NanoFlow: Towards Optimal Large Language Model Serving Throughput

Kan Zhu
University of Washington

Yilong Zhao*
University of Washington
Shanghai Jiao Tong University

Liangyu Zhao
University of Washington

Gefei Zuo
University of Michigan

Yile Gu
University of Washington

Dedong Xie
University of Washington

Yufei Gao*
University of Washington
Tsinghua University

Qinyu Xu*
University of Washington
Tsinghua University

Tian Tang*
University of Washington
Tsinghua University

Zihao Ye
University of Washington

Keisuke Kamahori
University of Washington

Chien-Yu Lin
University of Washington

Stephanie Wang
University of Washington

Arvind Krishnamurthy
University of Washington

Baris Kasikci
University of Washington

Abstract

The increasing usage of Large Language Models (LLMs) has resulted in a surging demand for planet-scale serving systems, where tens of thousands of GPUs continuously serve hundreds of millions of users. Consequently, throughput (under reasonable latency constraints) has emerged as a key metric that determines serving systems' performance. To boost throughput, various methods of inter-device parallelism (e.g., data, tensor, pipeline) have been explored. However, existing methods do not consider overlapping the utilization of different resources within a single device, leading to underutilization and sub-optimal performance.

We propose NanoFlow, a novel serving framework that exploits intra-device parallelism, which overlaps the usage of resources including compute, memory, and network within a single device through operation co-scheduling. To exploit intra-device parallelism, NanoFlow introduces two key innovations: First, NanoFlow splits requests into *nano-batches* at the granularity of operations, which breaks the dependency of sequential operations in LLM inference and enables overlapping; then, to get benefit from overlapping, NanoFlow uses an operation-level pipeline with execution unit scheduling, which partitions the device's functional units and simultaneously executes different operations in each unit. NanoFlow automates the pipeline setup using a parameter search algorithm, which enables easily porting NanoFlow to different models. We implement NanoFlow on NVIDIA GPUs and evaluate end-to-end serving throughput on several popular models such as LLaMA-2-70B, Mixtral 8×7B, LLaMA-3-8B, etc.. With practical workloads, NanoFlow provides 1.91× throughput boost compared to state-of-the-art serving systems achieving 59% to 72% of optimal throughput across ported models.

Keywords: LLM Inference, Parallelism, Pipeline

1 Introduction

Large language models (LLMs) have transformed applications such as chatbots, search engines, and office software [27, 33, 43]. Meanwhile, LLMs are extremely resource-intensive. Recent reports showed that nearly 30,000 A100 GPUs are required to serve over 100 million weekly active users just for ChatGPT [36, 42]. Given the massive scale of the LLM serving demands and limited GPUs availablity [15, 48], it is critical to fully utilize available hardware resources. In particular, throughput, i.e., tokens per device per second, has emerged as crucially important since it directly affects the cost of serving.

Compared to previous deep neural networks (DNNs), LLMs have two unique characteristics that make efficient inference significantly harder. First, the model size can be orders of magnitude larger than previous state-of-the-art models. For example, the GPT-3 model, released in 2020, has 175B parameters [8] and requires 5×A100 GPUs just to hold the model weights, while other closed-source LLMs are estimated to be larger. Second, the self-attention mechanism [52], a key component of LLM architecture, requires both time and memory scaling linearly with the context length during token generation. Typically, LLM serving systems will cache the state per request in a *key-value cache* (KV-cache), whose size depends on the context length. Thus, the KV-cache memory usage can even exceed that of the model weights [45].

*Work done at UW.
Corresponding author: Kan Zhu (kanzhu@cs.washington.edu)

Because of these characteristics, LLM inference is commonly assumed to be memory-intensive. Indeed, there have been previous works that study memory optimizations for LLM inference, including optimizing memory accesses [12] and reducing memory fragmentation [20], both of which are critical for improving overall throughput.

However, while each forward pass of an LLM is indeed memory-intensive under small batch sizes, this is not the full story. For throughput-oriented serving where requests form large batches, there are significant differences in resource utilization at the *operation* level . For example, some operations require general matrix-matrix multiplication (GEMM), which consumes significantly more compute resources than memory bandwidth at large batch sizes. Meanwhile, decoding self-attention requires the general matrix-vector multiplication (GEMV) operation, which has the opposite characteristics. Furthermore, for large models, the operation must be sharded across multiple GPUs [24]. This requires collective communication operations such as AllReduce, which is network-intensive.

Different resource usage of operations, when combined with how operations are dispatched, leads to the under-utilization of the most constrained resource. For a single request, each operation depends on the previous operation's output. Thus, within a GPU, existing frameworks execute these operations sequentially, one operation at a time, over the batch of inputs, which is conceptually simple. However, a purely sequential execution approach can lead to underutilization of the most constrained resource. This happens when operations limited by other resources occupy the GPU, leaving the constrained resource idle. Prior work has explored *request-level* and *phase-level* scheduling to improve hardware utilization, including continuous batching [54] and disaggregation of the decode and prefill requests [37, 58]. However, they still do not address the problem of operation level *intra-device* resource under-utilization for the batched requests, which can only be achieved by concurrent execution of different types of operations.

In this paper, we investigate how to approach throughput-optimal LLM serving. Conceptually, by concurrent execution of individual operations that have different compute, memory, and network bottlenecks, we can produce an operation schedule that achieves high utilization of the most constrained resource and approach the maximum serving throughput on a given device. Towards this end, we derive and validate a cost model for LLM serving to show that most of the modern serving workloads are compute-bound. We then propose NanoFlow, which enables intra-device parallelism by overlapping the utilization of compute with memory and network resources inside a single device. NanoFlow has two key innovations: *nano-batching* to expose more concurrency opportunities between operations and *execution unit scheduling* to explicitly map operations to hardware resources.

We use the term nano-batch to differentiate from a micro-batch [18], which pipelines execution *across devices* and at the granularity of one or multiple model layers. In contrast, NanoFlow utilizes intra-device parallelism which pipelines execution *within a device* and at the granularity of *individual operations* within a layer.

To overlap nano-batches, NanoFlow implements execution unit scheduling, which partitions execution units (Streaming Multiprocessors for Nvidia GPUs, Computing Units for AMD GPUs, etc) within a device and overlaps operations by mapping each operation to different partitions. By adjusting the ratio of units assigned to different operations, NanoFlow manages device resources at a finer granularity than existing systems and enables resource overlapping (§ 5.1) to increase throughput.

However, nano-batching and execution unit scheduling alone are insufficient to achieve optimal resource overlapping for the most constrained resource. There is a significant challenge in operation scheduling as there are multiple diverse operations to schedule within each LLM layer, including GEMM, self-attention, communication operations, etc.. Each operation's execution time can differ, depending on the nano-batch size, the number of assigned hardware units, and the kernel implementation. This is in contrast to the previous inference frameworks [24, 54], where the design space is limited to individual kernel implementations due to sequential operation execution. To explore the large design space introduced by intra-device parallelism, NanoFlow provides an automated search mechanism that uses a combination of topological sorting and greedy search to find the optimal schedule given the offline profiles of compute-, memory-, and network-bound kernels (§ 5.5). The automated search enables porting NanoFlow to wide ranges of LLMs with minimal human efforts.

We evaluate NanoFlow on LLaMA-2-70B model [50] using one NVIDIA 8×A100 DGX node [30]. Under practical workloads collected from ShareGPT [1], LMSys [56] and Splitwise [37], NanoFlow achieves on average 1.91× greater throughput compared with state-of-the-art serving frameworks including vLLM [20], DeepSpeed-FastGen [17] and TensorRT-LLM [32], which is 68.5% of the theoretically optimal offline throughput. NanoFlow achieves similar latency compared with TensorRT-LLM while serving up to 1.64× higher online request rate. Besides LLaMA-2-70B, we port NanoFlow to other popular LLMs (LLaMA-3-70B, LLaMA-3-8B, QWen2-72B, Deepseek-67B, and Mixtral 8×7B) and achieve 59%-72% of optimal throughput.

In summary, we contribute the following:

- A detailed analysis and validation of the workload characteristics and the theoretically optimal throughput of LLM serving systems.

- Intra-device parallelism, a novel parallelism paradigm that exploit nano-batching to maximize hardware utilization.
- NanoFlow, an end-to-end serving framework that implements efficient operation co-scheduling using execution unit scheduling and introduces automatic parameter search to automate schedule solutions.
- A comprehensive evaluation of NanoFlow that achieves 1.91× throughput gain compared to baselines and 68.5% of theoretical maximum throughput.

2 Background

In this section, we first briefly introduce the inference process of transformer-based LLMs. Based on that, we classify the operations according to their characteristics. Finally, we introduce the existing optimizations to scale LLM serving.

2.1 LLM inference workflow

Recent LLMs like GPT-4, LLaMA, and Mistral [19, 34, 50] are based on the decoder-only transformer architecture [52]. The inference process of these LLMs consists of two phases [54]: (1) the *prefill* phase, which processes the input prompt all at once, and (2) the *decode* phase, which generates output tokens one at a time auto-regressively. The prefill phase initializes the *KV-cache* [38], which keeps the per-request state to speed up the decode phase.

Both the prefill and decode phases utilize the same inference workflow, as illustrated in Figure 1. Upon receiving an inference request, the input traverses identical decoder layers to produce the subsequent tokens. In each decoder layer, the input is multiplied by weight matrices W_Q, W_K, and W_V to form Query, Key, and Value, with Key and Value concatenated into the existing KV-cache. The Query is then multiplied with the existing Keys and normalized to assess the similarity between the current token and all previous tokens. This similarity is employed to compute the weighted average of the Value, aggregating the context information. The outcome undergoes O-projection through a linear transformation using W_O followed by a layer normalization operation [7]. The activation is subsequently multiplied by W_{up} and W_{gate} separately to generate o_u and o_g. Next, o_u passes through an activation function (e.g., SiLU [14]), followed by an element-wise multiplication with o_u. Finally, W_{down} is applied as Down-projection to generate the output, which serves as the input to the next layer.

2.2 Operation characterization

According to the characteristics of the operations during inference, we can classify them into three categories.

Dense operations The KQV generation, O projection, Up, Gate, and Down projection compute the multiplication of activation and model weights, which we define as dense operations. In the prefill phase, all tokens of new requests form a large batch for these dense operations. In the decode phase, batched decode aggregates the activation from all the decode requests for dense operations [54]. Moreover, as the prefill phase and decode phase share the same weight matrices, the activations from both phases can be combined to further exploit the batching effect by amortizing the weight loading cost [3]. As a result, dense operations become *compute-bound*.

