vLLM: An Efficient Inference Engine for Large Language Models

by

Woosuk Kwon

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Fall 2025

vLLM: An Efficient Inference Engine for Large Language Models

Abstract

vLLM: An Efficient Inference Engine for Large Language Models

by

Woosuk Kwon

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Large language models (LLMs) have emerged as transformative technology capable of human-level or superhuman performance across diverse tasks, from writing complex software systems to discovering novel algorithms and processing multimodal data. Despite these remarkable capabilities, deploying LLMs at scale presents significant challenges due to their enormous computational and memory requirements. State-of-the-art models contain trillions of parameters and perform tens of thousands of generation steps, executed across large GPU clusters, often under strict latency constraints. These challenges are further compounded by the rapidly evolving model architectures and the growing diversity of hardware accelerators.

To address these challenges, this thesis presents the design and implementation of vLLM, an efficient and flexible open-source LLM inference engine. We first introduce PagedAttention, vLLM's core memory management algorithm that enables high-throughput LLM inference. We then examine vLLM's system design in detail, highlighting its scheduling mechanisms, extensible architecture, and key performance optimizations that enable it to meet a wide range of deployment requirements.

Together, these contributions establish vLLM as a comprehensive solution to LLM inference, delivering high performance, architectural flexibility, and the strength of a rapidly growing open-source ecosystem. Through vLLM, this thesis illustrates how principled systems design can effectively bridge the widening gap between the accelerating evolution of modern LLMs and the demanding practical constraints of large-scale, real-world deployment.

To my family.

Contents

Contents **ii**

List of Figures **iv**

List of Tables **vi**

1 Introduction **1**

2 Challenges in Efficient LLM Inference **3**
2.1 Autoregressive Generation and KV Cache . . . 3
2.2 Mixture-of-Experts Models . . . 7
2.3 Latency Constraints . . . 7
2.4 Trends . . . 9

3 PagedAttention: Efficient Memory Management for LLM Inference **10**
3.1 Introduction . . . 10
3.2 Background . . . 13
3.3 Memory Challenges in LLM Serving . . . 15
3.4 Method . . . 17
3.5 Implementation . . . 26
3.6 Evaluation . . . 27
3.7 Ablation Studies . . . 33
3.8 Discussion . . . 35
3.9 Related Work . . . 35
3.10 Conclusion . . . 36

4 vLLM: The Design and Implementation **37**
4.1 Overview . . . 37
4.2 Interface . . . 41
4.3 Model Authoring . . . 45
4.4 Inference Optimizations . . . 49
4.5 Scheduling Policy . . . 56
4.6 CPU Overheads . . . 60

4.7 Conclusion 68

5 Conclusion and Future Work **69**

5.1 Summary of Contributions 69

5.2 Future Directions 70

5.3 Concluding Remarks 71

Bibliography **73**

List of Figures

2.1 Autoregressive LLM inference. Given an input prompt, the model first processes all prompt tokens through the transformer layers in a single forward pass. It then generates output tokens one at a time with each new token requiring a complete forward pass through all transformer layers. This sequential token-by-token generation fundamentally limits parallelism. 4

3.1 *Left:* Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemerally for activation. *Right:* vLLM smooths out the rapid growth curve of KV cache memory seen in existing systems [63, 110], leading to a notable boost in serving throughput. 11
3.2 Average percentage of memory wastes in different LLM serving systems during the experiment in §3.6.2. 12
3.3 KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions. 17
3.4 vLLM system overview. 18
3.5 Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory. 19
3.6 Block table translation in vLLM. 20
3.7 Storing the KV cache of two requests at the same time in vLLM. 21
3.8 Parallel sampling example. 22
3.9 Beam search example. 23
3.10 Shared prompt example for machine translation. The examples are adopted from [14]. 24
3.11 Input and output length distributions of the (a) ShareGPT and (b) Alpaca datasets. 28
3.12 Single sequence generation with OPT models on the ShareGPT and Alpaca dataset 29
3.13 Average number of batched requests when serving OPT-13B for the ShareGPT (2 reqs/s) and Alpaca (30 reqs/s) traces. 30
3.14 Parallel generation and beam search with OPT-13B on the Alpaca dataset. . . . 31

3.15 Average amount of memory saving from sharing KV blocks, when serving OPT-13B for the Alpaca trace. 31
3.16 Translation workload where the input prompts share a common prefix. The prefix includes (a) 1 example with 80 tokens or (b) 5 examples with 341 tokens. 32
3.17 Performance on chatbot workload. 32
3.18 Ablation experiments. 33
3.19 (a) Overhead of recomputation and swapping for different block sizes. (b) Performance when serving OPT-13B with the ShareGPT traces at the same request rate. 34

4.1 Overview of vLLM's architecture. 39
4.2 Two-level scheduling with First-In First-Out and Shortest Remaining Token First scheduling policies. 59
4.3 vLLM's Asynchronous Scheduling. 61
4.4 Block table management in vLLM. 65
4.5 Comparison of full and piecewise CUDA graph approaches. 66
4.6 Performance comparison of execution modes on H100 GPUs. Batch size is 1, input length is 4000 tokens, and output length is 50 tokens. 67

List of Tables

3.1 Model sizes and server configurations. 27

4.1 Unified representation of various scheduling optimizations. All behaviors are expressed through updates to `num_computed_tokens` and `num_total_tokens`. . . 52

Acknowledgments

I feel incredibly fortunate to have pursued my PhD at the University of California, Berkeley, surrounded by such an inspiring community of people. My time here has profoundly shaped both my personal growth and my professional development, and I am deeply grateful to everyone who made this journey possible.

First and foremost, I would like to express my deepest gratitude to my advisor, Ion Stoica. He has been the ideal PhD advisor I could have ever imagined. His vision, guidance, and unwavering support shaped the direction of my research, and I am especially thankful for his trust, encouragement, and mentorship throughout my PhD.

I would also like to thank Joey Gonzalez, who has been a wonderful mentor and a constant source of guidance. I am grateful to Hao Zhang for being a close mentor and friend throughout my PhD, and for always providing thoughtful and realistic advice. Lastly, I thank Scott Shenker for serving on my thesis committee. I truly enjoyed working with him on the SkyPilot project, and I greatly appreciate his insights and perspective.

I feel extremely lucky to have worked on the vLLM project for several years with many exceptional collaborators, both inside and outside Berkeley. At Berkeley, I would like to thank Zhuohan Li, Simon Mo, Kaichao You, Chen Zhang, Lily Liu, Bowen Wang, Kuntai Du, and Jongseok Park. The countless days and nights we spent working together were both challenging and unforgettable. In industry, I am grateful to our close collaborators, including Roger Wang, Nick Hill, Lu Fang, Ye Qi, Robert Shaw, Michael Goin, Cody Hao Yu, and many other contributors whose efforts made this work possible.

Beyond vLLM, I would like to thank my colleagues in Sky Lab, including Wei-Lin Chiang, Zhanghao Wu, Zongheng Yang, Romil Bhardwaj, Frank Sifei Luan, Michael Luo, Yifan Qiao, Charlie Ruan, and Yilong Zhao, for the stimulating discussions, collaboration, and camaraderie that enriched my time at Berkeley.

I am also deeply grateful to my Korean friends, including Suhong Moon, Sam Son, Sehoon Kim, Minwoo Kang, Joseph Suh, Sunjin Choi, Hansung Kim, Junsun Choi, and Dayeol Lee, for the many meals, drinks, and moments we shared together. You made Berkeley feel like home. Because of you, I never felt alone during my PhD, and I truly felt that I had a family.

Over the past few years, I was also fortunate to work part-time at Google DeepMind and Thinking Machines Lab. I am especially thankful to my DeepMind mentor, Yanping Huang, for his invaluable life advice and research insights. I also believe that working with Yinghai Lu and Horace He at Thinking Machines Lab significantly contributed to my growth, both technically and professionally.

Last but certainly not least, I am deeply indebted to my family. Thank you, Mom and Dad, for your unconditional love, understanding, and support throughout my life. I would not have been able to complete this journey without you. I dedicate this dissertation to my family.

Chapter 1

Introduction

Large language models (LLMs) have rapidly emerged as one of the most transformative technologies today. Over just a few years, the scale and capability of these models have grown at an unprecedented pace. Early LLMs demonstrated impressive linguistic fluency; current state-of-the-art models such as GPT [75], Gemini [32], and Claude [7] now exhibit broad general intelligence, performing complex reasoning tasks, writing entire software systems, solving Olympiad-level mathematical problems, and processing multimodal inputs such as images, audio, and video. As these capabilities continue to expand, LLMs have become central to a wide range of scientific, industrial, and consumer applications, from conversational agents to code assistants, decision-support tools, biomedical analysis, and automated content generation.

Despite these breakthroughs, a central barrier remains: **efficiently deploying LLMs at scale**. The largest models contain trillions of parameters, and even smaller open-weight models have expanded in size and diversity. These models require substantial computational resources during inference, often requiring tens of thousands of generation steps distributed across a cluster of GPUs. Beyond raw computational demands, production deployments must satisfy stringent latency requirements—users expect near-instantaneous responses, even as models grow larger and more capable. The memory footprint of these models, particularly the key-value (KV) caches used in attention mechanisms, creates substantial bottlenecks that limit throughput and increase serving costs.

These challenges are further complicated by the rapidly evolving nature of the LLM ecosystem. The field moves at an extraordinary pace, with competitive open-weight models released every few weeks, each bringing architectural innovations and improved capabilities. Deployment environments are equally diverse, spanning multiple generations of NVIDIA GPUs [67, 66] to architecturally distinct accelerators like Google TPUs [43]. This diversity demands inference systems that are not only efficient but also flexible and adaptable to changing requirements and hardware platforms.

This rapidly changing landscape exposes a fundamental systems challenge: **how can we build an inference engine that delivers high performance at scale, while remaining flexible enough to accommodate new models, new optimization techniques, and new hardware backends?**

In this thesis, we present vLLM as a comprehensive solution to the LLM inference problem through three key pillars:

- **Algorithmic enhancement:** At its core, vLLM introduces PagedAttention, an efficient memory management algorithm for the KV cache. PagedAttention eliminates memory fragmentation and enables flexible, fine-grained sharing of KV-cache space across concurrent requests, thereby achieving significantly higher throughput in LLM inference.
- **System design and implementation:** vLLM is architected and implemented to flexibly support diverse model architectures, inference optimizations, and hardware backends, providing a robust and extensible foundation for both research exploration and production deployment.
- **Open-source ecosystem:** To address rapidly evolving requirements, vLLM leverages the power of open-source development through active collaboration among model vendors, hardware vendors, cloud providers, and research communities.

The remainder of this thesis is organized as follows:

- **Chapter 2** provides an overview of LLM inference and outlines the key challenges in deployment at scale.
- **Chapter 3** focuses on memory management of KV cache and introduces PagedAttention as an effective solution to memory fragmentation.
- **Chapter 4** describes vLLM's system design and implementation, demonstrating how it achieves high performance while supporting diverse models and inference optimizations.
- **Chapter 5** concludes the thesis and discusses directions for future work.

Chapter 2

Challenges in Efficient LLM Inference

Large language models (LLMs) have emerged as a transformative technology, powering applications ranging from conversational assistants to code generation and autonomous agents. To deploy these applications at scale, we must run LLM inference efficiently on accelerators such as GPUs. However, LLM inference presents fundamental computational challenges that make it both slow and expensive: the models are enormous, often containing trillions of parameters, and their autoregressive generation process is inherently sequential. These characteristics combine to create a workload that severely underutilizes modern GPU hardware, limiting the success and broader adoption of LLMs today.

This chapter examines these challenges in detail. We begin by analyzing the autoregressive generation process and its implications for computational efficiency. We then discuss how Mixture-of-Experts architectures introduce additional complexity. Finally, we examine the latency constraints imposed by real-world applications and the trends shaping the future of LLM inference.

2.1 Autoregressive Generation and KV Cache

Modern LLMs generate text through *autoregressive decoding*: tokens are produced one at a time, with each new token conditioned on all preceding tokens. As illustrated in Figure 2.1, this process is inherently sequential—the model cannot generate the next token until it has generated the current one. Each token generation requires a complete forward pass through the entire transformer stack, from the embedding layer through all N transformer layers. This sequential dependency fundamentally constrains the parallelism available during text generation and has profound implications for system design.

The inference process consists of two distinct phases:

1. **Prefill phase**: The model processes the entire input prompt in a single forward pass, computing internal representations for all input tokens simultaneously. In Figure 2.1, this corresponds to the leftmost forward pass that processes "Artificial Intelligence is". This phase is computationally intensive but highly parallelizable, as all prompt tokens can be processed together.

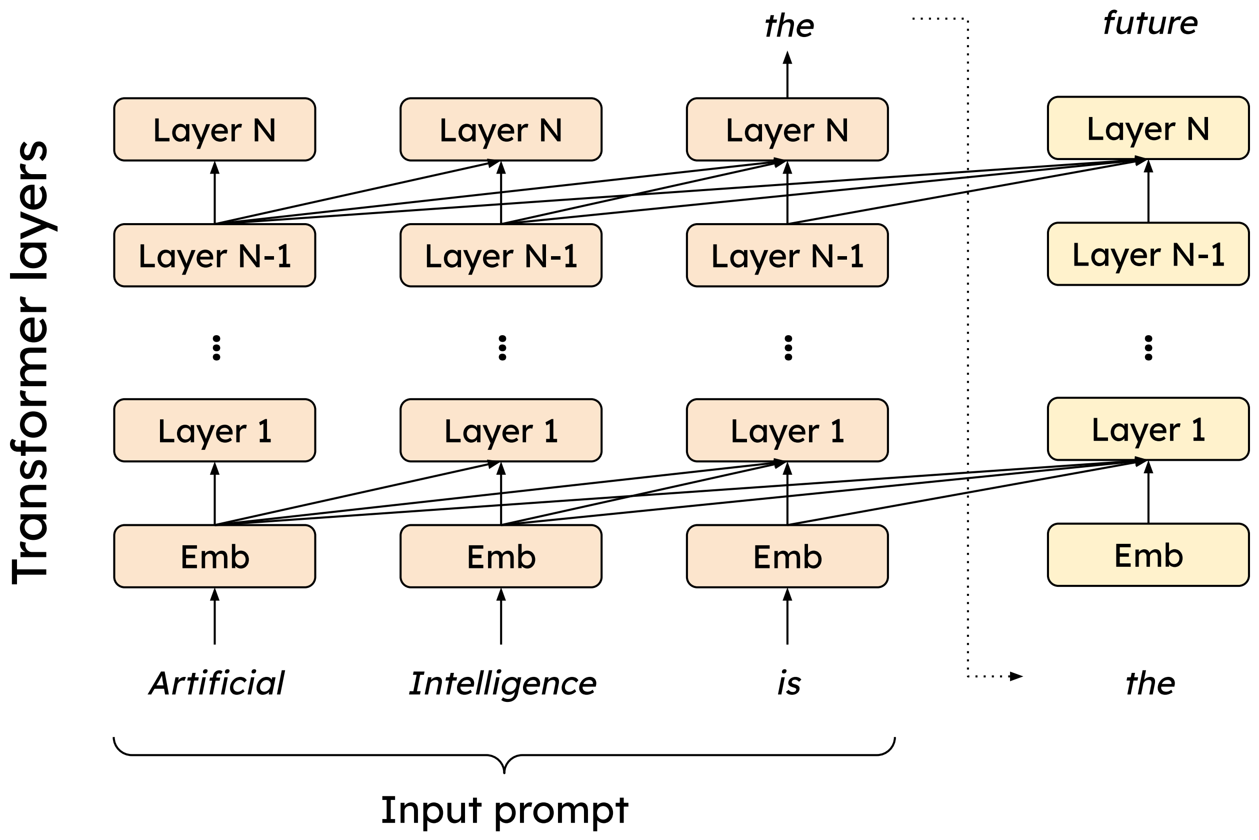


Figure 2.1: Autoregressive LLM inference. Given an input prompt, the model first processes all prompt tokens through the transformer layers in a single forward pass. It then generates output tokens one at a time with each new token requiring a complete forward pass through all transformer layers. This sequential token-by-token generation fundamentally limits parallelism.

