire inference as well, i.e., more than 97% of TTFT and TPOT is spent on MatMuls.

Figure 6 demonstrates that the computational load is not uniformly distributed among inference threads for all operators. Here, LLaMA3.2-1B-F16 is run by two threads. In (a), we see that most computations in the activation function are performed by one thread while the other is idle for almost 80% of the operator's runtime. However, (b) shows that the computations in a MatMul can be distributed almost equally between two threads.

Figure 7 demonstrates ProfTime's capability to show how LLM inferences run across multiple backends. (a) and (c) shows graph partitioning in llama.cpp, where an inference is decomposed into multiple computational graphs that run on different backends, i.e., CPU and Rockchip NPU backends in (a) and CPU and OpenCL (for Adreno GPU) backends in (c). Here, the backend is determined at the graph level. (b) shows nested backends, i.e., the CLBlast backend is implemented under the CPU backend. CLBlast backend is at the operator level, e.g., we see that when a MatMul is encountered by the CPU backend, it offloads the operation via CLBlast to the ARM Mali GPU on Orange Pi 5 Ultra. In all three cases, ProfTime shows the exact backend at the graph level. (a) and (b) further shows the operator's runtime directly from our tracing results, while for OpenCL, we

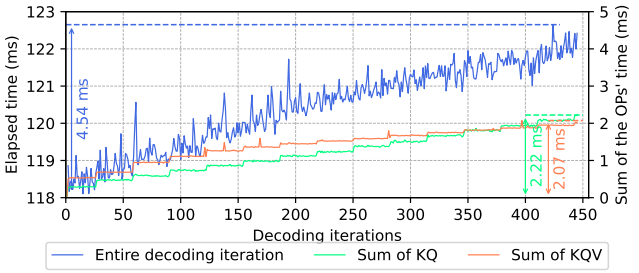


Figure 9. The elapsed time for generating one token and the sum of the *KQ* and *KQV* across layers in each decoding iteration. The model is LLaMA3.2-1B, which runs with 2 threads. It outputs 450 tokens. The increase in the iteration time is caused by *KQ* and *KQV*.

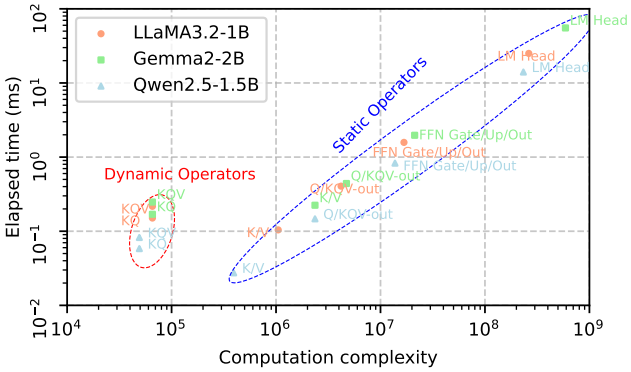


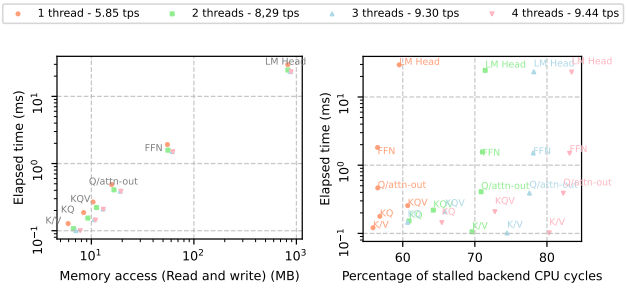
Figure 10. The execution time of the operator matrix multiplication in the first decoding iteration with 2 Cortex-A76 CPUs as the backend. The x-axis represents the computational complexity, i.e., $\mathcal{O}(M \times N \times K \times H)$, given a matrix multiplication $A_{(M,K)} \times B_{(K,N)}$. KQ and KQV also introduce the dimension of attention heads H. M is 1 in the decoding stage.

need to turn on the flag `GGML_OPENCL_PROFILING` to enable the built-in profiling tool and combine it with our results. The reason is that the OpenCL backend selects the OpenCL kernels for each operator and enqueues them into a command queue (CQ), from which the kernels are dispatched asynchronously.

