Towards Coarse-to-Fine Evaluation of Inference Efficiency for Large Language Models

Yushuo Chen[1], **Tianyi Tang**[1†], **Erge Xiang**[2†], **Linjiang Li**[2†],
Wayne Xin Zhao[1‡], **Jing Wang**[2‡], **Yunpeng Chai**[2‡], **Ji-Rong Wen**[1,2]
[1]Gaoling School of Artificial Intelligence, Renmin University of China
[2]School of Information, Renmin University of China
chenyushuo1999@foxmail.com,steventianyitang@outlook.com,
{ergeda,lilinjiang,jwang,ypchai,jrwen}@ruc.edu.cn,batmanfly@gmail.com

Abstract

In real world, large language models (LLMs) can serve as the assistant to help users accomplish their jobs, and also support the development of advanced applications. For the wide application of LLMs, the inference efficiency is an essential concern, which has been widely studied in existing work, and numerous optimization algorithms and code libraries have been proposed to improve it. Nonetheless, users still find it challenging to compare the effectiveness of all the above methods and understand the underlying mechanisms. In this work, we propose a coarse-to-fine method that encompasses both experimental and analytical components. This method can be applied across various models and inference libraries. Specifically, we examine four usage scenarios within two practical applications. We further provide both theoretical and empirical fine-grained analyses of each module in the Transformer architecture. Our methods can be a general and invaluable method for researchers to evaluate various code libraries and improve inference strategies across different LLMs. We open-source the supporting dataset, code, and evaluation scripts at the link: https://github.com/RUCAIBox/Inference-Efficiency-Evaluation.

Keywords: Inference Efficiency , Coarse-to-Fine Analysis , Large Language Model

1 Introduction

With the wide spread of large language models (Zhao et al., 2023), the enhancement of inference efficiency in LLMs has emerged as an important topic of contemporary research (Kim et al., 2023; Miao et al., 2023). To achieve superior inference speed without significant performance degradation, researchers have proposed diverse inference optimization algorithms and libraries.

Currently, several prominent libraries have been widely used in the market, such as vLLM (Kwon et al., 2023), DeepSpeed-MII (Microsoft, 2023), and TensorRT-LLM (NVIDIA, 2023), *etc*. These libraries have notably elevated inference efficiency through sophisticated methodologies such as optimization algorithms and parallel computing. Nevertheless, a notable deficiency exists in the absence of a standardized evaluation benchmark for comprehensively comparing the performance across existing libraries. To address it, this work meticulously devises a series of evaluation experiments with the goal of impartially and objectively assessing the inference efficiency of each library.

Concretely, this paper clearly defines two types of evaluation experiments: coarse-grained and fine-grained. In the coarse-grained evaluation, four text generation datasets with diverse length distributions are designed to simulate various generation tasks. We then explore two practical applications: *offline batch inference* and *online network service*. The former involves assessments conducted in batch mode offline while the latter pertains to real-time online service scenarios. We assess the efficiency of each library in offline inference and also evaluate their performance at different request frequencies.

[†] Authors contributed equally.
[‡] Corresponding authors.

Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

Libraries	Evaluations #Real	Evaluations #Syn.	Optimization Technologies KV Cache	Optimization Technologies FA	Optimization Technologies Batching
Transformers			Vanilla		
vLLM	3		Blocked	✓	✓
DeepSpeed-MII		✓	Blocked	✓	✓
TGI		✓	Blocked	✓	✓
TenserRT-LLM	1	✓	Blocked	✓	✓
llama.cpp		✓	Sequence		✓
LightLLM	1		Token	✓	✓
LMDeploy		✓	Blocked		✓
StreamingLLM		✓	W-Sink		

Table 1: Comparison of current open-sourced inference libraries, including evaluation methods and optimization technology. “#Real" indicates the number of real world data scenarios. “Syn." indicates synthetic data. “KV Cache" indicates KV cache management methods: “Vanilla" denotes naive method, “Blocked" denotes *PagedAttention*, “Token" denotes *token attention* and “W-Sink" denotes *window with attention sink* method. “FA" indicates FlashAttention. “Batching" indicates *in-flight batching*, *continuous batching* or *Dynamic SplitFuse*.

In the fine-grained analysis experiment, we provide an intricate examination of the requisite number of *floating-point* and *memory* operations for each module, to acquire a more holistic comprehension of the distribution of inference time. Besides, to pinpoint the efficiency bottleneck more accurately, we introduce the concept of arithmetic intensity and conducted an in-depth efficiency performance analysis of each module based on this concept. Furthermore, to validate the theoretical analysis, two representative libraries are selected for detailed and specific time analysis testing.

In conclusion, this investigation endeavors to delve into the inference efficiency of large language models through comprehensive and objective evaluation experiments. First, we propose a comprehensive benchmark which covers different task scenarios and can be applied across various models and inference libraries, and use them to evaluate different libraries in different usage scenarios, filling the gap in the inference benchmark. Second, we propose a fine-grained complexity analysis formula for each module of LLaMA, which reflects the bottleneck in decoding by calculating FLOPs, MOPs, and arithmetic intensity, and provides direction for subsequent decoding evaluation. Finally, we have open-source the above dataset, code, and evaluation scripts, which are available in `https://github.com/RUCAIBox/Inference-Efficiency-Evaluation`. Our detailed analysis is applicable to a wide range of rapid-update libraries across various Transformer-based LLMs . The findings of this study will not only offer valuable insights for enhancing existing inference libraries but also establish a robust groundwork for the advancement of future inference algorithms and libraries.

2 Preliminary

2.1 Background of Transformer

In contemporary LLMs, the prevailing architecture is the Transformer decoder (Vaswani et al., 2017). Utilizing the LLaMA (Touvron et al., 2023a; Touvron et al., 2023b) model as a paradigmatic illustration, its design encompasses two principal components: the multi-head attention block (MHA module) and the feed-forward network (FFN module). Both of these modules are followed by an RMS normalization (Zhang and Sennrich, 2019) and a residual network. Although various configuration variants exist for LLMs, the core components–MHA and FFN–remain consistent.

The MHA module transforms the input $\boldsymbol{X}$ into three matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ through different linear transformations, calculate the multi-head attention, and aggregate the results from multiple heads using the following formulas:

Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q, \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K, \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V, \tag{1}$$

$$\boldsymbol{O} = \mathrm{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^\intercal}{\sqrt{d}})\boldsymbol{V}, \tag{2}$$

$$\boldsymbol{X} = \boldsymbol{O}\boldsymbol{W}_O, \tag{3}$$

where $\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V, \boldsymbol{W}_O \in \mathbb{R}^{h\times h}$ denote learnable parameters.

The FFN module uses the SwiGLU activation function (Shazeer, 2020) to expand the intermediate state dimension with gated linear units, and then obtains the output result of the module through a linear transformation:

$$\boldsymbol{X} = [\mathrm{Swish}(\boldsymbol{X}\boldsymbol{W}_G) \odot (\boldsymbol{X}\boldsymbol{W}_U)]\boldsymbol{W}_D, \tag{4}$$

where $\odot$ is Hadamard product and $\boldsymbol{W}_G, \boldsymbol{W}_U \in \mathbb{R}^{h\times h'}$ and $\boldsymbol{W}_D \in \mathbb{R}^{h'\times h}$ denote parameters.

Algorithm 1 Greedy search with KV cache

Require: Model $\mathcal{M}$, input token id list $\boldsymbol{x}$
Ensure: Response token id list $\boldsymbol{y}$
1: $P, \boldsymbol{K}_{past}, \boldsymbol{V}_{past} = \mathcal{M}(\boldsymbol{x})$
2: $x' = \arg\max P$
3: $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus [x']$
4: **while** x' is not EOS and $|\boldsymbol{x}| \leq$ max-length **do**
5: $\quad P, \boldsymbol{K}, \boldsymbol{V} = \mathcal{M}(x', \boldsymbol{K}_{past}, \boldsymbol{V}_{past})$
6: $\quad x' = \arg\max P$
7: $\quad \boldsymbol{x} \leftarrow \boldsymbol{x} \oplus [x']$
8: $\quad \boldsymbol{K}_{past}, \boldsymbol{V}_{past} \leftarrow \boldsymbol{K}_{past} \oplus \boldsymbol{K}, \boldsymbol{V}_{past} \oplus \boldsymbol{V}$
9: **end while**
10: $\boldsymbol{y} \leftarrow \boldsymbol{x}$

After training, the inference of LLMs typically involves auto-regressive generation. Algorithm 1 represents an enhancement of auto-regressive generation, delineated into two distinct phases: the *prefill* phase and the *decoding* phase. During the prefill phase (lines 1-3), the model generates the initial token and stores the $\boldsymbol{K}$ and $\boldsymbol{V}$ matrices corresponding to the input tokens, called *KV cache* (Pope et al., 2022). Subsequently, in the decoding phase (lines 4-9), the model iteratively generates the next token by reusing the KV cache and updates the cache for future $\boldsymbol{K}$ and $\boldsymbol{V}$ matrices.