2. **Decode phase**: The model generates output tokens one at a time. Each generation step requires a full forward pass through all transformer layers, but processes only a single new token. In the figure, this corresponds to the subsequent forward passes that generate "the", "future", and so on. This phase dominates the total inference time for longer outputs, as the number of forward passes equals the number of generated tokens.

2.1.1 Arithmetic Intensity Analysis

To understand why LLM inference is challenging, we must examine its *arithmetic intensity*—the ratio of floating-point operations (FLOPs) to bytes transferred from memory. This metric determines whether a workload is *compute-bound* (limited by the GPU's computational throughput) or *memory-bound* (limited by memory bandwidth).

Modern GPUs exhibit a significant imbalance between compute capability and memory bandwidth. For example, an NVIDIA H100 GPU provides approximately 1,000 TFLOPS of BF16 compute throughput but only 3.35 TB/s of memory bandwidth [67]. The ratio of these quantities—roughly 300 FLOPs per byte—defines the *roofline* threshold: workloads with arithmetic intensity below this threshold are memory-bound, while those above are compute-bound.

Consider a simple matrix-vector multiplication $y = Wx$, where $W \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$. This operation performs $2mn$ FLOPs (one multiply and one add per element of W) while transferring $2mn$ bytes to read the weight matrix (assuming BF16 precision), plus negligible overhead for x and y. The arithmetic intensity is therefore approximately 1 FLOP/byte—far below the roofline threshold. Matrix-vector multiplication is severely memory-bound.

Now consider matrix-matrix multiplication $Y = WX$, where $X \in \mathbb{R}^{n \times b}$ represents a batch of b vectors. This operation performs $2mnb$ FLOPs while transferring $2mn$ bytes for W (which is read once and reused across all b vectors). The arithmetic intensity becomes approximately b FLOPs/byte. With sufficient batch size b (e.g., ≥ 300), matrix-matrix multiplication becomes compute-bound.

This analysis reveals the core challenge of LLM inference:

- **Prefill** processes many tokens simultaneously, enabling batched matrix multiplications with high arithmetic intensity. The prefill phase can achieve good GPU utilization.
- **Decode** processes only one token per request per step. As shown in Figure 2.1, each forward pass during decode handles just a single token traversing through all layers. Even with multiple requests batched together, the effective batch size is often insufficient to saturate GPU compute. The decode phase is typically memory-bound, with the GPU spending most of its time waiting for weight transfers rather than performing computation.

To quantify this, consider a 8-billion parameter model. Storing the weights in BF16 requires 16 GB of memory. At 3.35 TB/s bandwidth, simply reading all weights takes approximately 5 milliseconds. If we are decoding a single request, this memory transfer time dominates the forward pass latency, regardless of the GPU's computational capability. To achieve compute-bound operation during decode, we would need to batch hundreds of requests together—a requirement that may conflict with latency constraints or exceed available memory for storing per-request state.

2.1.2 KV Cache

The attention mechanism in Transformers [97] computes:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V are the query, key, and value matrices derived from the input sequence. Critically, generating each new token requires attending to *all* previous tokens in the sequence. The key and value vectors for previous tokens must therefore be available at each generation step.

A naive implementation would recompute K and V for all previous tokens at every forward pass during the decode phase. For a sequence of length L, this would require $O(L)$ computation per step, making the total generation cost $O(L^2)$—prohibitively expensive for long sequences.

The *KV cache* addresses this inefficiency by storing the key and value vectors computed during previous forward passes. At each generation step, only the key and value vectors for the new token are computed; the cached values for previous tokens are retrieved from memory. This reduces the per-step computation from $O(L)$ to $O(1)$ for the key-value projection, making total generation cost $O(L)$ rather than $O(L^2)$.

However, the KV cache introduces its own challenges:

Memory consumption. The KV cache size grows linearly with sequence length and batch size. For each token, the cache stores key and value vectors across all layers and attention heads. For a model with L layers, H attention key-value heads per layer, and head dimension d, each token requires:

$$\text{KV cache per token} = 2 \times L \times H \times d \times \text{sizeof(dtype)}$$

bytes, where the factor of 2 accounts for both keys and values.

For a model with 100 layers, 8 key-value heads, and 128 head dimension, each token requires approximately 400 KB of KV cache in BF16. A single request with a 100,000-token context consumes 40 GB of KV cache. Even with shorter contexts, the KV cache often dominates GPU memory usage, limiting the number of requests that can be served concurrently.

Memory bandwidth pressure. During decode, the attention operation must read the entire KV cache for all previous tokens. For long sequences, this memory transfer can become a significant bottleneck, adding to the memory-bound nature of the decode phase. For the previous 100-layer model with 100,000-token context, it takes 12 milliseconds to just read the single request's KV cache.

Memory fragmentation. Different requests have different sequence lengths, and these lengths change dynamically as generation proceeds. Pre-allocating memory for the maximum possible sequence length wastes memory when sequences are shorter. Dynamic allocation can lead to fragmentation, where free memory exists but cannot be used because it is not contiguous. Chapter 3 addresses this challenge through PagedAttention, which manages KV cache memory in fixed-size blocks.

The tension between KV cache memory requirements and the desire for large batch sizes creates a fundamental tradeoff in LLM inference system design. Larger batches improve arithmetic intensity and GPU utilization, but require more memory for KV caches. Efficient memory management for KV caches is therefore critical for achieving high throughput.

2.2 Mixture-of-Experts Models

Mixture-of-Experts (MoE) architectures [86, 47] have emerged as a powerful approach for scaling model capacity without proportionally increasing computational cost. In an MoE model, each token is processed by only a subset of *expert* networks, selected by a learned routing mechanism. This *sparse activation* allows models to have far more parameters than they use for any single token.

For example, a model might have 256 experts per MoE layer but route each token to only 8 experts. This provides a $32\times$ reduction in per-token computation compared to a dense model with equivalent total parameters. Models like DeepSeek V3 [56] and Qwen3 [107] leverage this approach to achieve strong performance with reduced inference cost.

However, MoE architectures introduce new challenges for inference efficiency:

Increased memory footprint. While MoE models reduce computation per token, they do not reduce memory requirements. All expert weights must be stored in GPU memory, even though only a fraction are used for any given token. A model with $32\times$ sparsity has $32\times$ more parameters than a dense model with equivalent per-token computation, requiring $32\times$ more memory for weights.

Amplified batch size requirements. The memory-bound nature of decode is exacerbated in MoE models. Consider the arithmetic intensity analysis from Section 2.1.1. For a dense model, processing a batch of b tokens achieves arithmetic intensity proportional to b. For an MoE model with E experts where each token uses k experts, the tokens in the batch are distributed across experts according to the routing decisions. If routing is approximately uniform, each expert receives only $\frac{b \cdot k}{E}$ tokens on average.

The effective batch size *per expert* is therefore reduced by a factor of $\frac{E}{k}$ compared to a dense model. For a model with 256 experts and top-8 routing ($E = 256$, $k = 8$), this factor is $32\times$. To achieve the same arithmetic intensity as a dense model with batch size b, an MoE model requires batch size $32b$.

This amplification has significant implications: an MoE model that matches a dense model's per-token computation may **require $32\times$ larger batches to achieve comparable GPU utilization**. Meeting this requirement while respecting memory constraints for KV caches and expert weights is challenging.

2.3 Latency Constraints

Real-world LLM applications impose strict latency requirements that constrain system design. Users expect responsive interactions, and many applications have hard deadlines for generating outputs. Understanding these constraints is essential for building practical inference systems.

2.3.1 Time-to-First-Token

Time-to-first-token (TTFT) measures the latency from when a request is submitted until the first output token is generated. This metric captures the responsiveness of the system from the user's perspective—how long they must wait before seeing any output.

TTFT is dominated by the prefill phase, which must process the entire input prompt before generation can begin. For long prompts, prefill can take hundreds of milliseconds or even seconds, creating noticeable delays.

Several factors influence TTFT:

- **Prompt length**: Longer prompts require more computation during prefill. TTFT scales roughly linearly with prompt length for compute-bound prefill, or super-linearly when attention computation becomes the bottleneck.
- **Queuing delay**: When the system is under load, requests may wait in a queue before being scheduled. This queuing delay adds directly to TTFT.
- **Batching interference**: If prefill computation is batched with ongoing decode operations, the prefill may take longer due to resource contention.

Applications have varying TTFT requirements. Interactive chat applications typically require TTFT under 1-2 seconds to feel responsive. Real-time applications like voice assistants may require TTFT under 500 milliseconds. Batch processing applications may tolerate much longer TTFT if overall throughput is prioritized.

2.3.2 Time-per-Output-Token

Time-per-output-token (TPOT), also called *inter-token latency* (ITL), measures the time between consecutive output tokens during the decode phase. This metric determines the perceived speed of text generation once output begins.

TPOT is determined by the decode phase latency, which depends on:

- **Batch size**: Larger batches amortize fixed overheads but increase per-step latency. The optimal batch size balances throughput against TPOT requirements.
- **Sequence length**: Longer sequences require reading larger KV caches during attention, increasing per-token latency.
- **System overheads**: Scheduling, memory management, and other system operations add to per-step latency.

Human reading speed provides a natural reference point for TPOT requirements. Humans read approximately 250 words per minute [15], or roughly 4 words per second. At approximately 1.3 tokens per word, this corresponds to about 5 tokens per second, or 200 milliseconds per token. For text that will be read in real-time, TPOT below this threshold provides output faster than users can consume it.

However, many applications have stricter requirements. Streaming speech synthesis requires consistent, low-latency token generation to avoid audio stuttering. Agentic applications that chain multiple LLM calls benefit from fast generation to reduce end-to-end latency. Code completion in IDEs must generate suggestions quickly enough to feel instantaneous.

The tension between throughput and latency is fundamental. Maximizing throughput favors large batches that improve GPU utilization. Minimizing latency favors smaller batches that reduce per-step time. Practical systems must navigate this tradeoff based on application requirements.

2.4 Trends

Several trends are shaping the landscape of LLM inference and intensifying the challenges described above:

Increasing model scale. State-of-the-art models continue to grow in size. While scaling laws [44] suggest diminishing returns at extreme scales, models with hundreds of billions or even trillions of parameters remain competitive for many tasks. Larger models require more memory, more computation, and more communication in distribution settings.

Longer context lengths. Applications increasingly demand longer context windows. Retrieval-augmented generation [50], document analysis, and multi-turn conversations all benefit from longer contexts. Models now support context lengths of 128,000 tokens or more [75, 7], with some models handling a million of tokens [32]. Longer contexts dramatically increase KV cache memory requirements and attention computation costs.

Multimodal inputs. Modern LLMs increasingly process images [57], audio [40], and video [55] alongside text. These modalities generate large numbers of tokens—a single image may produce thousands of tokens—amplifying prefill costs and KV cache requirements. Video inputs, which comprise many frames, can generate tens or hundreds of thousands of tokens.

Agentic workloads. LLM-powered agents that interact with tools, browse the web, or execute code are becoming prevalent [108, 83]. These workloads involve many sequential LLM calls with complex dependencies, making end-to-end latency critical. They also exhibit variable request patterns that challenge traditional batching strategies.

Hardware evolution. GPU capabilities continue to improve, with each generation providing more compute throughput, more memory bandwidth, and larger memory capacity. However, compute capability is growing faster than memory bandwidth, widening the gap that makes decode memory-bound. This trend increases the importance of techniques that improve arithmetic intensity and reduce memory access.

Efficiency pressure. The cost of LLM inference at scale is substantial. Large-scale deployments may require thousands of GPUs, consuming megawatts of power and millions of dollars annually. This cost pressure drives demand for more efficient inference, creating strong incentives for the optimizations described in subsequent chapters.

These trends collectively raise the stakes for efficient LLM inference. The challenges described in this chapter are not static—they are intensifying as models grow larger, contexts grow longer, and applications grow more demanding. Meeting these challenges requires innovations across the entire inference stack, from memory management to scheduling to kernel optimization. The following chapters describe such innovations, beginning with PagedAttention for efficient KV cache management in Chapter 3.

Chapter 3

PagedAttention: Efficient Memory Management for LLM Inference

3.1 Introduction

The emergence of large language models (*LLMs*) like GPT [74, 14] and PaLM [19] have enabled new applications such as programming assistants [30, 16] and universal chatbots [72, 31] that are starting to profoundly impact our work and daily routines. Many cloud companies [71, 84] are racing to provide these applications as hosted services. However, running these applications is very expensive, requiring a large number of hardware accelerators such as GPUs. According to recent estimates, processing an LLM request can be 10× more expensive than a traditional keyword query [82]. Given these high costs, increasing the throughput—and hence reducing the cost per request—of *LLM serving* systems is becoming more important.

At the core of LLMs lies an autoregressive Transformer model [97]. This model generates words (tokens), *one at a time*, based on the input (prompt) and the previous sequence of the output's tokens it has generated so far. For each request, this expensive process is repeated until the model outputs a termination token. This sequential generation process makes the workload *memory-bound*, underutilizing the computation power of GPUs and limiting the serving throughput.

