# Shift Parallelism: Low-Latency, High-Throughput LLM Inference for Dynamic Workloads

Mert Hidayetoglu, Aurick Qiao, Michael Wyatt, Jeff Rasley, Yuxiong He, and Samyam Rajbhandari
Snowflake AI Research

## Abstract

Efficient parallelism is necessary for achieving low-latency, high-throughput inference with large language models (LLMs). Tensor parallelism (TP) is the state-of-the-art method for reducing LLM response latency, however GPU communications reduces combined token throughput. On the other hand, data parallelism (DP) obtains a higher throughput yet is slow in response latency. Best of both worlds does not exist, and it is not possible to combine TP and DP because of the KV cache variance across the parallelisms.

We notice Sequence Parallelism (SP—Ulysses in training) has similar properties as DP but with KV cache invariance. We adapt SP to inference, and combine it with TP to get the best of both worlds. Our solution: Shift Parallelism.

Shift Parallelism dynamically switches across TP and SP, and minimizes latency in low traffic without losing throughput in high traffic. The efficient GPU communications of Shift Parallelism yields up to i) 1.51× faster response in interactive workloads and ii) 50% higher throughput in batch workloads, compared to a TP-only solution.

We evaluate Shift Parallelism with real-world production traces with dynamic traffic patterns as well as synthetic benchmarking patterns across models, context sizes, and arrival rates. All results affirm the same: Shift Parallelism has a better the latency vs. throughput tradeoff than TP or DP, and hence obtains low latency without degrading throughput in dynamic workloads.

## 1 Introduction

LLM inference has become the dominant workload in AI as its applications span agentic systems, chatbox (interactive) applications, model post-training (e.g. reinforcement learning), and image/video generation. The efficiency of inference systems is critical to both the performance and cost of AI applications. As of today, GPU parallelization is the prominent way of enabling large-scale AI production, and advanced multi-GPU parallelization techniques make nontrivial trade-offs across key performance metrics.

The parallelism techniques for inference are largely inherited from training, yet inference is different from training in terms of workload characteristics. Training workloads are typically homogeneous, stable, and do not care about latency but only care about throughput. Inference, on the contrary, is bursty and dynamic, and often has unpredictable traffic patterns. Furthermore, different workloads have different performance requirements. As a result, leveraging parallelism techniques designed for training in inference results in complex performance and cost trade-offs.

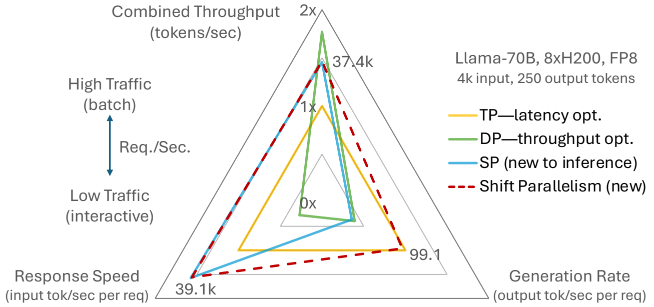


**Figure 1.** Comparison of response speed (#input tok./TTFT) and generation rate (1/TPOT), and throughput (tokens/sec). Shift Parallelism obtains a higher throughput than TP in high traffic, and lower latency than TP and DP in low traffic.

### Inference Workload Characteristics

When people talk about inference systems, they often refer to interactive workloads, or batch workloads.

***Interactive workloads*** process requests with low concurrency to minimize the completion time of each request. The completion time latency is important when there is a chain of interactions between the user and the LLM, such as in REST applications. The completion latency depends on the time to first token (TTFT)—and time per output token (TPOT), which are both critical in real-time applications.

***Batch workloads*** involve a number of requests to be processed concurrently, and the latency of an individual request is not critical. For example, workloads such as batched summarization or translation of hundreds or thousands of documents can cause high-traffic bursts that require high combined throughput of input and output tokens to minimize the cost per token.

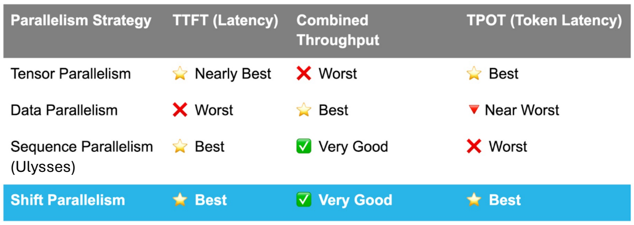
In enterprise systems, the request traffic pattern is often mixed and dynamically changes over time in unpredictable ways. In such dynamic settings, it is a challenge to optimize for different traffic patterns simultaneously, since existing parallelization techniques impose significant trade-offs.

### Latency vs. Throughput vs. Cost Tradeoff

Existing parallelisms exhibit prohibitive latency vs. throughput (cost) trade-offs, as explained below.

***Tensor parallelism (TP)*** partitions the model weights and computation in each layer. It has to synchronize the

**Table 1.** Performance tradeoffs of inference parallelisms.



| Parallelism Strategy | TTFT (Latency) | Combined Throughput | TPOT (Token Latency) |
|---|---|---|---|
| Tensor Parallelism | ☆ Nearly Best | ✗ Worst | ☆ Best |
| Data Parallelism | ✗ Worst | ☆ Best | ▼ Near Worst |
| Sequence Parallelism (Ulysses) | ☆ Best | ✓ Very Good | ✗ Worst |
| **Shift Parallelism** | **☆ Best** | **✓ Very Good** | **☆ Best** |

embeddings across layers with costly all-reduce communications. By splitting model weights and computation across GPUs, it optimizes for latency (i.e., TTFT and TPOT), yet the communication overhead increases the cost (i.e., reduces throughput).

***Data parallelism (DP)*** parallelizes across request boundaries in embarrassingly parallel, providing high throughput. Yet, DP cannot speed up work within a single request, and therefore unsuitable for highly interactive workloads.

The first two rows of Table 1 shows the performance tradeoffs of TP and DP. We do have the choice to deploy TP and DP in separate nodes and route latency- and throughput-oriented requests, respectively. However, duplicating the node count (one for TP and one for DP) doubles the deployment cost and add complexity.

***Why can't we combine both?*** A performant and low-cost inference system should be able to switch between latency-oriented and throughput-oriented parallelisms swiftly in a single deployment based on traffic demands. But this is not viable with TP and DP because they have different attention layouts. Specifically, their KV cache memory layouts are incompatible, and switching requires complex and costly data movement.

However, in this work, we notice that ***Sequence Parallelism (SP)*** [6]—another form of parallelism developed and used in training—can offer a potential solution to resolving the challenge. SP splits the input sequence across GPUs to parallelize work within a single request to reduce TTFT. Unlike TP, it avoids costly all-reduce communication, while still achieving high GPU utilization. And while SP cannot parallelize decoding steps, resulting in the worst TPOT compared to TP and DP (Table 1), it has the same KV cache layout as TP, allowing for dynamic switching to TP when TPOT is critical. We call this dynamic approach Shift Parallelism, which is the focus of this work.

***Shift Parallelism*** dynamically chooses the parallelization strategy between SP and TP based on the real-world traffic pattern, identified by the number of batched tokens in each iteration. By a given threshold, Shift Parallelism uses:

- TP for small batches—minimizing TPOT.
- SP for large batches—minimizing TTFT and achieving near-optimal throughput.

This is possible because the KV cache memory layout remains invariant between TP and SP, allowing Shift Parallelism to switch modes seamlessly, based on batch size and traffic patterns. More specifically, the KV cache layout does not change when switching across SP and TP.

Figure 1 benchmarks the latency and throughput tradeoffs of related parallelisms. Shift parallelism provides 1.5× higher throughput than TP in high traffic and 1.5× faster response in low traffic, 2× faster generation than DP in low traffic while losing only 17% throughput in high traffic.