ProfTime can pinpoint if an operator is delayed due to interference from a high-priority task, as shown in Figure 8. Here, two inference threads are bound to CPUs 6 and 7, while the interfering thread is running on CPU 6. We see that the two threads share CPU7 while CPU6 is occupied. After CPU6 becomes available, both threads run simultaneously on the two CPUs. The experiment shows how ProfTime can help debug unexpected delays in the inference.

5.3.3 *ProfStat visualization*

To analyze the performance across the generated tokens, we process the trace data obtained while running the inference



(a) Memory access (b) Backend bound

Figure 11. Memory access and stalled cycle ratio of matrix multiplications in the decoding stage with different numbers of thread settings.

with the CPU backend. Note that this configuration offers a decoding speed very close to the maximum obtained across all different backend combinations that we have tried in this work. In the decoding stage, the KV-cache management reduces computations, resulting in a stable decoding throughput. However, as the KV cache grows, the elapsed time for each iteration (TPOT) increases, as shown in Figure 9. After analyzing all operators, we have identified *KQ* and *KQV* operators to account for most of this change. Also, we observe a staircase-like increase in the execution times of these operators as the token count (i.e., the context length) grows.

Looking closely at all MatMul operators in a single decoding iteration, we observe that they are all matrix–vector multiplications, and some of them (*KQ* and *KQV*) involve broadcasting across the attention-head dimension. Except for *KQ* and *KQV*, the runtimes of others are proportional to the computational complexity, as shown in Figure 10. Qwen2.5-1.5B has a slightly lower intercept and thus a faster speed for an operator with the same dimension, which is possibly caused by the smaller hidden dimension (1536).

As decoding is memory-intensive, we read two per-core counters of Cortex-A76 CPUs to calculate the amount of data transferred between the last-level cache and main memory. Figure 11a illustrates the elapsed time of each MatMul with respect to the memory access. We observe that increasing the number of threads improves performance significantly only up to two threads. Beyond that point, additional threads provide only marginal speedups while pushing the workload toward the memory-bandwidth bottleneck and introducing unnecessary cache misses. Figure 11b also shows the ratio of stalled backend cycles with respect to the total CPU cycles. We see that four threads can cause more than 80% of stalled cycles, which is a waste of resources. From these observations, we can conclude that the MatMul operations scale linearly and, therefore, the decoding speed of a dense model becomes predictable given the hyperparameters.

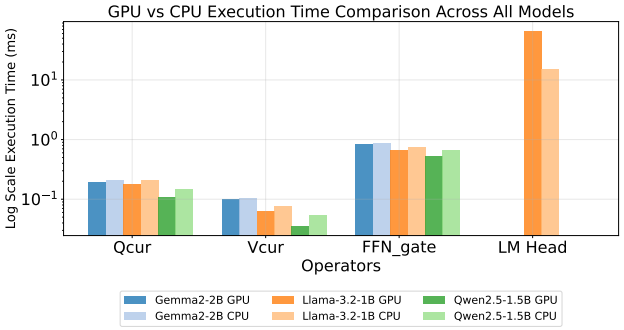


Figure 12. The comparison of matrix multiplication operators of the GPU and CPU on Rubik Pi. Both settings are at their peak decoding speeds.

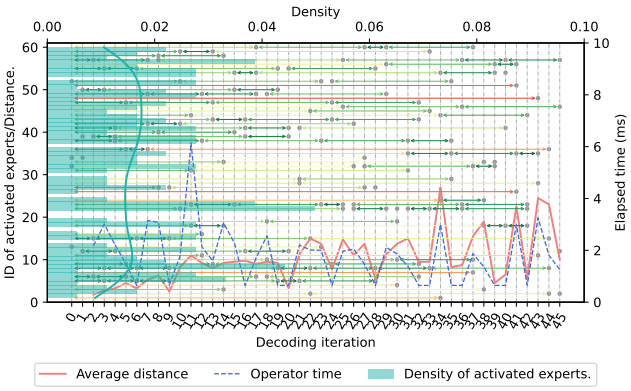


Figure 13. The activated experts for the *ffn_moe_up-0* operator in Qwen1.5-MoE-A2.7B-Q4 (60 experts in total with 4 activated experts each time). Lower x-axis: Decoding iterations. Upper x-axis: Density distribution of the activated experts. Left y-axis: Expert IDs. Right y-axis: Elapsed time of the operator. We compute the average distance between the current decoding iteration and the iteration when each activated expert was last used.