Attention operations In transformer architectures, the relation between tokens is captured using the attention operation. In the prefill phase, all of the query and key vectors are aggregated into Q and K matrices, respectively. The attention score between each token pair is computed using $\text{Softmax}(Q \cdot K^T)$. The attention score is then multiplied by the V matrix to get the attention output. All of K, Q, V matrices have dimension (p, D_{model}), where p is the number of tokens in the prompt and D_{model} is the hidden dimension. Since the p is large, prefill attention is compute-intensive. In contrast, in the decode phase, only one token is generated in each iteration where the token's embedding vector is multiplied by the K and V matrices formed by previous tokens. While the compute is minimal, loading large K and V matrices from the KV-cache makes the decode attention phase memory-bound. Moreover, since different requests have distinct contexts (K and V matrices), batching decode attention doesn't amortize the KV-cache loading time.

Other operations We classify the rest of the operations in transformer architecture, e.g., layer norms, position embeddings, etc., as other operations, due to their short runtime compared to dense or attention operations.

2.3 Serving LLMs at scale

With large models, multiple GPUs are required to provide enough resources, including both memory and compute, for efficient LLM serving [24]. Existing works [16, 41, 57] have comprehensive evaluations of different *inter-device parallelism* paradigms:

Tensor Parallelism [41, 57] splits each weight matrix on different GPUs and executes each operation collectively, thus avoiding weight duplication thereby scaling compute capacity. However, frequent collective communications after operations are required to synchronize each operation's results, as shown in Figure 1. Collective communications involve network-intensive operations, including AllGather (AG) and AllReduce (AR), which calls for high bandwidth interconnection between nodes like NVLink [31]. For dense operations, when weight matrices are split column-wise, AG gathers different output columns to form the entire output. In this case, each GPU sends its sector of output to all other GPUs. However, splitting weight matrix row-wise produces partial GEMM outputs, which are aggregated by AR to get the final result. Specifically, GPU i retrieves ith part of the partial result from all other GPUs and adds all matrices locally. The sum is then broadcast to other GPUs using AG.

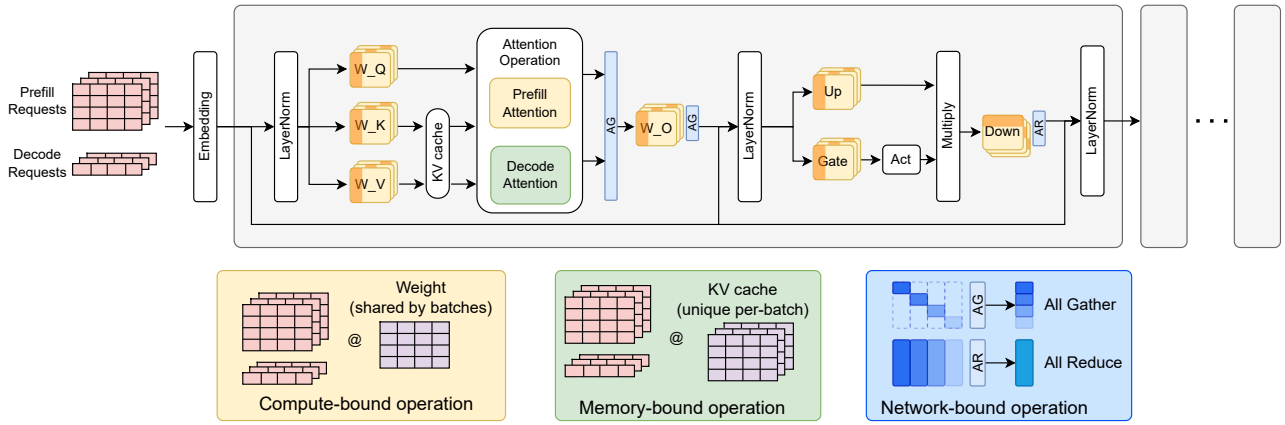


Figure 1. Transformer architecture. The operations in the yellow boxes have large batch sizes and share model weight parameters across different requests; hence, they are compute-bound. On the other hand, operations in green boxes require loading a unique KV cache for each request; hence, they are memory-bound. The blue box represents network operations that perform synchronization between operations.

Pipeline Parallelism [18] splits the model into stages by layers so that each GPU only needs to hold part of the weights. While all devices with tensor parallelism execute on the same batch of data, devices with pipeline parallelism operate on different micro-batches of data split by stages. Since the number of on-the-fly requests is bounded by memory capacity [20], pipeline parallelism must have smaller micro-batch sizes than tensor parallelism.

In practice, these parallelism paradigms are used in combination. Tensor parallelism is widely used on GPUs within a node to enable a larger batch size, while pipeline parallelism is used across nodes to further scale the cluster.

NanoFlow utilizes tensor parallelism within a single node following prior work [41]. Attention is split by heads as described in §4.4, and one AG is then used to generate the full input for the O-projection. O-projection is partitioned by columns, and its output is gathered for the Up and Gate projections. These projections are also split into columns to avoid the high communication costs associated with the large intermediate dimension. Finally, one AR is needed after the Down projection to produce the final output. In total, two AGs and one AR operation are needed for NanoFlow's inference workflow.

3 Analysis

In this section, we first introduce the key factors that determine the throughput of LLM serving (§3.1) and model the resource requirements of LLM serving (§3.2). Based on this modeling, we provide some case studies (§3.3), showing that most popular workloads are compute-bound. Then, for the compute-bound scenarios, we determine optimal serving throughput (§3.4) and validate the cost model with a case study (§3.5). We then explain why existing systems achieve throughput that is far from the optimal (§3.6) and how our proposed approach, intra-device parallelism, solves this issue and allows approaching optimal throughput (§3.7).

3.1 Key factors of serving throughput

In this paper, we define throughput as the *total throughput* of LLM serving, which is the number of tokens processed per second by both the prefill and the decode phases. Although there are other types of throughput metrics [20, 54], like decoding throughput and requests per second (RPS), these can be easily derived from the total throughput, as shown later.

Throughput is determined by three key factors, namely, hardware specification, model configuration, and user query statistics.

Hardware specification. The following factors determine the available GPU hardware resources:

- $MemBW$ (GB/s): the maximum memory bandwidth
- $MemSize$ (GB): the maximum memory capacity
- $Compute$ (GFLOP/s): the maximum compute capacity
- $NetBW$ (GB/s): the interconnect bandwidth

We list the hardware resources for widely used GPUs from both NVIDIA and AMD in Table 1.

Model configuration. The following factors regarding the model architecture determine the computation, memory, and network demand when serving LLMs:

- D_{model}: hidden dimension size
- L: number of layers in the model
- P_{Model}: number of parameters in the model
- R_{GQA}: group size of GQA [1]

[1]The number of attention heads for a shared KV head [6].

- *Dtype* (Bytes): size of the data type for model parameters measured in bytes (e.g., 2 for FP16)

User query statistics. The following user input statistics also affect the system throughput:

- p: average number of tokens in prompts to be prefilled
- d: average number of tokens in output to be decoded

Therefore, an inference request from a user corresponds to p prefill tokens and d decode tokens on average, for a total of $p + d$ tokens that we use when calculating the total throughput. Using user query statistics, total throughput (tokens generated per second) can be easily converted to other throughput metrics:

$$\text{Throughput}_{\text{decoding}} = \frac{d \cdot \text{Throughput}_{\text{total}}}{p + d}$$

$$\text{Throughput}_{\text{rps}} = \frac{\text{Throughput}_{\text{total}}}{p + d}$$

3.2 Modeling of optimal LLM serving

We first note that an optimal serving system needs to process large batches to achieve high throughput. For compute-bound dense operations, e.g. GEMMs, a larger batch amortizes the weight loading overhead and increases execution unit occupancy, which provides higher compute utilization. For memory- and network-bound operations, all requests share the same warmup cost (e.g., kernel launching overhead, pipeline setup overhead, network synchronization overhead, etc.), so a larger batch size leads to higher throughput. In both cases, beyond a certain threshold, the batching effect diminishes, and the system throughput converges to peak throughput. Therefore, when considering the optimal system throughput, we assume that the system always operates at the largest batch size at which the total available memory can hold the model weights and all the KV caches for the requests. For simplicity, we assume that the memory consumption of activations is negligible compared to that of model parameters and the KV-cache, since prior work[20] demonstrated that less than 5% of GPU memory is consumed by activations during LLM inference.

Next, we derive the latency of an LLM serving iteration (where a batch of user requests are being processed by all the transformer layers) from the perspectives of required memory, compute, and network resources.

Memory. From the memory perspective, latency is:

$$T_{mem} = \frac{MemSize}{MemBW} \tag{1}$$

This is because the entire device memory content needs to be loaded into the GPU caches and registers once for each iteration (typical in modern serving systems [20]) when using the largest batch size. Although the model weights are shared between iterations, due to the long reuse distance, it is infeasible to cache the model weights and avoid loading from the device memory.

Table 1. Release year, network bandwidth, memory bandwidth and memory bandwidth to compute ratio for various GPU models. For modern GPUs, the memory bandwidth to compute ratio tends to stabilize around 250 FLOP/B.

GPU Model	Release Year	NetBW (GB/s)	Compute (FP16 GFLOP/s)	MemBW (GB/s)	Ratio (FLOP/B)
V100	2017	300	125,000	900	139
A100 - 40G	2020	600	312,000	1555	200
A100 - 80G	2021	600	312,000	2000	156
H100	2023	600	989,000	3352	295
H200	2024	900	989,000	4800	206
B100	2024	1800	1,800,000	8000	225
B200	2024	1800	2,250,000	8000	281
MI250	2021	800	362,000	3352	107
MI300	2023	1024	1,307,000	5300	246

Compute. The latency of an iteration from the compute perspective depends on the total number of computations (floating point additions and multiplications) involved in dense operations, prefill attention, decode attention, and networking. Dense operations produce the vast majority of computations (as we validate in §3.5), therefore we model the compute latency of dense operations below. More details on the other, relatively minor, contributors to compute latency are in Appendix A.

To estimate the number of computations in dense operations, we first note that for every GEMM in dense operations, $2B_{Dense}N_wK_w$ computations need to be performed on weights and activations[2], where N_w and K_w are dimensions of the weight matrices, and B_{Dense} is the batch size for dense operations. We derive the total compute for dense operations as $2B_{Dense}\sum N_wK_w$. Note that the $\sum N_wK_w$ term is the number of weight elements in all layers, which can be approximated using P_{Model} (the parameters in the model), thus, we simplify the expression as $2B_{Dense}P_{Model}$.