In this paper, we

1. We characterize inference workloads and identify latency vs. throughput tradeoffs with existing inference parallelization (TP, DP).
2. Adapt SP from training to inference and generalize it for a diverse set of inference models supporting GQA [2], load-balancing at small batch sizes, combination of TP, and KV cache replication when paralleism degree is higher than number of KV heads.
3. Propose Shift Parallelism for dynamically switching across SP and TP for mitigating latency vs througput tradeoff for dynamic workloads.
4. Test Shift Parallelism with real-world production workloads and evaluate the performance characteristics via extensive benchmarking demonstrating up to 1.5× faster response time with 50% throughput saving.
5. Open source our implementation along with other SoTA techniques that are used in practice.

The rest of the paper is organized as follows. Section 2 provides mode details about production traffic patterns high-performance LLM inference. Section 3 presents SP and Shift Parallelism. Section 4 evaluates the performance of proposed techniques on real-life patterns. Section 5 about related work, and Section 6 concludes the paper.

## 2 Background

### 2.1 Production Traffic Patterns

Today, LLMs are a foundational component of AI applications. A single deployment, such as Llama-3.3-70B, can serve diverse use cases including sentiment analysis [11, 20], retrieval-augmented generation (RAG) [4, 9], coding agents [7, 19], and more. These heterogeneous use cases produce dynamic traffic patterns that must be efficiently managed by the underlying infrastructure. For example, a coding agent typically issues a small number of repeated

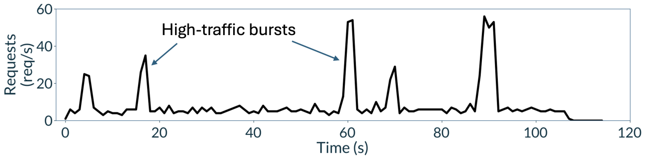


**Figure 2.** Bursty workload.

requests in a closed loop to iteratively refine its generated code, whereas a sentiment analysis workload may submit a large batch of requests in parallel to process text stored in a database.

In production, we typically observe two main *classes* of requests. First, *interactive* or *latency-sensitive* requests (e.g., agentic or chatbot applications) generally arrive one or a few at a time, with response latencies—TTFT and TPOT (see Sec. 2.2)—directly shaping the user experience. Second, *batch* or *throughput-sensitive* requests usually arrive in large volumes (thousands to millions at once), where aggregate throughput (tokens/s) determines job completion time. When these two classes of workloads are mixed, the result is a highly *bursty* traffic pattern, with different requests subject to different quality-of-service metrics (latency versus throughput). Fig. 2 illustrates an example traffic pattern that reflects our production environment.

### 2.2 Performance Metrics

Since LLM use cases are diverse, metrics that measure their inference performance is also multi-faceted. In our paper, we focus on three main metrics that cover the most important aspects of interactive and batch workloads:

- **Time-to-first-token (TTFT, ms):** The time after a client submits a prompt until the first characters of response text (tokens) are received.
- **Time-per-output-token (TPOT, ms):** After the first response token is received, the time between each subsequent token until the response is completed.
- **Combined throughput (tokens/s):** The total number of tokens (both prompt and response) processed by the inference system per unit of time.

Typically, TTFT and TPOT shape the quality of service for interactive applications, while combined throughput shapes the quality of service for batch use cases and also impacts the cost of running the service for the model provider.

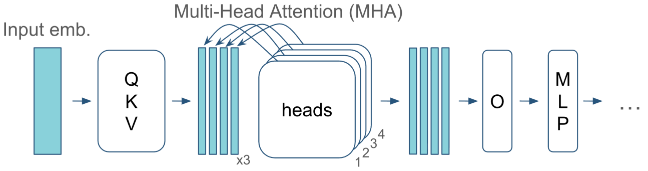


**Figure 4.** Vanilla transformer architecture and the attention mechanism.

### 2.3 Transformer Architecture

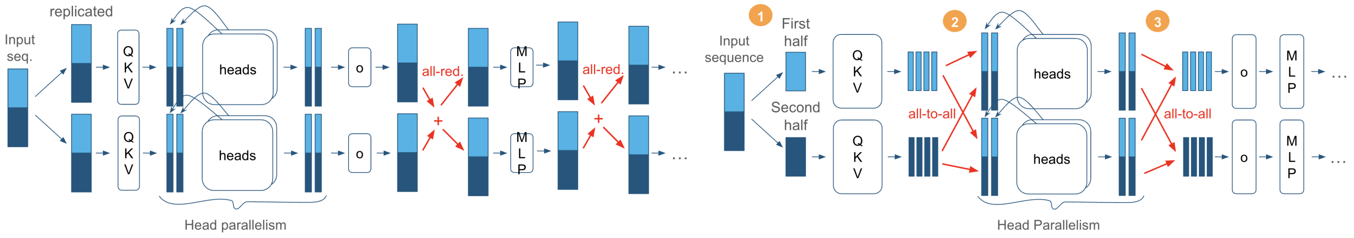
A vanilla LLM involves a series of transformer layers, and each transformer layer consists i) an attention mechanism and a ii) multi-layer perceptron (MLP). The weights in the transformer layer correspond to the QKV (which is a concatenation of q−querry, k−key, and v−value) and O matrices in the attention, as shown in Figure 4.

First, the QKV matrix projects the input embeddings into the QKV space, where attention is applied. The Multi-Head Attention (MHA) consists multiple heads, each "attends" a different column of the input sequence. After the attention, the O matrix projects the attention output back to the embedding space to be further processed by the MLP layer.

Each LLM request involves a sequence of input and output tokens. In prefill, the input tokens are batched and propagated altogether over all of the transformer layers and initializes the KV cache for the attention layers. At the end of the prefill, the first output token is decoded according to the resulting probability distribution over all tokens in the vocabulary. For decoding the full output sequence, each new token is appended to the context sequence, and the attention patterns of subsequent context are reused from the KV cache.

### 2.4 Existing Parallelism Approaches

DP runs multiple replicas across requests and do not accelerate processing of a single request. For accelerating, TP partitions the weight matrices either row-wise or column-wise, as depicted in Figure 3a. Yet, row parallelization requires all-reduce with $O(n)$ communication cost, where $n$ is the sequence length. For a fixed sequence length, the

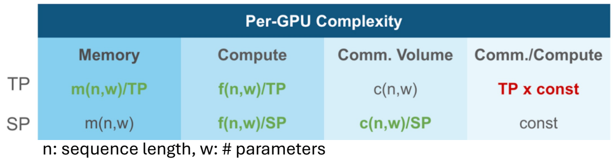


**(a)** Tensor Parallelism (TP=2)

**(b)** Ulysses Sequence Parallelism (SP=2)

**Figure 3.** Parallelization of the vanilla transformer on two GPUs with TP and SP. The attention has four heads which are parallelized across across heads. In (b), SP (1) partitions the input sequence, (2) switches to head parallelism using an all-to-all communication, applies head parallelization to attention, and (3) returns back to SP.

**Table 2.** Computational Complexity of TP and SP.



| | Per-GPU Complexity | | | |
|---|---|---|---|---|
| | Memory | Compute | Comm. Volume | Comm./Compute |
| TP | m(n,w)/TP | f(n,w)/TP | c(n,w) | TP x const |
| SP | m(n,w) | f(n,w)/SP | c(n,w)/SP | const |

n: sequence length, w: # parameters

communication-to-compute ration increases with the TP degree as shown in the last column of Table 2.

*Head Parallelism* is commonly used with SP and TP, where the attention heads are distributed across the GPUs equally. This is done with no additional cost by column-wise partitioning of the QKV matrix, as Figure 3a shows. Head parallelism cannot be scaled beyond the number of heads.

*Ulysses Sequence Parallelism* partitions the embedding sequence for parallelizing inference. Yet, each attention head requires the full sequence, resulting in all-to-all communications before and after the attention layer, as shown in Figure 3b. Nevertheless, the communication cost does not increase with SP as shown in the last column of Table 2.