Improving the throughput is possible by batching multiple requests together. However, to process many requests in a batch, the memory space for each request should be efficiently managed. For example, Fig. 3.1 (left) illustrates the memory distribution for a 13B-parameter LLM on an NVIDIA A100 GPU with 40GB RAM. Approximately 65% of the memory is allocated for the model weights, which remain static during serving. Close to 30% of the memory is used to store the dynamic states of the requests. For Transformers, these states consist of the key and value tensors associated with the attention mechanism, commonly referred to as *KV cache* [80], which represent the context from earlier tokens to generate new output tokens in sequence. The remaining small percentage of memory is used for other data, including activations – the ephemeral tensors created when evaluating the LLM. Since the model weights are constant and the activations only occupy a small fraction of the GPU memory, the way the KV cache is managed is critical in determining the

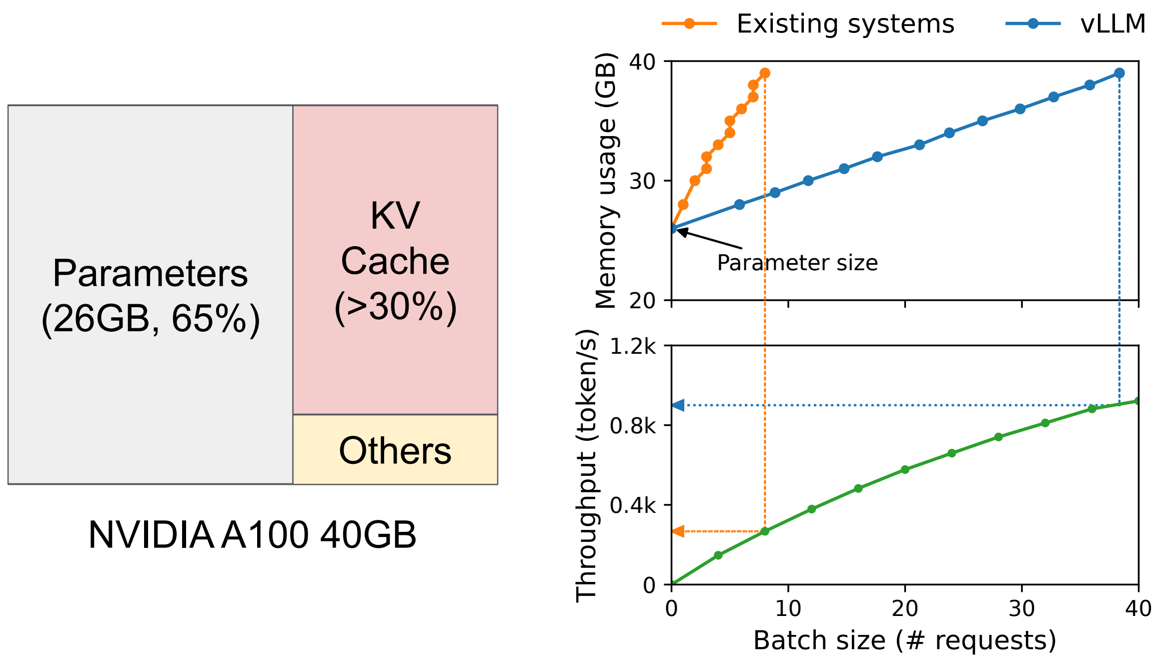


Figure 3.1: *Left:* Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemerally for activation. *Right:* vLLM smooths out the rapid growth curve of KV cache memory seen in existing systems [63, 110], leading to a notable boost in serving throughput.

maximum batch size. When managed inefficiently, the KV cache memory can significantly limit the batch size and consequently the throughput of the LLM, as illustrated in Fig. 3.1 (right).

In this paper, we observe that existing LLM serving systems [110, 63] fall short of managing the KV cache memory efficiently. This is mainly because they store the KV cache of a request in contiguous memory space, as most deep learning frameworks [78, 70] require tensors to be stored in contiguous memory. However, unlike the tensors in the traditional deep learning workloads, the KV cache has unique characteristics: it dynamically grows and shrinks over time as the model generates new tokens, and its lifetime and length are not known a priori. These characteristics make the existing systems' approach significantly inefficient in two ways:

First, the existing systems [110, 63] suffer from internal and external memory fragmentation. To store the KV cache of a request in contiguous space, they *pre-allocate* a contiguous chunk of memory with the request's maximum length (e.g., 2048 tokens). This can result in

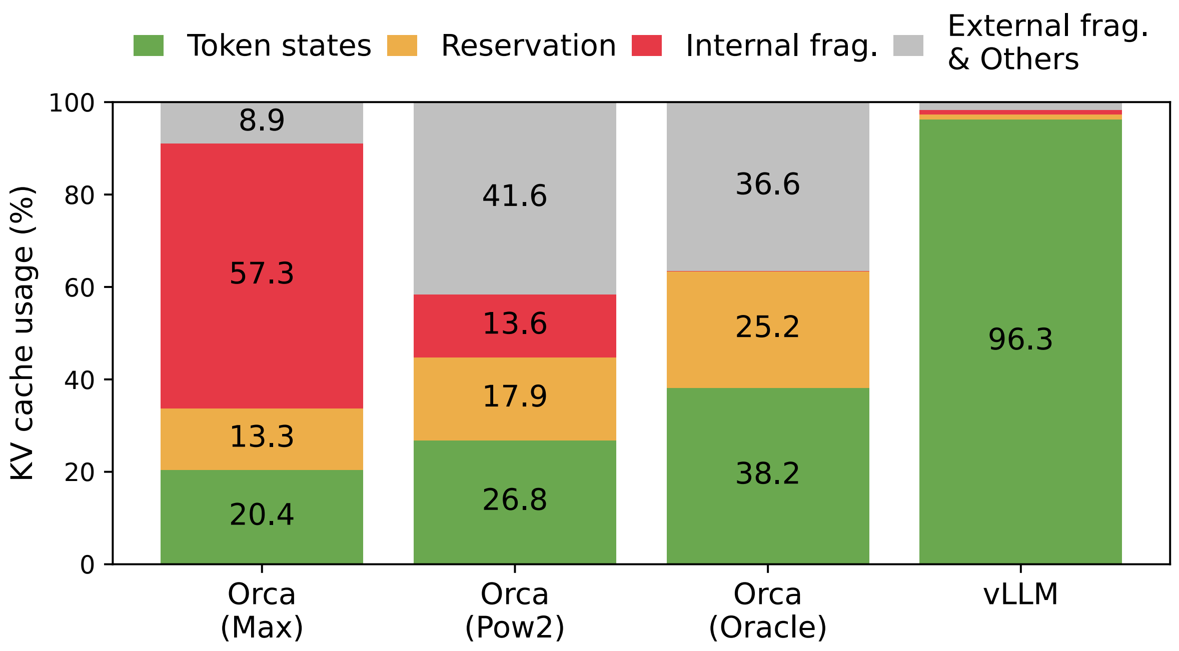


Figure 3.2: Average percentage of memory wastes in different LLM serving systems during the experiment in §3.6.2.

severe internal fragmentation, since the request's actual length can be much shorter than its maximum length (e.g., Fig. 3.11). Moreover, even if the actual length is known a priori, the pre-allocation is still inefficient: As the entire chunk is reserved during the request's lifetime, other shorter requests cannot utilize any part of the chunk that is currently unused. Besides, external memory fragmentation can also be significant, since the pre-allocated size can be different for each request. Indeed, our profiling results in Fig. 3.2 show that only 20.4% - 38.2% of the KV cache memory is used to store the actual token states in the existing systems.

Second, the existing systems cannot exploit the opportunities for memory sharing. LLM services often use advanced decoding algorithms, such as parallel sampling and beam search, that generate multiple outputs per request. In these scenarios, the request consists of multiple sequences that can partially share their KV cache. However, memory sharing is not possible in the existing systems because the KV cache of the sequences is stored in separate contiguous spaces.

To address the above limitations, we propose *PagedAttention*, an attention algorithm inspired by the operating system's (OS) solution to memory fragmentation and sharing: *virtual memory with paging*. PagedAttention divides the request's KV cache into blocks, each of which can contain the attention keys and values of a fixed number of tokens. In

PagedAttention, the blocks for the KV cache are not necessarily stored in contiguous space. Therefore, we can manage the KV cache in a more flexible way as in OS's virtual memory: one can think of blocks as pages, tokens as bytes, and requests as processes. This design alleviates internal fragmentation by using relatively small blocks and allocating them on demand. Moreover, it eliminates external fragmentation as all blocks have the same size. Finally, it enables memory sharing at the granularity of a block, across the different sequences associated with the same request or even across the different requests.

In this work, we build *vLLM*, a high-throughput distributed LLM serving engine on top of PagedAttention that achieves near-zero waste in KV cache memory. vLLM uses block-level memory management and preemptive request scheduling that are co-designed with PagedAttention. vLLM supports popular LLMs such as GPT [14], OPT [113], and LLaMA [96] with varying sizes, including the ones exceeding the memory capacity of a single GPU. Our evaluations on various models and workloads show that vLLM improves the LLM serving throughput by 2-4× compared to the state-of-the-art systems [110, 63], without affecting the model accuracy at all. The improvements are more pronounced with longer sequences, larger models, and more complex decoding algorithms (§3.4.3). In summary, we make the following contributions:

- We identify the challenges in memory allocation in serving LLMs and quantify their impact on serving performance.
- We propose PagedAttention, an attention algorithm that operates on KV cache stored in non-contiguous paged memory, which is inspired by the virtual memory and paging in OS.
- We design and implement vLLM, a distributed LLM serving engine built on top of PagedAttention.
- We evaluate vLLM on various scenarios and demonstrate that it substantially outperforms the previous state-of-the-art solutions such as FasterTransformer [63] and Orca [110].

3.2 Background

In this section, we describe the generation and serving procedures of typical LLMs and the iteration-level scheduling used in LLM serving.

3.2.1 Transformer-Based Large Language Models

The task of language modeling is to model the probability of a list of tokens $(x_1, \ldots, x_n)$. Since language has a natural sequential ordering, it is common to factorize the joint probability over the whole sequence as the product of conditional probabilities (a.k.a. *autoregressive decomposition* [11]):

$$P(x) = P(x_1) \cdot P(x_2 \mid x_1) \cdots P(x_n \mid x_1, \ldots, x_{n-1}). \tag{3.1}$$

Transformers [97] have become the de facto standard architecture for modeling the probability above at a large scale. The most important component of a Transformer-based language model is its *self-attention* layers. For an input hidden state sequence $(x_1, \ldots, x_n) \in \mathbb{R}^{n \times d}$, a self-attention layer first applies linear transformations on each position i to get the query, key, and value vectors:

$$q_i = W_q x_i, \; k_i = W_k x_i, \; v_i = W_v x_i. \tag{3.2}$$

Then, the self-attention layer computes the attention score a_{ij} by multiplying the query vector at one position with all the key vectors before it and compute the output o_i as the weighted average over the value vectors:

$$a_{ij} = \frac{\exp(q_i^\top k_j / \sqrt{d})}{\sum_{t=1}^{i} \exp(q_i^\top k_t / \sqrt{d})}, \; o_i = \sum_{j=1}^{i} a_{ij} v_j. \tag{3.3}$$

Besides the computation in Eq. 3.3, all other components in the Transformer model, including the embedding layer, feed-forward layer, layer normalization [8], residual connection [35], output logit computation, and the query, key, and value transformation in Eq. 3.2, are all applied independently position-wise in a form of $y_i = f(x_i)$.

3.2.2 LLM Service & Autoregressive Generation

Once trained, LLMs are often deployed as a conditional generation service (e.g., completion API [71] or chatbot [72, 31]). A request to an LLM service provides a list of *input prompt* tokens $(x_1, \ldots, x_n)$, and the LLM service generates a list of output tokens $(x_{n+1}, \ldots, x_{n+T})$ according to Eq. 3.1. We refer to the concatenation of the prompt and output lists as *sequence*.

Due to the decomposition in Eq. 3.1, the LLM can only sample and generate new tokens one by one, and the generation process of each new token depends on all the *previous tokens* in that sequence, specifically their key and value vectors. In this sequential generation process, the key and value vectors of existing tokens are often cached for generating future tokens, known as *KV cache*. Note that the KV cache of one token depends on all its previous tokens. This means that the KV cache of the same token appearing at different positions in a sequence will be different.

Given a request prompt, the generation computation in the LLM service can be decomposed into two phases:

The prompt phase takes the whole user prompt $(x_1, \ldots, x_n)$ as input and computes the probability of the first new token $P(x_{n+1} \mid x_1, \ldots, x_n)$. During this process, also generates the key vectors $k_1, \ldots, k_n$ and value vectors $v_1, \ldots, v_n$. Since prompt tokens $x_1, \ldots, x_n$ are all known, the computation of the prompt phase can be parallelized using matrix-matrix multiplication operations. Therefore, this phase can efficiently use the parallelism inherent in GPUs.

The autoregressive generation phase generates the remaining new tokens sequentially. At iteration t, the model takes one token x_{n+t} as input and computes the probability $P(x_{n+t+1} \mid x_1, \ldots, x_{n+t})$ with the key vectors $k_1, \ldots, k_{n+t}$ and value vectors $v_1, \ldots, v_{n+t}$. Note that the key and value vectors at positions 1 to $n + t - 1$ are cached at previous iterations, only the new key and value vector k_{n+t} and v_{n+t} are computed at this iteration. This phase completes either when the sequence reaches a maximum length (specified by users or limited by LLMs) or when an end-of-sequence (*eos*) token is emitted. The computation at different iterations cannot be parallelized due to the data dependency and often uses matrix-vector multiplication, which is less efficient. As a result, this phase severely underutilizes GPU computation and becomes memory-bound, being responsible for most portion of the latency of a single request.

3.2.3 Batching Techniques for LLMs

The compute utilization in serving LLMs can be improved by batching multiple requests. Because the requests share the same model weights, the overhead of moving weights is amortized across the requests in a batch, and can be overwhelmed by the computational overhead when the batch size is sufficiently large. However, batching the requests to an LLM service is non-trivial for two reasons. First, the requests may arrive at different times. A naive batching strategy would either make earlier requests wait for later ones or delay the incoming requests until earlier ones finish, leading to significant queueing delays. Second, the requests may have vastly different input and output lengths (Fig. 3.11). A straightforward batching technique would pad the inputs and outputs of the requests to equalize their lengths, wasting GPU computation and memory.

To address this problem, fine-grained batching mechanisms, such as cellular batching [28] and iteration-level scheduling [110], have been proposed. Unlike traditional methods that work at the request level, these techniques operate at the iteration level. After each iteration, completed requests are removed from the batch, and new ones are added. Therefore, a new request can be processed after waiting for a single iteration, not waiting for the entire batch to complete. Moreover, with special GPU kernels, these techniques eliminate the need to pad the inputs and outputs. By reducing the queueing delay and the inefficiencies from padding, the fine-grained batching mechanisms significantly increase the throughput of LLM serving.