We then derive B_{Dense} as a function of user requests. Let B_{req} be the total number of user requests in one batch. For each user request, the prefill stage occurs only for 1 iteration while the decode stages occur for d iterations. Hence, among B_{req} user requests, $\frac{B_{req}}{d+1}$ requests are in prefill stages, and $\frac{d \cdot B_{req}}{d+1}$ requests are in decode stages, on average. As one request in the prefill stage contains p tokens, this translates to p activation vectors. Therefore, the average batch size for dense operations, (B_{Dense}), is:

$$B_{Dense} = p \cdot \frac{B_{req}}{d+1} + \frac{B_{req}d}{d+1} = \frac{B_{req}(p+d)}{d+1} \tag{2}$$

Putting it all together, the latency for computing an iteration is:

[2] $MN(2K-1)$ arithmetic operations are needed for a GEMM of $[M, K] \times [K, N]$. For simplicity, we approximate this with $2MNK$.

$$T_{Compute} \approx \frac{2B_{Dense} \cdot P_{Model}}{Compute} \tag{3}$$

$$= \frac{2B_{req}P_{Model}}{Compute} \frac{p+d}{d+1} \tag{4}$$

Given our assumption that the system operates at the largest batch sizes, we further derive the average request batch size B_{req} from the KV-cache capacity limits. We define E_{kv} as the maximum number of elements in KV-cache that the system can hold, which can be calculated from $MemSize$ and P_{Model} (See §A). As each request requires a KV-cache with the capacity of $p + d/2$ tokens on average[3], and the KV-cache of one token consists of $\frac{2D_{model}L}{R_{GQA}}$ elements[4], B_{req} can be expressed as follows:

$$B_{req} = \frac{E_{kv}}{p + d/2} \frac{R_{GQA}}{2D_{model}L} \tag{5}$$

Therefore, Equation 4 can be rewritten as:

$$T_{Compute} \approx \frac{P_{Model}}{Compute} \frac{p+d}{d+1} \frac{E_{kv}}{p+d/2} \frac{R_{GQA}}{D_{model}L} \tag{6}$$

Network. Finally, for tensor parallelism, collective network communication primitives are needed to synchronize the results after performing operations in multiple GPUs over different shards of data. The size of the data transmitted with collective communication equals to the size of inputs for dense operations. To support tensor parallelism, two AGs and one AR are required. As described in §2.3, AR requires gathering output from other GPUs, performing local summation, and broadcasting results, thus needs to transfer twice the activations, while AG transfers the activation once. Therefore, we can estimate the total data movement in bytes as $4 \cdot B_{Dense}D_{model}Dtype \cdot L$, where the constant term 4 comes from two AG and one AR. We can further derive the latency of a serving iteration from the network perspective as:

$$T_{net} = 4 \cdot \frac{B_{Dense}D_{model}Dtype \cdot L}{NetBW} \tag{7}$$

3.3 Classifying LLM serving workloads

We classify LLM serving workloads by comparing the latency of an iteration derived based on memory, compute, and network requirements.

First, we compare Equation 7 (network time) with Equation 1 (memory time). As activation size $B_{Dense}D_{model}Dtype \cdot L$ is orders of magnitude smaller than the whole GPU memory size[20], while the network bandwidth is typically of the same magnitude as memory bandwidth (Table 1), the network is usually not a bottleneck.

Therefore, we focus on $T_{Compute}$ and T_{Mem} in the rest of the section. Comparing Equation 1 (memory time) and Equation 6 (compute time) determines the characteristics of the workload, namely, the following value

$$T_R = \frac{T_{Mem}}{T_{Compute}}$$

$$\approx \frac{Compute}{MemBW} \frac{MemSize}{E_{kv}} (d+1) \frac{p+d/2}{p+d} \frac{D_{model}L}{R_{GQA}P_{Model}} \tag{8}$$

If $T_R > 1$, the workload is memory-bound, and if $T_R < 1$, it is compute-bound.

Based on Equation 8, we can qualitatively analyze how different parameters affect workload characteristics. As shown in Table 1, the ratio of $Compute/Bandwidth$ is centered around 250 FLOP/B for modern data center GPUs, indicating the first multiplier in the Equation 8 remains relatively stable across GPU models. Smaller $Compute/Bandwidth$ moves the workload towards compute-bound. Since the KV cache would fill all the GPU memory excluding the model parameters, the ratio $\frac{MemSize}{E_{kv}}$ is also stable.

From a model perspective, a larger R_{GQA} reduces the value of T_R, indicating that models employing Group Query Attention tend towards being more compute-bound.

In terms of user query statistics, a longer decode length with fixed prefill length increases the value of T_R, pushing the workload towards being more memory-bound, while the term $\frac{p+d/2}{p+d} = 1 - \frac{1}{2}\frac{1}{p/d+1}$ limits the effect of prefill length.

Figure 2 shows the values of T_R with three representative models: LLaMA-2 7B [50] with 1×A100, Mistral 7B [19] with 1×A100, and LLaMA-2 70B with 8×A100s. As the figure

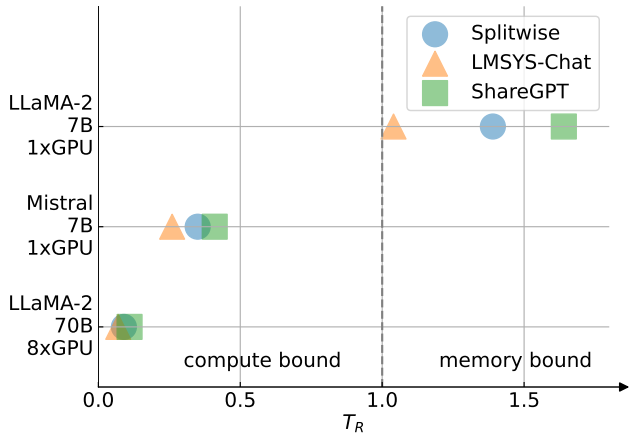


Figure 2. Classification of workloads. The figure shows the workload characteristics of the representative model and input datasets, according to Equation 8. Except for the case with the LLaMA-2 7B model on a single GPU, the workloads are heavily compute-bound.

[3]We add half of d because decode length gradually increases the KV-cache size over time, averaging half its final length due to linear accumulation.
[4]we multiply by 2 to account for Key and Value data, and divide by R_{GQA} because the KV-cache size is smaller than the hidden dimension size in models with GQA due to the reduced number of attention heads.

Table 2. Comparison of operation runtimes between cost model predictions and real-world measurements when serving the Llama2-70B model on 8 A100 GPUs with a 2K dense batch. We calculate the compute, memory, and network usage of every operation in an iteration and use hardware specifications to derive $T_{compute}$, T_{mem}, and T_{net}. For each operation, the most constrained resource, which has the longest estimated runtime, is shown in bold. The estimations align with the actual runtime presented in the last column. To model the overall serving cost, we sum $T_{compute}$, T_{mem}, and T_{net} for all operations. Since $T_{compute}$ is the dominant factor, serving large batches is primarily compute-bound.

Operation	Compute (GFLOP)	Memory Movement (GB)	Network Usage (GB)	$T_{compute}$ (ms)	T_{mem} (ms)	T_{net} (ms)	Measured Time (ms)
GEMM-KQV	27487.8	19.5	0	**11.01**	1.22	0	**16.08**
GEMM-O	21990.2	16.1	0	**8.81**	1.01	0	**16.01**
GEMM-UG	153931.6	96.6	0	**61.67**	6.04	0	**69.92**
GEMM-D	76965.8	49.7	0	**30.84**	3.11	0	**34.96**
Decode Attention	3665.9	462.2	0	1.47	**28.89**	0	**35.60**
Prefill Attention	916.3	2.1	0	**0.37**	0.13	0	**4.56**
Communication	18.8	75.2	75.2	0.01	4.70	**31.33**	**47.92**
Total				**114.17**	45.09	31.33	

shows, many of the workloads (Splitwise [37], LMSYS-Chat-1M [56], and ShareGPT [1]), except for LLaMA-2 7B model, are uniformly compute-bound. This is because Mixtral 7B and LLaMA-2 70B models adopt GQA, which is a common practice for state-of-the-art models including LLaMA-3 [49], NeMo [5], and Qwen2 [11]. Based on this analysis, we will focus our discussion and evaluation on compute-bound scenarios for the remainder of the paper.

3.4 How to achieve optimal serving throughput?

Optimal throughput (measured by token/s) can be achieved when the most limited resource is fully utilized. In the compute-bound case, using Equation 3, the optimal throughput of the system is derived as:

$$\begin{aligned} \text{Throughput}_{\text{optimal}} &= \frac{B_{Dense}}{T_{Iter}} = \frac{B_{Dense}}{T_{Compute}} \\ &= \frac{Compute}{2P_{Model}} \end{aligned} \tag{9}$$

Therefore, the optimal throughput depends **only** on the GPUs' aggregated computing power for the corresponding data type and the number of parameters in the model. Other factors, including the GPU memory size, bandwidth, the data type of the model, or the length of prefill/decode, do not affect the optimal throughput substantively.

For serving the LLaMA-2 70B model on 8×A100 GPUs, for instance, the optimal throughput is:

$$\frac{8 \times 312 \times 10^{12}}{2 \times 70 \times 10^{9}} \approx 17828 \text{ (token/s)}$$

3.5 Case study: serving Llama2-70B models at a large batch size

To validate our cost model, we apply our model to Llama2-70B serving at a dense batch size of 2048 requests with 8 NVIDIA A100 GPUs. We set the average prefill length to 512 and the average decode length to 1024. We calculate the compute GFLOP, memory movement, and network traffic of each operation, as shown in Table 2. Based on the numbers of operations, we then calculate $T_{compute}$, T_{mem}, and T_{net} using hardware specifications[5]. The longest estimated time $T_{op} = \max(T_{compute}, T_{mem}, T_{net})$ indicates the most constrained resource. To validate these results, we profile the running time of different operations. For most operations, T_{op} aligns with real-world profiling results. One exception is prefill attention, where the real measured time is quite larger than the times computed based on our model and hardware specifications. This is because even though prefill attention itself takes a relatively short time, the launching overhead of the associated kernels dominates total time.

The sum of all operations' $T_{compute}$, T_{mem}, and T_{net} shows the compute is the most constrained resource, which provides supporting evidence for our cost model.

3.6 Why do existing serving systems fall short?

Optimal LLM serving requires maximum resource utilization throughout the inference processes. However, existing serving systems do not fully utilize GPU resources. Figure 3 demonstrates the workflow of existing frameworks.

For Orca [54] and vLLM [20], prefill and decode are executed separately. For DeepSpeed-FastGen [17], the prefill and decode dense operations are fused, and then prefill and decode attention are performed separately. All existing frameworks support serving large models on multiple GPUs using network communication. However, since LLM serving is largely compute-bound, the aforementioned sequential execution of operations prevents achieving maximum resource

[5]We use one-way network bandwidth for T_{net} calculation

utilization, as the decode attention and network communications use negligible amount of compute.

3.7 How can intra-device parallelism solve the issue?

The aforementioned inefficiency of sequential operation execution motivates the design of intra-device parallelism, where we propose to overlap the execution of operations with distinct resource demands to maximize the utilization of the most limited resource, i.e. compute. Specifically, the idle compute resources when performing decode attention and network communications can be utilized by concurrent dense operations. Intra-device parallelism makes it possible to close the gap to optimal serving further.