Ulysses has been applied to training, and in this work, we extend Ulysses for inference handing inference specific nuances to allow for a generalized implementation [1].

## 3 System Design and Implementation

### 3.1 Overview

We design Shift Parallelism to enable switching between SP and TP, addressing the latency–throughput tradeoff in inference. The key insight is that both configurations must share the same KV cache layout—what we call KV cache invariance. This invariance allows us to switch seamlessly between SP and TP.

**Figure 5** illustrates KV cache invariance between TP=2 and SP=2. In the center, four attention heads are evenly distributed across two GPUs (two heads per GPU). This distribution is identical under both TP=2 and SP=2, allowing the two configurations to share a single attention mechanism and KV cache.

**3.1.1 SP for Inference.** Applying SP to inference is more nuanced than in training because of variable traffic patterns (e.g., load imbalance) and the lack of Grouped Query Attention (GQA) support in earlier designs, and parallelism that can exceed the number of KV attention heads. To address this, we develop a fully generic SP for inference that: i) supports GQA, ii) replicates KV cache as needed, iii) handles load balancing under low-traffic scenarios.

Furthermore, real-world inference is not simply a choice between SP *or* TP. For optimal performance, systems require arbitrary combinations of SP and TP. Our design supports this flexibility, enabling mixed SP–TP configurations.

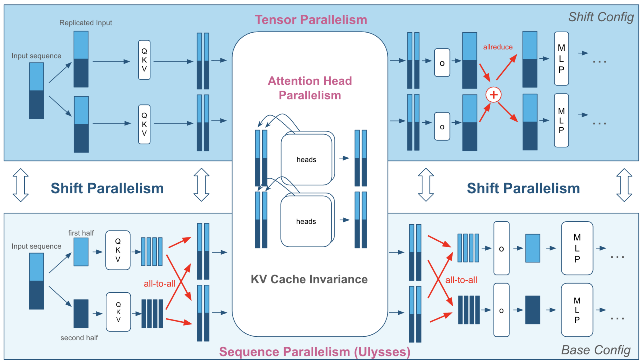


**Figure 5.** Although SP and TP are essentially different parallelisms, Shift Parallelism exploits the KV cache invariance between SP and TP for swiftly switch across them.

**3.1.2 Shift Parallelism.** Building on this flexible SP design, we implement Shift Parallelism using two configurations:

1. Base configuration: Uses SP or a mixed (SP, TP) setup, as long as SP× TP = P, where *P* is the total number of GPUs.
2. Shift configuration: Always (SP=1, TP=P), spanning the full node.

While conceptually simple, several technical challenges arise in enabling efficient transitions between these configurations:

**Cache invariance:** In general, the base and shift configurations are not automatically invariant. For arbitrary SP–TP combinations, head ordering in head parallelism breaks the invariance. We resolve this by developing a general process-to-data mapping to ensure KV cache consistency (3.3.1).

**Weight handling:** Transitioning between configurations requires that weights be compatible across both base and shift modes. We consider two strategies: (i) on-the-fly slicing, and (ii) explicit weight replication. Based on memory cost analysis, we adopt the latter as the preferred approach discussed in Section 3.3.2.

We designed Shift Parallelism for ease of use and adoption. It is integrated into vLLM via a plug-in system (Section 3.4) and is already deployed in production.

The rest of this section dives into the design details of SP and Shift Parallelism for inference.

### 3.2 SP for Inference

SP is essential for Shift Parallelism because it is the throughput-optimized counterpart of TP as they have the same KV cache layout, and therefore we can switch between two without changing the attention mechanism.

**3.2.1 Design for General Inference.** SP is originally implemented for training, and lacks important components,

[1] In the rest of the paper, we use Ulysses and SP interchangeably.

such as GQA mechanism [2], that is commonly used in inference models. The original MHA mechanism that is described in Section 2.4.

**GQA Extension:** In this work, we extend SP for GQA mechanism for adapting SP to a diverse set of LLMs that are used in inference. GQA saves memory by sharing each KV head with multiple query heads. However, multi-GPU scaling of GQA is nontrivial with models involving small #KV heads. For example, `Qwen-30B-A3B` attention has 4 KV heads, and it cannot be scaled to an 8×GPU node since there is not enough KV heads to be distributed across more than 4 GPUs.

TP solves this problem by replicating the KV weights in the QKV projection (see Section 2.3), which recomputes the KV cache within GQA groups redundantly. Unfortunately, this solution is not applicable to SP because each process owns only a slice of the input sequence and the missing slices cannot be replicated simply by recomputation.

**KV Cache Replication:** For SP in inference, we propose an in-network KV cache replication algorithm with multi-step neighborhood collectives. When $SP > h_{kv}$, i.e., #KV heads, we use a three step algorithm as demonstrated in Figure 6: We i) exchange available KV heads *across* the GQA groups, resulting neighborhood all-to-all communications within groups $(0, 2), (1, 3)$. Then ii) we replicate the heads in network using a neighborhood all-gather within GQA groups $(0, 1), (2, 3)$. As a result of the all-gather, each KV head is replicated within the GQA groups. We finally iii) interleave the incoming sequence to preserve the original sequence order.

Algorithm 1 shows the implementation. Query heads (*q_heads*) uses the SP communicator (Line 3) as usual, and KV heads (*kv_heads*) use $SP_{AA}$ for all-to-all communication (Line 4) and $SP_{AG}$ (Line 5) for all all-gather communication. Line 6 shows the reordering that depends on $SP_{AA}$ and $SP_{AG}$.

**Small Batch Size and Load Imbalance:** The main problem with SP is the load imbalance with small batch sizes, i.e., $n$ is small in Algorithm 2. This problem does not exist in training because batches are large and static, whereas in inference, the batch size varies according to the traffic.

Specifically, decoding in low traffic yields small batch sizes because there is only a few tokens produced at a time. Small batch sizes comparable to the SP degree cannot be evenly partitioned across GPUs, causing serious load imbalance. For example, when the batch size is 9 and $SP = 8$, all GPUs to process a single token except the one that processes two tokens, causing 50% efficiency. SP even breaks down when $SP >$ batch size, causing sparse communications.

decoding 中不同 SP degree 用 DP 的方式处理

To provide load balancing, we pad batches up to a multiple of SP degree so that we can evenly distribute them. Nevertheless, the padding results in redundant tokens, yielding a longer TPOT and hence longer request completion time compared to TP in low-traffic decoding.

rank 0: {S0}_{KV01}
rank 0: {S1}_{KV01}
rank 2: {S2}_{KV01}
rank 3: {S3}_{KV01}

rank 0: {S0S2}_{KV0}
rank 1: {S1S3}_{KV0}
rank 2: {S0S2}_{KV1}
rank 3: {S1S3}_{KV1}

rank 0: {S0S1S2S3}_{KV0}
rank 1: {S0S1S2S3}_{KV0}
rank 2: {S0S1S2S3}_{KV1}
rank 3: {S0S1S2S3}_{KV1}

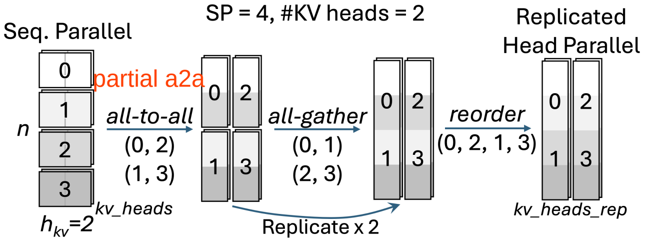


**Figure 6.** KV cache replication in a model with 2 KV heads on 4 GPUs. The GQA groups (0, 1) and (2, 3) replicates the KV cache as a result of neighborhood collectives.