3.3 Memory Challenges in LLM Serving

Although fine-grained batching reduces the waste of computing and enables requests to be batched in a more flexible way, the number of requests that can be batched together is still constrained by GPU memory capacity, particularly the space allocated to store the KV cache. In other words, the serving system's throughput is *memory-bound*. Overcoming this memory-bound requires addressing the following challenges in the memory management:

Large KV cache. The KV Cache size grows quickly with the number of requests. As an example, for the 13B parameter OPT model [113], the KV cache of a single token demands 800 KB of space, calculated as 2 (key and value vectors) $\times$ 5120 (hidden state size) $\times$ 40 (number of layers) $\times$ 2 (bytes per FP16). Since OPT can generate sequences up to 2048 tokens, the memory required to store the KV cache of one request can be as much as 1.6 GB. Concurrent GPUs have memory capacities in the tens of GBs. Even if all available memory was allocated to KV cache, only a few tens of requests could be accommodated. Moreover, inefficient memory management can further decrease the batch size, as shown in Fig. 3.2. Additionally, given the current trends, the GPU's computation speed grows faster than the memory capacity [29]. For example, from NVIDIA A100 to H100, The FLOPS increases by more than 2x, but the GPU memory stays at 80GB maximum. Therefore, we believe the memory will become an increasingly significant bottleneck.

Complex decoding algorithms. LLM services offer a range of decoding algorithms for users to select from, each with varying implications for memory management complexity. For example, when users request multiple random samples from a single input prompt, a typical use case in program suggestion [30], the KV cache of the prompt part, which accounts for 12% of the total KV cache memory in our experiment (§3.6.3), can be shared to minimize memory usage. On the other hand, the KV cache during the autoregressive generation phase should remain unshared due to the different sample results and their dependence on context and position. The extent of KV cache sharing depends on the specific decoding algorithm employed. In more sophisticated algorithms like beam search [91], different request beams can share larger portions (up to 55% memory saving, see §3.6.3) of their KV cache, and the sharing pattern evolves as the decoding process advances.

Scheduling for unknown input & output lengths. The requests to an LLM service exhibit variability in their input and output lengths. This requires the memory management system to accommodate a wide range of prompt lengths. In addition, as the output length of a request grows at decoding, the memory required for its KV cache also expands and may exhaust available memory for incoming requests or ongoing generation for existing prompts. The system needs to make scheduling decisions, such as deleting or swapping out the KV cache of some requests from GPU memory.

3.3.1 Memory Management in Existing Systems

Since most operators in current deep learning frameworks [78, 70] require tensors to be stored in contiguous memory, previous LLM serving systems [110, 63] also store the KV cache of one request as a contiguous tensor across the different positions. Due to the unpredictable output lengths from the LLM, they statically allocate a chunk of memory for a request based on the request's maximum possible sequence length, irrespective of the actual input or eventual output length of the request.

Fig. 3.3 illustrates two requests: request A with 2048 maximum possible sequence length and request B with a maximum of 512. The chunk pre-allocation scheme in existing systems has three primary sources of memory wastes:

- *reserved* slots for future tokens,
- *internal fragmentation* due to over-provisioning for potential maximum sequence lengths,
- *external fragmentation* from the memory allocator like the buddy allocator.

The external fragmentation will never be used for generated tokens, which is known before serving a request. Internal fragmentation also remains unused, but this is only realized after

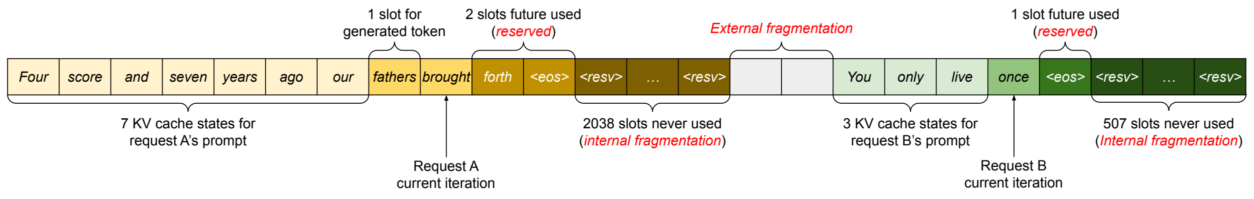

1 slot for generated token
2 slots future used (reserved)
External fragmentation
1 slot future used (reserved)
Four
score
and
seven
years
ago
our
fathers
brought
forth
<eos>
<resv>
…
<resv>
You
only
live
once
<eos>
<resv>
…
<resv>
7 KV cache states for request A's prompt
Request A current iteration
2038 slots never used (internal fragmentation)
3 KV cache states for request B's prompt
Request B current iteration
507 slots never used (Internal fragmentation)

Figure 3.3: KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

a request has finished sampling. They are both pure memory waste. Although the reserved memory is eventually used, reserving this space for the entire request's duration, especially when the reserved space is large, occupies the space that could otherwise be used to process other requests. We visualize the average percentage of memory wastes in our experiments in Fig. 3.2, revealing that the actual effective memory in previous systems can be as low as 20.4%. Notably, even when the request's output length is known perfectly a priori, as in Orca (Oracle), the reserved space still results in significant waste.

Although compaction [98] has been proposed as a potential solution to fragmentation, performing compaction in a performance-sensitive LLM serving system is impractical due to the massive KV cache. Even with compaction, the pre-allocated chunk space for each request prevents memory sharing specific to decoding algorithms in existing memory management systems.

3.4 Method

In this work, we develop a new attention algorithm, *PagedAttention*, and build an LLM serving engine, *vLLM*, to tackle the challenges outlined in §3.3. The architecture of vLLM is shown in Fig. 3.4. vLLM adopts a centralized scheduler to coordinate the execution of distributed GPU workers. The *KV cache manager* effectively manages the KV cache in a paged fashion, enabled by PagedAttention. Specifically, the KV cache manager manages the physical KV cache memory on the GPU workers through the instructions sent by the centralized scheduler.

Next, We describe the PagedAttention algorithm in §3.4.1. With that, we show the design of the KV cache manager in §3.4.2 and how it facilitates PagedAttention in §3.4.3, respectively. Then, we show how this design facilitates effective memory management for various decoding methods (§3.4.4) and handles the variable length input and output sequences (§3.4.5). Finally, we show how the system design of vLLM works in a distributed setting (§3.4.6).

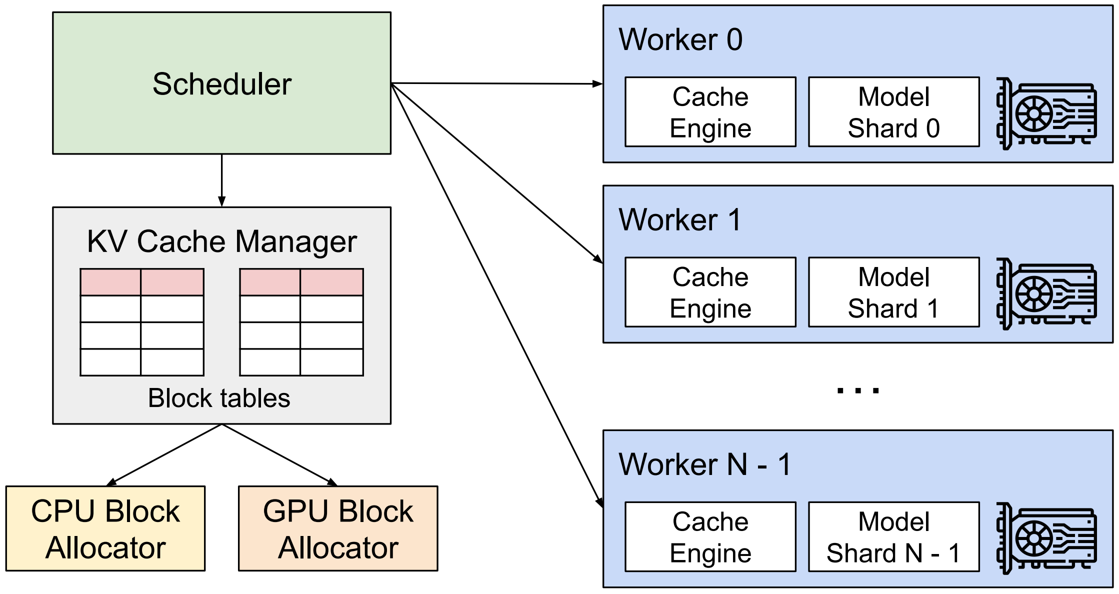


Figure 3.4: vLLM system overview.

3.4.1 PagedAttention

To address the memory challenges in §3.3, we introduce *PagedAttention*, an attention algorithm inspired by the classic idea of *paging* [45] in operating systems. Unlike the traditional attention algorithms, PagedAttention allows storing continuous keys and values in non-contiguous memory space. Specifically, PagedAttention partitions the KV cache of each sequence into *KV blocks*. Each block contains the key and value vectors for a fixed number of tokens,[1] which we denote as *KV block size* (B). Denote the key block $K_j = (k_{(j-1)B+1}, \ldots, k_{jB})$ and value block $V_j = (v_{(j-1)B+1}, \ldots, v_{jB})$. The attention computation in Eq. 3.3 can be transformed into the following block-wise computation:

$$A_{ij} = \frac{\exp(q_i^\top K_j/\sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t/\sqrt{d}) \cdot \mathbf{1}}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top, \tag{3.4}$$

where $A_{ij} = (a_{i,(j-1)B+1}, \ldots, a_{i,jB})$ is the row vector of attention score on j-th KV block.

During the attention computation, the PagedAttention kernel identifies and fetches different KV blocks separately. We show an example of PagedAttention in Fig. 3.5: The key and value vectors are spread across three blocks, and the three blocks are not contiguous on the physical memory. At each time, the kernel multiplies the query vector q_i of the query

[1]In Transformer, each token has a set of key and value vectors across layers and attention heads within a layer. All the key and value vectors can be managed together within a single KV block, or the key and value vectors at different heads and layers can each have a separate block and be managed in separate block tables. The two designs have no performance difference and we choose the second one for easy implementation.

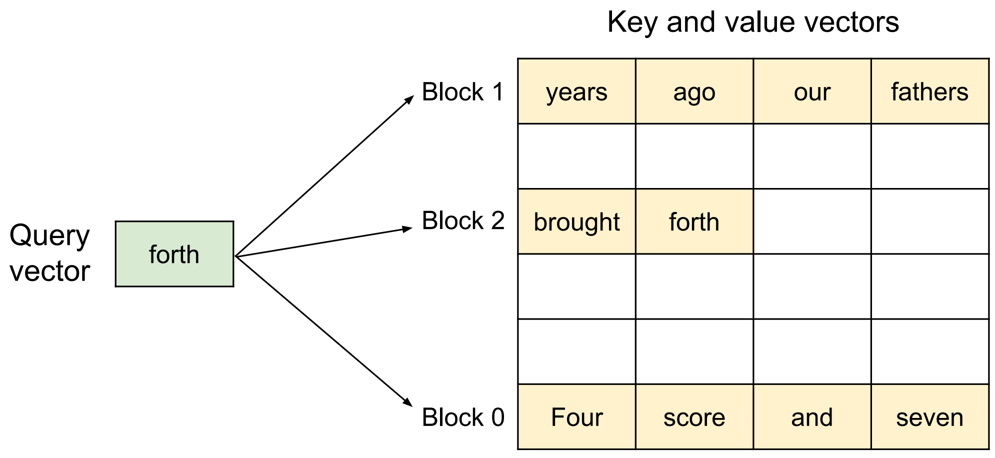


Figure 3.5: Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

token ("*forth*") and the key vectors K_j in a block (e.g., key vectors of "*Four score and seven*" for block 0) to compute the attention score A_{ij}, and later multiplies A_{ij} with the value vectors V_j in a block to derive the final attention output o_i.

In summary, the PagedAttention algorithm allows the KV blocks to be stored in non-contiguous physical memory, which enables more flexible paged memory management in vLLM.

3.4.2 KV Cache Manager

The key idea behind vLLM's memory manager is analogous to the *virtual memory* [45] in operating systems. OS partitions memory into fixed-sized *pages* and maps user programs' logical pages to physical pages. Contiguous logical pages can correspond to non-contiguous physical memory pages, allowing user programs to access memory as though it were contiguous. Moreover, physical memory space needs not to be fully reserved in advance, enabling the OS to dynamically allocate physical pages as needed. vLLM uses the ideas behind virtual memory to manage the KV cache in an LLM service. Enabled by PagedAttention, we organize the KV cache as fixed-size KV blocks, like pages in virtual memory.

A request's KV cache is represented as a series of *logical KV blocks*, filled from left to right as new tokens and their KV cache are generated. The last KV block's unfilled positions are reserved for future generations. On GPU workers, a *block engine* allocates a contiguous chunk of GPU DRAM and divides it into *physical KV blocks* (this is also done on CPU RAM for swapping; see §3.4.5). The *KV block manager* also maintains *block tables*—the mapping between logical and physical KV blocks of each request. Each block table entry records the corresponding physical blocks of a logical block and the number of filled positions. Separating logical and physical KV blocks allows vLLM to dynamically grow the KV cache memory without reserving it for all positions in advance, which eliminates most memory

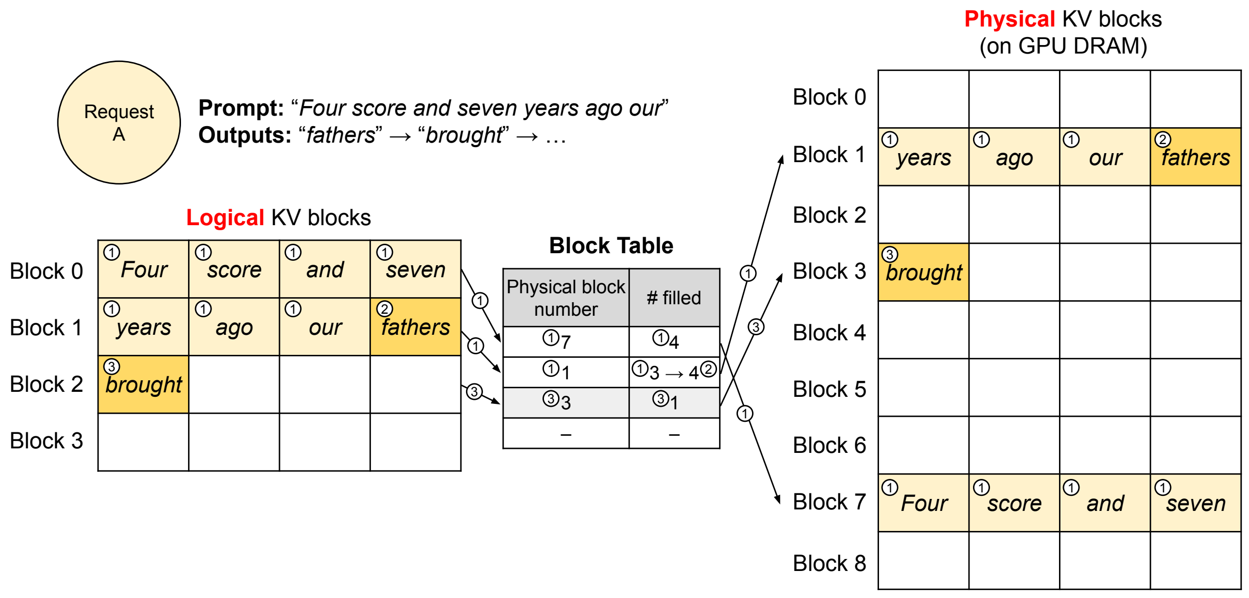

Request A
Prompt: "Four score and seven years ago our"
Outputs: "fathers" → "brought" → …
Logical KV blocks
Block 0: ① Four, ① score, ① and, ① seven
Block 1: ① years, ① ago, ① our, ② fathers
Block 2: ③ brought
Block 3
Block Table
Physical block number | # filled
①7 | ①4
①1 | ①3 → 4②
③3 | ③1
– | –
Physical KV blocks (on GPU DRAM)
Block 0
Block 1: ① years, ① ago, ① our, ② fathers
Block 2
Block 3: ③ brought
Block 4
Block 5
Block 6
Block 7: ① Four, ① score, ① and, ① seven
Block 8

Figure 3.6: Block table translation in vLLM.

waste in existing systems, as in Fig. 3.2.