4 Design

In this section, we first give a brief overview of NanoFlow's architecture in §4.1, followed by a detailed description of each system component(§4.2, §4.3, §4.4).

4.1 Overview of NanoFlow's architecture

The high-level design of NanoFlow is illustrated in Figure 5 alongside a potential deployment of NanoFlow on GPU clusters. As batch size is an important factor of hardware utilization, all incoming requests are handled by a global batch scheduler, which selects the best dense batch sizes (as those matter the most for improving compute efficiency) based on offline profiling as we explain shortly. Within each GPU, intra-device parallelism engine further splits the global batch into nano-batches and partitions the execution units to enable parallel nano-batch executions. Furthermore, to maximize memory utilization for larger batch sizes, NanoFlow eagerly accepts new requests with an estimation of GPU memory usage and timely offloads the KV-cache of retired requests to lower memory hierarchies (e.g., CPU and SSD).

4.2 Global batch scheduler

As the batch size is limited by memory capacity and it is crucial for increasing throughput, batch scheduler eagerly adds new requests into the global batch if the GPU memory allows (unlike for latency-optimized serving). To maximize the benefits of batching, the batch scheduler implements continuous batching introduced by Orca [54], which refills the on-the-fly global batch in each iteration. NanoFlow also adopts chunked prefill similar to Deepspeed-FastGen [17] and Sarathi [4], which combines prefill requests with decode requests to form a large batch. Besides, NanoFlow's batch scheduler introduces *discrete batching*, which chooses the dense batch size B_{dense} among a few high-performance batch sizes instead of arbitrary ones to maximize the efficiency of dense operations. For example, we find that the GEMM kernel performance in CUTLASS drops by 7% if only a single request is added to a batch of size 2048. Therefore, launching a batch of 2048 instead of 2049 is much more efficient. Batch scheduler identifies the high performance batch sizes through offline profiling and dynamically chooses the system batch size according to the system load.

In addition to the batching strategy, the batch scheduler also determines the assignment of on-the-fly requests to different GPUs. Since pipeline parallelism feeds micro-batches to different GPUs, which greatly decreases the batch size for dense operations(§2.3) and leads to underutilization of compute, batch scheduler adopts tensor parallelism within a node and simply feeds the entire batch into all GPUs instead of micro-batching.

4.3 Intra-device parallelism engine

After being scheduled by the batch scheduler, the requests are assigned to GPUs, where the intra-device parallelism engine splits a batch into nano-batches and overlaps them with execution unit scheduling.

Nano-batching. LLM operations have sequential dependencies, which makes it ordinarily hard to overlap the operations for a given batch of requests. Existing approaches to alleviate this such as micro-batching operate at the layer level [18], which is too coarse to achieve operation-level intra-device parallelism. The key idea in nano-batching is to split a batch of user requests into smaller nano-batches, thereby allowing overlapping operations across different nano-batches. For example, as shown in Figure 6, the network operation (Net) can be effectively overlapped with the compute operation (Com) with nano-batching.

Nano-batching enables intra-device parallelism in NanoFlow. We demonstrate the example workflow of LLaMA-2-70B in Figure 4, where solid and dotted blocks represent two different nano-batches. We note that within a given nano-batch, all of the operations follow sequential dependencies.

To overlap operations, at least two nano-batches are required. However, given a fixed global batch size, increasing the number of nano-batches also reduces the batch size for the operations and thus hurts performance due to reduced batching effect for dense operations. Therefore, in NanoFlow's design, wherever possible, we aim to minimize the number of nano-batches. Thus, we split O, UGD, and network operations to two nano-batches. However, since decode attention (GEMV) is data-dependent on KQV, splitting KQV only into two nano-batches would delay the launching of GEMV, thus further delaying O1 and O2, resulting in pipeline bubbles. Therefore, we use four nano-batches for both KQV and decode attention operations. Moreover, O2 needs to wait for all decode attention to finish. The GEMV AllGather between GEMV2-2 and O2 (shown as crossed out in Figure 4) would further push back the execution of O2, leading to sub-optimal performance. Therefore, we partition O2 row-wise to avoid the aforementioned AllGather and instead use an AllReduce after O2 to generate the final output, which can be effectively overlapped by UGD1 without pipeline bubbles.

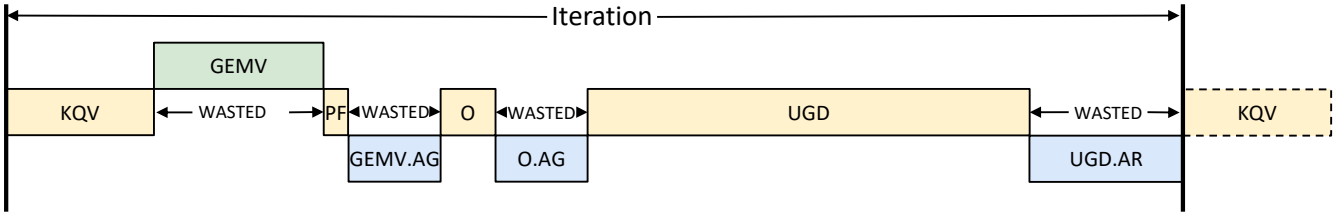


Figure 3. Execution pipeline of existing systems. The green, yellow, and blue colored operations correspond to memory-, compute-, and network-bound operations. GEMV denotes decode attention and PF denotes prefill attention. Operations in the previous and next layer are denoted by dotted borders. "wasted" shows the stages in the pipeline where the most constrained resource, compute, is underutilized. Small operations (i.e. layernorm, activation, etc.) are omitted for simplicity.

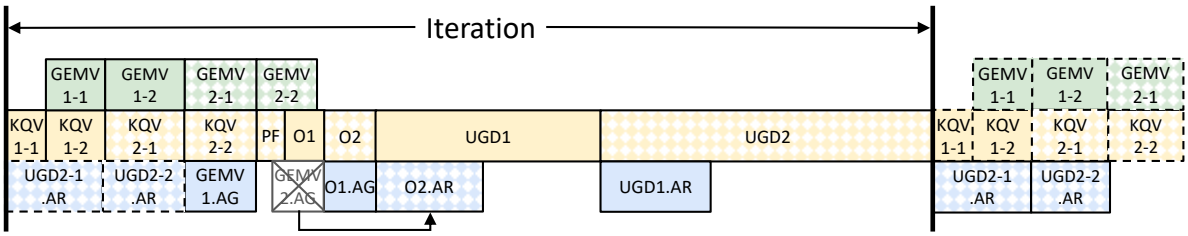


Figure 4. Execution pipeline of NanoFlow. The solid background and shaded background represents operations in two nano-batches that are co-scheduled on the same device. By overlapping the compute-, memory-, and network-intensive operations, NanoFlow increases compute utilization and improves the serving throughput.

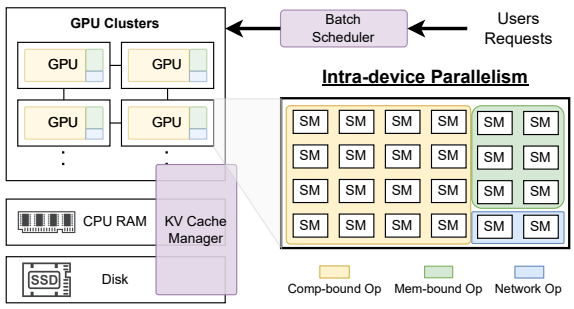


Figure 5. Architecture of NanoFlow on NVIDIA GPUs. NanoFlow can be applied to other accelerators (e.g., AMD MI300) that have multiple functional units within a single device.

NanoFlow's pipeline design is tailored for the most memory-intensive workload, where all requests are in the decode stage and the associated KV-caches of these requests fully occupy the GPU memory except for model weights. Due to this conservative design decision, in less extreme cases where there are fewer decode requests, dense operations can always effectively overlap decode attention. Therefore, the pipeline design can fit all workloads.

Execution unit scheduling. Even with nano-batching which enables overlapping operations, resource utilization can't be improved without the careful management of the execution of actual operations.

Given the pipeline design, NanoFlow automatically searches the execution plan to make sure operations can be effectively overlapped. The objective of the search is to determine the size of each nano-batch and the allocation of execution units to the operations in the pipeline. In line with prior work [26], a key goal of the automated search is to reduce kernel interference. This is crucial, because, as shown by some of our experiments, launching GEMM and GEMV operations together without managing the over-subscription of hardware execution units (e.g., SMs in NVIDIA GPUs) decreases the GEMM performance significantly by up to 2.5×. We describe the detailed implementation of the automated search in § 5.5.

4.4 KV-cache manager

Peak memory estimation. To maximize the usage of GPU memory while minimizing the chance of out-of-memory, NanoFlow's KV-cache manager predicts the peak future memory usage using workload characteristics. The KV-cache manager keeps track of the number of decoded tokens for each ongoing request and predicts the completion of the request, assuming the total decode length of the requests equals the average decode length. The KV-cache manager continuously calculates the highest future memory usage and dispatches new requests only if the estimated peak memory

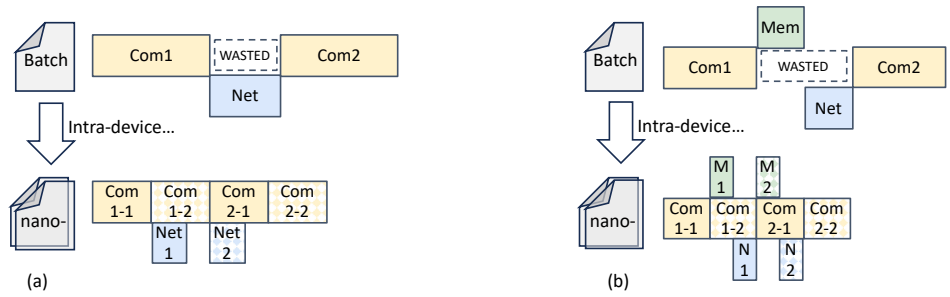


Figure 6. Illustration of nano-batches on typical workloads: the upper figures represent the original workflow while the below ones adopt intra-device parallelism. (a) Typical workload within FFN. By splitting nano-batches, compute operations (`Com1` and `Com2`) are executed in four stages with two nano-batches. Therefore, the network operations (`Net1` and `Net2`) can be overlapped with the `Com12` and `Com21`, respectively (b) Typical workload within self-attention operations. The memory and network operations can also be effectively overlapped, leaving only the compute operations on the critical path.

is less than the total GPU memory. Rarely, when the system runs out of memory, the KV-cache manager discards a request to reclaim memory.