Q heads 按照正常 n_rank=4 的 all-to-all 进行

**Algorithm 1** KV replication with $SP = SP_{AA} \times SP_{AG}$.

```
1: $qkv[n/SP, h + 2h_{kv}] \leftarrow embed[n/SP, d] * layer_i.qkv[d, h + 2h_{kv}]$
2: $q\_heads[n/SP, h], kv\_heads[n/SP, 2h_{kv}] \leftarrow qkv.split([h, 2h_{kv}])$
3: $q\_heads[n, h/SP] \leftarrow \boldsymbol{SP.all\_to\_all}(q\_heads)$
4: $kv\_heads[n/SP_{AG}, 2h_{kv}/SP_{AA}] \leftarrow \boldsymbol{SP_{AA}.all\_to\_all}(kv\_heads)$
5: $kv\_heads\_rep[n, 2h_{kv}/SP_{AA}] \leftarrow \boldsymbol{SP_{AG}.all\_gather}(kv\_heads)$
6: $kv\_heads\_rep \leftarrow kv\_heads\_rep.reorder((SP_{AG}, (SP_{AA}))$
7: $qkv\_heads[n, h/SP + 2h_{kv}/SP_{AA}] \leftarrow [q\_heads, kv\_heads\_rep]$
```

**Fusing Communications:** The QKV matrix fuses operations related to q, k, and v, bringing multiple communications down to a single matrix all-to-all communications together as shown in Algorithm 2 Line 4. When KV cache is replicated, the all-to-all symmetry in communications breaks down, and so the fusing. Fusing all the communications in Algorithm 1 communications is not trivial, yet we fuse the all-to-all and all-gather for k and v as a minor optimization.

### 3.2.2 Combined (SP, TP) Algorithm.

We need to combine SP with TP for handling large models that does not fit (or barely fits) in a single GPU. For a throughput-optimal config, we avoid partition the model with TP as much as each partition fits into GPU memory, and there is enough room for KV cache for providing concurrency and high throughput. Then rest of the GPUs can be efficiently employed using SP, which enlarges KV cache. For example, our evaluation involves `Llama-17B-16E` (FP8) has 109 GB memory footprint, yet needs at least $TP = 2$ for processing long contexts concurrently in within 141 GB GPU memory, and therefore we need a combination of $(TP = 2, SP = 4)$ for an optimal deployment on a node with 8 GPUs.

Algorithm 2 shows the forward pass algorithm with an arbitrary (SP, TP) configuration, where $n$ is the sequence length, $d$ is the hidden dimension, and $h$ is the number of heads. Algorithm 1 replaces Lines 3–4 of Algorithm 2 when KV cache is replicated.

## 3.3 Shift Parallelism

Shift parallelism (the main contribution of this paper) is designed to obtain low latency (TTFT and TPOT) in low traffic, and high throughput in high traffic by switching

**Algorithm 2** Combined ($SP, TP$) for the base config.

```
$embed[n/SP, d] \leftarrow SP.slice(input\_embeds[n, d])$
for $i = 1, \ldots, L$ do
    $qkv\_heads[n/SP, 3 \times h/TP] \leftarrow embed * layer_i.qkv[d, 3 \times h/TP]$
    $qkv\_heads[n, 3 \times h/(SP \times TP)] \leftarrow SP.\mathbf{all\_to\_all}(qkv\_heads)$
    $\boxed{attn\_o[n, h/(SP \times TP)] \leftarrow layer_i.attn(qkv\_heads)}$
    $attn\_o[n/SP, h/TP] \leftarrow SP.\mathbf{all\_to\_all}(attn\_o)$
    $embed[n/SP, d] \leftarrow attn\_o * layer_i.o[h/TP, d]$
    $TP.\mathbf{all\_reduce}(embed)$
    $act[n/SP, d'/TP] \leftarrow embed * layer_i.mlp\_up[d, d'/TP]$
    $embed[n/SP, d] \leftarrow act * layer_i.mlp\_down[d'/TP, d]$
    $TP.\mathbf{all\_reduce}(embed)$
end for
$output\_embeds[n, d] \leftarrow SP.\mathbf{all\_gather}(embed[n/SP, d])$
return $output\_embeds$
```

across parallelisms. The optimal parallelisms that we cover with Shift Parallelism are summarized in Table 3.

**Table 3.** Optimal Parallelisms Covered by Shift Parallelism.

| | **Low Traffic** | **High Traffic** |
|---|---|---|
| **TTFT** | SP | SP |
| **TPOT** | TP | SP |
| **Throughput** | SP* or TP | DP |

*SP for long input, TP for long output.

We can apply shift parallelism only across TP and SP because of their KV cache invariance property (Section 3.3.1), as a result, Shift Parallelism provides superior performance in the highlighted cases. The only case Shift Parallelism loses on DP (but wins on TP) is the throughput in high traffic, because parallel attention inevitably requires GPU communications.

In shift parallelism, we have two configurations; a) the base config that implements SP to optimize TTFT and throughput, and b) the shift configuration that implements full TP to optimize TPOT. The base configuration can optionally be a combination of TP and SP, of model does not fit into a single GPU (Section 3.2.2).

***How do we shift?*** The main criteria of switching between configurations is simple: We choose the base model for large batch size and the shift model for small batch size. Therefore, we decide on a shift parallelism threshold, if the batch size is larger than the threshold, we choose $(SP, TP)$ configuration, and choose the shift configuration, i.e., full-TP on $(SP \times TP)$ group as Algorithm 3 describes.

**Algorithm 3** Shift parallel ($SP \times TP$) forward pass.

```
if $n > threshold$ then
    return Algorithm 2[SP, TP]($input\_embed[n, d]$)
else
    return Algorithm 2[1, SP × TP]($input\_embed[n, d]$)
end if
```

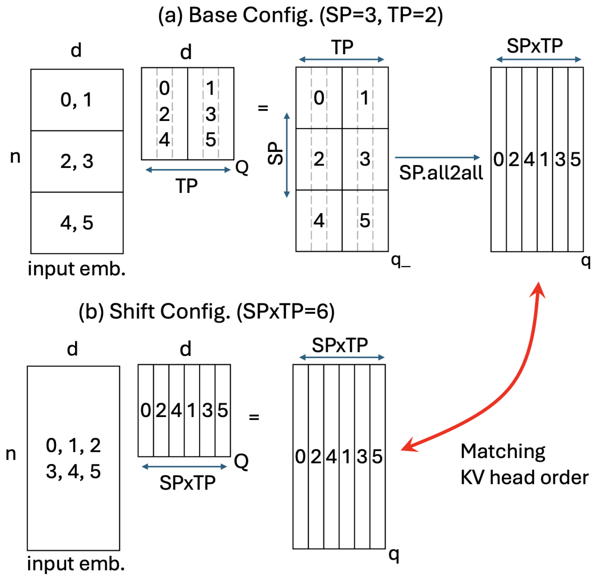


**Figure 7.** KV cache invariance of six heads across the base and shift configs. The shift config should shard the Q weights according to SP and TP degrees of the base config.

### 3.3.1 General KV Cache Invariance.

The KV cache invariance does not only require the same attention head layouts, but also the same ordering of the heads. Interestingly, the invariance across SP and TP breaks down when shifting across arbitrary $(SP, TP)$ and $(SP \times TP)$ configurations. When the base configuration involves a combination of SP and TP, e.g., $(SP = 3, TP = 2)$ the attention head order does not follow the same order as in $TP = 6$, i.e., not $(0, 1, 2, 3, 4, 5)$, anymore but $(0, 2, 4, 1, 3, 5)$. The attention mechanism do not care about the order of the heads as long as we lock in to an order (i.e., base config's) and stick to it when switching to the shift config as depicted in Figure 7.