3.4.3 Decoding with PagedAttention and vLLM

Next, we walk through an example, as in Fig. 3.6, to demonstrate how vLLM executes PagedAttention and manages the memory during the decoding process of a single input sequence: ① As in OS's virtual memory, vLLM does not require reserving the memory for the maximum possible generated sequence length initially. Instead, it reserves only the necessary KV blocks to accommodate the KV cache generated during prompt computation. In this case, The prompt has 7 tokens, so vLLM maps the first 2 logical KV blocks (0 and 1) to 2 physical KV blocks (7 and 1, respectively). In the prefill step, vLLM generates the KV cache of the prompts and the first output token with a conventional self-attention algorithm (e.g., [23]). vLLM then stores the KV cache of the first 4 tokens in logical block 0 and the following 3 tokens in logical block 1. The remaining slot is reserved for the subsequent autoregressive generation phase. ② In the first autoregressive decoding step, vLLM generates the new token with the PagedAttention algorithm on physical blocks 7 and 1. Since one slot remains available in the last logical block, the newly generated KV cache is stored there, and the block table's #filled record is updated. ③ At the second decoding step, as the last logical block is full, vLLM stores the newly generated KV cache in a new logical block; vLLM allocates a new physical block (physical block 3) for it and stores this mapping in the block table.

Globally, for each decoding iteration, vLLM first selects a set of candidate sequences for batching (more in §3.4.5), and allocates the physical blocks for the newly required logical blocks. Then, vLLM concatenates all the input tokens of the current iteration (i.e., all to-

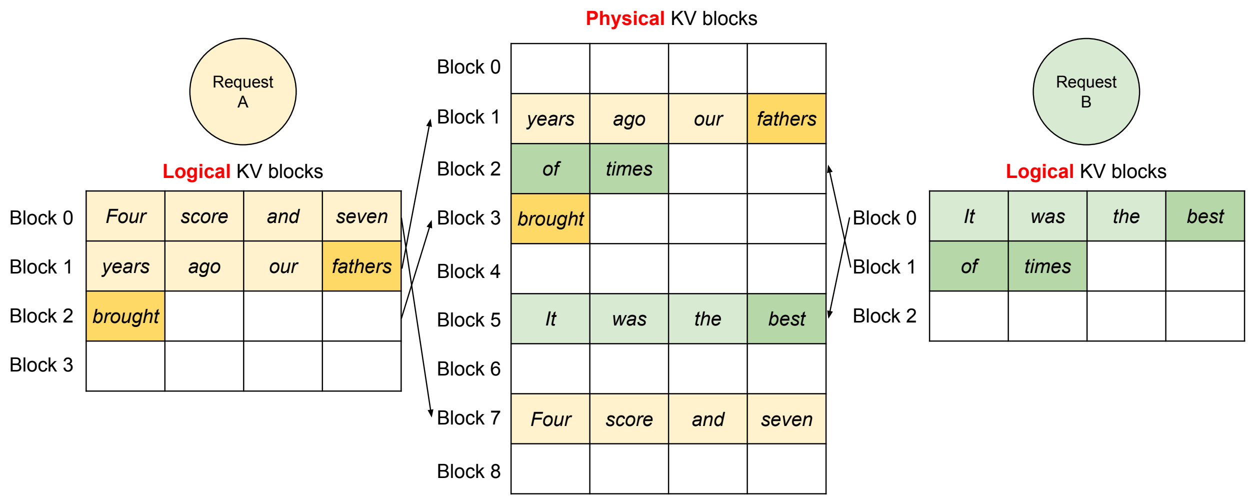


Figure 3.7: Storing the KV cache of two requests at the same time in vLLM.

kens for prompt phase requests and the latest tokens for generation phase requests) as one sequence and feeds it into the LLM. During LLM's computation, vLLM uses the PagedAttention kernel to access the previous KV cache stored in the form of logical KV blocks and saves the newly generated KV cache into the physical KV blocks. Storing multiple tokens within a KV block (block size ¿ 1) enables the PagedAttention kernel to process the KV cache across more positions in parallel, thus increasing the hardware utilization and reducing latency. However, a larger block size also increases memory fragmentation. We study the effect of block size in §3.7.2.

Again, vLLM dynamically assigns new physical blocks to logical blocks as more tokens and their KV cache are generated. As all the blocks are filled from left to right and a new physical block is only allocated when all previous blocks are full, vLLM limits all the memory wastes for a request within one block, so it can effectively utilize all the memory, as shown in Fig. 3.2. This allows more requests to fit into memory for batching—hence improving the throughput. Once a request finishes its generation, its KV blocks can be freed to store the KV cache of other requests. In Fig. 3.7, we show an example of vLLM managing the memory for two sequences. The logical blocks of the two sequences are mapped to different physical blocks within the space reserved by the block engine in GPU workers. The neighboring logical blocks of both sequences do not need to be contiguous in physical GPU memory and the space of physical blocks can be effectively utilized by both sequences.

3.4.4 Application to Other Decoding Scenarios

§3.4.3 shows how PagedAttention and vLLM handle basic decoding algorithms, such as greedy decoding and sampling, that take one user prompt as input and generate a single output sequence. In many successful LLM applications [30, 71], an LLM service must offer more complex decoding scenarios that exhibit complex accessing patterns and more oppor-

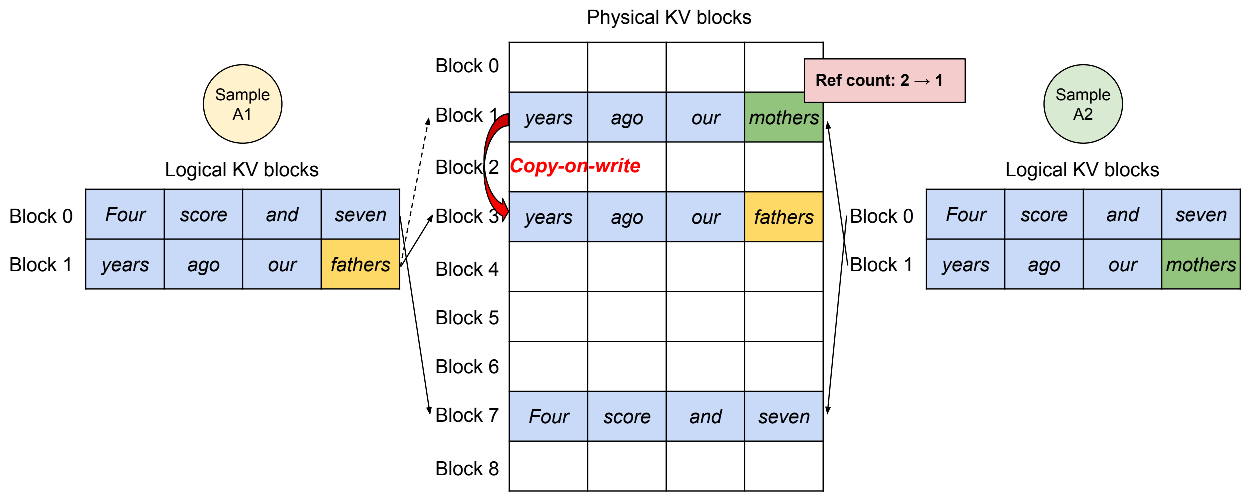


Figure 3.8: Parallel sampling example.

tunities for memory sharing. We show the general applicability of vLLM on them in this section.

Parallel sampling. In LLM-based program assistants [16, 30], an LLM generates multiple sampled outputs for a single input prompt; users can choose a favorite output from various candidates. So far we have implicitly assumed that a request generates a single sequence. In the remainder of this paper, we assume the more general case in which a request generates multiple sequences. In parallel sampling, one request includes multiple samples sharing the same input prompt, allowing the KV cache of the prompt to be shared as well. Via its PagedAttention and paged memory management, vLLM can realize this sharing easily and save memory.

Fig. 3.8 shows an example of parallel decoding for two outputs. Since both outputs share the same prompt, we only reserve space for one copy of the prompt's state at the prompt phase; the logical blocks for the prompts of both sequences are mapped to the same physical blocks: the logical block 0 and 1 of both sequences are mapped to physical blocks 7 and 1, respectively. Since a single physical block can be mapped to multiple logical blocks, we introduce a *reference count* for each physical block. In this case, the reference counts for physical blocks 7 and 1 are both 2. At the generation phase, the two outputs sample different output tokens and need separate storage for KV cache. vLLM implements a *copy-on-write* mechanism at the block granularity for the physical blocks that need modification by multiple sequences, similar to the copy-on-write technique in OS virtual memory (e.g., when forking a process). Specifically, in Fig. 3.8, when sample A1 needs to write to its last logical block (logical block 1), vLLM recognizes that the reference count of the corresponding physical block (physical block 1) is greater than 1; it allocates a new physical block (physical block 3), instructs the block engine to copy the information from physical block 1, and decreases the reference count to 1. Next, when sample A2 writes to physical block 1, the reference count

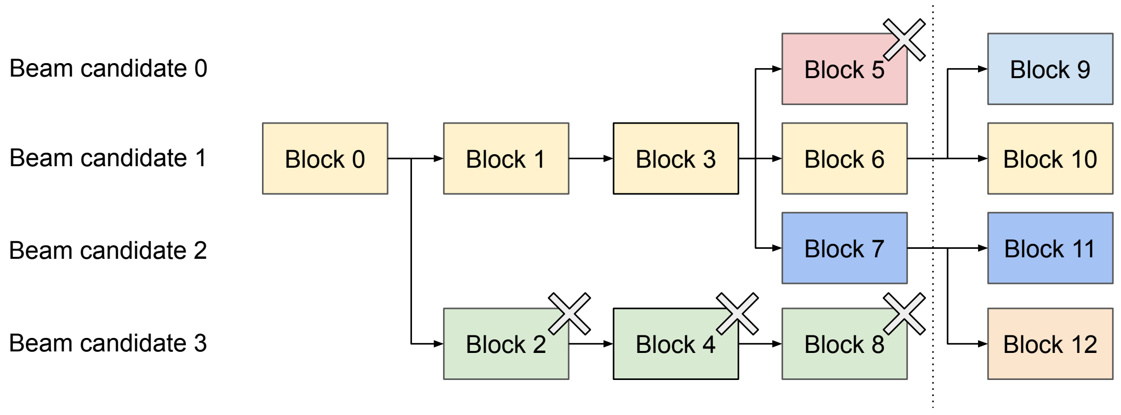


Figure 3.9: Beam search example.

is already reduced to 1; thus A2 directly writes its newly generated KV cache to physical block 1.

In summary, vLLM enables the sharing of most of the space used to store the prompts' KV cache across multiple output samples, with the exception of the final logical block, which is managed by a copy-on-write mechanism. By sharing physical blocks across multiple samples, memory usage can be greatly reduced, especially for *long input prompts*.

Beam search. In LLM tasks like machine translation [105], the users expect the top-k most appropriate translations output by the LLM. Beam search [91] is widely used to decode the most probable output sequence from an LLM, as it mitigates the computational complexity of fully traversing the sample space. The algorithm relies on the *beam width* parameter k, which determines the number of top candidates retained at every step. During decoding, beam search expands each candidate sequence in the beam by considering all possible tokens, computes their respective probabilities using the LLM, and retains the top-k most probable sequences out of $k \cdot |V|$ candidates, where $|V|$ is the vocabulary size.

Unlike parallel decoding, beam search facilities sharing not only the initial prompt blocks but also other blocks across different candidates, and the sharing patterns dynamically change as the decoding process advances, similar to the process tree in the OS created by compound forks. Fig. 3.9 shows how vLLM manages the KV blocks for a beam search example with $k = 4$. Prior to the iteration illustrated as the dotted line, each candidate sequence has used 4 full logical blocks. All beam candidates share the first block 0 (i.e., prompt). Candidate 3 digresses from others from the second block. Candidates 0-2 share the first 3 blocks and diverge at the fourth block. At subsequent iterations, the top-4 probable candidates all originate from candidates 1 and 2. As the original candidates 0 and 3 are no longer among the top candidates, their logical blocks are freed, and the reference counts of corresponding physical blocks are reduced. vLLM frees all physical blocks whose reference counts reach 0 (blocks 2, 4, 5, 8). Then, vLLM allocates new physical blocks (blocks 9-12) to store the new KV cache from the new candidates. Now, all candidates share blocks 0, 1,

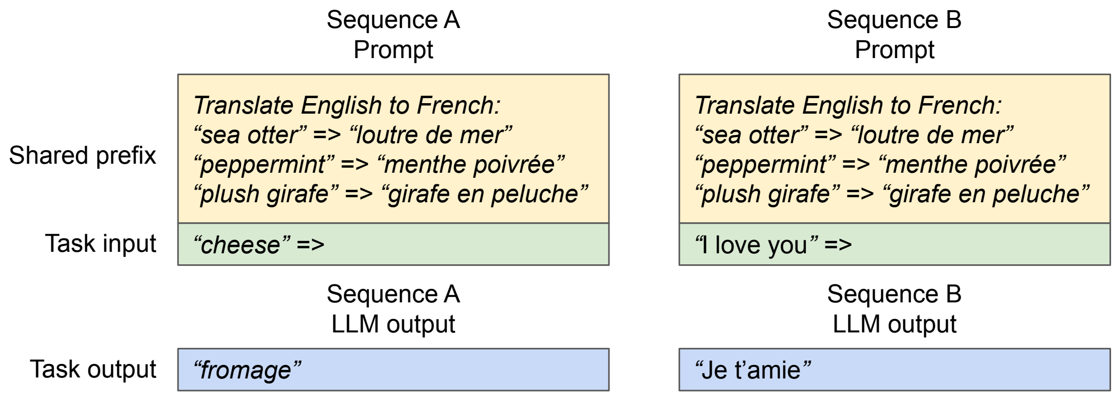


Figure 3.10: Shared prompt example for machine translation. The examples are adopted from [14].