Head-parallelism for the KV-cache NanoFlow, in line with prior work [41], splits the KV-cache by the head dimension to efficiently scale over GPUs, which naturally guarantees workload balance across GPUs. Moreover, NanoFlow partitions the weight matrices for KQV generations column-wise so that GPU i gets the i-th portions of the W_k, W_q and W_v matrices which generates the corresponding attention heads of K, Q, and V used for attention without any network communications.

Asynchronous KV-cache offload To efficiently serve multi-round workloads, NanoFlow eagerly offloads the KV-cache of finished requests to SSDs to minimize re-computation. In one iteration, NanoFlow selects the KV-cache of the finished request and copies them to the host in parallel to the on-the-fly dense operations, e.g., Up and Gate projection operations, and store to SSDs asynchronously. The SSDs feature extremely high capacity, which can easily reach $\sim 100\times$ of the GPU memory and $\sim 20\times$ of the CPU memory, making them a perfect device to hold numerous requests' KV-caches. Moreover, modern SSDs deliver high bandwidth, which can reach $3GB/s/SSD$. The peak offloading bandwidth can be estimated using the system's optimal throughput, as all the KV-caches of generated tokens are finally moved to SSDs. Therefore, for LLaMA-2-70B model, offloading bandwidth is only $5.4GB/s$[6], which can be handled by two SSDs. The memory hierarchy is managed by the Least-Recent-Used (LRU) policy. When an existing request is re-activated in multi-round conversation, NanoFlow loads the KV-cache to the GPU in parallel to the dense operations and starts the prefill stage in the following iteration. Note that both offloading and uploading are executed layer-by-layer to avoid pipeline bubbles.

[6]Offloading bandwidth = throughput $\times 2 \times Dtype \times D_{model} \times L/R_{GQA}$ = $17828 \times 2 \times 2 \times 8192 \times 80/8 = 5.4$ GB/s

5 Implementation

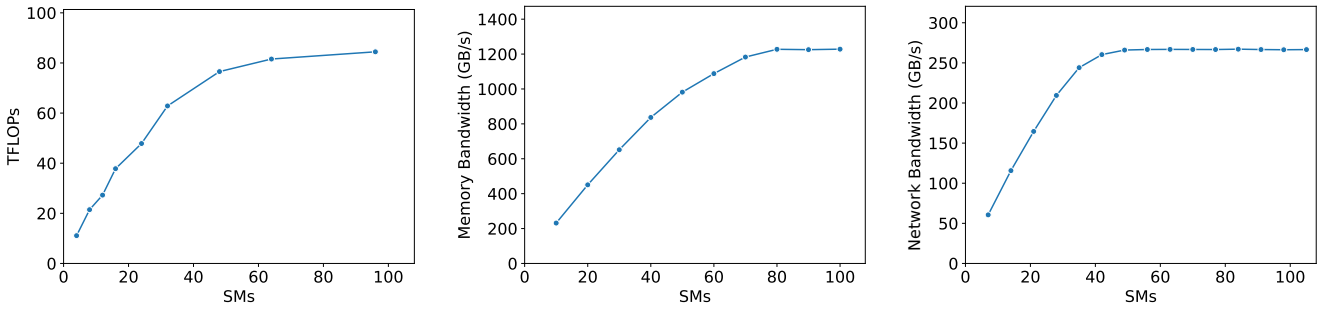
We implement the proposed NanoFlow pipeline on NVIDIA GPUs, one of the widely used hardware accelerators for LLM workloads. In this section, we provide implementation details of execution unit scheduling (§5.1), CUDA kernels of operations (§5.2), top-level scheduler (§5.3), and KV-cache manager (§5.4). We further discuss the automated search for the adaptive deployment of NanoFlow(§5.5), which we showcase by porting NanoFlow to other representative models in §5.6. NanoFlow is implemented with $\sim$ 4000 lines of CUDA, and $\sim$ 2000 lines of Python code.

5.1 Execution unit scheduling for intra-device parallelism

NanoFlow utilizes intra-device parallelism to increase hardware utilization and maximize throughput. To enable intra-device parallelism, operations with distinct resource demands, such as compute, memory, and network, must be executed simultaneously.

One possible way to achieve kernel overlapping is Horizontal Fusion [22]. By integrating multiple operations into one single kernel, we can control the hardware mapping at fine granularity. However, horizontal fusion needs to manually fuse complex kernels (e.g., FlashAttention, GEMM), leading to substantial implementation burdens. Moreover, GPU kernels such as GEMMs in CUTLASS are tuned for individual GPU generations and are fast evolving. Horizontal fusion has limited flexibility in embracing these kernel changes. Additionally, fused kernels are treated as one compilation unit, which is the smallest granularity the compiler uses to allocate hardware resources like registers and shared memory. Since operations have distinct hardware resource demands, a single scheduling plan cannot accommodate both effectively, resulting in performance degradation.

Another common way to run concurrent kernels on a GPU is launching them via CUDA Multi-Stream [23]. This approach does not break the kernel abstraction and enables



(a) GEMM **(b)** Decode Attention **(c)** All-gather Communication

Figure 7. Performance of compute-, memory- and network-bound operations with different numbers of SMs. The non-linear relationship between the number of SMs and the kernel performance exists for all kinds of operations.

the kernels to evolve independently. However, our experiments show that the GPU scheduler may refuse to run multiple kernels concurrently if both kernels demand a large number of execution units. Even worse, kernels can have strong interference with each other. For example, directly launching GEMM and GEMV kernels together via CUDA Stream will lead to 2.5× performance degradation of GEMM.

Therefore, we design a custom execution unit scheduling, which limits the execution units (i.e. SMs for NVIDIA GPU) usage of each kernel when launching different operations with CUDA Multi-Stream. With controllable execution units, execution unit scheduling effectively mitigates kernel interference and exploits the benefit of high resource utilization.

The key insight that supports execution unit scheduling is the non-linearity of GPU resource utilization. We demonstrate the non-linearity of memory, network, and compute in Figure 7, which shows the relative performance with different numbers of SMs assigned. For example, with only 35 SMs out of 108 SMs (32%), network kernels can achieve up to 92% peak performance, which indicates that the network resource utilization is *non-linear* to the number of SMs. In other words, per-SM performance achieves maximum before the operator uses all the SMs. Therefore, by overlapping multiple operations with limited SM usage, every individual operation can execute with better SM efficiency, which contributes to higher resource usage.

5.2 CUDA kernels

To implement execution unit scheduling, we provide our dedicated kernel implementations for each operation used in NanoFlow, including dense operations, self-attention, network, and other operations. Note that the implementation mechanism is compatible with existing kernel libraries, enabling NanoFlow's flexibility on different models.

Dense Operations. For both generality and efficiency considerations, we use state-of-the-art open-sourced GEMM kernel library, CUTLASS [46], to support NanoFlow's GEMM operations with different shapes and numbers of SMs. For each input shape, we use the CUTLASS Profiler [47] to determine the most efficient GEMM kernel configuration.

Attention. We implement attention operations based on a widely-used kernel library, FlashInfer [53]. To implement SM-constrained attention operations with less human effort, we propose a thread block mapping mechanism. In an operation with M thread blocks, we remap the workloads to only N running thread blocks (where $M > N$) without changing the workload partition. Instead, each thread block will sequentially execute the workload that was originally assigned to $\lceil M/N \rceil$ thread blocks.

Network. Network kernels in NanoFlow are implemented using an inter-GPU communication library, MSCCL++ [10]. We implement SM-constrained versions of AllGather, ReduceScatter, and AllReduce kernels.

Other operations. NanoFlow does not specialize in other operations, including Layer Normalization, Rotary Position Embedding (RoPE) [44], and GeLU [52], since these operations consume fewer resources compared to the aforementioned ones. Following common practice, NanoFlow fuse these operations with dense, attention or network kernels to further reduce the overhead.

5.3 Top-level scheduling

NanoFlow uses batch scheduler and KV-cache manager to conduct the top-level scheduling, which is adapted from a multi-tenants serving framework, Punica [9]. We illustrate our design in Figure 9.

To mitigate the scheduling overhead of CPU, both batch formation and KV-cache management are designed to operate parallel to the GPU executions of kernels via an asynchronous manner. At any iteration i, NanoFlow makes batching decisions and allocates the KV-cache entries for the next iteration before the end of the current iteration. NanoFlow

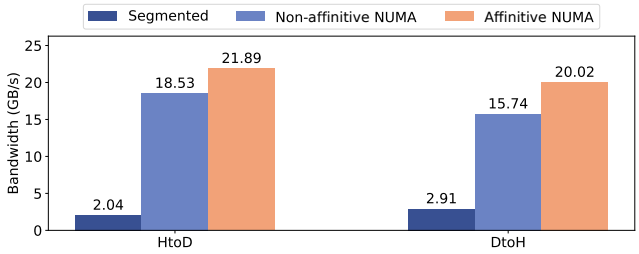


Figure 8. Host-to-devide and Device-to-host bandwidth under various design choices. Moving small memory segments is significantly slower than moving contiguous data. Data movement to a non-affinitive NUMA node can result in a noticeable slowdown.

directly launches iteration $i+1$ without detecting the end-of-sequence (EOS) tokens generated in iteration i. After launching iteration $i+1$, the EOS is detected. The finished request will be removed from the global batch by the batch formation of iteration $i+2$. Experiments show that this asynchronous top-level scheduling reduces the scheduling overhead to 2% of the pipeline iteration time, while a synchronized version would take more than 10%.

However, this mechanism generates one useless token after the EOS token. Fortunately, since the average decode length surpasses 100 for typical workloads (See Table 3), the overhead is negligible ($< 1\%$) compared to the benefit.

5.4 KV-cache offload

In every iteration, NanoFlow offloads the retired KV-cache from the last iteration and loads the KV-cache of new requests. As the latency of offloading is crucial for serving throughput, we propose efficiency optimizations, including KV-cache on-device rearrangement and NUMA-aware device-host copy to speed up offloading.

KV-cache on-device rearrangement With PageAttention [20], NanoFlow organizes KV-cache at the granularity of pages. Each page holds a segment of the KV-cache (e.g., 16 tokens), which is roughly 8KB per GPU with head parallelism. Each request holds hundreds of pages scattered in GPU memory. Thus, naive page-by-page offloading will incur significant overhead, as moving fragmented data is inefficient between host and device. We quantify this effect in Figure 8. On the other hand, memory movement inside GPU exhibits much lower overhead. Thus, NanoFlow implements a page aggregation kernel to collect offloading pages to a continuous space. Similarly, NanoFlow implements a page distribution kernel to scatter the loaded continuous pages to appropriate places on GPUs for the next iteration.

NUMA-aware device-host copy On NUMA architectures, the bandwidth of data movement between host and GPU is significantly affected by the NUMA affinity, as shown in Figure 8. Moving data between the NUMA-affinitive (directly attached) CPU and GPU can lead to 1.27× bandwidth gain compared to non-affinitive ones. NanoFlow ensures the KV-cache is copied to and from the affinitive NUMA node via thread binding.