2.2 Arithmetic Intensity

In model inference, temporal overhead mainly stems from GPU computation and memory access , which are measured in *floating-point operations (FLOPs)*, and bytes of read/write *memory operations (MOPs)* (Kim et al., 2023). Furthermore, the concept of *arithmetic intensity* (Luebke et al., 2004) is introduced as the ratio of the FLOPs to MOPs:

$$\text{Arithmetic Intensity} = \frac{\#\text{FLOPs}}{\#\text{MOPs}}. \tag{5}$$

Each computational operation (*e.g.,* linear transformation) and hardware component (*e.g.,* GPU) possesses a arithmetic intensity. When the arithmetic intensity of an operation surpasses that of the GPU, it suggests that the operation's efficiency is constrained by the GPU's computational capacity, defining a *compute-bound* scenario. Conversely, if the operation's intensity is lower than the GPU's, the limitation is due to the GPU's memory bandwidth, defining a *memory-bound* scenario.

Given this background, we are poised to undertake a evaluation of existing inference libraries through both overall (Section 3) and fine-grained analyses (Section 4). This dual approach allows us to thoroughly assess the performance of LLMs decoding and identify its primary bottlenecks.

3 Coarse-grained Evaluation and Analysis

In this section, we conduct an coarse-grained evaluation of the inference efficiency of existing libraries. We introduce a series of evaluation datasets tailored for two distinct usage scenarios.

3.1 Evaluation Scenarios

As shown in Table 1, existing libraries either rely on synthesized data or limited real-world evaluations, which may result in biased assessments and fail to capture the full range of functionalities across different libraries. Therefore, we propose to construct a comprehensive evaluation to examine two real-world usage scenarios: batch inference and server-based inference, across four specially constructed datasets to encompass a range of task scenarios.

3.1.1 Task Scenarios

We develop four datasets focusing on the generation tasks in various real-world scenarios. The input-output length distribution of these datasets is shown in Figure 3.

• **Short-to-Short Dataset.** This dataset encompasses scenarios such as question answering and daily assistance, characterized by brief inputs and outputs. We meticulously select 1,000 examples from the Alpaca dataset (Taori et al., 2023), ensuring that both the input and output lengths are predominantly under 50 tokens.

• **Short-to-Long Dataset.** Tailored for tasks like math problem solving and code generation, this dataset comprises scenarios with short inputs and more lengthy outputs. From the Alpaca dataset, we curate 1,000 instances where the input length does not exceed 50 tokens, while the output length varies up to 1,000 tokens.

• **Short-to-16k Dataset.** Building on the concept of the short-to-long dataset, we delve into scenarios demanding exceptionally long-text generation, such as story generation. We select instances from the Vicuna dataset (Chiang et al., 2023), requiring the model to produce outputs of exactly 16,000 tokens.

• **Long-to-Short Dataset.** Aimed at reflecting text summarization or multi-turn dialogue scenarios, this dataset features lengthy inputs with concise outputs. Compiled from the ShareGPT dataset (ShareGPT, 2023), it includes examples where the input ranges from 1,100 to 1,500 tokens and the output is limited to 120 tokens or less.

3.1.2 Usage Scenarios

We mainly consider two usage scenarios:

• **Batching Inference.** In evaluating the capabilities of LLMs, it is necessary to process extensive amounts of input data in bulk offline. This context does not require a specific order or delay to process each input, allowing for the flexible arrangement of generation . We employ the four datasets to assess the time taken by different libraries to process the entire dataset, along with the token throughput.

• **Serving Inference.** Contrary to batch inference, which is mainly used in research scenarios, serving inference is predominantly utilized in the network deployment to facilitate applications akin to ChatGPT. The metrics for this scenario include sequence and token throughput, measuring the system's efficacy in managing data sequences and tokens, respectively. To account for initial stabilization and concluding operations within the system, our analysis omits the first and last 100 requests. The evaluation allows for an in-depth investigation into how various libraries fare under simulated network service conditions, elucidating their capacity to manage varying loads and respond within acceptable timeframes.

3.2 Evaluation Setup

• **Libraries.** The libraries under evaluation encompass Transformers (TRF), vLLM, Deepspeed-MII (MII), TensorRT-LLM (TRT), and llama.cpp (L.CPP). For batching inference, we manually configure the batch size for TRF, while leveraging the built-in batching strategies for the remaining four libraries, as TRF does not provide a native batching strategy. For serving inference, we assess the performance of vLLM and MII. Here we will describe in detail the various software versions used. CUDA version is 12.1, PyTorch version is 2.1.2, Transformers library version is 4.36.2, vLLM version is 0.2.6, DeepSpeed MII version is 0.1.3. In addition, TensorRT LLM's Git submission number is `0ab9d17a59c284d2de36889832fe9fc7c8697604`, while llama.cpp's Git submission number is `122ed4840cc6d209df6043e027f8a03aee01da`. These version information are critical to ensure the stability and reproducibility of the project.

• **LLMs.** We utilize five models for evaluation: `Llama-2-7b-chat-hf`, `Llama-2-13b-chat-hf` (Touvron et al., 2023b), `mistralai/Mistral-7B-Instruct-v0.2` (Jiang et al., 2023), `vicuna-7b-v`

1.5-16k, and vicuna-13b-v1.5-16k (Chiang et al., 2023). The LLaMA-2 models, which are widely used in chat applications, are chosen to assess their performance across three scenarios: short-to-short (S2S), short-to-long (S2L), and long-to-short (L2S). The Mistral model, equipped with grouped-query attention, are evaluated with the same setting. The Vicuna models, designed for handling long contexts, are employed to evaluate performance on the short-to-16k (S-16k) dataset. Note that our evaluation primarily focuses on different libraries and is orthogonal to existing LLMs.

• **Hardwares.** To assess the influence of various hardware platforms on influence efficiency, we conduct experiments using three NVIDIA GPUs: RTX-3090, RTX-4090, and A100. Table 7 presents key specifications of these GPUs, encompassing GPU memory capacity, bandwidth, and BF16 floating-point operations (FLOPs) per second.

Data	Hardware	GPU Memory	Model	TRF	vLLM	MII	TRT	L.CPP
S2S	**3090**	**24 GB**	**LLaMA-7B**	98.14	23.85	27.66	73.36	49.21
	4090	**24 GB**	**LLaMA-7B**	70.84	13.89	27.05	58.79	83.74
	A100	**80 GB**	**LLaMA-7B**	65.09	12.39	18.53	41.62	41.81
	A100	**80 GB**	**Mistral-7B**	4368.88	11.01	14.04	-	-
	A100	**80 GB**	**LLaMA-13B**	248.46	24.33	29.98	76.41	39.70
S2L	**3090**	**24 GB**	**LLaMA-7B**	4762.62	567.79	792.67	1342.81	1590.07
	4090	**24 GB**	**LLaMA-7B**	5600.64	427.99	713.94	1206.16	1688.04
	A100	**80 GB**	**LLaMA-7B**	4876.83	177.91	597.84	760.17	1271.06
	A100	**80 GB**	**Mistral-7B**	4562.00	104.71	406.17	-	-
	A100	**80 GB**	**LLaMA-13B**	5879.23	256.03	825.18	1419.02	1036.68
L2S	**3090**	**24 GB**	**LLaMA-7B**	1177.80	441.62	485.65	540.80	695.22
	4090	**24 GB**	**LLaMA-7B**	864.07	269.55	329.86	294.15	876.12
	A100	**80 GB**	**LLaMA-7B**	756.04	166.84	236.08	197.03	2559.78
	A100	**80 GB**	**Mistral-7B**	7346.61	249.85	1709.11	-	-
	A100	**80 GB**	**LLaMA-13B**	3076.05	369.72	893.91	360.94	2879.51
S-16k	**A100**	**80 GB**	**Vicuna-7B**	50566.78	5980.50	6913.22	10464.32	36158.40
	A100	**80 GB**	**Vicuna-13B**	75257.35	11074.52	14186.84	33659.65	46040.82

Table 2: The total time cost in seconds for batch inference using LLaMA-2 (7B) and (13B).

Data	Hardware	GPU Memory	Model	TRF	vLLM	MII	TRT	L.CPP
S2S	**3090**	**24 GB**	**LLaMA-7B**	272.26	1121.62	1072.79	356.72	406.95
	4090	**24 GB**	**LLaMA-7B**	379.65	1924.46	1094.31	445.09	238.98
	A100	**80 GB**	**LLaMA-7B**	413.12	2167.77	1596.45	628.72	479.58
	A100	**80 GB**	**Mistral-7B**	10.22	1682.80	1324.70	-	-
	A100	**80 GB**	**LLaMA-13B**	147.75	1515.21	1235.56	342.50	496.52
S2L	**3090**	**24 GB**	**LLaMA-7B**	102.74	860.99	610.83	358.96	223.05
	4090	**24 GB**	**LLaMA-7B**	87.65	1154.60	674.17	399.63	210.57
	A100	**80 GB**	**LLaMA-7B**	101.74	2771.10	808.82	634.09	280.53
	A100	**80 GB**	**Mistral-7B**	67.82	2175.36	568.41	-	-
	A100	**80 GB**	**LLaMA-13B**	77.74	1797.52	562.07	339.68	326.98
L2S	**3090**	**24 GB**	**LLaMA-7B**	48.18	124.29	112.85	98.21	58.71
	4090	**24 GB**	**LLaMA-7B**	64.78	205.32	166.64	180.57	46.67
	A100	**80 GB**	**LLaMA-7B**	73.92	331.19	232.79	269.58	15.92
	A100	**80 GB**	**Mistral-7B**	30.58	616.57	100.52	-	-
	A100	**80 GB**	**LLaMA-13B**	30.68	254.89	96.63	147.15	13.51
S-16k	**A100**	**80 GB**	**Vicuna-7B**	25.31	214.03	185.15	122.32	35.40
	A100	**80 GB**	**Vicuna-13B**	17.01	115.58	90.22	38.03	27.80

Table 3: The token throughput (token/s) for batch inference using LLaMA-2 (7B) and (13B).