3; candidates 0 and 1 share block 6, and candidates 2 and 3 further share block 7.

Previous LLM serving systems require frequent memory copies of the KV cache across the beam candidates. For example, in the case shown in Fig. 3.9, after the dotted line, candidate 3 would need to copy a large portion of candidate 2’s KV cache to continue generation. This frequent memory copy overhead is significantly reduced by vLLM’s physical block sharing. In vLLM, most blocks of different beam candidates can be shared. The copy-on-write mechanism is applied only when the newly generated tokens are within an old shared block, as in parallel decoding. This involves only copying one block of data.

Shared prefix. Commonly, the LLM user provides a (long) description of the task including instructions and example inputs and outputs, also known as *system prompt* [73]. The description is concatenated with the actual task input to form the prompt of the request. The LLM generates outputs based on the full prompt. Fig. 3.10 shows an example. Moreover, the shared prefix can be further tuned, via prompt engineering, to improve the accuracy of the downstream tasks [51, 48].

For this type of application, many user prompts share a prefix, thus the LLM service provider can store the KV cache of the prefix in advance to reduce the redundant computation spent on the prefix. In vLLM, this can be conveniently achieved by reserving a set of physical blocks for a set of predefined shared prefixes by the LLM service provider, as how OS handles shared library across processes. A user input prompt with the shared prefix can simply map its logical blocks to the cached physical blocks (with the last block marked copy-on-write). The prompt phase computation only needs to execute on the user’s task input.

Mixed decoding methods. The decoding methods discussed earlier exhibit diverse memory sharing and accessing patterns. Nonetheless, vLLM facilitates the simultaneous processing of requests with different decoding preferences, which existing systems *cannot* efficiently do. This is because vLLM conceals the complex memory sharing between different sequences via a common mapping layer that translates logical blocks to physical blocks. The

LLM and its execution kernel only see a list of physical block IDs for each sequence and do not need to handle sharing patterns across sequences. Compared to existing systems, this approach broadens the batching opportunities for requests with different sampling requirements, ultimately increasing the system's overall throughput.

3.4.5 Scheduling and Preemption

When the request traffic surpasses the system's capacity, vLLM must prioritize a subset of requests. In vLLM, we adopt the first-come-first-serve (FCFS) scheduling policy for all requests, ensuring fairness and preventing starvation. When vLLM needs to preempt requests, it ensures that the earliest arrived requests are served first and the latest requests are preempted first.

LLM services face a unique challenge: the input prompts for an LLM can vary significantly in length, and the resulting output lengths are not known a priori, contingent on both the input prompt and the model. As the number of requests and their outputs grow, vLLM can run out of the GPU's physical blocks to store the newly generated KV cache. There are two classic questions that vLLM needs to answer in this context: (1) Which blocks should it evict? (2) How to recover evicted blocks if needed again? Typically, eviction policies use heuristics to predict which block will be accessed furthest in the future and evict that block. Since in our case we know that all blocks of a sequence are accessed together, we implement an all-or-nothing eviction policy, i.e., either evict all or none of the blocks of a sequence. Furthermore, multiple sequences within one request (e.g., beam candidates in one beam search request) are gang-scheduled as a *sequence group*. The sequences within one sequence group are always preempted or rescheduled together due to potential memory sharing across those sequences. To answer the second question of how to recover an evicted block, we consider two techniques:

Swapping. This is the classic technique used by most virtual memory implementations which copy the evicted pages to a swap space on the disk. In our case, we copy evicted blocks to the CPU memory. As shown in Fig. 3.4, besides the GPU block allocator, vLLM includes a CPU block allocator to manage the physical blocks swapped to CPU RAM. When vLLM exhausts free physical blocks for new tokens, it selects a set of sequences to evict and transfer their KV cache to the CPU. Once it preempts a sequence and evicts its blocks, vLLM stops accepting new requests until all preempted sequences are completed. Once a request completes, its blocks are freed from memory, and the blocks of a preempted sequence are brought back in to continue the processing of that sequence. Note that with this design, the number of blocks swapped to the CPU RAM never exceeds the number of total physical blocks in the GPU RAM, so the swap space on the CPU RAM is bounded by the GPU memory allocated for the KV cache.

Recomputation. In this case, we simply recompute the KV cache when the preempted sequences are rescheduled. Note that recomputation latency can be significantly lower than the original latency, as the tokens generated at decoding can be concatenated with the original user prompt as a new prompt—their KV cache at all positions can be generated in one prompt phase iteration.

The performances of swapping and recomputation depend on the bandwidth between CPU RAM and GPU memory and the computation power of the GPU. We examine the speeds of swapping and recomputation in §3.7.3.

3.4.6 Distributed Execution

Many LLMs have parameter sizes exceeding the capacity of a single GPU [14, 19]. Therefore, it is necessary to partition them across distributed GPUs and execute them in a model parallel fashion [114, 53]. This calls for a memory manager capable of handling distributed memory. vLLM is effective in distributed settings by supporting the widely used Megatron-LM style tensor model parallelism strategy on Transformers [89]. This strategy adheres to an SPMD (Single Program Multiple Data) execution schedule, wherein the linear layers are partitioned to perform block-wise matrix multiplication, and the the GPUs constantly synchronize intermediate results via an all-reduce operation. Specifically, the attention operator is split on the attention head dimension, each SPMD process takes care of a subset of attention heads in multi-head attention.

We observe that even with model parallel execution, each model shard still processes the same set of input tokens, thus requiring the KV Cache for the same positions. Therefore, vLLM features a single KV cache manager within the centralized scheduler, as in Fig. 3.4. Different GPU workers share the manager, as well as the mapping from logical blocks to physical blocks. This common mapping allows GPU workers to execute the model with the physical blocks provided by the scheduler for each input request. Although each GPU worker has the same physical block IDs, a worker only stores a portion of the KV cache for its corresponding attention heads.

In each step, the scheduler first prepares the message with input token IDs for each request in the batch, as well as the block table for each request. Next, the scheduler broadcasts this control message to the GPU workers. Then, the GPU workers start to execute the model with the input token IDs. In the attention layers, the GPU workers read the KV cache according to the block table in the control message. During execution, the GPU workers synchronize the intermediate results with the all-reduce communication primitive without the coordination of the scheduler, as in [89]. In the end, the GPU workers send the sampled tokens of this iteration back to the scheduler. In summary, GPU workers do not need to synchronize on memory management as they only need to receive all the memory management information at the beginning of each decoding iteration along with the step inputs.

3.5 Implementation

vLLM is an end-to-end serving system with a FastAPI [27] frontend and a GPU-based inference engine. The frontend extends the OpenAI API [71] interface, allowing users to customize sampling parameters for each request, such as the maximum sequence length and the beam width k. The vLLM engine is written in 8.5K lines of Python and 2K lines of C++/CUDA code. We develop control-related components including the scheduler and the block manager in Python while developing custom CUDA kernels for key operations such as PagedAttention. For the model executor, we implement popular LLMs such as GPT [14],

Table 3.1: Model sizes and server configurations.

Model size	**13B**	**66B**	**175B**
GPUs	A100	4×A100	8×A100-80GB
Total GPU memory	40 GB	160 GB	640 GB
Parameter size	26 GB	132 GB	346 GB
Memory for KV cache	12 GB	21 GB	264 GB
Max. # KV cache slots	15.7K	9.7K	60.1K

OPT [113], and LLaMA [96] using PyTorch [78] and Transformers [104]. We use NCCL [65] for tensor communication across the distributed GPU workers.

3.5.1 Kernel-level Optimization

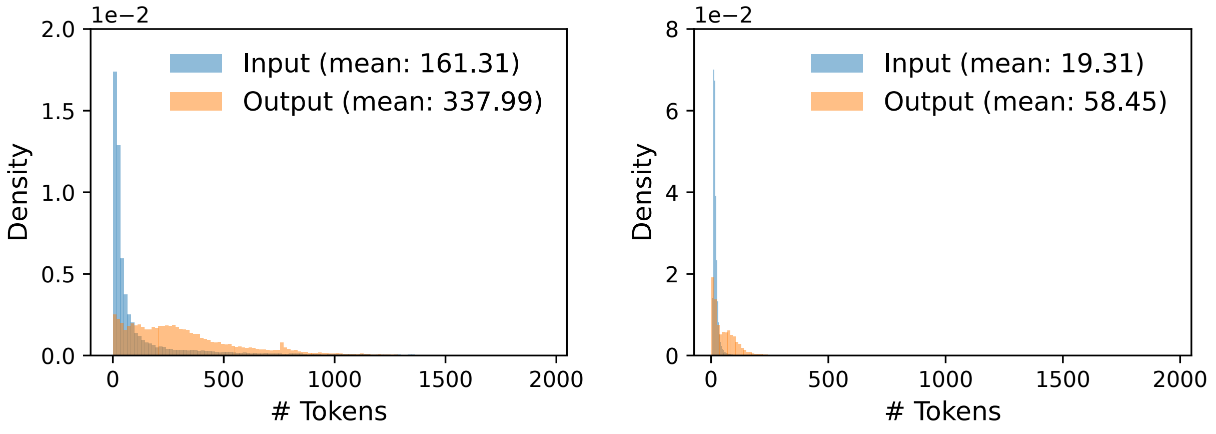
Since PagedAttention introduces memory access patterns that are not efficiently supported by existing systems, we develop several GPU kernels for optimizing it. (1) *Fused reshape and block write.* In every Transformer layer, the new KV cache are split into blocks, reshaped to a memory layout optimized for block read, then saved at positions specified by the block table. To minimize kernel launch overheads, we fuse them into a single kernel. (2) *Fusing block read and attention.* We adapt the attention kernel in FasterTransformer [63] to read KV cache according to the block table and perform attention operations on the fly. To ensure coalesced memory access, we assign a GPU warp to read each block. Moreover, we add support for variable sequence lengths within a request batch. (3) *Fused block copy.* Block copy operations, issued by the copy-on-write mechanism, may operate on discontinuous blocks. This can lead to numerous invocations of small data movements if we use the `cudaMemcpyAsync` API. To mitigate the overhead, we implement a kernel that batches the copy operations for different blocks into a single kernel launch.

3.5.2 Supporting Various Decoding Algorithms

vLLM implements various decoding algorithms using three key methods: `fork`, `append`, and `free`. The `fork` method creates a new sequence from an existing one. The `append` method appends a new token to the sequence. Finally, the `free` method deletes the sequence. For instance, in parallel sampling, vLLM creates multiple output sequences from the single input sequence using the `fork` method. It then adds new tokens to these sequences in every iteration with `append`, and deletes sequences that meet a stopping condition using `free`. The same strategy is also applied in beam search and prefix sharing by vLLM. We believe future decoding algorithms can also be supported by combining these methods.

3.6 Evaluation

In this section, we evaluate the performance of vLLM under a variety of workloads.



(a) ShareGPT (b) Alpaca

Figure 3.11: Input and output length distributions of the (a) ShareGPT and (b) Alpaca datasets.

3.6.1 Experimental Setup

Model and server configurations. We use OPT [113] models with 13B, 66B, and 175B parameters and LLaMA [96] with 13B parameters for our evaluation. 13B and 66B are popular sizes for LLMs as shown in an LLM leaderboard [77], while 175B is the size of the famous GPT-3 [14] model. For all of our experiments, we use A2 instances with NVIDIA A100 GPUs on Google Cloud Platform. The detailed model sizes and server configurations are shown in Table 3.1.

Workloads. We synthesize workloads based on ShareGPT [93] and Alpaca [92] datasets, which contain input and output texts of real LLM services. The ShareGPT dataset is a collection of user-shared conversations with ChatGPT [72]. The Alpaca dataset is an instruction dataset generated by GPT-3.5 with self-instruct [101]. We tokenize the datasets and use their input and output lengths to synthesize client requests. As shown in Fig. 3.11, the ShareGPT dataset has 8.4× longer input prompts and 5.8× longer outputs on average than the Alpaca dataset, with higher variance. Since these datasets do not include timestamps, we generate request arrival times using Poisson distribution with different request rates.

Baseline 1: FasterTransformer. FasterTransformer [63] is a distributed inference engine highly optimized for latency. As FasterTransformer does not have its own scheduler, we implement a custom scheduler with a dynamic batching mechanism similar to the existing serving systems such as Triton [69]. Specifically, we set a maximum batch size B as large as possible for each experiment, according to the GPU memory capacity. The scheduler takes up to B number of earliest arrived requests and sends the batch to FasterTransformer for processing.