5.5 Automatic parameter search

Given a specific model architecture, NanoFlow needs manual investigation of the operations' resource usage to determine the nano-batch splitting strategy and overlapping schemes. However, this is still far from a feasible execution plan, as the nano-batch sizes, SM assignment plans, and kernel implementations still form a huge search space. Therefore, we implement an *automatic parameter search* for NanoFlow, which uses greedy search to derive optimal configurations.

Specifically, autosearch uses topological sorting to find the critical path (the chain of dependent operations that takes the longest execution time) and assign more SMs to the operators on the critical path to shorten the critical path. To control interference, autosearch limits total execution unit usage according to the number of SMs in GPU. Therefore, assigning more SMs to the kernels on the critical path would slow down concurrent operations due to fewer available units, possibly causing critical path changes. The autosearch iteratively identifies new critical paths and optimizes the critical path until the best configuration is reached. After applying the above critical path optimization algorithm to all possible combinations of nano-batch sizes, we choose the scheduling with the shortest critical path. The autosearch estimates the kernel performance based on offline profiling, which is efficient and highly parallelizable. The typical running time is within 10 minutes, which is a one-time offline cost without runtime overhead.

5.6 Porting NanoFlow to other LLMs

Although splitting nano-batches and constructing the initial NanoFlow pipeline from scratch requires human efforts, virtually all of the pipeline can be reused across the most popular LLMs as they share the decoder-only architecture. Minor modifications are needed for porting NanoFlow to small models and MoE models, as we discuss later. Moreover, the kernel implementations of different LLMs are very similar, making kernel modification almost a one-time cost.

To illustrate this, we port NanoFlow to 5 representative models that covers prevailing model sizes (8B, 70B) and model architectures (dense models, MoE models). The process of porting NanoFlow is shown below. We also show the performance of NanoFlow on these models in § 6.7.

LLaMA-3-70B (Meta)[28] LLaMA-3-70B has an identical architecture as LLaMA-2-70B. Therefore, no effort is needed to port NanoFlow.

Qwen2-72B (Alibaba Cloud) [11] Qwen2 shares a similar architecture except for the FFN layer dimensions. Porting NanoFlow to Qwen2 requires profiling GEMM kernels for

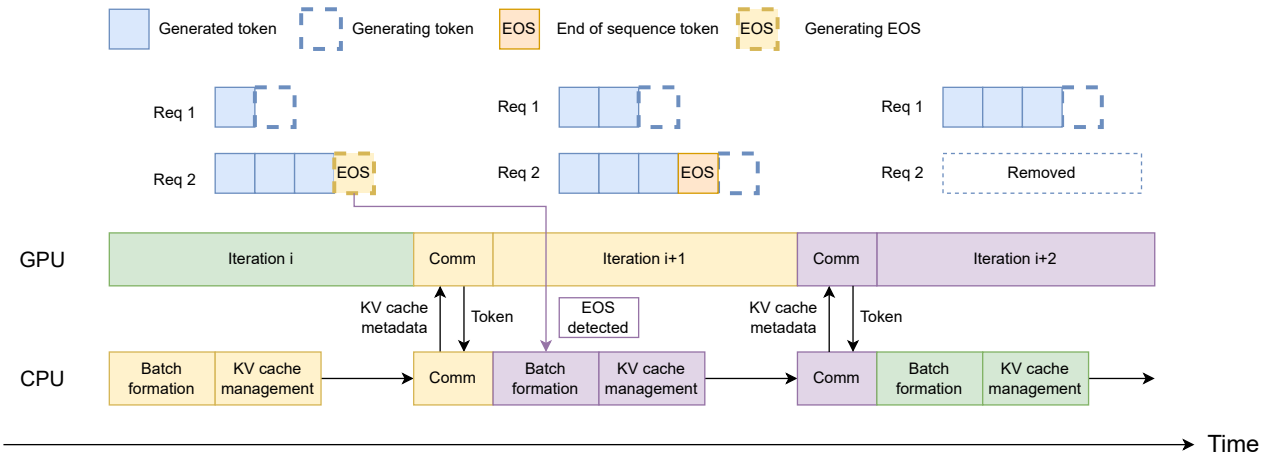


Figure 9. Asynchronous top-level scheduling. Request 2 generates an EOS token at iteration i. Due to the asynchronous execution, the EOS token is detected at iteration $i + 1$ while the request is removed from the batch at iteration $i + 2$. This approach hides the overhead of batch formation and KV-cache management, leading to superior performance.

new dimensions and an autosearch to derive new nano-batch partitions, which can be done in a few hours with ~ 50 lines of code changes.

Deepseek-67B (Deepseek) [13] For Deepseek, the FFN dimension and the number of layers are different. We rerun profiling and autosearch to consider these changes, which takes a few hours to complete.

Mixtral 8×7B (Mistral AI) [19] Mixtral is an MoE model, for which the gating operation is required for expert selection. We inserted gating into the original pipeline, reimplemented the network kernels, and changed the dimensions of FFN layers. Note that these changes do not change individual operation characteristics or operation dependency. Therefore, the previous pipeline design can be reused. Due to the kernel implementations, porting to Mixtral 8*7B takes a few days and requires 200 lines of code changes.

LLaMA-3-8B (Meta) [28] Adapting NanoFlow to small models such as LLaMA3-8B requires a new pipeline design, as no network is involved in small model serving. We use a straightforward pipeline design where FFN for the first nano-batchoverlaps with Attention for the second nano-batchand vice versa. We run profiling and autosearch to get the optimal schedule. Porting to LLaMA3-8B can be completed within one day with around 200 lines of code changes.

6 Evaluation

In the evaluation, we mainly focus on answering the following research questions regarding NanoFlow:

- How does NanoFlow enhance serving throughput compared to existing systems, and how does its throughput compare to the optimal value? (§6.2)
- How does the latency of NanoFlow change for different request rates, and how is the latency distributed? (§6.3, §6.4)
- How do the techniques proposed in NanoFlow contribute to the end-to-end performance? (§6.5)
- How does NanoFlow use compute, memory and network resources over time? (§6.6)
- How does NanoFlow perform when applied to other representative LLMs? (§6.7)

6.1 Experiment Setup

Hardware. We run our experiments on 8× A100 80GB SXM GPUs with NVLink.

Models. We mainly evaluate NanoFlow using the LLaMA-2-70B model, one of the most widely-used open-source LLMs. We demonstrate the flexibility of NanoFlow by evaluating NanoFlow on 5 other representative LLMs.

Baselines. We consider three widely-used serving frameworks as baselines.

The vLLM [20] is a state-of-the-art serving system delivering high throughput. For evaluation, we change the `max-num-seqs` parameter in the vLLM engine to tune the maximum number of sequences allowed per iteration and use the one that results in the highest throughput.

DeepSpeed-FastGen [17] is a serving framework developed by Microsoft. It dynamically composes prefill with decode requests to ensure that the engine is operating in a high throughput regime. We vary the `max-ragged-batch-size` to tune the batch size for dynamic batching and select the one with the highest throughput.

TensorRT-LLM [32] is a high-performance LLM inference engine built upon NVIDIA's TensorRT SDK. We set `max-num-tokens` by calculating the maximum capacity for

Dataset	Avg. Input (Std)	Avg. Output (Std)
Splitwise [37]	1155 (1109)	211 (163)
LMSYS-Chat-1M [56]	102 (169)	222 (210)
ShareGPT [1]	246 (547)	322 (244)

Table 3. The average and standard deviation of input and output lengths in the sampled datasets.

the KV-cache in the GPU memory. We also enable paged KV-cache and dynamic batching optimizations when compiling the models. [7]

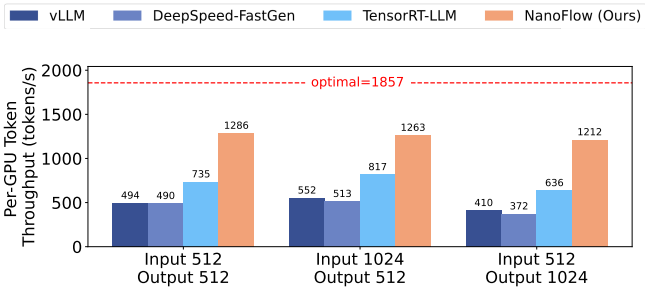
Datasets. Splitwise [37] is a conversation trace collected from real production environment, with a total of around 20000 requests. LMSYS-Chat-1M [56] is a large-scale dataset with 1 million real-world conversations from 25 different LLMs. ShareGPT [1] is a dataset with conversations collected from the ShareGPT API. We use the the full trace from Splitwise and randomly sampled 50,000 requests from ShareGPT and LMSYS-Chat-1M for our evaluation. Table 3 shows the average input length and output length in tokens for the sampled datasets we use.

6.2 Offline throughput

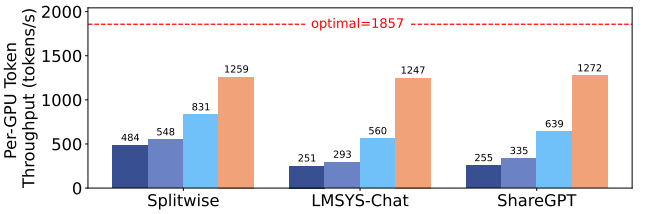
We first evaluate offline throughput to simulate the purely throughput-oriented use cases [39]. Since these use cases do not factor in the latency of prefill or decode, we assume the requests arrive all at once at the beginning of the experiment. We sample actual input and output length from the datasets and add additional experiments with constant length. We then deploy NanoFlow with dense batch size 2048 to measure the *total throughput* defined as the total number of input and output tokens divided by the execution time. We compare the performance of NanoFlow with optimal throughput derived by Equation 9, using the peak compute usage of CUTLASS, 260 TFLOPs, at the batch size of 2048. Since the optimal throughput is independent of user query statistics, for all the experiments, the optimal throughput is 1857 tokens/s/GPU.

Figure 10 shows the offline throughput of NanoFlow compared to baselines. NanoFlow has the highest throughput in all settings. NanoFlow is able to achieve 68.5% of the theoretical optimal throughput in the best case. For constant input and output length, NanoFlow has 2.62× higher offline throughput than vLLM, 2.78× higher than DeepSpeed-FastGen and 1.73× higher than TensorRT-LLM on average. For input and output length from datasets, NanoFlow has 4.18× higher offline throughput than vLLM, 3.45× higher than DeepSpeed-FastGen and 1.91× higher than TensorRT-LLM on average.

[7] We use vLLM v0.5.3.post1 (commit ID 38c4b7e), DeepSpeed-FastGen v0.2.3 (commit ID 429bc5c), and TensorRT-LLM v0.8.0 (commit ID 5955b8a).



