Forecasting LLM Inference Performance via Hardware-Agnostic Analytical Modeling

Rajeev Patwari, Ashish Sirasao, and Devleena Das

Advanced Micro Devices (AMD), San Jose, California

Abstract

Large language models (LLMs) have been increasingly deployed as local agents on personal devices with CPUs, NPUs and integrated GPUs. However, forecasting inference performance on devices with such heterogeneity remains challenging due to the dynamic compute and memory demands. Existing approaches rely on GPU benchmarking or machine learning-based latency predictors, which are often hardware-specific and lack generalizability. To this end, we introduce LIFE, a lightweight and modular analytical framework that is comprised of modular analytical model of operators, configurable to characterize LLM inference workloads in a hardware and dataset-agnostic manner. LIFE characterizes the influence of software and model optimizations, such as quantization, KV cache compression, LoRA adapters, chunked prefill, different attentions, and operator fusion, on performance metrics such as time-to-first-otken (TTFT), time-per-output-token (TPOT) and tokens-per-second (TPS). LIFE enables performance forecasting using only hardware specifications, such as TOPS and memory bandwidth, without requiring extensive dataset benchmarking. We validate LIFE's forecasting with inference on AMD Ryzen CPUs, NPUs, iGPUs and NVIDIA V100 GPUs, with Llama2-7B variants, demonstrating the utility of LIFE in forecasting LLM performance through lens of system efficiency to enable efficient LLM deployment across different hardware platforms.

1. Introduction

The Generative Pre-Trained Transformer [27], a decoder only LLM architecture based on the original Transformer architecture [35], has become the foundation for modern large language models (LLMs) such as Phi-4 [1], Llama2 [34] and Llama3 [12]. Recent advancements from Deepseek-V3 [24], have shown that smaller LLMs, with only a few billion parameters, can offer improved performance compared to earlier models of similar size. Furthermore, parameter-efficient fine-tuning (PEFT) [29] techniques, such as Low Rank Adaptation (LoRA) [15] has enabled efficient LLMs for task-specific applications. These efforts have led to an explosion of interest in local LLMs for faster and private on-device inference on laptops and mobile devices with heterogeneous hardware comprising of Neural Processing Units (NPU), Graphics Processing Units (GPU) and Central Processing Units (CPU), like AMD Ryzen APUs [5].

Enabling efficient LLM inference on personal devices with CPUs, NPUs and integrated GPUs remains challenging due to architectural heterogeneity with distributed and limited memory bandwidth, and difficulties in forecasting performance due to the lack of hardware and dataset-agnostic performance models. Modern heterogeneous devices contain distinct memory and hardware specifications of compute capacity in Tera Operations Per Second (TOPS, or TOPs/sec) and bandwidth, Giga Bytes per second (GBps). The peak performance and memory utilization specifications are not sufficient to forecast performance as the efficiency of hardware varies with the inherent dynamism in the LLM inference workload. Specifically, LLM inference workloads are non-uniform where compute and memory demands vary with prompt and generation lengths, and token generation latency change over time due to KV cache growth. For example, TOPs required for prompt length 128 is much less than 2048 tokens. Similarly, time per output token (TPOT) of the first token generated is different from that of 1000th token due to increased memory in KV Cache. While software optimizations such as operator fusion and model optimizations such as KV cache compression [13], quantization [23] to 4-bit, micro-scaling precision formats [31] and attention mechanisms can boost inference performance, it is imperative to understand their influences on performance prediction. Existing work in the area of LLM performance forecasting focuses on GPUs like NeuSight [21], [9] and ASTRA [37] that use simulators to get insight into heterogeneous architectures or GPUs. However, there is a gap in understanding dynamic nature of LLM workload on hardware efficiency and vice versa. This brings us to the core motivation of our work to answer the following: *(1) What are the fundamental workload requirements for LLM inference? (2) What types of dynamically changing conditions emerge in LLM inference? and (3) How do variations in hardware efficiency impact these dynamic behaviors and affect LLM inference performance?*

In this work, we present the **LLM Inference Forecast Engine (LIFE) Framework**, a lightweight analytical framework for modeling LLM inference workloads in a hardware and data-agnostic manner. LIFE consists of analytical models of core LLM operators, which are then utilized to build a configurable analytical workload model that supports various datatypes, software and model optimization techniques [22]. Using the analytical LLM workload model, LIFE simulates and quantifies the dynamic compute and memory utilization across the LLM inference timeline. We showcase LIFE's workload characterization capabilities across variants of Llama2-7b [33], focusing on both the LLM decode and

prefill phase, and quantify the impact of software and model optimizations and demonstrate the impact of variable hardware efficiency on performance. Our results reveal critical bottlenecks in hardware utilization, highlighting opportunities in designing improved and optimized hardware for LLM inference. The contributions of our work is summarized as follows:

1) **Hardware and dataset agnostic analytical framework for LLM inference**: We introduce LIFE, a set of modular operator-level analytical models that can be configurable to support different datatypes, model and software optimizations to capture the dynamic compute and memory behaviors of LLM inference.
2) **Characterization of LLM inference phases with optimization trade-offs**: LIFE provides empirical analysis of the prefill and decode phases under varying prompt/generation lengths and optimizations including quantization, KV cache compression, chunked prefill, different attention mechanisms and operator efficiencies.
3) **Forecasting hardware-aware LLM performance without datasets**: LIFE predicts TTFT, TPOT and TPS using only TOPS and bandwidth, enabling hardware-aware performance estimation without requiring benchmarking datasets. We very forecasted performance with true inference of Llama2-7B on AMD Ryzen CPU, NPU, iGPU and NVIDIA V100 GPU hardware.

2. Background

There are several factors that can influence workload efficiency which we motivate below. Our LIFE framework enables these factors and provides workload analysis and forecasting in a hardware and dataset independent manner.

2.1. LLM Architecture

Fig. 1 shows a simplified view of the LLM architecture, highlighting the two distinct phases in inference: prefill and decode phase. In the prefill phase, the input is processed to set the context for the model in KV Cache [29]. During decode, a new token is generated auto-regressively, updating the KV cache. The fundamental building blocks of LLMs are Embedding layer, consecutive Decoder layers, followed by a Language Model (LM) Head layer and sampling, with Elementwise and Normalization layers in between. The Attention, MLP and normalization layers together form the Decoder layer. Attention layer comprises of Query, Key, Value computations, Rotational Position Encoding (RoPE), an attention mechanism, i.e. either Multi Head-Attention (MHA) [35], Group-Query Attention (GQA) [4], Multi Query Attention (MQA) [32] or Multi-Head-Latent-Attention (MLA) [24] followed by output projection. The MLP consists of projection layers and an activation function. These finite operators in LLMs enable us to develop a low footprint analytical model within the LIFE framework.

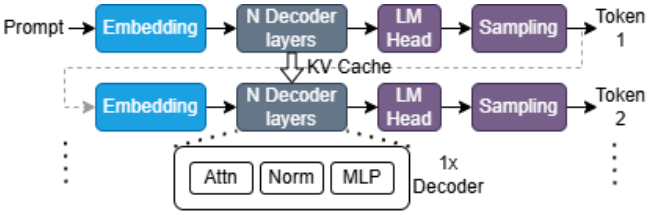

Figure 1: LLM Architecture

2.2. Software optimizations for inference

The following software optimizations are modeled in LIFE. Operator fusion and memory reuse are widely adopted software optimizations for inference speed up. Operator fusion reduces kernel dispatch calls, kernel launch (dispatch) latency and memory utilization between sequential operations in a model. Operator fusion results in reduced memory access, but the compute operations remain unchanged. Flash Attention [10] is an example of operator fusion for performance improvement. Approximations of activation functions using polynomial or piecewise linear methods [19] is another optimization technique used in Single Instruction Multiple Data (SIMD) accelerators.