Figure 7 shows the distributed memory layout of Q projection with the (a) base config and (b) the shift config. The original config yields TP groups $(0, 1), (2, 3), (4, 5)$ and SP groups $(0, 2, 4), (1, 2, 5)$. TP groups partition the Q weights across heads (i.e. columns) and replicates the input embeddings. SP groups partitions the embeddings across the sequence (i.e., rows), and replicates the Q weights. As a result of the 2D partitioning, the output of the linear layer ($q\_$) has a global layout as shown in the figure. As a result of the all-to-all communication within the SP groups, the resulting head partitions ($q$) results in interleaved head ordering $(0, 2, 4, 1, 3, 5)$. We need to adjust the attention head ordering of (b) shift config accordingly to provide KV cache consistency.

### 3.3.2 Memory Management.

There are two ways of implementing Shift Parallelism with generalized KV cache invariance. 1) slicing the model weights on-the-fly and 2) loading separate models that shares the attention mechanism (and the KV cache). We use 2) in our implementation.

***On-the-fly slicing.*** This implementation modifies the linear layer implementation of the original code such that each GPU multiplies a slice of the base model's weight partition. To preserve KV cache invariance, each GPU must have the slice according to their SP ranks. For example, global ranks 2, and 3 in Figure 7 gets heads 1, and 4 which are already in the base model's respective weight partition.

Slicing provides the same effect with TP, and has no memory overhead since the running buffer can be reused for all layers. Nevertheless, it is not as performant as the next solution because of each slicing requires matrix transposition due to an FP8 hardware limitation of Hopper tensor cores.

***Separate Models.*** The separate model solution do not share the weights across the base and shift configs, but replicates the weights. In this implementation, we load two separate models, one for the base configuration and one for the shift configuration, and these models share the same KV cache.

When loading the weights for the shift model, we use a separate group, `SP_TP` that spans both `SP` and `TP` groups, but with the order of `SP` group to preserve KV cache coherency do that the shift model will load the right wieght shards as shown in Figure 7.

- `TP: [[0, 1], [2, 3], [4, 5]]`
- `SP: [[0, 2, 4], [1, 3, 5]]`
- `SP_TP: [[0, 2, 4, 1, 3, 5]]`

With the separate model solution, we can write the total weight footprint as

$$w_{total} = \frac{w_{base}}{TP} + \frac{w_{base}}{SP \times TP}, \tag{1}$$

where the first and second terms represent the base and shift models' weights, respectively. As a results, the memory overhead of the shift model is $1/SP$, i.e., an base model with more SP and less TP alleviates the memory overhead of the shift model. For example, when SP=8, the shift model's memory overhead is 12.5%.

### 3.4 Integration into vLLM

None of the existing inference frameworks implement SP (Ulysses), and also modifying existing enterprise frameworks (such as vLLM) is tedious. We overcame the problem by developing a plug-in system [17] for implementing the proposed techniques in this paper.

For achieving low latency in inference, it is crucial to enable compilation and CUDA graph capture mechanisms in vLLM because the batch size can be small in low traffic. However, capturing all-to-all communications caused compilation issues because their dynamic memory management involved dynamic behaviors that are not supported by vLLM's compilation infrastructure. As a remedy, we modified compilation by relaxing the assumptions that vLLM make for fully capturing additional GPU kernels and communications related to Shift Parallelism.

## 4 Evaluation

In this section, we demonstrate the Shift-Parallelism can mitigate the latency vs throughput tradeoffs commonly seen in TP and DP. More specifically, we show

1. Shift Parallelism can adapt to bursty real-world traffic pattern, achieving simultaneously lowest latency (up to 3.23× lower) and near optimal throughput compared to TP and DP .
2. On low traffic Shift Parallelism can achieve up to 6.97× lower TTFT, 2.45× lower TPOT than DP, and up to 51% higher throughput than TP.
3. Extensive evaluation of Shift Parallelism over a wide range of sequences and request traffic demonstrating consistently superior performance (1.67×–6.97× faster response, 1×–2.45× faster generation and 1.51× higher throughput) compared to TP and DP, guaranteeing lowest latency with low cost over the entire spectrum, even in high traffic.
4. Shift Parallelism can accelerate real-world production deployment offering fastest open-source inference solution (3.4× lower completion time, and 1.06× higher throughput) by composing with SoTA inference technologies like SwiftKV and Speculative Decoding,
5. The cost breakdown analysis of DP, TP, and Shift Parallelism and explore further tradeoffs across dense and sparse models.

### 4.1 Experimental Setup

#### 4.1.1 Hardware.

Unless specified, we use AWS instances with 8xH200 GPUs each, i.e., `p5en.48xlarge`. Each GPU has 141 GB memory with 4.8 TB/s bandwidth, and also provides a peak dense matrix multiplication of 1,979 FP8 TFLOPS with tensor cores. The GPUs are interconnected with an NVSwitch network with 900 GB/s rated bandwidth.

#### 4.1.2 Software.

Unless specified, we use our implementation (Sec. 3.4) plugged into vLLM v0.9.2. For comparison, we use SGLang [21] v0.4.6, TRT-LLM [12] v0.18.2.

**Table 4.** Models used in evaluation.

|  | Num. | Num. | Hidden | # Heads |  |
|---|---|---|---|---|---|
| **Model Name** | **Params.** | **Lay.** | **Size** | **Q** | **KV** |
| `Llama-70B` | 70B | 80 | 8192 | 64 | 8 |
| `Qwen-32B` | 32B | 64 | 5120 | 64 | 8 |
| `Llama-17B-16E` | 109B/17B | 48 | 5120 | 40 | 8 |
| `Qwen-30B-A3B` | 30B/3B | 48 | 2048 | 32 | 4 |

#### 4.1.3 Models.

We use the models listed in Table 4, all with FP8 quantization. Shift Parallelism is originally designed for dense models, therefore we first present the main evaluation for L70B and Q32B, and then we discuss the performance limitations with Q30B-3B and L17B-16E which are mixture of experts (MoE) models—their static and active number of parameters are shown separately in Table 4.

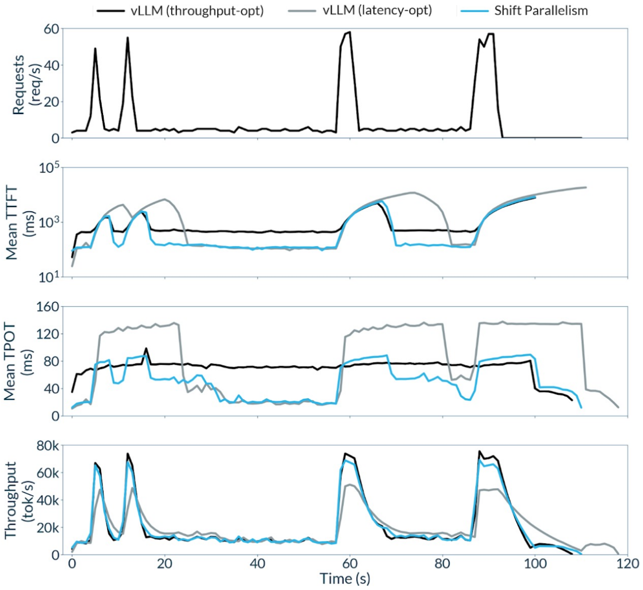


**Figure 8.** Shift Parallelism achieves the lowest response, fastest generation and near-optimal throughput under dynamic traffic. We used Llama-70B and a modified version vLLM's serving benchmark that makes requests a steady stream of request at low frequency with occasional bursts of high frequency requests.

**4.1.4 Datasets.** We use three three types of datasets. i) synthetic requests with random data for parameterized benchmarking, ii) a trace that is a mixture of requests from HumanEval [3] and from a CodeAct agent [10] running against SWEBench [7] (HumenEval is one-shot and SWEBench is agentic), iii) a filtered real-life dataset that matches the synthetic dataset requests for the sake of running speculative decoding, and iv) a bursty traffic pattern that resembles real-life production environment.