(a) LLaMA-2-70B, 8 GPU, TP=8, Constant Input & Output Length



(b) LLaMA-2-70B, 8 GPU, TP=8, Input & Output Length from Dataset

Figure 10. Comparison of offline throughput. NanoFlow outperforms all baselines for all the workload settings in the throughput measured by the total number of tokens processed (both prefill and decode) per second per GPU. TP stands for the number of GPUs used in tensor parallelism.

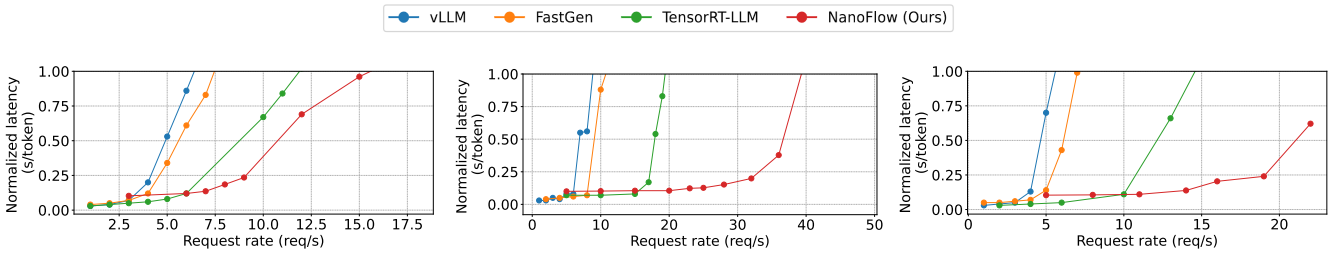
6.3 Online latency

We evaluate the online latency of NanoFlow, for which we sample the request arrival interval via an exponential distribution, following previous work [20]. We generate 5 minutes of request traces from the datasets in Table 3 for various request rate. We then measure the *normalized latency* by first calculating end-to-end request latency divided by output length in tokens, then taking the average for all requests. For each framework, we gradually increase the request rate until the system is saturated (i.e., the total throughput is stable) and measure the normalized latency.

Figure 11 demonstrates the online latency of NanoFlow compared to baselines. NanoFlow is able to sustain a higher request rate with low latency compared to baselines among all the datasets. On the LMSys-Chat-1M dataset, NanoFlow can achieve 1.64× higher request rate compared to TensorRT-LLM within 200ms latency constraint.

6.4 Online latency distribution

Besides the average latency, we also evaluate the tail latency of NanoFlow at 90% of maximum throughput for all of the datasets. We show the per-token latency CDF in Figure 12. Since the discrete batching strategy chooses constant dense batch size at high workload, NanoFlow's 99 percentile latency is only 1.07× of the average latency, caused by the variation in sequence lengths of the on-the-fly batches.



(a) Splitwise **(b)** LMSYS-Chat-1M **(c)** ShareGPT

Figure 11. Comparison of the online latency. The x-axis shows the number of incoming requests per second and the y-axis shows the normalized latency . NanoFlow retains lower latency at a high request rate, while achieving similar latency with existing systems at a low request rate.

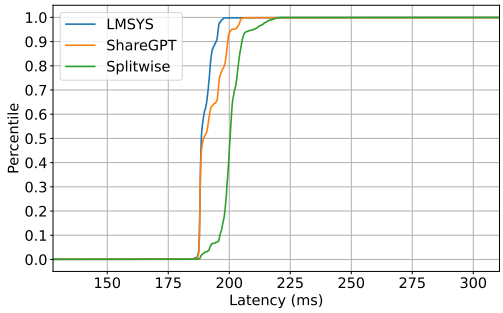


Figure 12. The per-token latency distribution for NanoFlow at 90% of maximum throughput. Due to the constant global batch, NanoFlow features low latency variations.

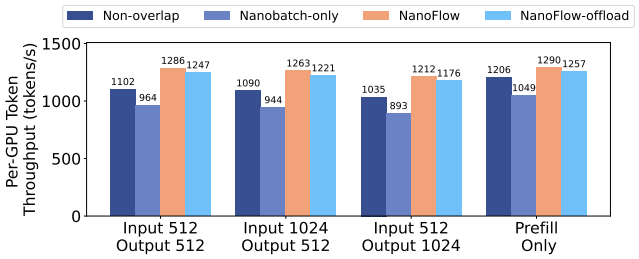


Figure 13. Ablation study for NanoFlow. Compared with sequentially running all operations (non-overlap), splitting into nano-batches (nano-batch-only) does incur an overhead. However, by overlapping network and memory-bound operations with compute-bound operations, NanoFlow significantly surpasses the non-overlapping baseline. Offloading only incurs a small overhead while boosting the system performance for multi-round conversations.

6.5 Ablation Study

To showcase the overhead and benefit of proposed techniques, including nano-batching, overlapping operations, and offloading, we compare NanoFlow with baselines that share the same set of scheduling systems and underlying kernels.

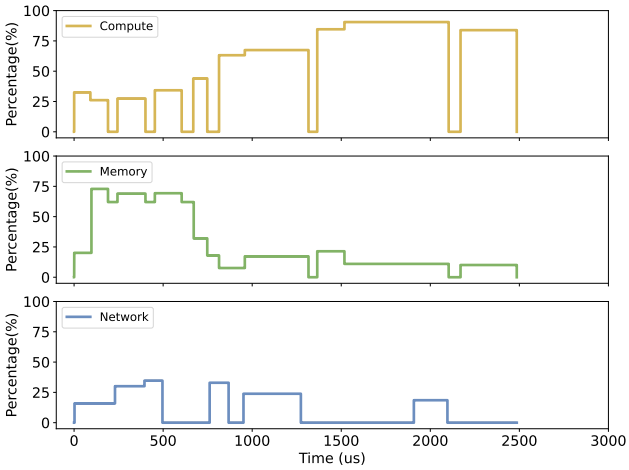
We first build the non-overlapping baseline, which does not split the input into nano-batches and simply runs every operation sequentially. We further build a nano-batch-only baseline, where the requests are split into the same nano-batches with NanoFlow while still executing operations sequentially to evaluate the nano-batch overhead. As shown in Figure 13, splitting requests to nano-batches reduces the performance by 13.2%.

To estimate the benefit of overlapping network- and memory-bound kernels, we compare NanoFlow and baselines under various prefill-decode ratios, as shown in Figure 13. We use prefill-only workloads (Input 512, Output 0) to indicate the benefit of overlapping network-bound kernels, as no memory-bound attention is involved. Additionally, we use decode heavy workloads (Input 512, Output 1024) to evaluate the benefit of overlapping both network- and memory-bound kernels. NanoFlow achieves 1.07× speedup from overlapping network-bound kernels and achieves 1.17× speedup when overlapping both compared with non-overlapping baseline.

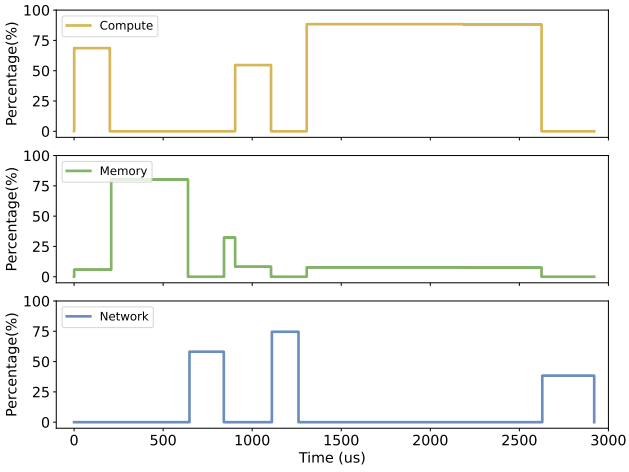
Moreover, we quantify the performance degradation by offloading in Figure 13. Enabling offloading would slow down the pipeline by 3.0%. However, for n-round conversations, offloading can avoid repetitive prefill process of history tokens, thus reduce the total computation from $O(n^2)$ to $O(n)$, significantly increasing efficiency.

6.6 Resource usage

To reason about the performance gain, we compare the resource usage of NanoFlow and the non-overlapping baseline over time in Figure 14. While non-overlapping baseline mostly uses one kind of resource at a particular timestamp, NanoFlow can concurrently utilize multiple resources due to intra-device parallelism and execution unit scheduling and achieves higher compute utilization. However, due to inevitable kernel interference, NanoFlow still has a gap between optimal compute usage.



(a) NanoFlow resource usage over time



(b) Non-overlap baseline resource usage over time

Figure 14. Comparison of resources usage during inference of a single layer. NanoFlow achieve high compute resource usage across the whole pipeline while baseline only uses one of the resources at every timestamp resulting in underutilization of compute when performing memory and network bound operations.

6.7 Porting NanoFlow to other LLMs

We ported NanoFlow to 5 representative models to showcase its flexibility. We evaluate the offline throughput of NanoFlow on these LLMs with constant length of input 1024 and output 512, compared with the vLLM baseline and the optimal throughput. All evaluations are done on the same 8×A100 80GB SXM GPUs, except for the LLaMA-3-8B on a single A100 80GB SXM. The results are summarized in Figure 15. We found that NanoFlow improves throughput to between 59% and 72% of optimal throughput for these models, which greatly surpasses vLLM.

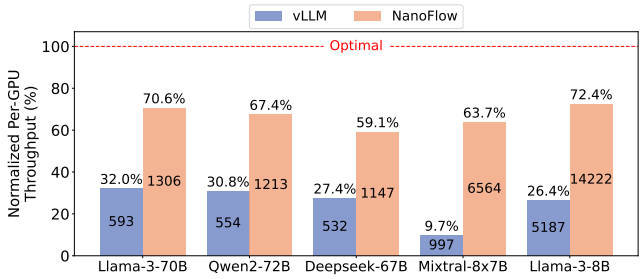


Figure 15. Performance of NanoFlow in terms of tokens per second per GPU when ported to different models. Compared with vLLM, NanoFlow significantly increases throughput. NanoFlow achieves up to 72.4% of optimal throughput.

7 Related Work

LLM serving optimizations. Previous works investigated optimizations for improving serving throughput at different granularities. Orca [54] proposes request-level continuous batching, which refills the on-the-fly batch to maximize batch size at the granularity of iteration. DistServe [58] and Splitwise [37] explore phase-level scheduling, which disaggregates the prefill and decode phases into different clusters. DeepSpeed-FastGen [17] and Sarathi-Serve [2] propose chunked prefill policy, which splits a prefill request into multiple smaller chunks and batches decode and prefill requests together to improve overall utilization. However, none of these works are scheduled at the granularity of operations. NanoFlow exploits intra-device parallelism, which features fine-grained resource management. Other works tackle the inference inefficiency by optimizing specific operations, including PageAttention [20], quantization [25, 55], etc. NanoFlow can easily adopt new operation implementations using execution unit scheduling. Moreover, to reduce the decode latency, various speculative decoding works focus on predicting and verifying future tokens [21, 29]. However, speculative decoding cannot boost the system throughput due to limited acceptance rate and redundant computations.