2.3. Model optimizations

Model quantization reduces the total memory footprint of model parameters. Since generating each new token in an LLM requires a full model pass, reducing model size directly improves token generation throughput. GPTQ [11] and AWQ [23] quantization algorithms enable weight reduction from FP16/BF16 to 4-bit without significant loss of accuracy.

KV Cache compression is another model optimization technique that reduces the memory footprint of the KV cache. Reducing size of KV cache reduces the amount of history read for every new token generation, thus reducing TPOT and increasing the TPS. The KV cache can be compressed to 4-bit or 8-bit [13], whereas MLA exhibits another form of KV compression without quantizing the contents of KV cache [24].

In LIFE, we analyze compute and memory utilized with or without KV cache compression and weight quantization.

2.4. LoRA Finetuned LLMs

LoRA [15] is a PEFT method that updates two smaller matrices during finetuning, as opposed to all trainable parameters. Finetuned adapters are merged with a pre-trained base model's weights either prior to inference as a single step, or dynamically merged for every single GEMM operator call [17]. In LIFE, we enable analyzing workload and performance impact for both dynamic and static ahead-of-time LoRA adapter merging

3. Methodology

Fig.2 presents LIFE which takes as input (A) a config file that specifies the operating conditions and optimizations to

configure (B) the analytical LLM, comprised of hierarchical analytical models of operators shown in (C). The simulation scripts (D) simulate LLM scenarios defined by the config file and input operating conditions of past/present sequence lengths (E). The analytical LLM and operator models update the statistics database (F) and the analysis script (G) analyzes metrics from the database, with hardware specifications described in (H) to analyze workload characterization and forecast performance metrics (I). The statistics database collect metrics for characterization and forecasting that are hardware agonistic and dataset independent, enabling LIFE's generalizability to different hardware.

3.1. Analytical Model of Operators

The lightweight analytical model of operators in LIFE measures compute and memory utilization required by each operator based on input conditions. This abstraction makes LIFE hardware and data agnostic, as actual computations are not computed, allowing to quickly gather information across various LLM operating configurations. Data movement operations like resize and transpose are considered as fused to the compute operations in LIFE's analytical model. We categorize operators into foundational and derived types. The analytical model assumes that if the tensors do not fit into the on chip chip, they are tiled and executed on accelerator which increases the number of dispatch calls. LIFE's framework provides means to model and measure dispatch calls.

Foundational Operators are computed on hardware with a single operation. Table 1 provides a list of foundational operators modeled in LIFE and the corresponding compute and memory utilization, which is a function of datatype, represented by *nbytes* and *qbytes* (eg. *nbytes* = 2 for bf16 and *qbytes* = 0.5 for 4-bit). The input tensor shapes are generally denoted by $(m, k) and (k, n)$ for GEMM, (m, n) for elementwise, $(b, m, k), (b, k, n)$ for BMM, etc. Non-linear approximations [14] generally implemented on accelerators are also modeled. The analytical model of the Linear operator is shown as an example in appendix-8.1.

Derived Operators in LLM are derived from one or more foundational operators, listed in Table 2. Eg. MHA is a combination of BMM, softmax and element wise add, multiply operations. inverse [25] and inverse square root [36] approximations are also modeled.

3.2. Software Optimizations in LIFE

3.2.1. Operator Fusion & Memory Reuse. Operator fusion is analytically modeled by removing memory access between operations. For example, with a fused MHA, like Flash Attention [10], the output of the first BMM is directly used by the softmax, and second BMM, without writing and reading from memory. Fusion significantly reduces memory overhead between ops. LIFE updates the statistics database accordingly when fusion is enabled, capturing the reduced memory usage and dispatch calls.

3.2.2. Dynamic Shape Padding. Padding arbitrary tensors to nearest supported operator shape is widely used to support dynamic shapes at runtime. In the decode phase, the KV cache incrementally increases by one token at a time. LIFE provides an analytical model of the attention operators to study padding during LLM decode.

TABLE 1: Foundational Operators

Operator	*Compute Ops*	*MemRD + MemWR[Bytes]*
Linear (GEMM+Bias)	$2mkn$	$((mk) + (kn) + n) + (mn)$
(De)Quantize (Shift+Scale)	$2num\_el$	$num\_el \times nbytes + num\_qparams \times nbytes + num\_el \times qbytes$
BMM	$2bmkn - bmn$	$((bmk) + (bkn)) \times nbytes + (bmn) \times nbytes$
Elemw	mn	$2mn \times nbytes + mn \times nbytes$
Non-Linear (Piece Wise Linear)	$2num\_el$	$(num\_el + tables) \times nbytes + num\_el \times nbytes$
Non-Linear (Polyonimal Approx.)	$((n(n + 1)/2) + n) \times num\_el$	$(num\_el + n) \times nbytes + num\_el \times nbytes$
Embedding	1	$vocabsize \times hiddensize \times nbytes + hiddensize \times nbytes$

TABLE 2: Derived Operators

Operators	*Foundational Operator Used*
(Quantized) Linear	Linear (GEMM+Bias), Dequantize (Elemw Add, Mul) - int4, int8, int16, MXFP8, MXINT8, etc.
(Quantized) LoRA Linear	Linear (GEMM+Bias), Elemw Add, MatMul with optional LoRA
Inverse Square-root	Elemw Add, Mul
Inverse	Elemw Add, Mul
RoPE	Elemw Add, Mul
Norm	Elemw Add, Mul, Inverse
Softmax	NLF, Elemw Add, Mul, Inverse
MLP	Linear (GEMM+Bias), NLF Elemw Add, Mul
MHA	BMM, Softmax, Elemw Add, Mul
MLA	Linear, ElemW Add, Mul, MHA, Norm, RoPE

3.3. Model Optimizations in LIFE

3.3.1. Model Quantization. Quantization techniques enable parameter compression, resulting in low precision arithmetic and reduction in model parameter size. AWQ [23], GPTQ [11] and QuaRot [8] have proven that LLMs can be quantized with minimal loss of accuracy. Quantization requires dequantizing weights to higher precision within the compute operator before applying the linear affine transformation on the input activations. This behavior is modeled in LIFE's analytical model to reflect associated compute and memory overhead.

3.3.2. Attention mechanisms. Different attention mechanisms, MHA, GQA, MQA or MLA are analytically modeled in LIFE. There is an emerging interest to convert existing

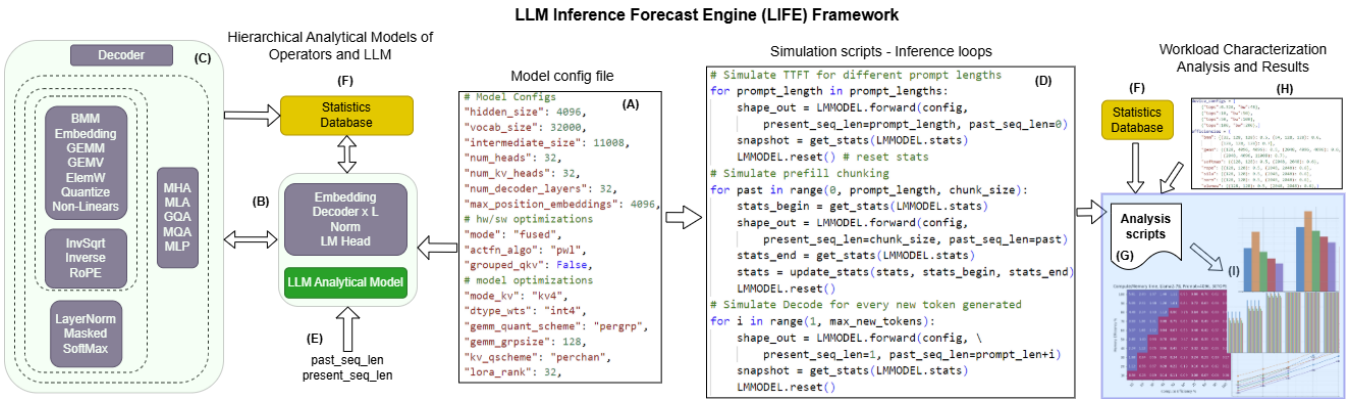

Figure 2: Overview of the LIFE framework. (A) configuration file defines operating conditions and optimizations. (B) Analytical LLM model is built using (C) hierarchical operators. (D) Simulation scripts run LLM scenarios with input sequence lengths (E). (F) The statistics database collects hardware and dataset agnostic metrics and (G) analysis scripts combine these with (H) hardware specs to forecast performance (I).