3.3 Evaluation Results

Firstly from the scenario of batching inference in Table 2 and Table 3, we find that GPU computational performance is pivotal for short input-output pairs, whereas memory bandwidth becomes critical as

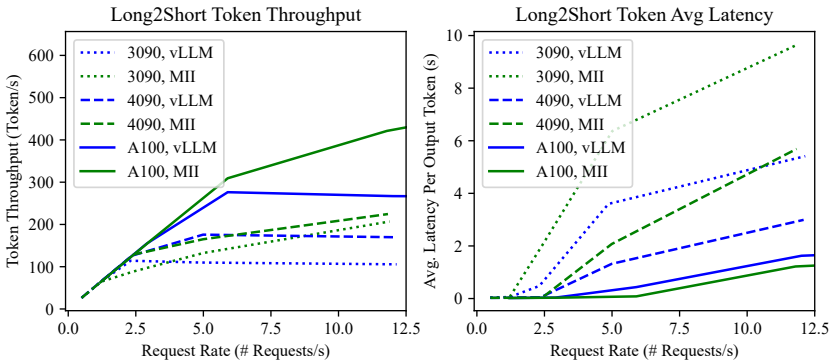


Figure 1: The throughput and latency for serving inference of vLLM and MII using LLaMA-2 (7B) under different request frequencies on the Long2Short dataset.

sequences elongate. We have observed that the 4090 significantly outperforms the 3090 in the S2S dataset. However, this advantage diminishes in S2L and L2S datasets. Conversely, the A100 consistently excels across all datasets. We hypothesize that the observed performance discrepancies are attributable to the differing specifications of the GPUs (see Table 7). The 4090 boasts double the computational power of the 3090, yet their bandwidths are comparable. In contrast, the A100 doubles both the computational power and bandwidth relative to the 4090.

Secondly, vLLM and MII demonstrate superior efficiency compared to other libraries in batching scenarios. This advantage is primarily attributable to their optimization technologies, including KV cache and batching strategies. When analyzing the results from the 7B and 13B models, it is evident that the 13B model's processing time is nearly double that of the 7B model for both vLLM and MII. This phenomenon is not observed in other libraries. Given that the computational FLOPs for the 13B model are twice those of the 7B model, a corresponding increase in processing time is expected. This indicates that the other libraries have room for improvement in memory management. As for GQA, vLLM offers robust support, leading to improved inference performance; however, other libraries lack full support for this feature.

Thirdly, the Dynamic SplitFuse strategy of MII demonstrates enhanced efficiency in serving inference scenarios of long sequences, as evidenced by the results depicted in Figures 1, 4, and 5. It is observed that with an increasing evaluation rate of requests, the vLLM initially exhibits a surge, followed by a gradual decline after reaching its peak performance. In contrast, the token throughput for MII consistently rises, although the rate of increase gradually diminishes. This phenomenon becomes more evident as the sequences lengthen (Figures 1 and 5), because the Dynamic SplitFuse strategy enables more fine-grained segmentation for longer sequences. Regarding token latency, as the rate of requests escalates, both vLLM and MII show a steady increase in latency. The latency of the Dynamic SplitFuse is observed to be higher when the GPU memory is limited (*i.e.,* 3090 and 4090).

4 Fine-grained Modular Evaluation and Analysis

In this section, we conduct a both *theoretical analysis* and *practical evaluation* to quantify the time, floating point operations, memory read/write volumes, and arithmetic intensity of each module in LLaMA. This granular investigation provides a thorough understanding of the model's computational characteristics. Comparing these modules of Transformers and vLLM, we can derive insights into the optimization paths of current libraries and yield crucial guidance for future improvement. Our method is also applicable to analyze the bottleneck of other libraries.

4.1 Theoretical Analysis

In this section, we dissect each operation within the LLaMA decoder layer, deriving theoretical formulas for the number of floating-point operations and the volume of memory reads/writes, as well as the resulting arithmetic intensity, which quantifies the balance between computation and data movement, guiding optimizations to maximize processing efficiency and minimize memory bottlenecks in compute-heavy

	FLOPs Form.	I/O Form.	A.I. Form.	Transformers Time	Transformers I/O	Transformers A.I.	vLLM Time	vLLM I/O	vLLM A.I.
$\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}_{QKV}$	$6bsh^2$	$\Theta(bsh+h^2)$	$\Theta\left(\frac{1}{\frac{1}{h}+\frac{1}{bs}}\right)$	77.22	20.55	642.15	72.59	29.31	450.12
$\boldsymbol{Q},\boldsymbol{K}=\mathrm{RoPE}(\boldsymbol{Q},\boldsymbol{K})$	$6bsh$	$\Theta(bsh)$	$\Theta(1)$	32.79	17.66	0.18	5.34	3.47	0.93
$\boldsymbol{O}=\mathrm{Attn}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})$	$4bs^2h+4bs^2n$	$\Theta(bs^2n+bsh)$	$\Theta\left(\frac{1+\frac{1}{d}}{\frac{1}{d}+\frac{1}{s}}\right)$	112.65	77.52	14.32	23.97	8.14	136.44
$\boldsymbol{X}=\boldsymbol{O}\boldsymbol{W}_O$	$2bsh^2$	$\Theta(bsh+h^2)$	$\Theta\left(\frac{1}{\frac{1}{h}+\frac{1}{bs}}\right)$	25.75	6.85	642.12	23.51	9.26	475.09
$\boldsymbol{X}=\mathrm{Add\&Norm}(\boldsymbol{X})$	$5bsh$	$\Theta(bsh+h)$	$\Theta\left(\frac{1}{1+\frac{1}{bs}}\right)$	18.47	19.80	0.14	4.99	3.40	0.79
$\boldsymbol{G},\boldsymbol{U}=\boldsymbol{X}[\boldsymbol{W}_G,\boldsymbol{W}_U]$	$4bshh'$	$\Theta(bsh+bsh'+hh')$	$\Theta\left(\frac{1}{\frac{1}{h}+\frac{1}{h'}+\frac{1}{bs}}\right)$	119.91	37.52	630.02	128.37	53.02	445.85
$\boldsymbol{D}=\mathrm{Swish}(\boldsymbol{G})\odot\boldsymbol{U}$	$2bsh'$	$\Theta(bsh')$	$\Theta(1)$	9.23	13.60	0.21	9.15	8.11	0.36
$\boldsymbol{X}=\boldsymbol{D}\boldsymbol{W}_D$	$2bshh'$	$\Theta(bsh+bsh'+hh')$	$\Theta\left(\frac{1}{\frac{1}{h}+\frac{1}{h'}+\frac{1}{bs}}\right)$	55.85	17.15	689.38	62.40	21.56	548.33
$\boldsymbol{X}=\mathrm{Add\&Norm}(\boldsymbol{X})$	$5bsh$	$\Theta(bsh+h)$	$\Theta\left(\frac{1}{1+\frac{1}{bs}}\right)$	18.47	19.80	0.14	4.99	3.40	0.79

Table 4: Theoretical and practical results of in prefill stage ($b=8, s=512$).