With respect to the prefill stage, existing works have shown performance improvements by exploiting parallelism through heterogeneous backends (Song et al., 2024; Xu et al., 2025). On CPUs, certain libraries can better exploit cache locality for MatMuls, such as BLIS (Van Zee & van de Geijn, 2015). Using BLIS on four Cortex-A76 cores doubles the prefill speed compared to the default CPU backend. By reading l3d_cache_refill and mem_access_wr PMCs and analyzing the results, we find that BLIS reduces memory access by 75% relative to the default configuration.

When offloading the layers to the accelerators, *ProfInfer* can provide a fine-grained comparison on different backends. Taking the Adreno GPU in Rubik Pi as an example, Figure 12 depicts the execution time of several static MatMuls on both CPU (obtained peak performance using four threads) and GPU corresponding to the three LLMs with 4-bit quantization. For Gemma2-2B and Qwen2.5-1.5B, the MatMul in *LM Head* cannot be offloaded due to the size limit of OpenCL kernels. We observe that the GPU outperforms the CPU only in a certain range of dimensions. In the LM Head where the matrix size is very large, there is a dramatic drop in GPU performance. Thus, regardless of the data transfer between CPU and GPU, selective operator offloading based on the tensor dimension can potentially accelerate the inference speed.

Figure 13 shows the profiling result of a MatMul in an FFN layer of an MoE model, Qwen1.5-MoE-A2.7B (Bai et al., 2023). The model comprises 60 experts at each FFN layer and activates 4 of them each time. Even with 4-bit quantization, it still requires 8.9 GB of storage, which cannot be fully loaded into the main memory of the Orangepi 5 Ultra and is only runnable by using `mmap`. The lower x-axis shows the decoding iterations, while the left y-axis represents expert IDs. The top x-axis indicates the density distribution of the activated experts over iterations, where we plot the frequency with which each expert is activated. The right y-axis represents the elapsed time. Inside the figure, we annotate the distance between two consecutive activations of an expert and calculate the average distance of the four experts in the current iteration. After a few iterations, we observe that the operator time is approximately proportional to the average distance. This indicates that the long-distance experts are likely to be evicted from the main memory. By monitoring major page faults and memory accesses, we conclude that the bottleneck in MoE model inferences lies in disk I/O rather than memory bandwidth.

6 Related Works

Built-in profilers in general-purpose ML frameworks. Profiling and observability are critical for identifying performance bottlenecks in machine learning inference engines. Traditional frameworks such as TensorFlow Lite (TensorFlow Team, 2024), ONNX Runtime (Microsoft, 2025), and TensorRT (NVIDIA Corporation, 2024) offer built-in profilers to report operator execution time, memory allocation, and threading behavior. Recent LLM inference engines, such as vLLM (Kwon et al., 2023) and TensorRT-LLM (NVIDIA Corporation, 2025), optimize batching and memory scheduling for high-throughput generation. However, these profilers are typically intrusive, *i.e.*, requiring runtime flags or compilation options. Moreover, their observability features remain coarse-grained, typically exposing aggregate throughput, token rate, and latency.

eBPF-based profiling and system tracing. Recent advances leverage eBPF for performance analysis in deep learning systems. *eInfer* (Chu et al., 2025) demonstrates fine-grained tracing for distributed LLM inference using eBPF to collect runtime call graphs without source modifi-

cation. Similarly, *eGPU* (Yang et al., 2025) extends eBPF programmability to GPUs, and Craun et al. (Craun et al., 2024) propose techniques to eliminate eBPF tracing overhead on untraced processes. These studies show the promise of eBPF for low-intrusion telemetry but stop short of providing semantic alignment between ML operators and hardware performance counters. Moreover, existing tools are evaluated on server-class CPUs or GPUs and overlook the constraints of on-device LLM inference and dynamic execution behaviors (*e.g.*, mixture-of-experts routing).