LLMs trained with MHA/GQA/MQA to use MLA [18] as it reduces KV size and improves performance for long prompts. The analytical models in LIFE capture these attention mechanisms, including MLA, and a specific attention mechanism can be chosen using the config file.

3.3.3. KV Cache compression. KV compression [22] reduces memory overhead of LLM inference [13]. We consider two techniques for KV compression in the analytical model, (1) compression provided by the MLA operator (2) KV quantization to 4-bit or 8-bit. The resulting memory reduction is modeled by the LIFE framework. Since KV quant requires dequantization during the attention operation, this overhead is also modeled in LIFE's analytical models.

3.3.4. LLM Chunked Prefill. Chunked prefill splits large prompts into smaller chunks of equal sizes, computing the prefill for each chunk while reusing the KV cache from previous chunks [3]. Chunked prefill enables long prompts without the need for special hardware. LIFE models chunked prefill in its simulation framework. The code snippet is shown in Fig. 2.

3.3.5. LoRA adaptation. LoRA adapters are merged with base model weights prior to computing the linear affine transformation in Linear layers as shown in Equation-7. LIFE models the impact of LoRA on the Linear operator to characterize LoRA adaptation performed during inference for every token generation or at a one-time merge.

3.4. Simulation Scripts & Statistical Database

LIFE's simulation software runs the LLM analytical model to characterize inference workloads. The memory and compute utilization are accumulated in LIFE's statistics database. Because LIFE is dataset independent and does not use actual model weights or perform real inference, the time taken to characterize the workload is in the range of few seconds to minutes on a laptop, allowing for fast simulations. The configuration files configure the LLM analytical model to operate in required settings.

4. Experimental Setup

Below we enumerate the experiments performed with LIFE to characterize LLM inference workload and forecast the performance in a hardware and dataset agnostic way.

4.1. Types of Workload Experiments

We organize LIFE's workload characterization abilities into two types: (1) operator-workload and (2) LLM workload. Furthermore, we consider both the workloads in both the prefill and decode stages of inference.

4.1.1. Operator workload. Operator workload focuses on understanding operator workload in isolation, across prefill, decode and prefill-chunking. We study Linear layer with/without LoRA; BMM operation; different attentions in both prefill and decode modes.

4.1.2. LLM inference workload. LIFE also understands workload at the model level, invoking sequences of operators that constitute the model architecture. LIFE utilizes the model and software optimizations enumerated in its config file to generate different LLM scenarios for which LLM workload is studied. An example configuration file for Llama2-7B with MLA is shown in Appendix 8.2.

4.2. Metrics

We categorize LIFE's metrics into two groups, (1) workload metrics and (2) performance metrics.

4.2.1. Workload Metrics. LIFE's simulation computes the following hardware agnostic workload metrics in the statistics database: ***(1) Compute Operations***, ***(2) Total Memory read/write*** in bytes, ***(3) KV Cache read/write*** in bytes, ***(4) Number of Dispatch Calls***.

4.2.2. Performance Metrics. Analysis scripts in LIFE analyze the workload metrics with reference to a desired hardware system with TOPS, bandwidth (BW) (GBps)

and operating efficiencies to forecast inference with the following performance metrics.

To accurately forecast performance metrics, LIFE expects compute and memory efficiency of operator for specific shapes and extrapolates to other shapes. Efficiency on a hardware can be measured using unit tests of operators without running LLM inference workload. Alternatively, LIFE's analysis can forecast performance metrics for all ranges of efficiencies to give insights into what is possible on a given hardware configuration.

(1) Time-To-First-Token (TTFT). For a given hardware platform with known TOPS (TOPs/sec) with an average operator efficiency, the latency of the prefill phase is estimated as shown in Eq.1 to Eq.3.

$$t_c = \sum_{op} \left(\frac{TOPs_{op}}{(ec_{op} * TOPS)} \right) + \sum_{op} t\_dispatch_{op} \quad (1)$$

$$t_m = \sum_{op} \left(\frac{Mem_{op}}{(em_{op} * BW)} \right) + \sum_{op} t\_dispatch_{op} \quad (2)$$

$$TTFT = \max(t_c, t_m) \quad (3)$$

Specifically, t_c is time taken to compute the operation and t_m is time taken for memory access. $t\_dispatch_{op}$ is the dispatch latency of the operator. ec_{op} and em_{op} are the efficiencies of the compute operation and memory utilization of the operator op. When operator efficiency is 100%, ec_{op} and em_{op} are both 1.

(2) Token-Per-Output-Token (TPOT). We observe that the $t_c << t_m$ during the decode phase for the models and the conditions studied. Thus, we define TPOT to be solely dependent on t_m. Note, em_{avg} is the average memory utilization efficiency during the lifetime of LLM inference simulation.

$$TPOT = \sum_{ops} \frac{(BW * em_{op})}{MEM_{op}} + \sum_{op} t\_dispatch_{op} \quad (4)$$

$$TPOT = \frac{(BW * em_{avg})}{MEM} + t\_dispatch_{total} \quad (5)$$

(3) Token-Per-Second(TPS).

$$TPS = (1/TPOT)\ tokens/sec \quad (6)$$

(4) LoRA adapter update time. t_{lora} is the time taken to merge the product of LoRA adapters A, B with the base weights of the corresponding linear layer.

$$t_{lora} = \sum_{linear} \left(W_{linear} + B_{linear} A_{linear} \right) \quad (7)$$

4.3. Models Studied

We perform workload characterization of variants of Llama2-7B as shown in Table 3. Each model variant is a combination of software and model optimizations.

Variant	Weights, Activations	KV, Attention	LoRA	Fusion
bf16-bf16	bf16, bf16	bf16, MHA	No	No
bf16-int4	bf16, int4	bf16, MHA	No	No
bf16-int4-fused	bf16, int4	bf16, MHA	No	Yes
bf16-int4-kv4	bf16, int4	int4, MHA	No	Yes
bf16-int4-mla	bf16, int4	bf16, MLA	No	Yes
bf16-int4-lora	bf16, int4	bf16, MHA	Yes	Yes
QuaRot-w4a4kv4	int8, int4	int4, MHA	No	Yes
fp16-fp16	fp16, fp16	fp16, MHA	No	No

TABLE 3: LLAMA2-7B MODEL VARIANTS STUDIED

4.4. Verification Setup

To verify LIFE's forecasted performance metrics, we use three hardware platforms. (1) AMD Ryzen 9 HX 370 CPU [5] with 326.4 *GFLOPS* and 240 *GBps* bandwidth for both prefill and decode, and (2) AMD Ryzen AI Max+ 395 [7] with an overall capacity of 126 *TOPS*, 50 *TOPS* on NPU for prefill and 256 *GBps* bandwidth for iGPU for decode and (3) NVIDIA V100 GPU, with 126 *TOPS* and 900 *GBps* bandwidth [26]. For (1) and (3) we use Pytorch 2.6 [28] and 4.49.0 version of HuggingFace transformers [16]. For (2), we use the RyzenAI hybrid-llm software [6]. LIFE simulation and analysis is run on setup (1).

5. Results

We perform in-depth characterization of LLM inference workload to collect workload metrics for different operating conditions and forecast the performance metrics for different hardware configurations and efficiencies.