Baseline 2: Orca. Orca [110] is a state-of-the-art LLM serving system optimized for

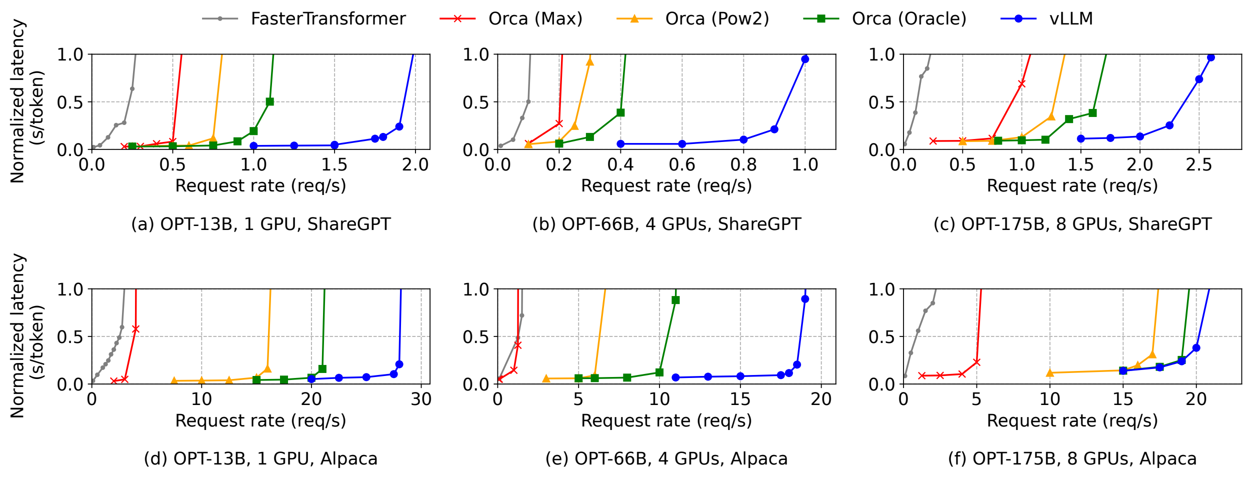


Figure 3.12: Single sequence generation with OPT models on the ShareGPT and Alpaca dataset

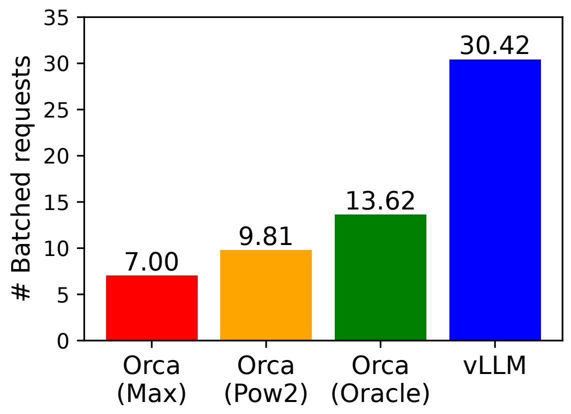
throughput. Since Orca is not publicly available for use, we implement our own version of Orca. We assume Orca uses the buddy allocation algorithm to determine the memory address to store KV cache. We implement three versions of Orca based on how much it over-reserves the space for request outputs:

- **Orca (Oracle).** We assume the system has the knowledge of the lengths of the outputs that will be actually generated for the requests. This shows the upper-bound performance of Orca, which is infeasible to achieve in practice.
- **Orca (Pow2).** We assume the system over-reserves the space for outputs by at most 2×. For example, if the true output length is 25, it reserves 32 positions for outputs.
- **Orca (Max).** We assume the system always reserves the space up to the maximum sequence length of the model, i.e., 2048 tokens.

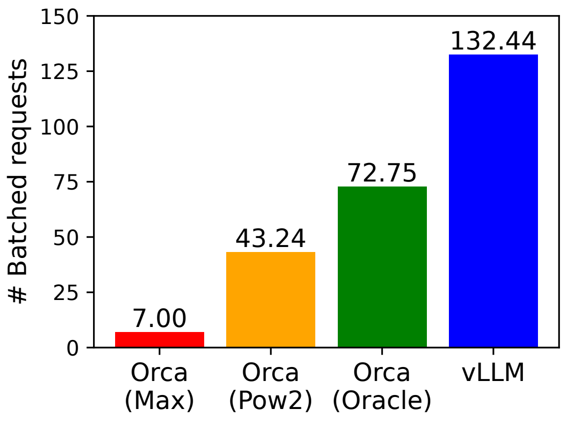
Key metrics. We focus on serving throughput. Specifically, using the workloads with different request rates, we measure *normalized latency* of the systems, the mean of every request's end-to-end latency divided by its output length, as in Orca [110]. A high-throughput serving system should retain low normalized latency against high request rates. For most experiments, we evaluate the systems with 1-hour traces. As an exception, we use 15-minute traces for the OPT-175B model due to the cost limit.

3.6.2 Basic Sampling

We evaluate the performance of vLLM with basic sampling (one sample per request) on three models and two datasets. The first row of Fig. 3.12 shows the results on the ShareGPT dataset. The curves illustrate that as the request rate increases, the latency initially increases



(a) ShareGPT



(b) Alpaca

Figure 3.13: Average number of batched requests when serving OPT-13B for the ShareGPT (2 reqs/s) and Alpaca (30 reqs/s) traces.

at a gradual pace but then suddenly explodes. This can be attributed to the fact that when the request rate surpasses the capacity of the serving system, the queue length continues to grow infinitely and so does the latency of the requests.

On the ShareGPT dataset, vLLM can sustain 1.7×–2.7× higher request rates compared to Orca (Oracle) and 2.7×–8× compared to Orca (Max), while maintaining similar latencies. This is because vLLM's PagedAttention can efficiently manage the memory usage and thus enable batching more requests than Orca. For example, as shown in Fig. 3.13a, for OPT-13B vLLM processes 2.2× more requests at the same time than Orca (Oracle) and 4.3× more requests than Orca (Max). Compared to FasterTransformer, vLLM can sustain up to 22× higher request rates, as FasterTransformer does not utilize a fine-grained scheduling mechanism and inefficiently manages the memory like Orca (Max).

The second row of Fig. 3.12 and Fig. 3.13b shows the results on the Alpaca dataset, which follows a similar trend to the ShareGPT dataset. One exception is Fig. 3.12 (f), where vLLM's advantage over Orca (Oracle) and Orca (Pow2) is less pronounced. This is because the model and server configuration for OPT-175B (Table 3.1) allows for large GPU memory space available to store KV cache, while the Alpaca dataset has short sequences. In this setup, Orca (Oracle) and Orca (Pow2) can also batch a large number of requests despite the inefficiencies in their memory management. As a result, the performance of the systems becomes compute-bound rather than memory-bound.

3.6.3 Parallel Sampling and Beam Search

We evaluate the effectiveness of memory sharing in PagedAttention with two popular sampling methods: parallel sampling and beam search. In parallel sampling, all parallel sequences in a request can share the KV cache for the prompt. As shown in the first row of

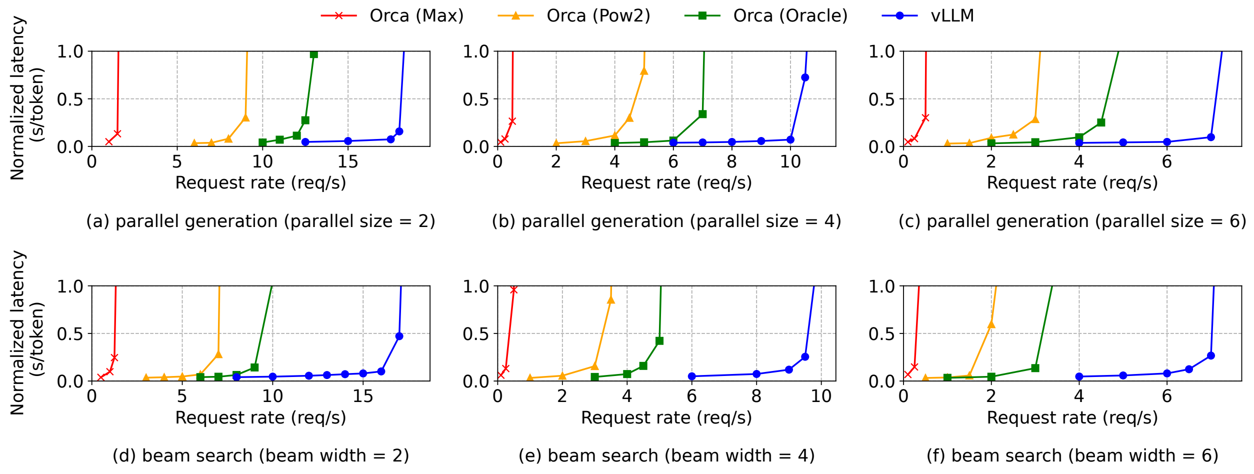


Figure 3.14: Parallel generation and beam search with OPT-13B on the Alpaca dataset.

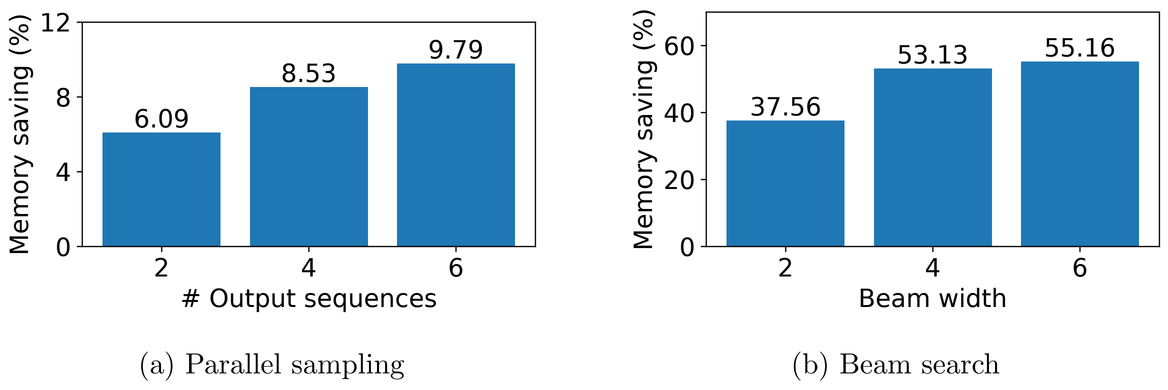


Figure 3.15: Average amount of memory saving from sharing KV blocks, when serving OPT-13B for the Alpaca trace.

Fig. 3.14, with a larger number of sequences to sample, vLLM brings more improvement over the Orca baselines. Similarly, the second row of Fig. 3.14 shows the results for beam search with different beam widths. Since beam search allows for more sharing, vLLM demonstrates even greater performance benefits. The improvement of vLLM over Orca (Oracle) on OPT-13B and the Alpaca dataset goes from 1.3× in basic sampling to 2.3× in beam search with a width of 6.

Fig. 3.15 plots the amount of memory saving, computed by the number of blocks we saved by sharing divided by the number of total blocks without sharing. We show 6.1% - 9.8% memory saving on parallel sampling and 37.6% - 55.2% on beam search. In the same experiments with the ShareGPT dataset, we saw 16.2% - 30.5% memory saving on parallel sampling and 44.3% - 66.3% on beam search.

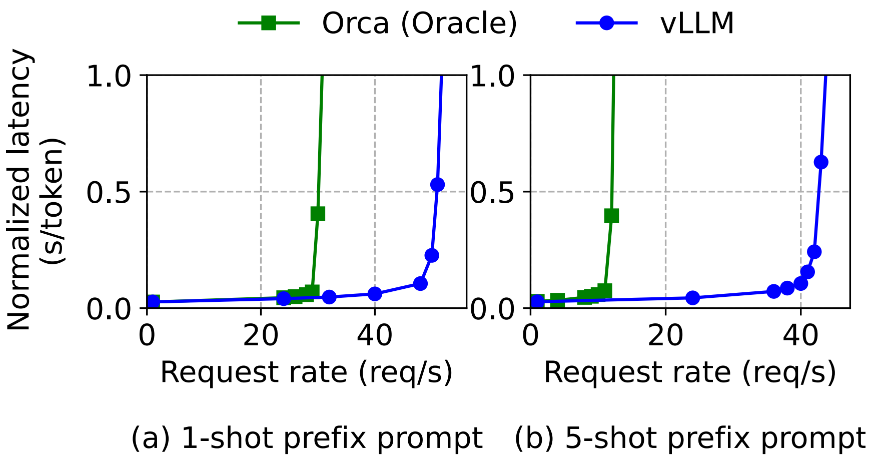


Figure 3.16: Translation workload where the input prompts share a common prefix. The prefix includes (a) 1 example with 80 tokens or (b) 5 examples with 341 tokens.

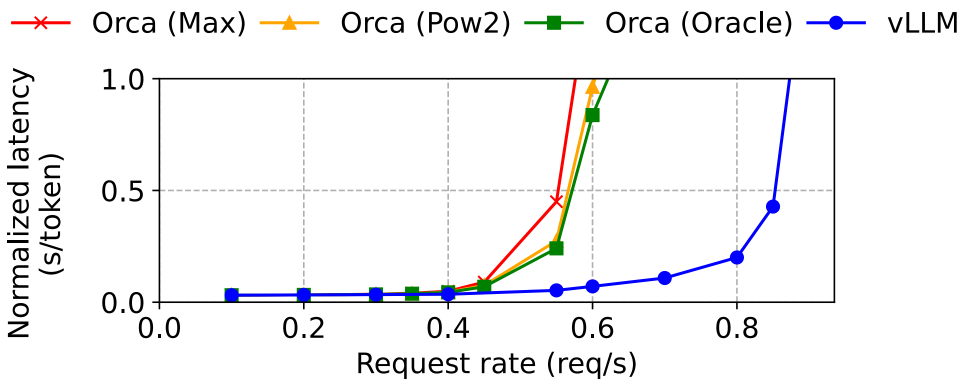
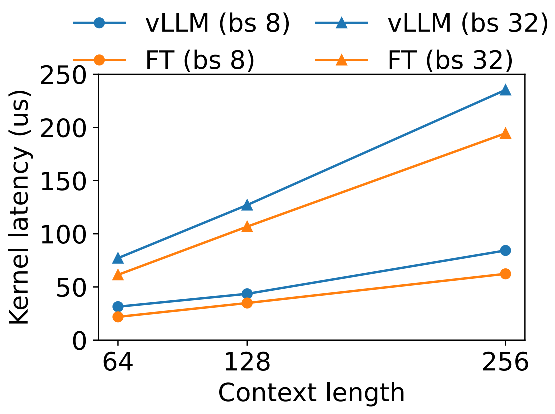


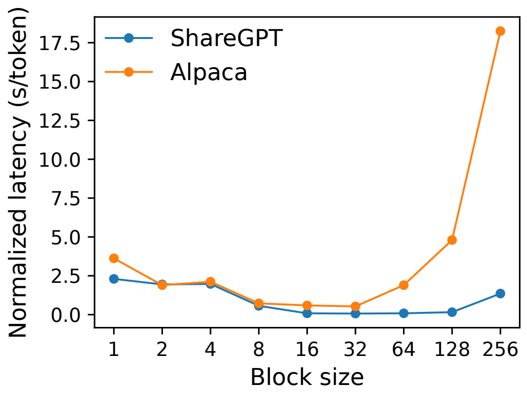
Figure 3.17: Performance on chatbot workload.

3.6.4 Shared prefix

We explore the effectiveness of vLLM for the case a prefix is shared among different input prompts, as illustrated in Fig. 3.10. For the model, we use LLaMA-13B [96], which is multilingual. For the workload, we use the WMT16 [13] English-to-German translation dataset and synthesize two prefixes that include an instruction and a few translation examples. The first prefix includes a single example (i.e., one-shot) while the other prefix includes 5 examples (i.e., few-shot). As shown in Fig. 3.16 (a), vLLM achieves 1.67× higher throughput than Orca (Oracle) when the one-shot prefix is shared. Furthermore, when more examples are shared (Fig. 3.16 (b)), vLLM achieves 3.58× higher throughput than Orca (Oracle).



(a) Latency of attention kernels.



(b) End-to-end latency with different block sizes.

Figure 3.18: Ablation experiments.