	FLOPs Form.	I/O Form.	A.I. Form.	Transformers Time	Transformers I/O	Transformers A.I.	vLLM Time	vLLM I/O	vLLM A.I.
$\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}=\boldsymbol{x}\boldsymbol{W}_{QKV}$	$6bh^2$	$\Theta(bh+h^2)$	$\Theta\left(\frac{1}{\frac{1}{h}+\frac{1}{b}}\right)$	2.72	3.23	7.98	2.11	3.22	7.99
$\boldsymbol{q},\boldsymbol{k}=\mathrm{RoPE}(\boldsymbol{q},\boldsymbol{k})$	$6bh$	$\Theta(bh)$	$\Theta(1)$	2.66	0.03	0.24	0.31	0.00	1.48
$\boldsymbol{K},\boldsymbol{V}=\mathrm{Cache}(\boldsymbol{k},\boldsymbol{v})$	-	$\Theta(bh)$ or $\Theta(bsh)$	-	10.89	3.46	-	1.82	2.22	-
$\boldsymbol{o}=\mathrm{Attn}(\boldsymbol{q},\boldsymbol{K},\boldsymbol{V})$	$4bsh+4bsn$	$\Theta(bsn+bsh+bh)$	$\Theta\left(\frac{1+\frac{1}{d}}{1+\frac{1}{d}+\frac{1}{s}}\right)$	3.52	2.23	0.97	1.60	2.22	0.98
$\boldsymbol{x}=\boldsymbol{o}\boldsymbol{W}_O$	$2bh^2$	$\Theta(bh+h^2)$	$\Theta\left(\frac{1}{\frac{1}{h}+\frac{1}{b}}\right)$	0.91	1.08	7.98	0.90	1.08	7.98
$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	$5bh$	$\Theta(bh+h)$	$\Theta\left(\frac{1}{1+\frac{1}{b}}\right)$	1.83	0.03	0.18	0.26	0.00	1.19
$\boldsymbol{g},\boldsymbol{u}=\boldsymbol{x}[\boldsymbol{W}_G,\boldsymbol{W}_U]$	$4bhh'$	$\Theta(bh+bh'+hh')$	$\Theta\left(\frac{1}{\frac{1}{h}+\frac{1}{h'}+\frac{1}{b}}\right)$	3.87	5.78	7.99	3.66	5.77	8.00
$\boldsymbol{d}=\mathrm{Swish}(\boldsymbol{g})\odot\boldsymbol{u}$	$2bh'$	$\Theta(bh')$	$\Theta(1)$	0.27	0.02	0.33	0.42	0.01	0.50
$\boldsymbol{x}=\boldsymbol{d}\boldsymbol{W}_D$	$2bhh'$	$\Theta(bh+bh'+hh')$	$\Theta\left(\frac{1}{\frac{1}{h}+\frac{1}{h'}+\frac{1}{b}}\right)$	2.05	2.89	7.98	2.03	2.89	7.98
$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	$5bh$	$\Theta(bh+h)$	$\Theta\left(\frac{1}{1+\frac{1}{b}}\right)$	1.83	0.03	0.18	0.26	0.00	1.19

Table 5: Theoretical and practical results in decoding stage ($b=8, s=512$).

tasks. This analysis is strictly limited to a single decoder layer; to extrapolate to real-world applications, one must multiply these findings by the total number of decoder layers. Our analysis are detailed in Tables 4 and 5. In the following analysis, b represents the batch size, s represents the input sequence length, h represents the hidden size, h' represents the intermediate size of FFN module, n represents the number of attention "heads", and d represents the size of each "head" (n and d satisfying $h=nd$).

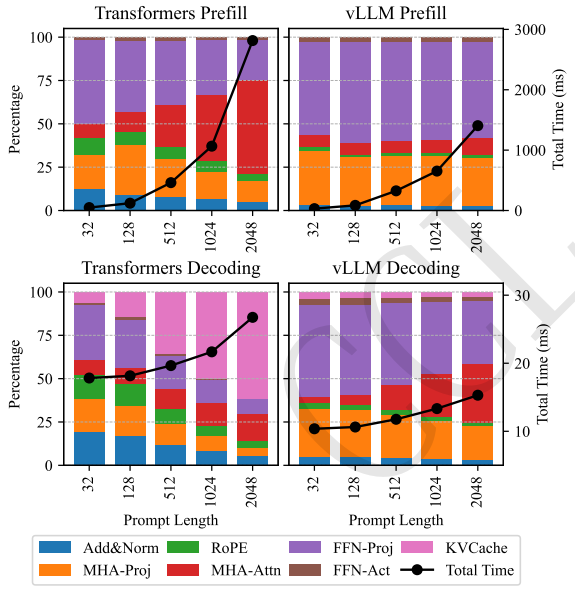
• **FLOP Analysis.** First, let's analyze the *prefill* phase: For the MHA module, the three linear projections can be expressed as matrix multiplications (Equation 1), requiring $6bsh^2$ FLOPs. The calculation of RoPE involves 4 multiplications and 2 additions, requiring $6bsh$ FLOPs. Regarding the attention calculation (Equation 2), the multiplication of matrix $\boldsymbol{Q}$ and matrix $\boldsymbol{K}$ requires $2bs^2h$ FLOPs. Dividing by $\sqrt{d}$ and calculating the softmax requires $4bs^2n$ FLOPs. Finally, multiplying with matrix $\boldsymbol{V}$ requires $2bs^2h$ FLOPs. Therefore, the attention calculation requires a total of $4bs^2h+4bs^2n$ FLOPs. The final linear transformation (Equation 3) in the MHA module also requires $2bsh^2$ FLOPs. For the FFN module (Equation 4), the initial two linear projections require $4bshh'$ FLOPs. The calculation of the activation function involves both multiplication and the Swish function, requiring $2bsh'$ FLOPs. The final linear projection requires $2bshh'$ FLOPs. The calculation of the RMS normalization (RMSNorm) and the residual networks requires $5bsh'$ FLOPs. For the *decoding* phase, apart from the attention calculation, the FLOPs required for other parts can be obtained by substituting $s=1$ into the corresponding formulas from the prefill phase. The FLOPs for the attention becomes $4bsh+4bsn$.

• **MOPs Analysis.** Due to the fact that matrix multiplication is calculated in blocks in practical operations, memory read and write volumes can only be expressed in the form of progressive complexity

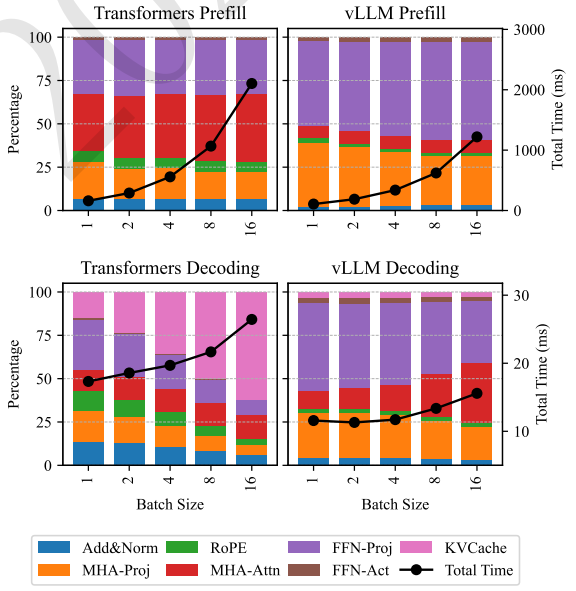
Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

Θ. First, let's analyze the *prefill* phase: For the MHA module, the three linear projections can be expressed as matrix multiplications (Equation 1), requiring $\Theta(bsh)$ MOPs. The calculation of RoPE involves $\Theta(bsh)$ MOPs. Regarding the attention calculation (Equation 2), the multiplication of matrix $\boldsymbol{Q}$ and matrix $\boldsymbol{K}$ requires $\Theta(bsh + bs^2n)$ MOPs. Dividing by $\sqrt{d}$ and calculating the softmax requires $\Theta(bs^2n)$ MOPs. Finally, multiplying with matrix $\boldsymbol{V}$ requires $\Theta(bsh + bs^2n)$ MOPs. Therefore, the attention calculation requires a total of $\Theta(bsh + bs^2n)$ MOPs. The final linear transformation (Equation 3) in the MHA module also requires $\Theta(bsh + bsh' + hh')$ MOPs. For the FFN module (Equation 4), the initial two linear projections $\Theta(bsh + bsh' + hh')$ MOPs. The calculation of the activation function involves both multiplication and the Swish function, requiring $\Theta(bsh')$ MOPs. The final linear projection requires $\Theta(bsh + bsh' + hh')$ MOPs. The calculation of the RMSNorm and the residual networks requires $\Theta(bsh + h)$ MOPs. For the *decoding* phase, apart from the attention calculation, the MOPs required for other parts can be obtained by substituting $s = 1$ into the corresponding formulas from the prefill phase. The MOPs required for the attention calculation become $\Theta(bsn + bsh + bh)$.

• **Arithmetic Intensity Analysis.** Based on the analysis of FLOPs and MOPs, the arithmetic intensity of each module can be determined by dividing these two quantities. During the prefill stage, from the formulas in Table 4, it is evident that attention module exhibits the lowest arithmetic intensity, excluding components such as RoPE, RMSNorm, and residual networks. During the decoding stage, the arithmetic intensity of each linear transformation is approximately $\Theta(b)$. However, the arithmetic intensity of the attention module is approximately $\Theta(1)$. Hence, optimizing the implementation of attention, RoPE, RMSNorm, and residual networks is crucial for reducing MOPs during the inference stage of LLMs, which leads to the development of FlashAttention and PagedAttention . Additionally, maximizing the batch size in the decoding stage is necessary to enhance the arithmetic intensity of linear transformations, which necessitates the advance of batching strategies (Kwon et al., 2023; Microsoft, 2023).



(a) The distributions of total time and module time for both Transformers and vLLM libraries across different input lengths ranging from 32 to 2048 tokens.



(b) The distribution of total time and module time for both Transformers and vLLM libraries across different batch size ranging from 1 to 16.

Figure 2: Results of Fine Grained Modular Evaluation.

4.2 Evaluation Setup

To accurately measure the execution time and memory read/write volume (MOPs) of various modules during real-world execution, we employ two tools: NVIDIA Nsight Compute CLI (NCU) and `torch.profile`. NCU is adept at quantifying the execution time and MOPs for individual CUDA kernels, while `torch.profile` offers detailed call stacks of CUDA kernels, enabling precise identification of

specific modules.