## 4.2 Latency and Throughput in Real-World Traffic

For testing Shift Parallelism on real-life environment, we create a bursty dataset by changing the arrival using vLLM's burstiness benchmark. Figure 8 (top) shows resulting traffic pattern that has four high-traffic bursts. The rest of the results show the input latency (i.e., TTFT) and the output latency (i.e., TPOT) that is experienced by a request in milliseconds, and also the combined input/output token throughput of all requests in tokens per second.

To obtain Figure 8, we randomly mix two real-life datasets that are described in Section 4.1.4. The mix involves both latency- and throughput-critical requests with variable sizes.

Table 5 summarizes the latency and throughput statistics that are collected from the trace of the bursty workload experiment (Figure 8). The experiment trace with vLLM's TP and DP, and also the proposed Shift Parallelism shows that

**Table 5.** Performance stats with the bursty workload.

| | Median TTFT | Median TPOT | Peak Throughput |
|---|---|---|---|
| vLLM (throughput opt.—DP) | 1,355 ms | 83 ms | 75,535 tok/s |
| vLLM (latenct opt.—TP) | 3,930 ms | 85 ms | 51,162 tok/s |
| vLLM+Shift Parallelism | 148 ms | 51 ms | 69,147 tok/s |

- Shift Parallelism obtains the lowest latency across TP and DP with bursty (dynamic) traffic since can sustain low latency with a higher traffic than the other two .
- Shift Parallelism obtains a higher peak throughput than TP, and therefore processes the batches in a shorter time. As a result, there wait time for latency-critical request is reduced significantly, i.e., TTFT does not explode with Shift Parallelism (148 ms vs. 3.9 sec.).

Overall, Shift Parallelism can handle the high-traffic bursts better than both TP ad DP, achieving up to 9.16× lower TTFT and 1.63× lower TPOT that both, while at the same time achiveing nearly as good throughput as DP during high-traffic periods, ultimately, improving the quality of service.

## 4.3 Performance Benchmarks

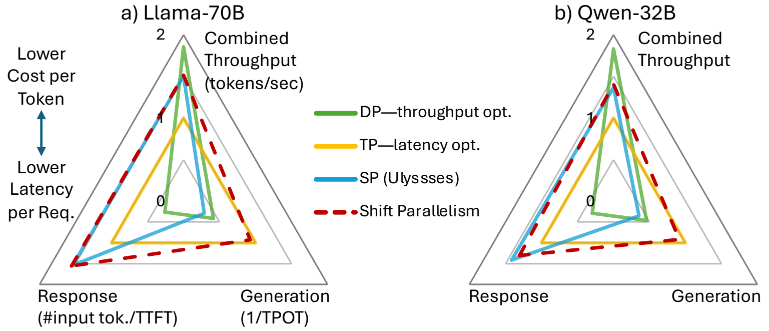
In this section we send synthetic requests for presenting performance characteristics via parameterized experiments.

***Latency vs. throughput tradeoff.*** We evaluate the latency vs. throughput tradeoff across parallelism using a uniform request size of 4k input tokens and 250 output tokens. For finding the peak throughput, we send a batch of requests (thousands) and provide sufficient concurrency to saturate the GPU throughput. For finding the lowest latency, we process requests sequentially, i.e., a single request at a time.

Figure 1 compares DP, TP, SP, and Shift Parallelism across Llama-70B and Qwen-32B. Shift Parallelism achieves the lowest TTFT that is 1.56× and 6× lower than TP and DP for Llama, and 4.45× and 1.31× for Qwen. Shift Parallelism achieves that lowest TPOT that is 9.34 ms for Llama and 8.68 ms for Qwen. Shift Parallelism experiences significantly less throughput degradation compared to TP. Specifically, TP loses 46% and 45% throughput with Llama and Qwen, yet Shift Parallelism only loses 18% and 23%, respectively. A bit part of it comes from vLLM's engine overhead that we discuss in Section 4.5.

***Variations across context sizes.*** For investigating TTFT, TPOT, and throughput with various input context sizes, we repeat the experiment in Figure 9 for input sequences with 2k–128k tokens and 250 output tokens. Figure 14 presents the input and output latency in low traffic, and throughput in high traffic.

Shift Parallelism provides a 6.97× and 1.56× faster response than DP and TP, respectively, because it uses SP for prefill, which is more efficient than both DP and TP. As a result, Shift Parallelism is more responsive and also provides



c) Raw Measurement

| Model | Perf. Metric | Unit | DP | TP | SP | Shift Par. |
|---|---|---|---|---|---|---|
| Llama-70B | TTFT | Millisec. | 614 | 159 | **103** | **102** |
| | TPOT | Millisec. | 22.5 | **9.34** | 32.5 | **10.1** |
| | Comb. Throughput | Tok./Sec. | 45.9k | 24.7k | **37.2k** | **37.4k** |
| Qwen-32B | TTFT | Millisec. | 385 | 113 | **79.8** | **86.41** |
| | TPOT | Millisec. | 18.8 | **8.68** | 24.4 | **9.48** |
| | Comb. Throughput | Tok./Sec. | 70.1k | 38.3k | **52.3k** | **53.8k** |

**Figure 9.** Comparison of response and generation latency, and throughput, all in tokens/sec. with (a) Llama-70B and (b) Qwen-32B based on the (c) measurements. Shift Parallelism simultaneously obtain a higher throughput and a lower latency than TP, alleviating the latency vs. throughput tradeoff of existing parallelisms.

a faster completion time especially for long input and short output contexts where TTFT dominates the completion time (such as in summarization).

Shift Parallelism provides up to 2.45× faster generation than DP and a similar output latency to TP. Ideally, the output latency should not depend on the input context size in theory, but in practice TPOT increases with the input size (see Figure 10). The main reason is that each output token needs to read more number of tokens from the KV cache as input context grows, and eventually the system becomes memory bandwidth bound. TP and Shift Parallelism parallelizes the attention layer (Section 2.3), and hence the KV cache, providing memory bandwidth, mitigating the output latency for long inputs.

Shift Parallelism obtains up to 1.51× higher peak throughput than TP, meaning that processing the high-traffic bursts and also batch workloads is approx. 50% faster with Shift Parallelism. Nevertheless, the throughput drops significantly with larger contexts because attention time dominates the end-to-end generation. See Section 4.5 for analysis.

***Latency vs. Arrival Rate.*** To investigate the performance between extremely high and low traffic rates, we test Shift Parallelism across a wide range of intermediate traffic by varying the request arrival rates. We measure TTFT and TPOT of an individual request, which both increases with higher traffic, and calculate the completion time as TTFT + #output tok.×TPOT. The question is, where does the tradeoff