Mobile and on-device inference profiling. Mobile inference frameworks prioritize efficiency under tight resource budgets. For example, MNN (Jiang et al., 2020) provides a lightweight cross-platform runtime for DNNs on smartphones and embedded systems, focusing on latency and energy metrics rather than detailed operator introspection. Recent benchmarks such as PalmBench (Li et al., 2025) evaluate compressed and quantized LLMs across mobile devices, emphasizing throughput and accuracy. However, these evaluations still lack *fine-grained*, *non-intrusive* profiling, which prevents developers from understanding how different operators and dynamic execution behaviors impact on-device LLM performance.

In summary, prior profilers either (i) require intrusive instrumentation within ML runtimes, (ii) provide only coarse metrics through serving frameworks, or (iii) apply eBPF tracing without integrating operator semantics or supporting mobile hardware. *ProfInfer* bridges these gaps by combining eBPF-based dynamic probes with operator-level performance-counter monitoring. It achieves non-intrusive, fine-grained observability for LLM inference, correlating runtime events with hardware behavior and enabling detailed visualization of model execution on mobile platforms.

7 Conclusion and Discussion

We propose a lightweight eBPF-based profiling tool for on-device LLM inference. By recording events corresponding to the core functions of the llama.cpp software stack and combining them with hardware observability via performance monitoring counters, we obtain fine-grained operator-level profiles across three complementary views that reveal performance bottlenecks.

Portability to other inference engines. Porting *ProfInfer* to other IEs requires modest effort. Tools like `perf --callgraph` and `pahole` can be used to identify the relevant functions and data structures in an IE that provides similar information to the ones traced in llama.cpp. We can also attach u(ret)probes to the corresponding signatures, even with name mangling. For instance, MNN-LLM (Wang et al., 2024), a C++-based IE, implements `Llm::forwardVec`, `CPUAttention::onExecute`, and `AttentionExecution::onExecute` for token-level decoding and forward pass of each attention layer on CPU and GPU. `MNN::Tensor` contains similar information as in `ggml_tensor` in llama.cpp. While tracing scripts need to be updated, the analysis scripts can be reused, provided the trace format can be preserved.

Profile-guided optimization and lightweight runtime tracing. *ProfInfer* is primarily intended to help developers analyze performance bottlenecks across models and platforms, enabling them to better configure runtime compute resources, select more suitable models, and even optimize the IE. Moreover, its low tracing overhead makes it possible to support runtime optimization strategies, such as online performance tuning, interference analysis, and scheduling policies (e.g., SCHED-EXT), which are particularly helpful for scenarios with real-time requirements.

As future work, the operator-level profiling results can be analyzed to optimize graph partitioning, thereby exploiting inter-tensor parallelism and heterogeneous backends during decoding. Additionally, we could use a memory benchmark (Zuepke et al., 2024; Pradhan et al., 2025) that reveals the maximum memory-access capability to better understand the hardware's performance limits.

Acknowledgments

Binqi Sun was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation. We would also like to thank all the anonymous reviewers for their valuable feedback and constructive suggestions, which have helped improve the quality of this work.

References

Trace event format. https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU, 2025. Google Docs document.

Abaza, H., Roy, D., Fan, S., Saidi, S., and Motakis, A. Trace-Enabled Timing Model Synthesis for ROS2-based Autonomous Applications. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024.

Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R., Gunasekar, S., Harrison, M., Hewett, R. J., Javaheripi, M., Kauffmann, P., et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebron, F., and Sanghai, S. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head

Checkpoints. In *Proc. of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

Apple. Introducing apple's on-device and server foundation models. https://machinelearning.apple.com/research/introducing-apple-foundation-models, 2024. Accessed: 2025.

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Cassagnes, C., Trestioreanu, L., Joly, C., and State, R. The rise of eBPF for non-intrusive performance monitoring. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.

Chen, L., Feng, D., Feng, E., Wang, Y., Zhao, R., Xia, Y., Xu, P., and Chen, H. Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2025.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

Chu, K., Su, J., Zhang, Y., Zhao, C., Yang, Y., Zheng, Y., Lin, S., Zhao, S., and Zhang, W. eInfer: Unlocking Fine-Grained Tracing for Distributed LLM Inference with eBPF. In *Proc. of the 3rd Workshop on eBPF and Kernel Extensions*, 2025.

Craun, M., Hussain, K., Gautam, U., Ji, Z., Rao, T., and Williams, D. Eliminating eBPF Tracing Overhead on Untraced Processes. In *Proc. of the 2nd Workshop on eBPF and Kernel Extensions*, 2024.