In the following experiments, we utilize simulated data with input lengths ranging from 32 to 2048 using a fixed batch size of $b = 8$ for Figure 2a, Tables 8 and 9. We also conduct experiments varying different batch sizes with $s = 1024$ in Figure 2b, Tables 10 and 11, and experiments varying different hardware with $b = 8, s = 512$ in Tables 12 and 13. For each experiment, we employ both Transformers and vLLM libraries to generate two tokens using LLaMA-2 (7B) each on A100 GPU. This allows for execution of the prefill stage and the decoding stage once within each library.

4.3 Evaluation Results

Firstly, our practical time consuming results are consistent with our theoretical analysis results in Tables 4 and 5. Thus, we can estimate the runtime for each module during the prefill and decoding phases. For compute-bound operations (*e.g.,* the linear transformation of MHA in the prefill phase), the estimation primarily relies on the number of FLOPs, represented as bsh^2. For memory-bound operations, runtime is primarily influenced by MOPs. The corresponding estimation equations are detailed below:

$$T_{\text{prefill}} = \alpha \underbrace{bsh^2l}_{\text{MHA Proj.}} + \beta \underbrace{bshh'l}_{\text{FFN Proj.}} + \gamma \underbrace{bs^2nl}_{\text{Attn.}} + \eta \underbrace{bshl}_{\text{RoPE, Norm, Res., Attn.}} + \lambda \underbrace{bsh'l}_{\text{FFN Act.}} + \mu, \tag{6}$$

$$T_{\text{decoding}} = \phi \underbrace{bshl}_{\text{KV Cache, Attn.}} + \psi \underbrace{bsnl}_{\text{KV Cache}} + \omega \underbrace{bhl}_{\text{KV Cache, Attn.}} + \nu, \tag{7}$$

where $\alpha, \beta, \gamma, \eta, \lambda, \mu, \phi, \psi, \omega, \nu$ are the coefficients of different items. We can determine them through linear regression based on our experimental data, as presented in Table 6.

Libs.	α	β	γ	η	λ	μ
TRF	3.75×10^{-11}	3.69×10^{-11}	4.20×10^{-8}	1.70×10^{-7}	6.35×10^{-9}	3.28×10^{1}
vLLM	4.51×10^{-11}	3.35×10^{-11}	2.29×10^{-9}	5.88×10^{-8}	6.26×10^{-9}	-1.64×10^{0}
Libs.	ϕ	ψ	ω	ν		
TRF	2.31×10^{-8}	2.65×10^{-11}	3.32×10^{-12}	1.85×10^{1}		
vLLM	2.23×10^{-9}	1.75×10^{-11}	1.63×10^{-8}	1.12×10^{1}		

Table 6: The coefficients of running time (ms) estimation Equation 6 and 7.

Secondly, the attention module is the bottleneck during the prefill and decoding stage from the results in Figure 2a. Notably, during the prefill phase, the conventional attention mechanism emerges as the primary bottleneck, particularly as the input length escalates. To address this challenge, the integration of FlashAttention (Dao et al., 2022) presents an effective optimization strategy. Conversely, in the decoding phase, inadequate management of the KV cache can result in the update of the KV cache emerging as the principal bottleneck with increasing input lengths. vLLM employs block management techniques for KV cache and PagedAttention (Kwon et al., 2023) mechanisms to streamline KV cache updates and attention calculations, contributing to enhanced efficiency in decoding tasks.

Third, batching strategies are shown to be effective for increasing arithmetic intensity during the decoding stage. According to the formulas presented in Table 5, it is evident that all operations are memory-bound during decoding due to the low arithmetic intensity. For operations such as linear transformations and activations, increasing the batch size can enhance arithmetic intensity. Notably, even with larger batch sizes and input lengths, the processing time remains nearly consistent for these operations, as indicated in Tables 11 and 9. This consistency suggests that we can execute more FLOPs within a similar timeframe. Such findings support the use of strategies such as continuous batching and Dynamic SplitFuse to boost arithmetic intensity and thereby increase the overall token throughput.

In addition, CUDA kernel fusion also plays a significant role in improving decoding efficiency. The vLLM library features specially designed CUDA kernels tailored for operations such as RoPE, Swish, and RMSNorm. In contrast to the Transformers library, which exhibits the same arithmetic intensity complexity (as shown in Tables 4 and 5), vLLM refines the implementation of these operations to optimize memory access patterns and reduce execution time, as illustrated in Figure 2a.

Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

Finally, it is evident that linear transformation operations (*i.e.,* MHA projection and FFN projection) still occupy a substantial portion of time during both the prefill and decoding phases. As shown in Figure 2a, after various vLLM optimizations, linear transformations comprise the most time-consuming elements when the sequence length is short and they also account for over 50% of the total time as the sequence length increases. Although optimizing matrix multiplication presents inherent challenges, it offers a promising path for future inference enhancements.

5 Related Work

• **System Optimization.** There are many optimization algorithms for inference in large language models. To address the low efficiency issue of multi-head attention calculation, *FlashAttention* (Dao et al., 2022; Dao, 2023) leverages optimization strategies employed in GPU-based matrix multiplication. By reducing the frequency of memory accesses, this optimization technique increases the arithmetic intensity and improves the efficiency of the attention module. GQA (Ainslie et al., 2023) reduces the size of the KV matrix, thereby reducing the memory access time. To optimize the management of the KV cache memory in decoding phase, vLLM proposes *PagedAttention* (Kwon et al., 2023). This mechanism effectively mitigates the frequent update requirement of the KV cache and reduces memory fragmentation, leading to improved overall efficiency. As for practical applications, researchers have proposed batching strategy such as *continuous batching* (Kwon et al., 2023), *inflight batching* (NVIDIA, 2023) and *Dynamic SplitFuse* (Microsoft, 2023). Their strategy involves immediately substituting completed requests with new ones, eliminating the need for padding tokens.

• **Inference Libraries.** The Transformers (Wolf et al., 2020) library is a commonly used library in the field of natural language processing, providing code and archive points for many common models. TGI (Contributors, 2023a) is a library developed by HuggingFace for further optimization of inference based on the Transformers. vLLM (Kwon et al., 2023) mainly improves the utilization efficiency of KV cache by paging storage and combining with PagedAttention technology. DeepSpeed-MII has introduced Dynamic-SplitFuse technology to fully tap into the computing potential of GPUs. TensorRT-LLM is developed by Nvidia, which has been further optimized based on the previous FasterTransformer (NVIDIA, 2021) library, improving the efficiency of running large models on Nvidia GPUs. Llama.cpp is entirely based on C/C++ implementation, with good cross platform compatibility and the ability to run on various computing devices. Other code libraries such as LightLLM (ModelTC, 2023), LMDeploy (Contributors, 2023b), StreamLLM (Xiao et al., 2023), and Inferflow (Shi et al., 2024) have made different optimization implementations for inference.

6 Conclusion

In this work, we introduced a comprehensive benchmark that can encompass diverse task scenarios for the evaluation of various libraries. We integrated various common experimental settings in our framework, to provide a useful testbed for evaluating inference efficiency related libraries. Based on it, we proposed a detailed formula for analyzing the complexity of each component from LLaMA, which involves metrics such as FLOPs, MOPs, and arithmetic intensity. It can delineate the decoding bottlenecks in the inadequacy of memory bandwidth, and has been validated in our experiments. Besides, widely-used strategies such as FlashAttention, PagedAttention and CUDA kernel fusion demonstrated the mitigation of memory access constraints is helpful to enhance the inference efficiency. Our analysis is adaptable to different libraries utilizing a range of LLMs and will offer valuable insights for the advancement of future inference algorithms and libraries.

Acknowledgements

This work was partially supported by National Natural Science Foundation of China under Grant No. 92470205, Beijing Municipal Science and Technology Project under Grant No. Z231100010323009, and Beijing Natural Science Foundation under Grant No. L233008.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2 305.13245*.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March.

Hugging Face Contributors. 2023a. huggingface/text-generation-inference: Large language model text generation inference. https://github.com/huggingface/text-generation-inference.

LMDeploy Contributors. 2023b. Lmdeploy: A toolkit for compressing, deploying, and serving llm. https://github.com/InternLM/lmdeploy.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Proc. of NeurIPS*.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR*.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *CoRR*, abs/2310.06825.

Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. 2023. Full stack optimization of transformer inference: a survey. *CoRR*, abs/2302.14017.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*.

David Luebke, Mark J. Harris, Jens H. Krüger, Timothy J. Purcell, Naga K. Govindaraju, Ian Buck, Cliff Woolley, and Aaron E. Lefohn. 2004. GPGPU: general purpose computation on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2004, Los Angeles, California, USA, August 8-12, 2004, Course Notes*, page 33. ACM.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. 2023. Towards efficient generative large language model serving: A survey from algorithms to systems. *CoRR*, abs/2312.15234.

Teams Microsoft. 2023. microsoft/deepspeed-mii: Mii makes low-latency and high-throughput inference possible, powered by deepspeed. https://github.com/microsoft/DeepSpeed-MII.

Teams ModelTC. 2023. Modeltc/lightllm: Lightllm is a python-based llm (large language model) inference and serving framework, notable for its lightweight design, easy scalability, and high-speed performance. https://github.com/ModelTC/lightllm.