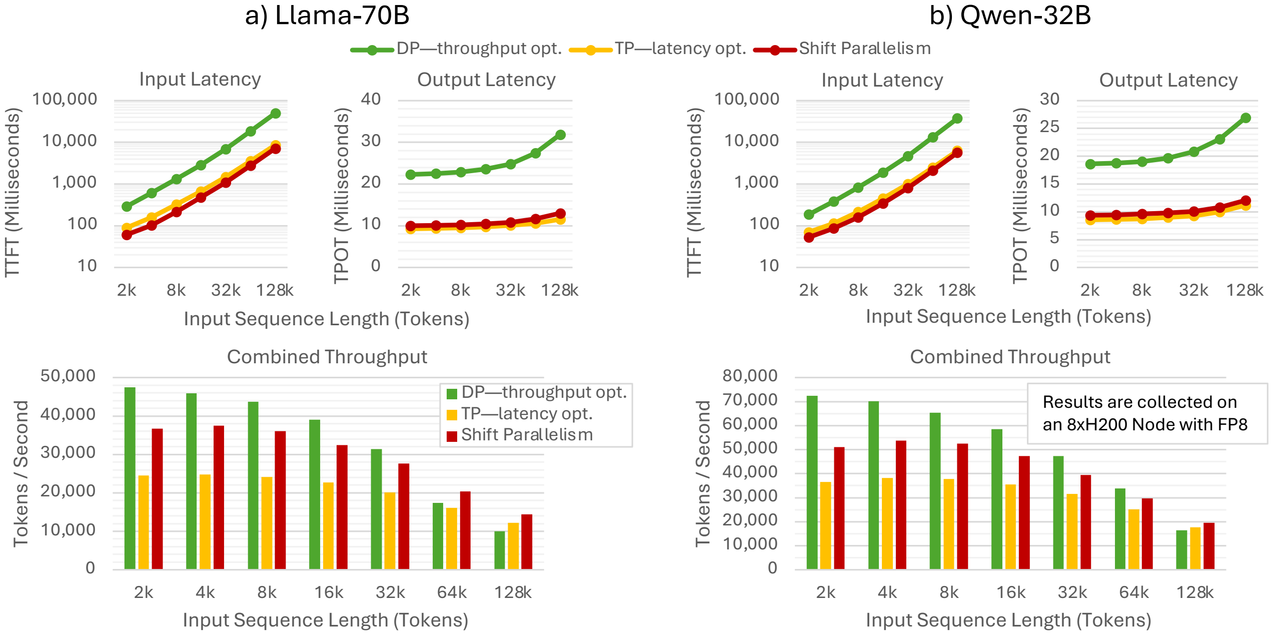


**Figure 10.** Performance variation across input sequence length: Minimum latency (TTFT, TPOT), and maximum throughput of (a) Llama-70B and (b) Qwen-32B. The throughput drops with large context sizes due to the excess attention time.

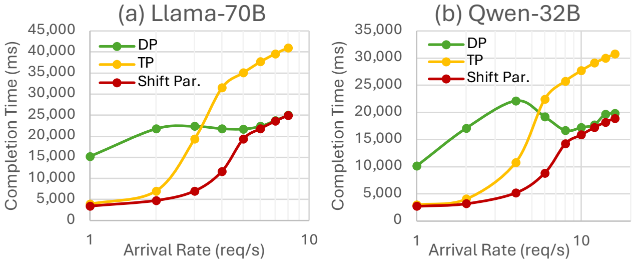


**Figure 11.** Request completion time vs. arrival rate. TP and DP make the performance tradeoff across arrival rates. Shift Parallelism strictly obtains the lowest completion time across arrival rates. Request size: 8k input, 250 output.

happens, and how well Shift Parallelism transitions from latency to throughput optimization?

Results in Figure 11 shows the performance variation across arrival rates. TP and DP curves cross over at a critical arrival rate (a few req/sec). Yet Shift Parallelism guarantees the lowest latency regardless of the arrival rate—strictly better than both DP and TP solutions. In low-to-medium rates (req/s), Shift Parallelism switches back-and-forth across SP and TP for minimizing the input (TTFT) and output (TPOT) latencies, respectively. In high traffic, Shift Parallelism uses SP to save combined throughput (tokens/sec).

### 4.4 Shift Parallelism in Production

We fully integrated Shift Parallelism in our existing production environment (see Section 3.4). Running efficiently in production is not only about parallelism, but it also requires a plethora of other state-of-the-art techniques. To that extent, we integrated Shift Parallelism with SwiftKV [14] and speculative decoding [13, 18] in our production environment.

Figure 12 shows that our combined production simultaneously achieves highest throughput (lowest cost) and lowest completion time—all in one deployment—outperforming the best open source systems optimized for each metric individually [2].

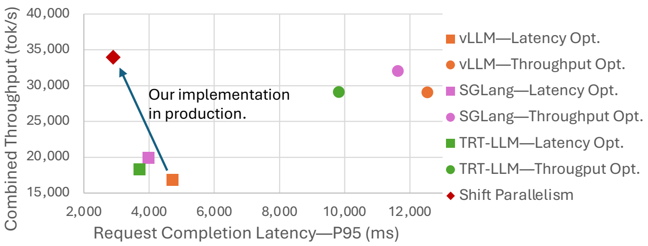


**Figure 12.** Comparison of our production environment with other frameworks using a real dataset on Llama-70B.

### 4.5 Cost Breakdown

We analyze the cost of individual system components by taking away one component at a time. Figure 13 shows the resulting breakdown of time to process 1,920 requests with Llama-70B and Qwen-32B models on a single node. We clearly see that SP (and hence Shift Parallelism) has a lower communication cost than TP. On the other hand, we observe two unaddressed performance bottlenecks that are not related to Shift Parallelism:

- Attention time grows significantly with the sequence size, and therefore reduces the combined throughput. Recent papers address this issue using sparse attention [cite] and it is out of scope of this paper.
- The parallelization cost of vLLM is significant in small models (e.g., compare Llama-70B, Qwen-32B). We find vLLM cost by removing the forward pass. This indicates that a large portion of the remaining throughput gap between DP and SP might actually be the vLLM overhead unrelated to SP.

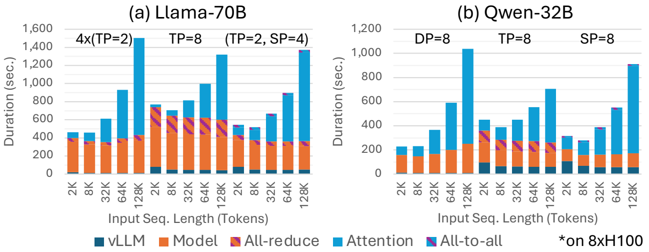


**Figure 13.** End-to-end cost breakdown* of time spent in a batch workload with (a) Llama-70B and (b) Qwen-32B. Shorter seq. → vLLM overhead, longer seq. → attn. time.

### 4.6 Limitations and Future Work

To investigate further on Shift Parallelism's performance behavior, we stretch our evaluation to two recently released sparse (i.e., MoE) models that are listed in the bottom two rows of Table 4.

To enable these models, we use the SP generalizations in Section 3: i) Llama-17B-16E barely fits into a single GPU and when SP=8 is used in the base config., there is no memory left in the KV cache to support large context sizes. To enable long contexts, we use Algorithm 2 for the base config. (SP=4, TP=2). ii), Qwen-30B-A3B suffers from scaling because it only has 4 KV heads, and we use Algorithm 1 to scale the model across SP=8 GPUs.

Figure 14 compares the throughput and latency performance in across the board. The models sorted from larger to smaller. The sparse models attains a higher throughput and lower latency than dense models, simply because they

[2] We enabled the best available speculative decoding for each framework. These experiments were run on data sets generated using real-world production traces to compute throughput, and a mixture of ShareGPT, HumanEval and SWEBench to measure latency. As a result, these results are representative of performance achievable in real-world deployments. For more details, see the evaluation methodology in the appendix.