5.1. Analysis of LLM Prefill

5.1.1. Operator distribution. We characterize the prefill phase of Llama2-7B across varying prompt lengths and show detailed results for prompt length of 2048. LIFE's simulation provides workload metrics for all operators. Table 4 shows the distribution of compute operations across operators at diff prompt lengths. We observe that for shorter prompts, the Linear operator dominates compute usage, followed by BMM and Softmax in MHA. As the prompt length increases, the BMM operators in MHA significantly dominate the compute utilization. Note that at shorter prompts, BMM efficiency has lesser impact on TTFT compared to longer prompts.

Prompt	GEMM	BMM	Softmax	TOPs	KV (GB)
256	99.0 %	1.0 %	0.0 %	3.42	0.1
1024	96.0 %	3.9 %	0.0 %	14.09	0.5
2048	92.4 %	7.5 %	0.1 %	29.29	1.0
4096	85.9 %	14.0 %	0.2 %	63.04	2.0
8192	75.2 %	24.5 %	0.3 %	143.87	4.0
16384	60.3 %	39.1 %	0.5 %	358.94	8.0
32768	43.2 %	56.0 %	0.7 %	1002.67	16.0
65536	27.5 %	71.6 %	0.8 %	3144.41	32.0

TABLE 4: TOPS VS PROMPT LENGTH FOR LLAMA2-7B BF16-BF16 MODEL IN PREFILL

We further analyze compute workload distribution across the model variants described in Table 3. Fig. 3 shows compute TOPs breakdown by operator for each

model variant. We observe that despite applying model optimizations, compute complexity remains the primary bottleneck during the prefill phase.

We also examine how compute and memory usage vary with (1) software optimizations (2) model quantization and (3) model optimizations with KV cache. Table 5 summarizes these results. We see that the TOPs remain largely unchanged across quantization and optimization techniques, but that the memory utilization varies significantly.

Model	Prompt Length	TOPs	MemRD (GB)	MemWR (GB)	KV (GB)
bf16-bf16	2048	29.2941	43.5	29.0	1
bf16-int4	2048	29.3074	34.4	29.0	1
bf16-int4-kv4	2048	29.3079	10.1	4.4	0.25
bf16-bf16	4096	63.0379	106.4	90.1	2
bf16-int4	4096	63.0511	97.3	90.1	2
bf16-int4-kv4	4096	63.0522	16.8	8.8	0.5

TABLE 5: Model workload metrics for Llama2-7B variants

5.1.2. Forecasting Prefill Performance. We use LIFE's analysis scripts to forecast performance metrics using the workload metrics for two model variants, bf16-int4 and bf16-int4-kv4, focusing on a 4096 prompt length. The simulation evaluates the impact of compute and memory efficiency across different hardware configurations ranging from from 10-100 TOPS of compute and from 10-100GBps of peak bandwidth. For the 100 hardware configurations, we measure t_c and t_m using Equation 1- 3, which loosely correlates with arithmetic intensity.

Fig. 4, shows a grid of these ratios. When $(t_c/t_m) > 1$, the TTFT is limited by compute bound and when $(t_c/t_m) < 1$, TTFT is bandwidth bound. For the bf16-int4 model at 100% efficiency, prefill is predominately memory-bound due to high memory read/writes (Fig. 4 top-left). Reducing compute to 50% and memory efficiency to 80% shifts the performance profile, altering the (t_c/t_m) balance (Fig. 4 top-right. In this case, although the total TOPS decreased, the drop in memory efficiency had a greater impact, shifting the (t_c/t_m) ratio, and altering the performance bottleneck. We repeat the same analyses with the bf16-int4-kv4 model variant (bottom left and right) and observe a significant difference in (t_c/t_m), highlighting the impact of KV compression on memory. Note that if $(t_c/t_m) > 1$ model optimization techniques do not help with TTFT. Overall, the results from Fig. 4 reveal that traditional roofline analyses are insufficient for forecasting TTFT in LLMs, as operator efficiencies vary greatly during inference. Single roofline models do not capture all dynamisms adequately; hardware efficiency varies with LLM operating conditions.

We further analyze a single configuration within the 100 hardware configurations–30 TOPS and 50 GBps–to study how varying compute and memory efficiency affect prefill latency. As shown in Fig. 5, TTFT can be accurately predicted when the compute and hardware efficiencies and how they vary in time and operator space is understood.

5.1.3. Prefill Forecast verification. We verified LIFE's forecasting and characterization on two hardware setups. Table 7 compares measured and forecasted performance. On setup 1 (CPU), we measure the bf16-bf16 variant. Interestingly, as the matrix dimensions increase, compute efficiency drops–contrary to expectations–due to increased dispatch calls and cache pressure which raise both t_m and t_c. On setup 2 (NPU), we measure the bf16-int4 variant. On setup (3) we measure fp16-fp16 variant. The absolute TTFT is much shorter compared to setup 1 due to faster hardware and quantized weights. We also observe that compute efficiency improves with longer prompts, indicating better NPU efficiency on longer prompts. In all three cases, LIFE's hardware and dataset agnostic forecasts closely match measurements on real hardware. All three hardwares are vastly different CPU, NPU, iGPU and data center GPU.

TABLE 6: Forecast vs Measurements - Prefill

Prompt Length	TOPs	Forecast TTFT		Measured TTFT	Measured Efficiency
		100% efficiency	50% efficiency		
AMD Ryzen 9 HX 370 CPU: bf16-bf16					
32	0.42	1.30	2.60	1.85	70.3 %
64	0.85	2.61	5.21	3.34	77.9 %
128	1.70	5.21	10.42	6.72	77.5 %
256	3.42	10.48	20.96	14.61	71.7 %
512	6.91	21.17	42.34	31.03	68.2 %
1024	14.09	43.17	84.34	72.99	59.1 %
2048	29.29	89.74	179.47	186.15	48.2 %
AMD Ryzen AI Max+ 395 NPU: bf16-int4					
128	1.70	0.04	0.07	0.3	11.3%
1536	21.55	0.43	0.86	1.8	23.9%
NVIDIA V100 GPU: fp16-fp16					
512	6.91	0.06	0.11	0.11	50.3 %
1024	14.09	0.11	0.22	0.2	56.3 %
2048	29.29	0.23	0.47	0.4	58.6 %

5.2. LLM Prefill with chunking

Chunked-prefill is an effective technique for supporting long prompts that exceed system limits. We study prefill-chunking for the Llama2-7B bf16-bf16 variant for a prompt length of 4096 tokens with different chunk sizes. Fig. 6 shows workload metric ratios relative to no chunking. We see that smaller chunk sizes increase memory pressure (orange bar), but as long as $t_c > t_m$, the process remains compute bound and chunked prefill does not slow down compared to regular prefill. We find that the efficiency of memory utilization must be kept high to keep the prefill compute bound. This means, to forecast accurate TTFT for prefill chunking, analyses shown in Sec. 5.1 must be done. Additionally, the number of dispatch calls also increases by 64x for smallest chunk-size (red bar). However, given the operator dispatch latency is several orders of magnitude smaller than the compute and memory latency, it has no impact to LLM prefill performance. We observe that the prefill chunk size inversely affects memory utilized, but compute load change minimally.