Teams NVIDIA. 2021. Nvidia/fastertransformer: Transformer related optimization, including bert, gpt. https://github.com/NVIDIA/FasterTransformer.

Teams NVIDIA. 2023. Nvidia/tensorrt-llm: Tensorrt-llm provides users with an easy-to-use python api to define large language models (llms) and build tensorrt engines that contain state-of-the-art optimizations to perform inference efficiently on nvidia gpus. tensorrt-llm also contains components to create python and c++ runtimes that execute those tensorrt engines. https://github.com/NVIDIA/TensorRT-LLM.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently scaling transformer inference. *CoRR*, abs/2211.05102.

Teams ShareGPT. 2023. Sharegpt: Share your wildest chatgpt conversations with one click. https://sharegpt.com/.

Noam Shazeer. 2020. GLU variants improve transformer. abs/2002.05202.

Shuming Shi, Enbo Zhao, Deng Cai, Leyang Cui, Xinting Huang, and Huayang Li. 2024. Inferflow: an efficient and highly configurable inference engine for large language models.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models. *CoRR*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models. *CoRR*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proc. of NeurIPS*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October. Association for Computational Linguistics.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv*.

Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 12360–12371.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

A Appendix

A.1 Extended Results Tables and Figures

	3090	**4090**	**A100**
Memory Size (GB)	24	24	80
Bandwidth (GB/s)	936	1008	2039
BF16 TFLOPs	71	165.2	312

Table 7: The details of three hardwares.

		$s = 32$	$s = 128$	$s = 512$	$s = 1024$	$s = 2048$
TRF	$\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_{QKV}$	7.33	26.27	77.22	129.09	256.14
	$\boldsymbol{Q}, \boldsymbol{K} = \mathrm{RoPE}(\boldsymbol{Q}, \boldsymbol{K})$	4.63	9.94	32.79	63.79	125.63
	$\boldsymbol{O} = \mathrm{Attn}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$	4.15	14.05	112.65	415.68	1544.94
	$\boldsymbol{X} = \boldsymbol{O}\boldsymbol{W}_O$	2.45	8.77	25.75	43.05	85.39
	$\boldsymbol{X} = \mathrm{Add\&Norm}(\boldsymbol{X})$	3.08	5.68	18.47	36.32	71.36
	$\boldsymbol{G}, \boldsymbol{U} = \boldsymbol{X}[\boldsymbol{W}_G, \boldsymbol{W}_U]$	23.73	50.92	175.76	341.57	665.18
	$\boldsymbol{D} = \mathrm{Swish}(\boldsymbol{G}) \odot \boldsymbol{U}$	0.79	2.38	9.23	18.42	36.80
	$\boldsymbol{X} = \boldsymbol{D}\boldsymbol{W}_D$	6.19	16.18	55.85	111.27	221.92
	$\boldsymbol{X} = \mathrm{Add\&Norm}(\boldsymbol{X})$	3.08	5.68	18.47	36.32	71.36
vLLM	$\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_{QKV}$	7.26	17.42	72.59	144.55	299.84
	$\boldsymbol{Q}, \boldsymbol{K} = \mathrm{RoPE}(\boldsymbol{Q}, \boldsymbol{K})$	0.67	1.38	5.34	10.33	20.97
	$\boldsymbol{O} = \mathrm{Attn}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$	2.21	6.00	23.97	54.08	147.36
	$\boldsymbol{X} = \boldsymbol{O}\boldsymbol{W}_O$	2.73	6.98	23.51	46.22	99.65
	$\boldsymbol{X} = \mathrm{Add\&Norm}(\boldsymbol{X})$	0.52	1.25	4.99	9.72	19.42
	$\boldsymbol{G}, \boldsymbol{U} = \boldsymbol{X}[\boldsymbol{W}_G, \boldsymbol{W}_U]$	11.07	32.18	128.37	255.03	527.56
	$\boldsymbol{D} = \mathrm{Swish}(\boldsymbol{G}) \odot \boldsymbol{U}$	0.83	2.39	9.15	17.97	36.36
	$\boldsymbol{X} = \boldsymbol{D}\boldsymbol{W}_D$	5.96	18.66	62.40	123.06	271.20
	$\boldsymbol{X} = \mathrm{Add\&Norm}(\boldsymbol{X})$	0.52	1.25	4.99	9.72	19.42

Table 8: Detailed running time (ms) of Transformers and vLLM when varying sequence length in the prefill stage ($b = 8$).

Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

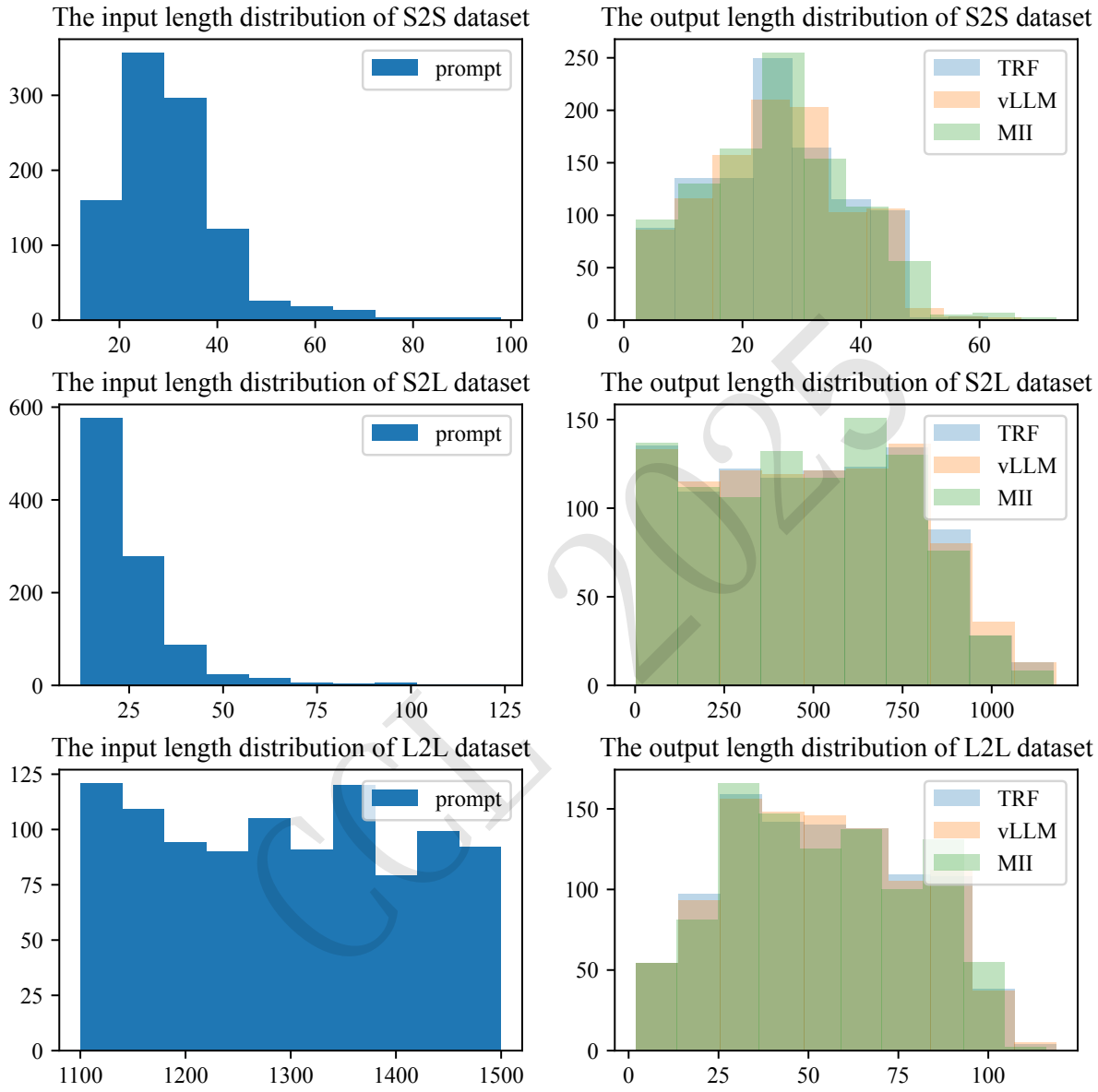


Figure 3: The distribution of input length and output length of three datasets.

Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

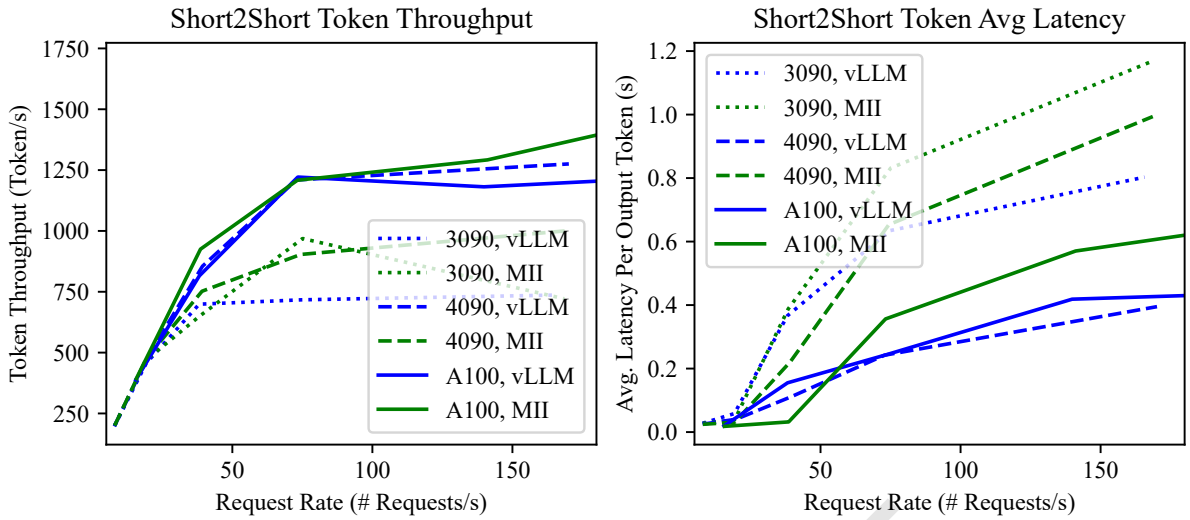


Figure 4: The throughput and latency for serving inference of vLLM and MII using LLaMA-2 (7B) under different request frequencies on the Short2Short dataset.

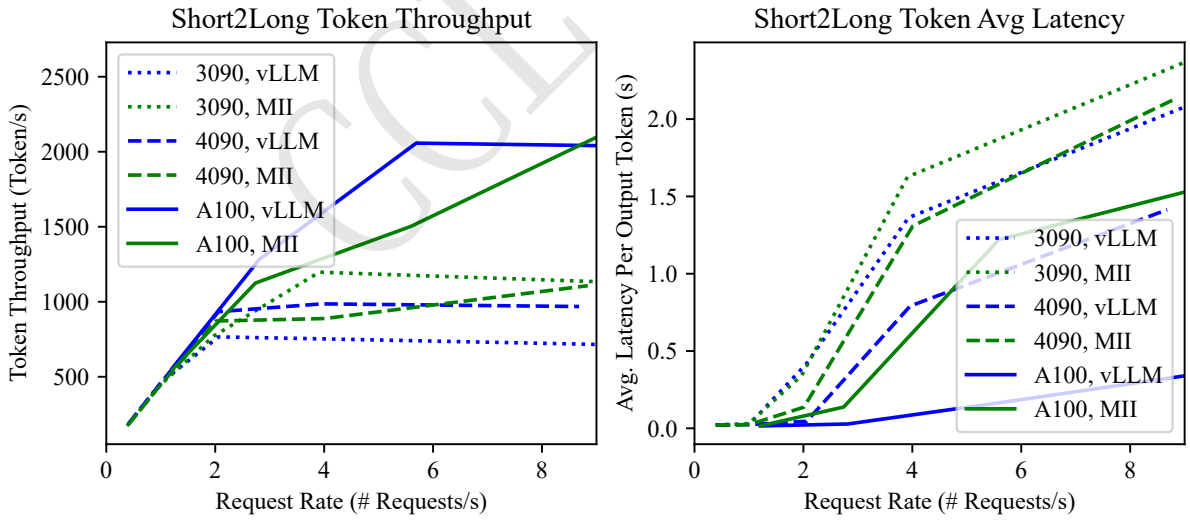


Figure 5: The throughput and latency for serving inference of vLLM and MII using LLaMA-2 (7B) under different request frequencies on the Short2Long dataset.

Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

		$s=32$	$s=128$	$s=512$	$s=1024$	$s=2048$
TRF	$\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}=\boldsymbol{x}\boldsymbol{W}_{QKV}$	2.73	2.72	2.72	2.72	2.70
	$\boldsymbol{q},\boldsymbol{k}=\text{RoPE}(\boldsymbol{q},\boldsymbol{k})$	2.68	2.68	2.66	2.67	2.67
	$\boldsymbol{K},\boldsymbol{V}=\text{Cache}(\boldsymbol{k},\boldsymbol{v})$	1.18	3.05	10.89	21.51	42.70
	$\boldsymbol{o}=\text{Attn}(\boldsymbol{q},\boldsymbol{K},\boldsymbol{V})$	1.65	2.01	3.52	5.48	10.70
	$\boldsymbol{x}=\boldsymbol{o}\boldsymbol{W}_{O}$	0.91	0.91	0.91	0.92	0.90
	$\boldsymbol{x}=\text{Add\&Norm}(\boldsymbol{x})$	1.83	1.82	1.83	1.84	1.80
	$\boldsymbol{g},\boldsymbol{u}=\boldsymbol{x}[\boldsymbol{W}_{G},\boldsymbol{W}_{U}]$	3.88	3.87	3.87	3.88	3.87
	$\boldsymbol{d}=\text{Swish}(\boldsymbol{g})\odot\boldsymbol{u}$	0.27	0.27	0.27	0.27	0.27
	$\boldsymbol{x}=\boldsymbol{d}\boldsymbol{W}_{D}$	2.05	2.05	2.05	2.05	2.04
	$\boldsymbol{x}=\text{Add\&Norm}(\boldsymbol{x})$	1.83	1.82	1.83	1.84	1.80
vLLM	$\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}=\boldsymbol{x}\boldsymbol{W}_{QKV}$	2.11	2.11	2.11	2.11	2.11
	$\boldsymbol{q},\boldsymbol{k}=\text{RoPE}(\boldsymbol{q},\boldsymbol{k})$	0.32	0.31	0.31	0.31	0.31
	$\boldsymbol{K},\boldsymbol{V}=\text{Cache}(\boldsymbol{k},\boldsymbol{v})$	0.38	0.38	0.38	0.38	0.38
	$\boldsymbol{o}=\text{Attn}(\boldsymbol{q},\boldsymbol{K},\boldsymbol{V})$	0.40	0.65	1.81	3.40	5.32
	$\boldsymbol{x}=\boldsymbol{o}\boldsymbol{W}_{O}$	0.90	0.89	0.89	0.89	0.90
	$\boldsymbol{x}=\text{Add\&Norm}(\boldsymbol{x})$	0.27	0.27	0.26	0.26	0.27
	$\boldsymbol{g},\boldsymbol{u}=\boldsymbol{x}[\boldsymbol{W}_{G},\boldsymbol{W}_{U}]$	3.67	3.68	3.66	3.66	3.68
	$\boldsymbol{d}=\text{Swish}(\boldsymbol{g})\odot\boldsymbol{u}$	0.42	0.42	0.42	0.42	0.42
	$\boldsymbol{x}=\boldsymbol{d}\boldsymbol{W}_{D}$	2.03	2.03	2.03	2.02	2.03
	$\boldsymbol{x}=\text{Add\&Norm}(\boldsymbol{x})$	0.27	0.27	0.26	0.26	0.27

Table 9: Detailed running time (ms) of Transformers and vLLM when varying sequence length in the decoding stage ($b=8$).

		$b=1$	$b=2$	$b=4$	$b=8$	$b=16$
TRF	$\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}_{QKV}$	26.28	39.20	77.26	129.09	256.15
	$\boldsymbol{Q},\boldsymbol{K}=\text{RoPE}(\boldsymbol{Q},\boldsymbol{K})$	9.64	17.48	32.82	63.79	125.74
	$\boldsymbol{O}=\text{Attn}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})$	53.97	105.54	209.38	415.68	827.96
	$\boldsymbol{X}=\boldsymbol{O}\boldsymbol{W}_{O}$	8.77	13.08	25.76	43.05	85.40
	$\boldsymbol{X}=\text{Add\&Norm}(\boldsymbol{X})$	5.66	9.77	18.46	36.32	71.41
	$\boldsymbol{G},\boldsymbol{U}=\boldsymbol{X}[\boldsymbol{W}_{G},\boldsymbol{W}_{U}]$	50.93	94.01	175.78	341.57	665.21
	$\boldsymbol{D}=\text{Swish}(\boldsymbol{G})\odot\boldsymbol{U}$	2.36	4.64	9.23	18.42	36.82
	$\boldsymbol{X}=\boldsymbol{D}\boldsymbol{W}_{D}$	16.18	33.54	55.85	111.27	221.92
	$\boldsymbol{X}=\text{Add\&Norm}(\boldsymbol{X})$	5.66	9.77	18.46	36.32	71.41
vLLM	$\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}_{QKV}$	33.58	53.01	85.13	138.82	271.39
	$\boldsymbol{Q},\boldsymbol{K}=\text{RoPE}(\boldsymbol{Q},\boldsymbol{K})$	2.58	3.77	6.22	10.14	20.18
	$\boldsymbol{O}=\text{Attn}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})$	8.16	14.05	26.04	49.95	97.63
	$\boldsymbol{X}=\boldsymbol{O}\boldsymbol{W}_{O}$	6.94	12.37	23.01	44.28	86.57
	$\boldsymbol{X}=\text{Add\&Norm}(\boldsymbol{X})$	1.21	2.43	4.86	9.76	19.53
	$\boldsymbol{G},\boldsymbol{U}=\boldsymbol{X}[\boldsymbol{W}_{G},\boldsymbol{W}_{U}]$	34.39	65.60	124.65	243.48	480.48
	$\boldsymbol{D}=\text{Swish}(\boldsymbol{G})\odot\boldsymbol{U}$	2.37	4.54	8.91	17.62	35.04
	$\boldsymbol{X}=\boldsymbol{D}\boldsymbol{W}_{D}$	18.59	33.70	63.38	115.57	224.88
	$\boldsymbol{X}=\text{Add\&Norm}(\boldsymbol{X})$	1.21	2.43	4.86	9.76	19.53

Table 10: Detailed running time (ms) of Transformers and vLLM when varying batch size in the prefill stage ($s=1024$).

Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

		$b=1$	$b=2$	$b=4$	$b=8$	$b=16$
TRF	$\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}=\boldsymbol{x}\boldsymbol{W}_{QKV}$	2.70	2.71	2.72	2.72	2.72
	$\boldsymbol{q},\boldsymbol{k}=\mathrm{RoPE}(\boldsymbol{q},\boldsymbol{k})$	2.34	2.45	2.61	2.67	2.72
	$\boldsymbol{K},\boldsymbol{V}=\mathrm{Cache}(\boldsymbol{k},\boldsymbol{v})$	2.98	5.59	10.91	21.51	42.77
	$\boldsymbol{o}=\mathrm{Attn}(\boldsymbol{q},\boldsymbol{K},\boldsymbol{V})$	2.47	3.22	3.97	5.48	9.50
	$\boldsymbol{x}=\boldsymbol{o}\boldsymbol{W}_O$	0.91	0.91	0.91	0.92	0.92
	$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	1.40	1.57	1.65	1.84	2.18
	$\boldsymbol{g},\boldsymbol{u}=\boldsymbol{x}[\boldsymbol{W}_G,\boldsymbol{W}_U]$	3.83	3.85	3.88	3.88	3.89
	$\boldsymbol{d}=\mathrm{Swish}(\boldsymbol{g})\odot\boldsymbol{u}$	0.25	0.25	0.26	0.27	0.28
	$\boldsymbol{x}=\boldsymbol{d}\boldsymbol{W}_D$	2.05	2.05	2.05	2.05	2.06
	$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	1.40	1.57	1.65	1.84	2.18
vLLM	$\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}=\boldsymbol{x}\boldsymbol{W}_{QKV}$	2.18	2.10	2.10	2.11	2.14
	$\boldsymbol{q},\boldsymbol{k}=\mathrm{RoPE}(\boldsymbol{q},\boldsymbol{k})$	0.30	0.31	0.31	0.31	0.32
	$\boldsymbol{K},\boldsymbol{V}=\mathrm{Cache}(\boldsymbol{k},\boldsymbol{v})$	0.35	0.37	0.37	0.38	0.39
	$\boldsymbol{o}=\mathrm{Attn}(\boldsymbol{q},\boldsymbol{K},\boldsymbol{V})$	1.22	1.37	1.78	3.41	5.56
	$\boldsymbol{x}=\boldsymbol{o}\boldsymbol{W}_O$	0.89	0.89	0.89	0.89	0.90
	$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	0.27	0.27	0.26	0.26	0.26
	$\boldsymbol{g},\boldsymbol{u}=\boldsymbol{x}[\boldsymbol{W}_G,\boldsymbol{W}_U]$	4.02	3.67	3.66	3.67	3.67
	$\boldsymbol{d}=\mathrm{Swish}(\boldsymbol{g})\odot\boldsymbol{u}$	0.40	0.41	0.42	0.42	0.43
	$\boldsymbol{x}=\boldsymbol{d}\boldsymbol{W}_D$	2.03	2.02	2.02	2.03	2.03
	$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	0.27	0.27	0.26	0.26	0.26

Table 11: Detailed running time (ms) of Transformers and vLLM when varying batch size in the decoding stage ($s=1024$).

		3090	**4090**	**A100**
TRF	$\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}_{QKV}$	243.31	115.54	77.22
	$\boldsymbol{Q},\boldsymbol{K}=\mathrm{RoPE}(\boldsymbol{Q},\boldsymbol{K})$	35.32	17.41	32.79
	$\boldsymbol{O}=\mathrm{Attn}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})$	141.28	89.81	112.65
	$\boldsymbol{X}=\boldsymbol{O}\boldsymbol{W}_O$	81.13	38.52	25.75
	$\boldsymbol{X}=\mathrm{Add\&Norm}(\boldsymbol{X})$	28.13	17.59	18.47
	$\boldsymbol{G},\boldsymbol{U}=\boldsymbol{X}[\boldsymbol{W}_G,\boldsymbol{W}_U]$	657.61	275.45	175.76
	$\boldsymbol{D}=\mathrm{Swish}(\boldsymbol{G})\odot\boldsymbol{U}$	17.07	12.53	9.23
	$\boldsymbol{X}=\boldsymbol{D}\boldsymbol{W}_D$	214.90	91.07	55.85
	$\boldsymbol{X}=\mathrm{Add\&Norm}(\boldsymbol{X})$	28.13	17.59	18.47
vLLM	$\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}_{QKV}$	252.45	100.47	84.92
	$\boldsymbol{Q},\boldsymbol{K}=\mathrm{RoPE}(\boldsymbol{Q},\boldsymbol{K})$	6.35	4.03	6.23
	$\boldsymbol{O}=\mathrm{Attn}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})$	33.26	20.10	22.74
	$\boldsymbol{X}=\boldsymbol{O}\boldsymbol{W}_O$	87.48	36.34	22.95
	$\boldsymbol{X}=\mathrm{Add\&Norm}(\boldsymbol{X})$	6.74	3.86	4.87
	$\boldsymbol{G},\boldsymbol{U}=\boldsymbol{X}[\boldsymbol{W}_G,\boldsymbol{W}_U]$	453.50	184.23	124.33
	$\boldsymbol{D}=\mathrm{Swish}(\boldsymbol{G})\odot\boldsymbol{U}$	12.28	8.49	8.92
	$\boldsymbol{X}=\boldsymbol{D}\boldsymbol{W}_D$	231.07	98.01	63.29
	$\boldsymbol{X}=\mathrm{Add\&Norm}(\boldsymbol{X})$	6.74	3.86	4.87

Table 12: Detailed running time (ms) of Transformers and vLLM when varying hardware in the prefill stage ($b=8, s=512$).

Proceedings of the 24th China National Conference on Computational Linguistics, pages 985-1002, Jinan, China, August 11-14, 2025.

		3090	4090	A100
TRF	$\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}=\boldsymbol{x}\boldsymbol{W}_{QKV}$	5.28	3.86	2.72
	$\boldsymbol{q},\boldsymbol{k}=\mathrm{RoPE}(\boldsymbol{q},\boldsymbol{k})$	1.79	1.04	2.66
	$\boldsymbol{K},\boldsymbol{V}=\mathrm{Cache}(\boldsymbol{k},\boldsymbol{v})$	11.16	6.41	10.89
	$\boldsymbol{o}=\mathrm{Attn}(\boldsymbol{q},\boldsymbol{K},\boldsymbol{V})$	3.64	3.66	3.52
	$\boldsymbol{x}=\boldsymbol{o}\boldsymbol{W}_{O}$	1.76	1.29	0.91
	$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	1.26	0.96	1.83
	$\boldsymbol{g},\boldsymbol{u}=\boldsymbol{x}[\boldsymbol{W}_{G},\boldsymbol{W}_{U}]$	7.13	6.35	3.87
	$\boldsymbol{d}=\mathrm{Swish}(\boldsymbol{g})\odot\boldsymbol{u}$	0.20	0.16	0.27
	$\boldsymbol{x}=\boldsymbol{d}\boldsymbol{W}_{D}$	4.35	3.23	2.05
	$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	1.26	0.96	1.83
vLLM	$\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}=\boldsymbol{x}\boldsymbol{W}_{QKV}$	3.90	3.59	2.11
	$\boldsymbol{q},\boldsymbol{k}=\mathrm{RoPE}(\boldsymbol{q},\boldsymbol{k})$	0.22	0.17	0.31
	$\boldsymbol{K},\boldsymbol{V}=\mathrm{Cache}(\boldsymbol{k},\boldsymbol{v})$	0.26	0.19	0.38
	$\boldsymbol{o}=\mathrm{Attn}(\boldsymbol{q},\boldsymbol{K},\boldsymbol{V})$	2.84	2.57	1.81
	$\boldsymbol{x}=\boldsymbol{o}\boldsymbol{W}_{O}$	1.75	1.26	0.89
	$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{X})$	0.20	0.15	0.26
	$\boldsymbol{g},\boldsymbol{u}=\boldsymbol{x}[\boldsymbol{W}_{G},\boldsymbol{W}_{U}]$	8.06	6.35	3.68
	$\boldsymbol{d}=\mathrm{Swish}(\boldsymbol{g})\odot\boldsymbol{u}$	0.27	0.20	0.42
	$\boldsymbol{x}=\boldsymbol{d}\boldsymbol{W}_{D}$	4.33	3.17	2.02
	$\boldsymbol{x}=\mathrm{Add\&Norm}(\boldsymbol{x})$	0.20	0.15	0.26

Table 13: Detailed running time (ms) of Transformers and vLLM when varying hardware in the decoding stage ($b=8, s=512$).