FNIRSI. FNB58 USB Fast Charge Tester. https://www.fnirsi.com/products/fnb58, 2026. Accessed 25 Mar 2025.

Gerganov, G. and contributors. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2025.

Google LLC. Perfetto: System profiling, app tracing and trace analysis. https://perfetto.dev/, 2025.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783*, 2024.

Hagberg, A. A., Schult, D. A., and Swart, P. J. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proc. of the 7th Python in Science Conference*, 2008.

Jiang, X., Wang, H., Chen, Y., Wu, Z., Wang, L., Zou, B., Yang, Y., Cui, Z., Cai, Y., Yu, T., Lyu, C., and Wu, Z. MNN: A Universal and Efficient Inference Engine. In *Proc. of Machine Learning and Systems (MLSys)*, 2020.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proc. of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *9th International Conference on Learning Representations (ICLR)*, 2021.

Li, Y., Liu, J., Zhang, H., Narayanan, M. B., Sharma, U., Zhang, S., Zeng, Y., Raghuram, J., and Banerjee, S. PALMBENCH: A COMPREHENSIVE BENCHMARK OF COMPRESSED LARGE LANGUAGE MODELS ON MOBILE PLATFORMS. In *International Conference on Learning Representations (ICLR)*, 2025.

Microsoft. ONNX Runtime: Profiling Tools. https://onnxruntime.ai/docs/performance/tune-performance/profiling-tools.html, 2025.

MLC team. MLC-LLM, 2023-2025. URL https://github.com/mlc-ai/mlc-llm.

NVIDIA Corporation. TensorRT Best Practices: Profiling Interfaces. https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-803/best-practices/index.html, 2024.

NVIDIA Corporation. NVIDIA TensorRT-LLM: Optimized Inference for Large Language Models. https://github.com/NVIDIA/TensorRT-LLM, 2025.

ONNX Runtime developers. ONNX Runtime. https://onnxruntime.ai/, 2021.

Pradhan, A., Ottaviano, D., Jiang, Y., Huang, H., Zhang, J., Zuepke, A., Bastoni, A., and Caccamo, M. Predictable Memory Bandwidth Regulation for DynamIQ Arm Systems. In *2025 IEEE 31st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2025.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving Language Understanding by Generative

Pre-Training. Technical report, OpenAI, 2018. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

Rice, L. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. O'Reilly Media, 2023.

Song, Y., Mi, Z., Xie, H., and Chen, H. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*, 2024.

Team, G. Gemma 2: Improving Open Language Models at a Practical Size, 2024. URL https://arxiv.org/abs/2408.00118.

Team, O. Ollama: Run large language models locally. https://github.com/ollama/ollama, 2023. Accessed: 2025-10-27.

TensorFlow Team. Optimize TensorFlow Performance Using the Profiler. https://www.tensorflow.org/guide/profiler, 2024.

Van Zee, F. G. and van de Geijn, R. A. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software*, pp. 14:1–14:33, 2015.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems (NeurIPS)*, 2017.

Wang, Y., Chen, H., Tang, Y., Guo, T., Han, K., Nie, Y., Wang, X., Hu, H., Bai, Z., Wang, Y., et al. PanGu-π: Enhancing Language Model Architectures via Nonlinearity Compensation. *arXiv preprint arXiv:2312.17276*, 2023.

Wang, Z., Yang, J., Qian, X., Xing, S., Jiang, X., Lv, C., and Zhang, S. MNN-LLM: A generic inference engine for fast large language model deployment on mobile devices. In *Proc. of the 6th ACM International Conference on Multimedia in Asia Workshops*, 2024.

Xu, D., Zhang, H., Yang, L., Liu, R., Huang, G., Xu, M., and Liu, X. Fast On-device LLM Inference with NPUs. *Proceedings from Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

Yang, Y., Yu, T., Zheng, Y., and Quinn, A. eGPU: Extending eBPF Programmability and Observability to GPUs. In *Proc. of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems*, 2025.

Zuepke, A., Bastoni, A., Chen, W., Caccamo, M., and Mancuso, R. Mempol: polling-based microsecond-scale per-core memory bandwidth regulation. *Real-Time Systems*, pp. 369–412, 2024.