5.3. Analysis of LLM Decode

The Table 7 shows the workload metrics, GOPs and total memory utilized by the Llama2-7B variants during decode. We see that compute workload is three orders of magnitude lower than in prefill phase in Table 4 . Although the GOPs

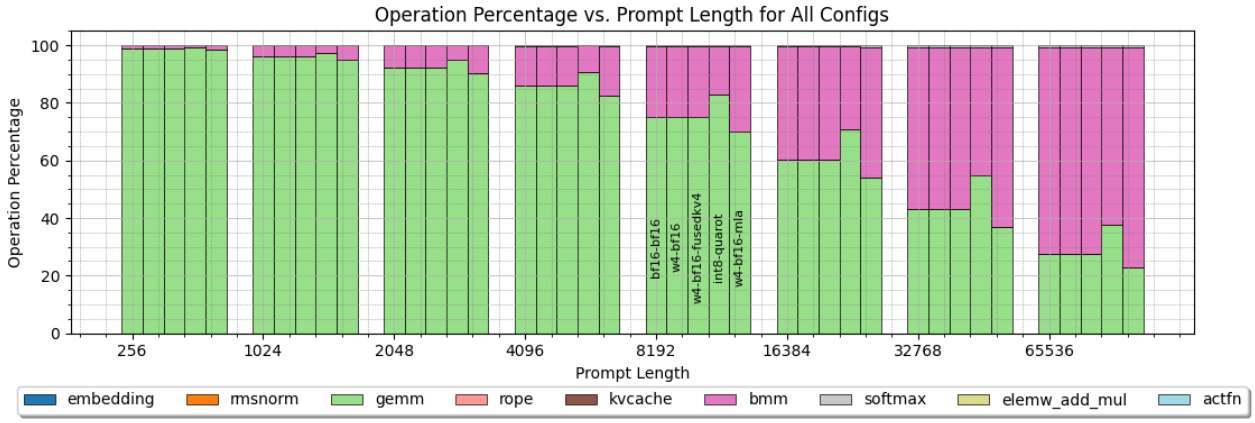

Figure 3: Prompt Length vs TOPs for Llama2-7B variants in Prefill: Each bar represents a breakdown of TOPs across different operators in that variant, (1) bf16-bf16 (2) bf16-int4 (3) bf16-int4-kv4 (4) QuaRot int8-int4 with Hadamard (5) bf16-int4-mla with Q, KV rank of 128. MLA with low rank adapters multiplied online at runtime

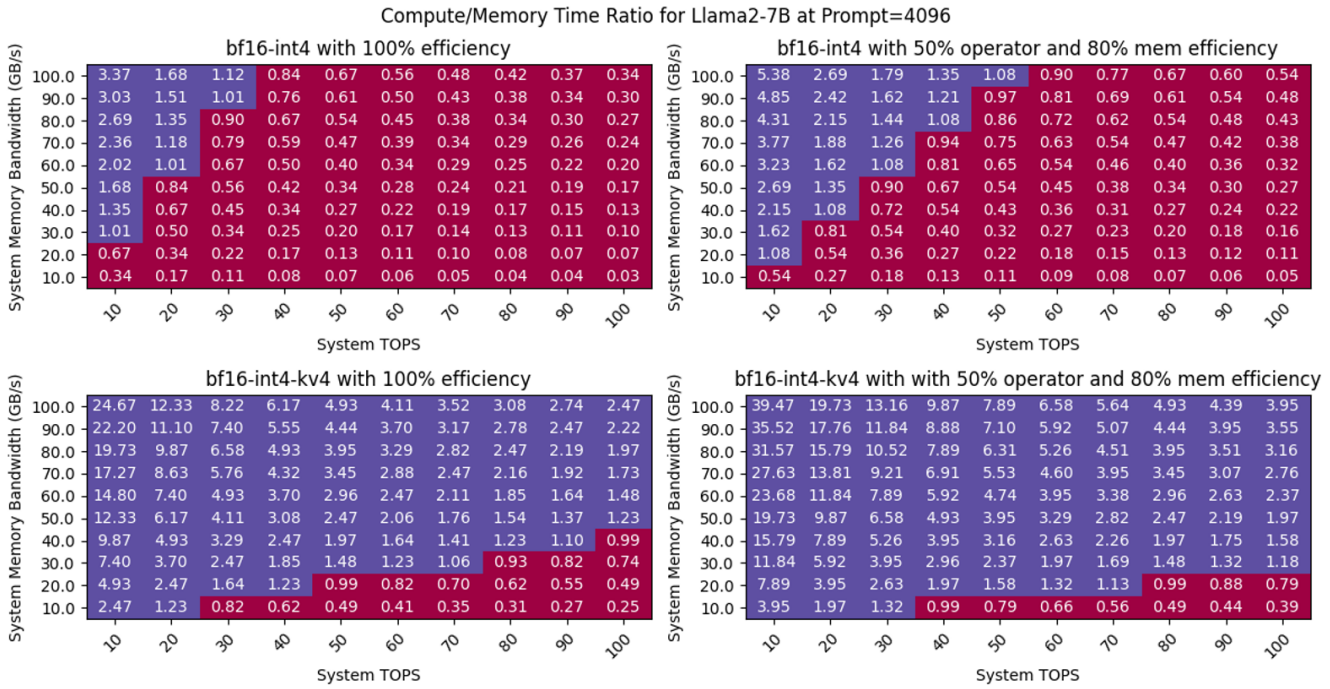

Figure 4: Llama2-7B: t_c/t_m for different hardware configurations, compute and memory efficiency

increase by 2x in the int4 model due to dequantization ops, the total compute is few GOPs. However, memory usage is substantial and grows with token generation. We show that the biggest factor for TPS improvement during decode is driven by int4 quantization. Additionally, while KV compression has minimal effects on small prompts, its impact is prominent at higher prompts.

5.3.1. Operator dispatch latency. There is a relatively low latency cost for every operator dispatch call. During the decode stage, the dispatch call latency adds to the

TABLE 7: Analysis of Llama2-7B variants' GOPs, Memory in decode phase

Prompt	GOPs			Memory (GB)		
	bf16 -bf16	bf16 -int4	bf16- int4-kv4	bf16 -bf16	bf16 -int4	bf16- int4-kv4
32	13.34	26.55	26.61	12.85	3.74	3.55
64	13.36	26.57	26.64	12.88	3.77	3.57
128	13.39	25.60	26.69	12.94	3.83	3.59
256	13.46	26.67	26.79	13.07	3.96	3.59
512	13.59	26.81	26.99	13.32	4.21	3.64
1024	13.86	27.08	27.40	13.82	4.71	3.73
2048	14.41	27.62	28.21	14.83	5.72	3.92

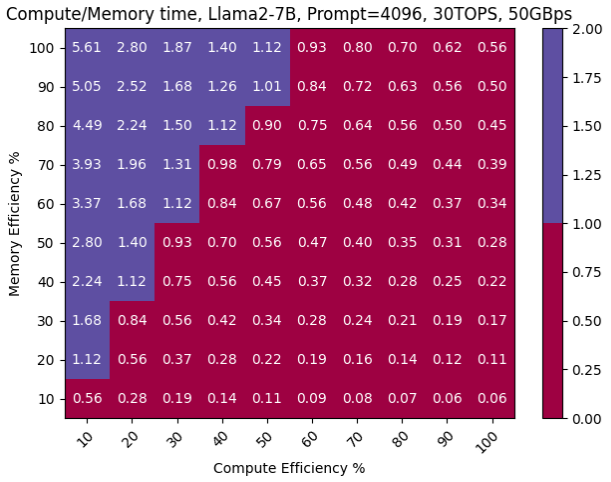

Figure 5: Llama2-7B tc/tm on 50TOPs, 30GBps vs compute and bandwidth efficiencies

TPOT as the workload and impacts TPOT. LIFE's simulation framework gathers dispatch calls as a workload metric. Table.8 shows number of dispatch calls for different models. Operator fusion reduces number of dispatch calls.

TABLE 8: Analysis of Dispatch calls during decode

Model	Dispatch Calls
Llama2-7B-int4	611
DeepSeek-Qwen2.5-1.5B-int4	535
DeepSeekv2-Lite-int4	516
Gemma2-2B-int4	497
Phi4-3.8B-int4	515
Qwen3-14B-int4	763
Qwen3-32B-int4	1219

5.3.2. Long context and generation. As text generation progresses, KV cache accumulates and the overall memory utilized to generate a single token increases. Quantifying this metric is critical to understand how the TPOT and TPS change over time for a given prompt length. Fig. 7 shows the memory utilized during model decode starting from a 4K initial prompt length. The X axis shows new tokens generated while the Y axis shows memory consumed. From Equations.1-6, TPOT is directly proportional to memory read. We observe that without KV compression, TPS drops as high as 50% and 26% for smaller and longer prompts respectively with same bandwidth efficiency. With KV compression, TPS does not drop more than 10% for any prompt length.