Operation-level parallelism. Some works focus on improving the efficiency of DNN workload with operation-level optimization. For example, Rammer [26] explores intra-operation parallelism by remapping the operations into different functional units. Unity [51] investigates the combination of parallelisms and equivalent transformation of operations. However, they have not explored increasing parallelism using nano-batch and are unaware of the different resource demands of operations. ASPEN [35] and Welder [40] break the boundary between operations by constructing and compiling a tile-level data graph. However, since tiles follows sequential dependency, the parallelism is still underutilized, while NanoFlow uses nano-batch to create more overlapping opportunity. Moreover, they need kernel compilation

from scratch, which is incompatible with the existing high-performance kernel library.

8 Conclusion

We analyzed the characteristics of different operations in LLM serving and revealed the compute-bound nature of modern LLM serving workloads. We identified that the resource is under-utilization inside a single GPU due to the sequential execution of compute-bound, memory-bound, and network-bound operations, leading to sub-optimal throughput. To address this, we proposed NanoFlow, a novel end-to-end LLM serving system that overlaps the usage of compute, memory, and network resources with a carefully designed pipeline of different operations to achieve intra-device parallelism. We presented an automatic parameter search system to generalize NanoFlow to diverse models. Our experiments show that NanoFlow achieves 1.91× throughput improvements over the state-of-the-art systems.

References

[1] 2023. ShareGPT. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. *arXiv preprint arXiv:2403.02310* (2024).

[3] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369* (2023).

[4] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 [cs.LG]

[5] Mistral AI. 2024. Mistral NeMo. https://mistral.ai/news/mistral-nemo/.

[6] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 4895–4901. https://doi.org/10.18653/v1/2023.emnlp-main.298

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. arXiv:1607.06450 [stat.ML]

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2023. Punica: Multi-Tenant LoRA Serving. arXiv:2310.18547 [cs.DC]

[10] Peng Cheng, Changho Hwang, Abhinav Jangda, Suriya Kalivardhan, Binyang Li, Shuguang Liu, Saeed Maleki, Madan Musuvathi, Olli Saarikivi, Aashaka Shah, Wei Tsui, and Ziyue Yang. [n. d.]. *MSCCL++: A GPU-driven communication stack for scalable AI applications.* https://github.com/microsoft/mscclpp

[11] Alibaba Cloud. 2024. Alibaba Cloud's Qwen2 with Enhanced Capabilities Tops LLM Leaderboard. https://www.alibabacloud.com/blog/alibaba-cloudâĂŹs-qwen2-with-enhanced-capabilities-tops-llm-leaderboard_601268/.

[12] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv:2307.08691 [cs.LG]

[13] DeepSeek-AI, :, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, A. X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. arXiv:2401.02954 [cs.CL] https://arxiv.org/abs/2401.02954

[14] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2017. Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning. arXiv:1702.03118 [cs.LG]

[15] Erin Griffith. 2023. *The Desperate Hunt for the A.I. Boom's Most Indispensable Prize.* Technical Report. The New York Times.

[16] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 2022. FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 120–134. https://doi.org/10.1145/3503221.3508418

[17] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. *arXiv preprint arXiv:2401.08671* (2024).

[18] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. arXiv:1811.06965 [cs.CV]

[19] 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. arXiv:2310.06825 [cs.CL]

[20] 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*. 611–626.

[21] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2022. Fast Inference from Transformers via Speculative Decoding. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:254096365

[22] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2020. Automatic Horizontal Fusion for GPU Kernels. arXiv:2007.01277 [cs.DC] https://arxiv.org/abs/2007.01277

[23] Hao Li, Di Yu, Anand Kumar, and Yi-Cheng Tu. 2014. Performance modeling in CUDA streams—A means for high-throughput data processing. In *2014 IEEE international conference on big data (big data)*. IEEE, 301–310.

[24] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[25] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving. arXiv:2405.04532 [cs.CL] https://arxiv.org/abs/2405.04532

[26] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[27] Yusuf Mehdi. 2023. Reinventing search with a new AI-powered Microsoft Bing and EDGE, your copilot for the web. https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/

[28] Meta. 2024. Build the future of AI with Meta Llama 3. https://llama.meta.com/llama3/.

[29] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2023. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781* (2023).

[30] NVIDIA. 2024. NVIDIA DGX Platform. https://www.nvidia.com/en-us/data-center/dgx-platform/.

[31] NVIDIA. 2024. NVLink and NVLink Switch. https://www.nvidia.com/en-us/data-center/nvlink/.

[32] NVIDIA. 2024. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[33] OpenAI. 2023. Introducing ChatGPT. https://openai.com/blog/chatgpt

[34] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[35] Jongseok Park, Kyungmin Bin, Gibum Park, Sangtae Ha, and Kyunghan Lee. 2023. ASPEN: Breaking Operator Barriers for Efficient Parallelization of Deep Neural Networks. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 68625–68638. https://proceedings.neurips.cc/paper_files/paper/2023/file/d899a31938c7838965b589d9b14a5ca6-Paper-Conference.pdf

[36] Dylan Patel and Afzal Ahmad. 2023. The Inference Cost Of Search Disruption – Large Language Model Cost Analysis. https://www.semianalysis.com/p/the-inference-cost-of-search-disruption

[37] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative LLM inference using phase splitting. arXiv:2311.18677 [cs.AR]

[38] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems* 5 (2023).

[39] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[40] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling Deep Learning Memory Access via Tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 701–718. https://www.usenix.org/conference/osdi23/presentation/shi

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

[42] Shubham Singh. 2024. ChatGPT Statistics (AUG 2024) - Users Growth Data — demandsage.com. https://www.demandsage.com/chatgpt-statistics/. [Accessed 14-08-2024].

[43] Jared Spataro. 2023. Introducing Microsoft 365 copilot – your copilot for work. https://blogs.microsoft.com/blog/2023/03/16/introducing-microsoft-365-copilot-your-copilot-for-work/

[44] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.

[45] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. Quest: Query-Aware Sparsity for Efficient Long-Context LLM Inference. arXiv:2406.10774

[46] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. *CUTLASS*. https://github.com/NVIDIA/cutlass

[47] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2024. *CUTLASS Profiler*. https://github.com/NVIDIA/cutlass/tree/main/tools/profiler

[48] CHENG TING-FANG. 2023. *TSMC sees AI chip output constraints lasting 1.5 years*. Technical Report. Nikkei Asia.

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[50] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[51] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 267–284. https://www.usenix.org/conference/osdi22/presentation/unger

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[53] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. 2024. Accelerating Self-Attentions for LLM Serving with FlashInfer. https://flashinfer.ai/2024/02/02/introduce-flashinfer.html

[54] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[55] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2023. Atom: Low-bit quantization for efficient and accurate llm serving. *arXiv preprint arXiv:2310.19102* (2023).

[56] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2023. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. arXiv:2309.11998 [cs.CL]

[57] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[58] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. *arXiv preprint arXiv:2401.09670* (2024).

A Detailed Modeling of Compute Latency

In §3.2, we derived the compute latency's dominant portion, which is due to the dense operations (Equation 6). Here, we also derive the compute latency due to prefill attention, decode attention and networking to demonstrate that their contribution is relatively minor as we validated through experimental analysis in §3.5.

Prefill attention. For prefill attention, first, every query vector of the tokens in a request is multiplied with all the key vectors in one layer. Since each key and query vector is of dimension D_{model} and a request contains on average p tokens, the total compute for one layer is $2p^2D_{model}$. Note that in grouped query attention, multiple attention heads in the query vector is multiplied with the same key head, where the key vector is of size D_{model}/R_{GQA}. However, the number of operations due to every query-key vector multiplication in one layer's prefill attention computation still remains $2D_{model}$. The result is then multiplied with the value vector (also of size $p \cdot D_{model}$), which results in another $2p^2D_{model}$ computations. Therefore, given $\frac{B_{req}}{d+1}$ prefill requests, the total number of compute operations due to prefill attention is $4 \cdot \frac{B_{req}}{d+1} \cdot p \cdot p \cdot D_{model} \cdot L$ for each iteration. Note that $\frac{B_{req}}{d+1} \cdot p < B_{Dense}$, which is $\frac{B_{req}(p+d)}{d+1}$. We further compare the term $p \cdot D_{model} \cdot L$ with P_{model}. We multiply D_{type} to both of the terms to get $p \cdot D_{model} \cdot L \cdot D_{type}$ and $P_{model} \cdot D_{type}$. The first term is the activation size for prefill tokens, while the second term is the memory size holding the model weight. Since activations occupies negligible memory [20], the term $p \cdot D_{model} \cdot L$ is significantly less than P_{model}. Therefore, the amount of computation for prefill attention ($4 \cdot \frac{B_{req}}{d+1} \cdot p \cdot p \cdot D_{model} \cdot L$) is usually significantly less than that for dense operations ($2B_{Dense}P_{Model}$).

Decode attention. Similarly, during decode attention computations, one query vector multiplies with all of the key vectors of previous tokens and further multiply with value tokens. Under Group Query Attention, each K and V heads are multiplied with R_{GQA} Q heads. Therefore, $2R_{GQA}$ computation operations are needed for every KV-cache element[8]. We denote the number of KV-cache elements as E_{kv}. Since each element needs to go through one GEMV during attention, the number of computations for attention is $2E_{kv}R_{GQA}$. Note that since we assume the system operates at the largest batch size and the activation size is negligible, all available memory that is not used for model parameters is used for the KV-cache. Therefore, E_{kv} can be calculated by $\frac{MemSize}{Dtype} - P_{Model}$, which is on par with P_{Model} for most of the use cases [20]. Since B_{Dense} is typically on the order of hundreds, the computation for decoding attention ($2E_{kv}R_{GQA}$) is also negligible compared to the computation for GEMM operations ($2B_{Dense}P_{Model}$).

Networking. AllReduce operations compute the sum of N_{GPU} pieces of partial results to produce the output. Therefore, every element in the final result is derived using $N_{GPU} - 1$ additions. Since the output of AllReduce serves as the input for dense operations, the number of total elements that result from AllReduce is $B_{dense}D_{model}$. Therefore, the total number of compute performed as a result of network communication is $(N_{GPU}-1) \cdot B_{dense}D_{model}L$. Compared to dense operations ($2B_{Dense}P_{Model}$), since $P_{Model} \sim D^2_{model}L >> (N_{GPU} - 1) \cdot D_{model}L$, the compute performed due to networking is negligible.

[8] We use element to refer to the values in the KV-cache, for example, each float value in the float typed KV-cache.