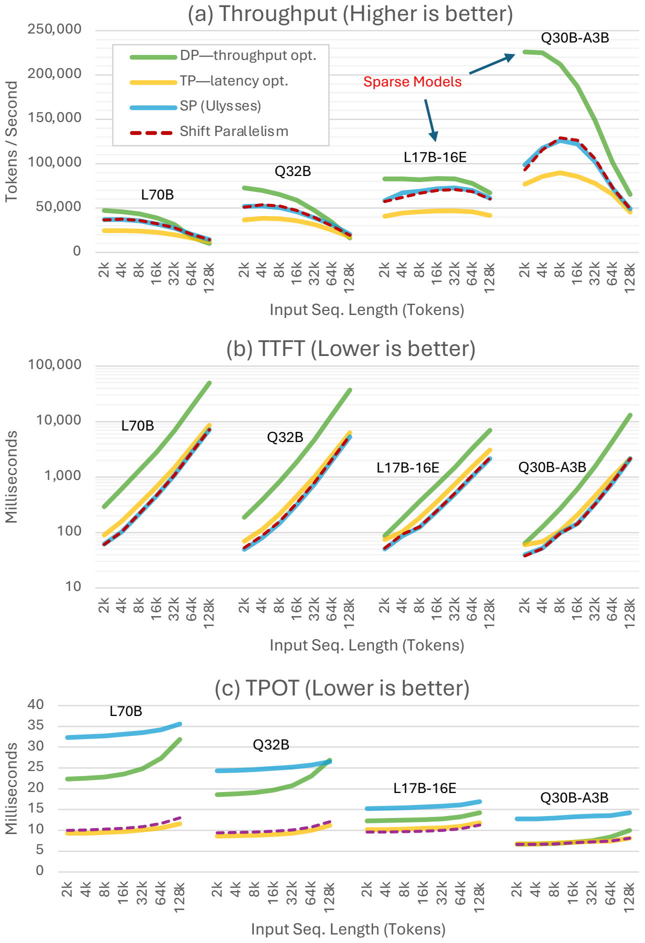


**Figure 14.** (a) Peak Throughput and minimum latency—(b) TTFT and (c) TPOT—comparison across parallelisms, models and input sequence lengths. Shift Parallelism switches across SP and TP for obtaining high throughput and low latency.

have less number of active parameters (see Table 4). Compared to TP, Shift parallelism shows excellent performance on both sparse models: Up to 50% higher throughput without increasing the latency.

The smallest model, Qwen-30B-A3B, attains a much higher throughput (250k tokens/sec.) than the other models with DP, and the throughput suddenly drops either with TP and SP, especially with smaller end of input sizes. The discrepancy is due to the vLLM's parallelization overhead with small models that is discussed in Section 4.5.

The sparse models beg further investigation in the context of expert parallelism (EP). Specifically, there is no prior work that combines SP with EP to to further optimize sparse models, which we will leave as a future work.

## 5 Related Work

The heterogeneous resource demands of LLM inference is one of its main challenges, and prior works have proposed different techniques to manage it. We discuss a few works related to Shift Parallelism in this section.

*Chunked Prefill* [1, 5] tackles the heterogeneous resource demand of LLM inference requests in their input-processing (prefill) stage vs their output-generation (decode) phase. During prefill, the input tokens are processed in parallel and is thus compute-intensive, while during decode, a single token is generated at a time but requires the full KV-cache of prior tokens, and is thus memory-intensive. Chunked prefill is proposed by DeepSpeed-FastGen [5] and Sarathi-Serve [1] and mixes requests in both prefill and decode phases in the same batch, thereby improving resource utilization and combined throughput. Today, it is default in many popular inference engines [8, 12, 21]. In comparison, Shift Parallelism is targeted at time-varying sizes of each batch. It is orthogonal and compatible with chunked prefill, and our experiments in Sec. 4 are run using their combination.

*Disaggragated Inference* [15, 22] uses separate GPU workers for prefill and decode so that prefill throughput and decode latency can be separately scaled and optimized. Compared with chunked-prefill systems and Shift Parallelism, disaggregated inference can eliminate interference between requests in the prefill and decode stages, but at the cost of dedicating additional resources to each stage. Additionally, the KV cache must be communicated from a prefill worker to a decode worker for each request, causing extra communication overhead. In contrast, Shift Parallelism with chunked-prefill overlaps prefill and decode, with decode tokens accessing the KV cache from local memory, resulting in more efficient resource utilization and less cost per token.

## 6 Conclusion

LLM's have diverse real-world applications that yields different traffic patterns that have different performance requirement. Fundamentally, existing parallelisms (TP and DP) for inference optimize either for latency (for interactive applications) or throughput (for batch workloads), but do not optimize for multiple applications in the same deployment.

For supporting dynamic workloads, we need to switch back-and-forth between parallelisms, yet cannot switch across TP and DP due to their KV cache mismatch. As a remedy, we bring in SP (Ulysses) that is originally applied to training for high throughput, and generalized it for inference, and providing the KV cache invariance across SP and TP by addressing corner cases. We call our solution Shift Parallelism, where a deployment has two configs. that switch back and forth across SP and TP.

Our extensive benchmarking show that Shift Parallelism addresses the latency vs. throughput tradeoff in a low cost way, providing up to 50% more throughput compared without losing latency across traffic rates. We fully, integrated Shift Parallelism along with other SoTA that we use in production, and open source our inference system [16].

## References

[1] 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:2403.02310 [cs.LG] https://arxiv.org/abs/2403.02310

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Felipe Lebrón, 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*. Association for Computational Linguistics, Singapore, 4895–4901. doi:10.18653/v1/2023.emnlp-main.298

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[4] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL] https://arxiv.org/abs/2312.10997

[5] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. arXiv:2401.08671 [cs.PF] https://arxiv.org/abs/2401.08671

[6] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. arXiv:2309.14509 [cs.LG] https://arxiv.org/abs/2309.14509

[7] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] https://arxiv.org/abs/2310.06770

[8] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:2309.06180 [cs.LG] https://arxiv.org/abs/2309.06180

[9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL] https://arxiv.org/abs/2005.11401

[10] Graham Neubig and Xingyao Wang. 2024. OpenHands CodeAct 2.1: An Open, State-of-the-Art Software Development Agent. *All Hands AI Blog* (1 November 2024). https://www.all-hands.dev/blog/openhands-codeact-21-an-open-state-of-the-art-software-development-agent

[11] Junichiro Niimi. 2024. Dynamic Sentiment Analysis with Local Large Language Models using Majority Voting: A Study on Factors Affecting Restaurant Evaluation. arXiv:2407.13069 [cs.CL] https://arxiv.org/abs/2407.13069

[12] NVIDIA Developer. 2023. NVIDIA TensorRT-LLM: An Open-Source Library for Accelerating LLM Inference. https://developer.nvidia.com/blog/optimizing-inference-on-llms-with-tensorrt-llm-now-publicly-available/. Accessed: August 19, 2025.

[13] Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. 2025. SuffixDecoding: Extreme Speculative Decoding for Emerging AI Applications. arXiv:2411.04975 [cs.CL] https://arxiv.org/abs/2411.04975

[14] Aurick Qiao, Zhewei Yao, Samyam Rajbhandari, and Yuxiong He. 2025. SwiftKV: Fast Prefill-Optimized Inference with Knowledge-Preserving Model Transformation. arXiv:2410.03960 [cs.LG] https://arxiv.org/abs/2410.03960

[15] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. arXiv:2407.00079 [cs.DC] https://arxiv.org/abs/2407.00079

[16] Samyam Rajbhandari, Mert Hidayetoglu, Aurick Qiao, Ye Wang, Juncheng Yang, Jeff Rasley, Michael Wyatt, and Yuxiong He. 2025. Arctic Inference with Shift Parallelism: Fast and Efficient Open Source Inference System for Enterprise AI. arXiv:2507.11830 [cs.DC] https://arxiv.org/abs/2507.11830

[17] Snowflake AI Research. 2025. ArcticInference: A vLLM plugin for low-latency, high-throughput LLM inference. https://github.com/snowflakedb/ArcticInference.

[18] Ye Wang, Gabriele Oliaro, Jaeseong Lee, Yuxiong He, Aurick Qiao, and Rajbhandari Samyam. 2025. Fastest Speculative Decoding in vLLM with Arctic Inference and Arctic Training. https://www.snowflake.com/en/engineering-blog/fast-speculative-decoding-vllm-arctic.

[19] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. arXiv:2401.07339 [cs.SE] https://arxiv.org/abs/2401.07339

[20] Wenxuan Zhang, Yue Deng, Bing Liu, Sinno Jialin Pan, and Lidong Bing. 2023. Sentiment Analysis in the Era of Large Language Models: A Reality Check. arXiv:2305.15005 [cs.CL] https://arxiv.org/abs/2305.15005

[21] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104

[22] 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:2401.09670 [cs.DC] https://arxiv.org/abs/2401.09670