5.3.3. Decode Forecast Verification. LIFE's analysis software computes t_m and $t\_dispatch_{op}$ and performance metrics with Equation.1 through Equation.6. We first forecast the TPS for a given efficiency and then measure the TPS on the actual hardware on both setups 1 and 2 describe in Section. refsec:verif-setup. The results are shown in Table. 10. We observe that the forecasted

TABLE 9: Analysis of Llama2-7B Memory during Decode with 4K prompt and 2000 new tokens

Prompt Length	Llama2-7B variant	Mem(GB) 1st token	Mem(GB) last token	Mem last/ Mem 1st
128	bf16-bf16	12.75	14.71	1.15x
128	bf16-int4	3.65	5.60	1.53x
128	bf16-int4-kv4	3.53	3.90	1.10x
4096	bf16-bf16	16.66	18.62	1.18x
4096	bf16-int4	7.55	9.51	1.26x
4096	bf16-int4-kv4	4.26	4.60	1.08x

TPS is comparable to measured TPS on two independent hardware platforms. This verified LIFE's hardware agnostic and dataset independent performance forecasting.

TABLE 10: Forecast vs Measurements Llama2-7B Decode

Prompt length	Measured			Forecast	
	TPOT(ms)	TPS	Efficiency	Efficiency	TPS
AMD Ryzen 9 HX 370 CPU: bf16-bf16					
32	629	1.59	9%	10%	1.87
64	608	1.64	9%	10%	1.86
128	769	1.30	7&	10%	1.85
256	574	1.74	10%	10%	1.84
512	903	1.11	6%	10%	1.80
1024	1152	0.87	5%	10%	1.74
2048	2203	0.45	3%	10%	1.62
AMD Ryzen AI Max+ 395 iGPU: bf16-int4					
128	28.7	34.5	52.5%	50%	33.4
1536	30.5	32.8	52.5%	50%	27.2
NVIDIA V100 GPU: fp16-fp16					
512	25	40.0	60%	50%	32.6
1024	27	36.9	57%	50%	30.3
2048	31	32.1	51%	50%	26.7

5.4. Analysis of Attention Mechanisms

We use LIFE's simulation framework to characterize all attention mechanisms during decode phase and compare them. LIFE gathers the statistics for 2000 consecutive output tokens starting from a prompt length of 8192. Table.11 shows the results of comparison. We observe that MLA consumes more memory during decode than MQA and GQA for long prompts with online computation of low rank adapters within Q and KV Linear layers of the MLA. However, when the KV cache is compressed along with MLA, memory consumed by MLA reduces to almost as much as GQA. In all cases, MQA consumes least memory. For long prompt, replacing MHA with MLA reduces memory consumption by almost 50%. The investigation conclusively shows that GQA with KV cache compressed is comparable to MLA in memory utilization.

TABLE 11: Attention mechanism memory comparison during LLM decode

Mode	Mem[MB]-1st token MHA/GQA/MQA/MLA	Mem[MB]-2000th token MHA/GQA/MQA/MLA
Eager	388 / 244 / 202 / 344	450 / 283 / 234 / 415
Fused	322 / 178 / 136 / 278	368 / 201 / 152 / 333
Fused-KV8	226 / 130 / 102 / 166	249 / 141 / 110 / 193
Fused-KV4	178 / 106 / 85 / 110	189 / 111 / 89 / 124

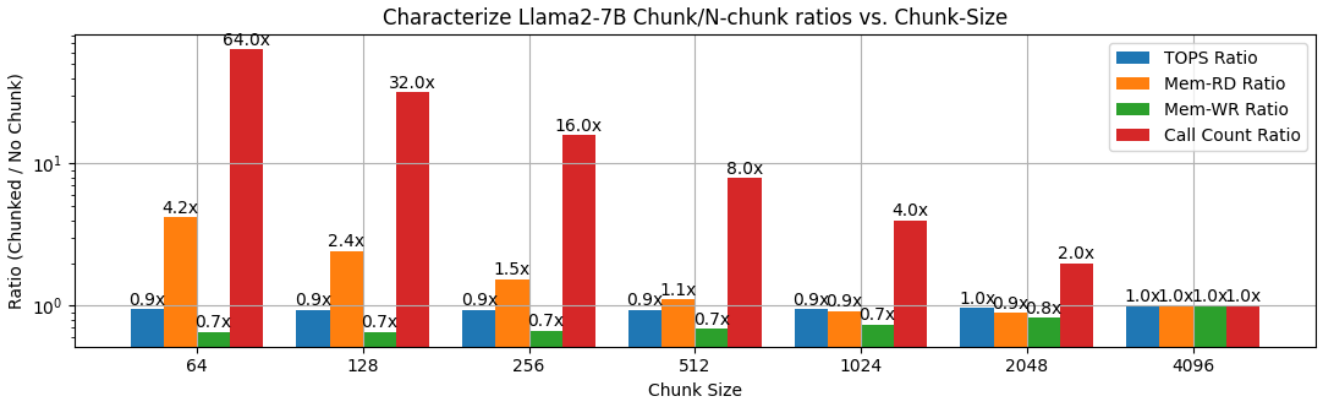

Figure 6: Llama2-7B: Ratio of metrics chunked-prefill/no chunked-prefill at prompt=4096

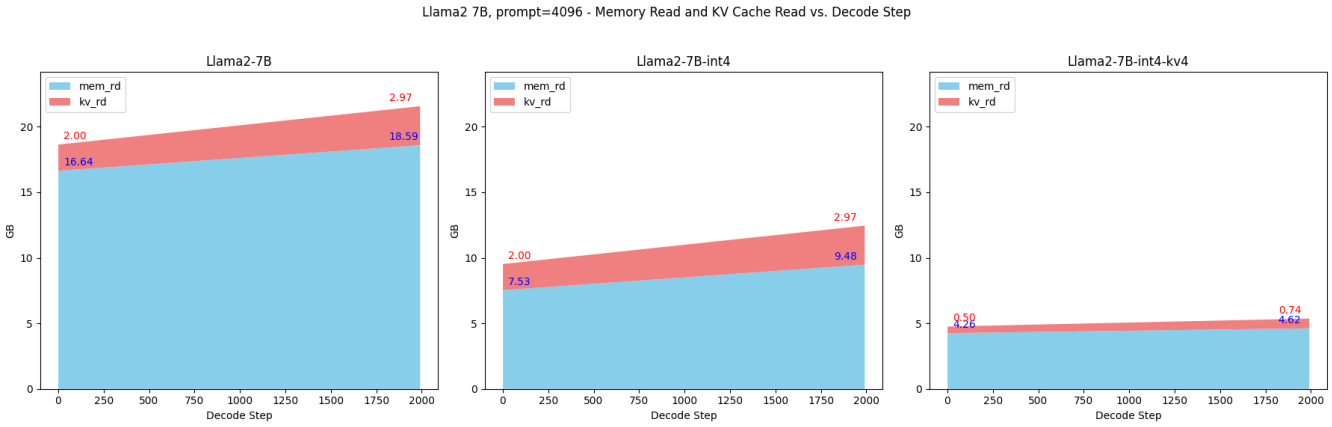

Figure 7: Llama2-7B Memory RD during Decode for prompt=4096 and 2000 new tokens: (left) bf16-bf16 (middle) bf16-int4 (right) bf16-int4-kv4

5.4.1. Analysis of MHA Efficiency. The decode phase increases the KV cache by one token for every new token generated, consequently incrementing the inputs to the BMM operator in MHA by one. BMM is often performed by padding the inputs to closest supported tiled implementation in hardware. The compute efficiency of BMM kernel drops due to under utilization. As the inputs increase, the padding required reduces, eventually BMM operates at maximum efficiency, and as the inputs increase further, the utilization for the last BMM tile drops and this repeats. With LIFE framework, we characterize and analyze the BMM operator in this specific operating condition to quantify the efficiency for various tile sizes. Fig.8 shows the results. The y-axis is compute ops. The solid black line is ideal compute. We observe that for a tile size of 64 (blue), the ideal BMM compute (dotted blue line) and the tiled BMM compute (solid blue line) differ by 1000x as sequence length increases (right). The different color lines show that irrespective of tile size selection, efficiency drops drastically as sequence length increases.

The sawtooth plot of BMM efficiency variation in decode show that the average efficiency reaches an asymptote as the number of new tokens generated increases. The asymptote defines the average BMM efficiency for long token generation with large KV cache. Thus, for long prompts, efficiency of BMM operator becomes critical. Forecasting TPS with varying MHA operator efficiency is a challenge due varying efficiencies.

5.5. Analysis of LoRA adaptation

We used the LIFE framework to investigate impact of LoRA adapter on TTFT. Fig. 9 shows compute ops for a single GEMM of size 4096x4096, appended with a LoRA adapter. We first analyze the impact on a single GEMM operator with LoRA adapter. We observe that for LoRA adapter merge inline with matmul, the compute operations increase vastly compared to the baseline of no LoRA adapters as the rank increases. This is shown in the first nine sets of bars on the plot. As the prompt length increases, compute required for adapter merge is much lesser than GEMM. As shown in the Fig. 4, absolute

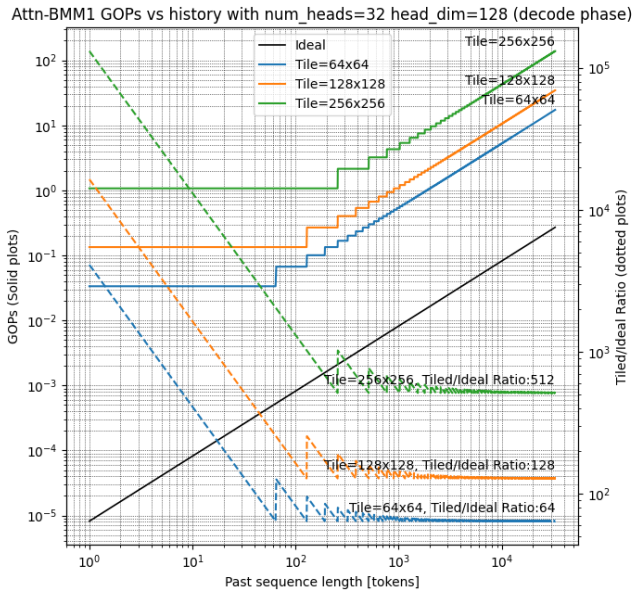

Figure 8: BMM Tiling efficiency

TOPs for TTFT phase for smaller prompts is order(s) of magnitude lower than for 2K or more prompt. We conclude that if the LoRA adapter is inline in the matmul op for every single GEMM call, TTFT of smaller prompts increases by almost 2x compared to longer prompts.

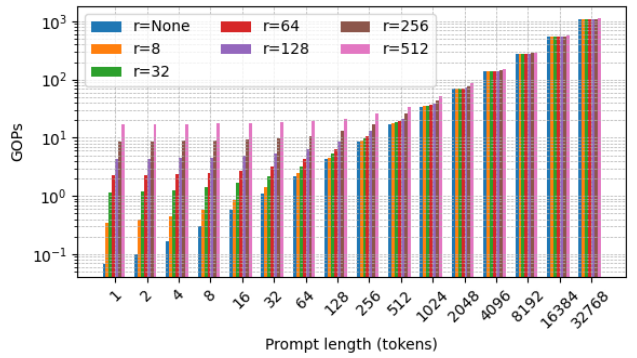

Figure 9: LoRA Linear: Ops vs prompt-size vs LoRA rank

Secondly we analyze the total TOPs required to do a complete model update with LoRA adapters on all 7 matmuls of Llama2-7B. A full model LoRA update requires considerable amount of compute as rank increases. Compare to compute for prefill in Table 4, TTFT for a prompt length of 256 requires 3.42 TOPs. LoRA update is almost half of that, at 1.67 TOPs for r=128. Therefore, continuous LoRA update during inference impacts TTFT by more than 50% for smaller prompts, where as the impact is lesser for larger prompts. If the update is done once, ahead of time, TTFT does not get affected due to LoRA.

TABLE 12: LLAMA2-7B LORA UPDATE COMPUTE OVERHEAD

Layer	K	N	r=16	r=32	r=64	r=128
q_proj	4096	4096	0.6	1.1	2.2	4.3
k_proj	4096	4096	0.6	1.1	2.2	4.3
v_proj	4096	4096	0.6	1.1	2.2	4.3
o_proj	4096	4096	0.6	1.1	2.2	4.3
gate_proj	4096	11008	1.5	3.0	5.9	11.6
up_proj	4096	11008	1.5	3.0	5.9	11.6
down_proj	11008	4096	1.5	3.0	5.9	11.6
Total (x32)	-	-	220.2	427.4	841.9	1670.8

6. Related Work

Forecasting LLM performance is of paramount importance due increasing LLM sizes and new hardware. ASTRA [30] and other simulators like [21], [9] have shown ways to simulate the workload on either specific networks, or specific hardware architectures such as GPU. Neusight [21] specifically targets GPU hardware performance ASTRA focusses on large scale training performance. Vidur [2] presents a large scale simulation framework that predicts the best deployment configuration to make inference efficient but does not address the causes of performance bottlenecks and impact of efficiency. PyTorch profiler [28] gives some insights into workload but it lacks efficiency based performance forecasting. [20] is another interesting work that targets modeling LLM performance but only for GPU. We found that a fundamental hardware and dataset agnostic analytical model of LLM inference like **LIFE** and performance forecasting through the lens of compute or memory efficiency has not been thoroughly studied.

7. Conclusion

Forecasting LLM performance using efficiency based analytical model is essential to quantitatively evaluate dynamically varying LLM inference workload. This paper delved into the understanding of the inherent dynamism of the LLM inference workload and performance forecasting through the lens of system efficiency. Our analysis using LIFE demonstrated that the compute and memory efficiency play a critical role in performance, beyond a simple roofline model. We showed the percentage impact to TTFT in prefill phase, TPS during decode phase for various software and model optimizations, for varying operating conditions of short and long prompts. We also analyzed and highlighted the role of BMM efficiency for long prompts and GEMM for LoRA adaptation. We showcased critical bottlenecks of the workload and how the analysis can be leveraged to forecast performance variation on hardware. We further showed how widely used LoRA adapters have impact on TTFT at different prompt lengths. Finally, we compared LIFE's forecasted performance metrics to real measurements on AMD CPU, NPU, iGPU and NVIDIA GPU and showed accuracy of our forecasting. We show that LIFE's framework can be applied to map LLMs on to any hardware. While we showcase our study on dense LLMs, extending this to Vision Language Models (VLMs), Mixture-of-Experts (MoEs) and Speculative Decoding is left for future exploration.

References

[1] A. Abouelenin, A. Ashfaq, A. Atkinson, H. Awadalla, N. Bach, J. Bao, A. Benhaim, M. Cai, V. Chaudhary, C. Chen *et al.*, "Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras," *arXiv preprint arXiv:2503.01743*, 2025.

[2] A. Agrawal, N. Kedia, J. Mohan, A. Panwar, N. Kwatra, B. Gulavani, R. Ramjee, and A. Tumanov, "Vidur: A large-scale simulation framework for llm inference," 2024. [Online]. Available: https://arxiv.org/abs/2405.05465

[3] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills," 2023. [Online]. Available: https://arxiv.org/abs/2308.16369

[4] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," 2023. [Online]. Available: https://arxiv.org/abs/2305.13245

[5] AMD, "Ryzen ai apu with cpu, npu and igpu," 2023. [Online]. Available: https://www.amd.com/en/products/processors/laptop/ryzen/ai-300-series/amd-ryzen-ai-max-plus-395.html

[6] AMD, "Ryzen AI LLM Software," https://ryzenai.docs.amd.com/en/latest/llm/overview.html, 2025, accessed: 2025-06-30.

[7] AMD, "Ryzen ai max+ 395," 2025. [Online]. Available: https://www.amd.com/en/products/processors/laptop/ryzen/ai-300-series/amd-ryzen-ai-max-plus-395.html

[8] S. Ashkboos, A. Mohtashami, M. L. Croci, B. Li, P. Cameron, M. Jaggi, D. Alistarh, T. Hoefler, and J. Hensman, "Quarot: Outlier-free 4-bit inference in rotated llms," 2024. [Online]. Available: https://arxiv.org/abs/2404.00456

[9] J. Cho, M. Kim, H. Choi, G. Heo, and J. Park, "Llmservingsim: A hw/sw co-simulation infrastructure for llm inference serving at scale," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Sep. 2024, p. 15–29. [Online]. Available: http://dx.doi.org/10.1109/IISWC63097.2024.00012

[10] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," 2022. [Online]. Available: https://arxiv.org/abs/2205.14135

[11] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022.

[12] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[13] C. Hooper, S. Kim, H. Mohammadzadeh, M. W. Mahoney, Y. S. Shao, K. Keutzer, and A. Gholami, "Kvquant: Towards 10 million context length llm inference with kv cache quantization," 2025. [Online]. Available: https://arxiv.org/abs/2401.18079

[14] W. G. Horner, "Horners method of polynomial approximation," 2025. [Online]. Available: https://en.wikipedia.org/wiki/Horner%27s_method

[15] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[16] HuggingFace, "Huggingface transformers github 4.49.0," 2025. [Online]. Available: https://github.com/huggingface/transformers/tree/v4.49.0

[17] HuggingFace, "LoRA in LLMs HuggingFace," 2025. [Online]. Available: https://huggingface.co/docs/peft/main/en/developer_guides/lora

[18] T. Ji, B. Guo, Y. Wu, Q. Guo, L. Shen, Z. Chen, X. Qiu, Q. Zhang, and T. Gui, "Towards economical inference: Enabling deepseek's multi-head latent attention in any transformer-based llms," 2025. [Online]. Available: https://arxiv.org/abs/2502.14837

[19] L. Jiang and F. Zhang, "Using piecewise polynomial activation functions and relevance attention for long-term time-series prediction," in *2024 International Joint Conference on Neural Networks (IJCNN)*, 2024, pp. 1–10.

[20] J. Kundu, W. Guo, A. BanaGozar, U. D. Alwis, S. Sengupta, P. Gupta, and A. Mallik, "Performance modeling and workload analysis of distributed large language model training and inference," 2024. [Online]. Available: https://arxiv.org/abs/2407.14645

[21] S. Lee, A. Phanishayee, and D. Mahajan, "Forecasting gpu performance for deep learning training and inference," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. ACM, Mar. 2025, p. 493–508. [Online]. Available: http://dx.doi.org/10.1145/3669940.3707265

[22] H. Li, Y. Li, A. Tian, T. Tang, Z. Xu, X. Chen, N. Hu, W. Dong, Q. Li, and L. Chen, "A survey on large language model acceleration based on kv cache management," 2025. [Online]. Available: https://arxiv.org/abs/2412.19442

[23] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," 2024. [Online]. Available: https://arxiv.org/abs/2306.00978

[24] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[25] L. Moroz, V. Samotyy, and O. Horyachyy, "An effective floating-point reciprocal," in *2018 IEEE 4th International Symposium on Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS)*, 2018, pp. 137–141.

[26] NVIDIA, "NVIDIA V100 technical specification," https://www.nvidia.com/en-us/data-center/tesla-v100/, 2025, accessed: 2025-06-30.

[27] OpenAI, "Chatgpt: Optimizing language models for dialogue," https://openai.com/blog/chatgpt, 2022, accessed: 2025-06-02.

[28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[29] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 606–624, 2023.

[30] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 81–92.

[31] B. D. Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf *et al.*, "Microscaling data formats for deep learning," *arXiv preprint arXiv:2310.10537*, 2023.

[32] N. Shazeer, "Fast transformer decoding: One write-head is all you need," *arXiv preprint arXiv:1911.02150*, 2019.

[33] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[34] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[36] Wikipedia, "Fast inverse square root in doom," 2025. [Online]. Available: https://en.wikipedia.org/wiki/Fast_inverse_square_root

[37] W. Won, T. Heo, S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Apr. 2023, p. 283–294. [Online]. Available: http://dx.doi.org/10.1109/ISPASS57527.2023.00035

8. Appendix

8.1. Analytical Model of Linear

```
def gemm(cls, opconfig:Dict={}, adapter_only:bool=False, strict:bool=False) -> Tuple:
    (m, k) = opconfig["shape_a"]
    (_, n) = opconfig["shape_b"]
    g = opconfig["grpsize"]
    opcount = m * k * n * 2 - (m * n)
    mem_rd  = (m * k) * cls.calc_nbytes(opconfig["dtype_a"])
    mem_wr =  (m * n) * cls.calc_nbytes(opconfig["dtype_out"])
    mem_rd_params = (k * n) * cls.calc_nbytes(opconfig["dtype_b"])
    if opconfig["bias"]:
        opcount += m * n
        mem_rd_params += n * cls.calc_nbytes(opconfig["dtype_a"])
    if (opconfig["dtype_b"] == "int4"): # per group
        # dequant
        opcount += (k * n) * 2
        mem_rd_params += ((k//g)*n) * cls.calc_nbytes(opconfig["dtype_a"]) # scale
        mem_rd_params += ((k//g)*n) * cls.calc_nbytes(opconfig["dtype_b"]) # zero
    # if LoRA
    if ("lora_rank" in opconfig.keys()):
        if (opconfig["lora_rank"] is not None):
            mem_rd_params += (k * opconfig["lora_rank"]) * cls.calc_nbytes(opconfig["dtype_lora"])
            mem_rd_params += (opconfig["lora_rank"] * n) * cls.calc_nbytes(opconfig["dtype_lora"])
            opcount += (k * opconfig["lora_rank"] * n) * 2 # lora A@B
            opcount += (k * n) # addition of adapter to original wt matrix
    cls.update_stats_ops("gemm", opcount, mem_rd, mem_wr, opconfig["mode"])
    cls.update_stats_ops("gemm", 0, mem_rd_params, 0, "eager")
    return  (m, n)
```

8.2. LLM configuration file with MLA

```
{
    "mode": "eager",
    "dtype_in": "bf16",
    "hidden_size": 4096,
    "vocab_size": 32000,
    "intermediate_size": 11008,
    "actfn_algo": "pwl",
    "actfn_table_size": 256,
    "dtype_wts": "int4",
    "gemm_quant_scheme": "pergrp",
    "gemm_grpsize": 128,
    "bias": false,
    "rope_table_size": 4096,
    "num_heads": 32,
    "num_kv_heads": 32,
    "num_decoder_layers": 32,
    "kv_qscheme": "none",
    "max_position_embeddings": 4096,
    "mla": true,
    "q_lora_rank": 128,
    "kv_lora_rank": 128,
    "qk_nope_head_dim": 128,
    "qk_rope_head_dim": 64,
    "v_head_dim": 128,
}
```