3.6.5 Chatbot

A chatbot [72, 31, 18] is one of the most important applications of LLMs. To implement a chatbot, we let the model generate a response by concatenating the chatting history and the last user query into a prompt. We synthesize the chatting history and user query using the ShareGPT dataset. Due to the limited context length of the OPT-13B model, we cut the prompt to the last 1024 tokens and let the model generate at most 1024 tokens. We do not store the KV cache between different conversation rounds as doing this would occupy the space for other requests between the conversation rounds.

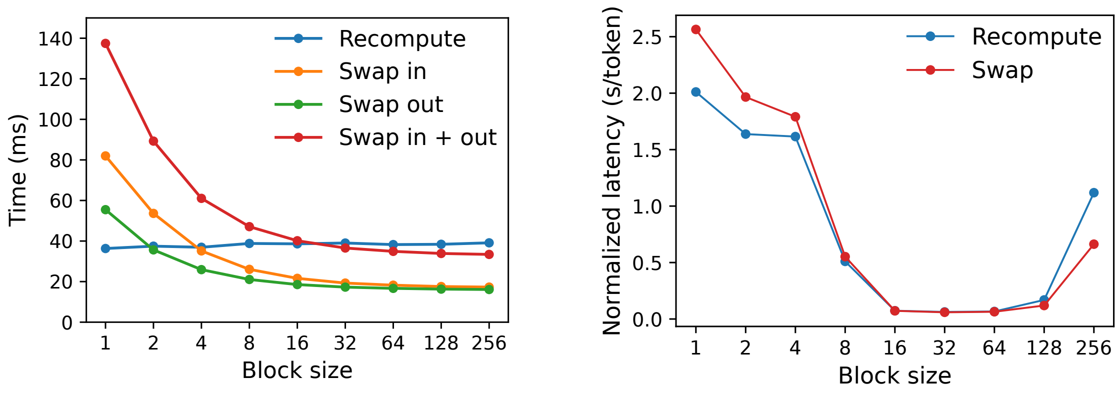
Fig. 3.17 shows that vLLM can sustain 2× higher request rates compared to the three Orca baselines. Since the ShareGPT dataset contains many long conversations, the input prompts for most requests have 1024 tokens. Due to the buddy allocation algorithm, the Orca baselines reserve the space for 1024 tokens for the request outputs, regardless of how they predict the output lengths. For this reason, the three Orca baselines behave similarly. In contrast, vLLM can effectively handle the long prompts, as PagedAttention resolves the problem of memory fragmentation and reservation.

3.7 Ablation Studies

In this section, we study various aspects of vLLM and evaluate the design choices we make with ablation experiments.

3.7.1 Kernel Microbenchmark

The dynamic block mapping in PagedAttention affects the performance of the GPU operations involving the stored KV cache, i.e., block read/writes and attention. Compared to the existing systems, our GPU kernels (§3.5) involve extra overheads of accessing the



(a) Microbenchmark

(b) End-to-end performance

Figure 3.19: (a) Overhead of recomputation and swapping for different block sizes. (b) Performance when serving OPT-13B with the ShareGPT traces at the same request rate.

block table, executing extra branches, and handling variable sequence lengths. As shown in Fig. 3.18a, this leads to 20–26% higher attention kernel latency, compared to the highly-optimized FasterTransformer implementation. We believe the overhead is small as it only affects the attention operator but not the other operators in the model, such as Linear. Despite the overhead, PagedAttention makes vLLM significantly outperform FasterTransformer in end-to-end performance (§3.6).

3.7.2 Impact of Block Size

The choice of block size can have a substantial impact on the performance of vLLM. If the block size is too small, vLLM may not fully utilize the GPU's parallelism for reading and processing KV cache. If the block size is too large, internal fragmentation increases and the probability of sharing decreases.

In Fig. 3.18b, we evaluate the performance of vLLM with different block sizes, using the ShareGPT and Alpaca traces with basic sampling under fixed request rates. In the ShareGPT trace, block sizes from 16 to 128 lead to the best performance. In the Alpaca trace, while the block size 16 and 32 work well, larger block sizes significantly degrade the performance since the sequences become shorter than the block sizes. In practice, we find that the block size 16 is large enough to efficiently utilize the GPU and small enough to avoid significant internal fragmentation in most workloads. Accordingly, vLLM sets its default block size as 16.

3.7.3 Comparing Recomputation and Swapping

vLLM supports both recomputation and swapping as its recovery mechanisms. To understand the tradeoffs between the two methods, we evaluate their end-to-end performance and microbenchmark their overheads, as presented in Fig. 3.19. Our results reveal that swapping incurs excessive overhead with small block sizes. This is because small block sizes often result in numerous small data transfers between CPU and GPU, which limits the effective PCIe bandwidth. In contrast, the overhead of recomputation remains constant across different block sizes, as recomputation does not utilize the KV blocks. Thus, recomputation is more efficient when the block size is small, while swapping is more efficient when the block size is large, though recomputation overhead is never higher than 20% of swapping's latency. For medium block sizes from 16 to 64, the two methods exhibit comparable end-to-end performance.

3.8 Discussion

Applying the virtual memory and paging technique to other GPU workloads. The idea of virtual memory and paging is effective for managing the KV cache in LLM serving because the workload requires dynamic memory allocation (since the output length is not known a priori) and its performance is bound by the GPU memory capacity. However, this does not generally hold for every GPU workload. For example, in DNN training, the tensor shapes are typically static, and thus memory allocation can be optimized ahead of time. For another example, in serving DNNs that are not LLMs, an increase in memory efficiency may not result in any performance improvement since the performance is primarily compute-bound. In such scenarios, introducing the vLLM's techniques may rather degrade the performance due to the extra overhead of memory indirection and non-contiguous block memory. However, we would be excited to see vLLM's techniques being applied to other workloads with similar properties to LLM serving.

LLM-specific optimizations in applying virtual memory and paging. vLLM re-interprets and augments the idea of virtual memory and paging by leveraging the application-specific semantics. One example is vLLM's all-or-nothing swap-out policy, which exploits the fact that processing a request requires all of its corresponding token states to be stored in GPU memory. Another example is the recomputation method to recover the evicted blocks, which is not feasible in OS. Besides, vLLM mitigates the overhead of memory indirection in paging by fusing the GPU kernels for memory access operations with those for other operations such as attention.

3.9 Related Work

General model serving systems. Model serving has been an active area of research in recent years, with numerous systems proposed to tackle diverse aspects of deep learning model deployment. Clipper [20], TensorFlow Serving [70], Nexus [87], InferLine [21], and Clockwork [33] are some earlier general model serving systems. They study batching, caching, placement, and scheduling for serving single or multiple models. More recently, DVABatch [22] introduces multi-entry multi-exit batching. REEF [34] and Shepherd [112] propose preemption for serving. AlpaServe [53] utilizes model parallelism for statistical multiplexing. However, these general systems fail to take into account the auto-regressive property and token state of LLM inference, resulting in missed opportunities for optimization.

Specialized serving systems for transformers. Due to the significance of the transformer architecture, numerous specialized serving systems for it have been developed. These systems utilize GPU kernel optimizations [100, 4, 63, 59], advanced batching mechanisms [26, 110], model parallelism [80, 110, 4], and parameter sharing [117] for efficient serving. Among them, Orca [110] is most relevant to our approach.

Comparison to Orca. The iteration-level scheduling in Orca [110] and PagedAttention in vLLM are complementary techniques: While both systems aim to increase the GPU utilization and hence the throughput of LLM serving, Orca achieves it by scheduling and interleaving the requests so that more requests can be processed in parallel, while vLLM is doing so by increasing memory utilization so that the working sets of more requests fit into memory. By reducing memory fragmentation and enabling sharing, vLLM runs more requests in a batch in parallel and achieves a 2-4$\times$ speedup compared to Orca. Indeed, the fine-grained scheduling and interleaving of the requests like in Orca makes memory management more challenging, making the techniques proposed in vLLM even more crucial.

Memory optimizations. The widening gap between the compute capability and memory capacity of accelerators has caused memory to become a bottleneck for both training and inference. Swapping [38, 99, 81], recomputation [17, 42] and their combination [79] have been utilized to reduce the peak memory of training. Notably, FlexGen [88] studies how to swap weights and token states for LLM inference with limited GPU memory, but it does not target the online serving settings. OLLA [90] optimizes the lifetime and location of tensors to reduce fragmentation, but it does not do fine-grained block-level management or online serving. FlashAttention [23] applies tiling and kernel optimizations to reduce the peak memory of attention computation and reduce I/O costs. This paper introduces a new idea of block-level memory management in the context of online serving.

3.10 Conclusion

This paper proposes PagedAttention, a new attention algorithm that allows attention keys and values to be stored in non-contiguous paged memory, and presents vLLM, a high-throughput LLM serving system with efficient memory management enabled by PagedAttention. Inspired by operating systems, we demonstrate how established techniques, such as virtual memory and copy-on-write, can be adapted to efficiently manage KV cache and handle various decoding algorithms in LLM serving. Our experiments show that vLLM achieves 2-4$\times$ throughput improvements over the state-of-the-art systems.

Chapter 4

vLLM: The Design and Implementation

Despite the algorithmic improvements introduced in Chapter 3, building a practical and high-performance inference engine for large language models (LLMs) involves far more than optimizing the core attention algorithm. Real-world LLM inference pipelines must handle heterogeneous user workloads, diverse model architectures, and strict performance constraints imposed by modern hardware. These challenges extend across the entire stack—from user-facing APIs to low-level GPU kernels—and require a system-wide design.

As a response to these challenges, this chapter presents **vLLM** [94], a widely adopted open-source inference engine that has rapidly become a de facto standard in both academia and industry. We describe vLLM from its overarching goals and guiding philosophies to the detailed design of its interfaces and internal components.

4.1 Overview

4.1.1 Goals and Philosophies

vLLM was created with several concrete goals derived from real-world deployment requirements:

- **Ease of Use**: How can we make deploying LLMs as simple, intuitive, and frictionless as possible? What is the absolute minimum interface a user should provide to get started? Users should not need to understand the internals of the inference engine to leverage its capabilities effectively.

- **Diverse Model Support**: How can we quickly and reliably support the rapidly evolving ecosystem of models? The pace of innovation in LLM architectures is unprecedented, with new models being released weekly. An inference engine must be able to adapt to this rapid evolution without requiring extensive re-engineering.

- **Inference Optimizations**: How can we incorporate state-of-the-art inference optimizations? The landscape of inference optimizations is vast and growing, including techniques such as speculative decoding, prefix caching, and disaggregated serving. An effective inference engine must be able to integrate these techniques seamlessly.

- **Zero CPU Overheads**: How can we keep the GPU fully utilized while avoiding unnecessary CPU bottlenecks? This consideration is becoming increasingly critical for performance on the latest generation of GPUs, where a forward pass of the model only takes a few milliseconds. Even small CPU overheads can significantly degrade overall throughput.

Achieving these goals introduces significant engineering and architectural challenges. Any solution must strike a careful balance between performance, flexibility, stability, and extensibility, while keeping the system understandable and maintainable. These requirements are often at tension with each other: for example, maximizing performance often requires specialization, which can conflict with flexibility; supporting extensibility can introduce complexity that threatens maintainability.

To navigate these competing requirements, vLLM follows three fundamental design philosophies:

- **Be general**: vLLM is designed around abstractions that generalize across diverse optimizations, model architectures, and hardware constraints. Rather than implementing each optimization as an isolated special case, vLLM seeks unified representations that capture the common structure underlying seemingly disparate techniques. This approach avoids the fragmentation and combinatorial complexity that arise when each feature requires its own code path, and allows new optimizations to be incorporated by mapping them onto existing abstractions rather than requiring architectural changes.

- **Be flexible**: The AI landscape evolves rapidly, with new models, modalities, and optimization techniques emerging continuously. vLLM intentionally avoids overfitting to present-day workloads, instead favoring abstractions that can accommodate emerging model types, new modalities, and future optimizations. This flexibility is achieved through careful interface design, modular architecture, and extensible abstractions that can evolve without requiring wholesale system redesign.

- **Be open**: The scope of efficient LLM inference is too large for any single organization to tackle comprehensively. vLLM embraces openness—open development, open-source collaboration, and transparent design—enabling contributions from academia, industry developers, and the research community. This openness has been instrumental in vLLM's rapid adoption and continuous improvement, as it allows the broader community to contribute optimizations, fix bugs, and extend functionality.

These principles have played a central role in shaping the architecture described in the following sections and have enabled vLLM to evolve rapidly while maintaining stability and performance.

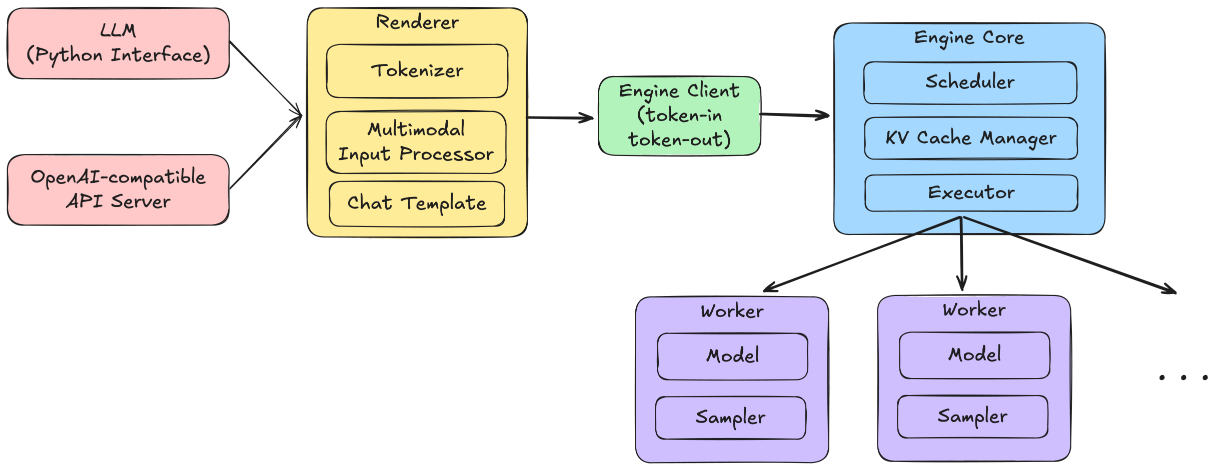


Figure 4.1: Overview of vLLM's architecture.

4.1.2 High-level Architecture

Figure 4.1 illustrates the high-level architecture of vLLM, which is organized into a series of layered components that communicate to process user requests and return generated outputs. Each layer has well-defined responsibilities and interfaces, enabling modularity and separation of concerns. We describe each component in detail below, proceeding from the outermost user-facing layer to the innermost execution layer.

User-Facing Interfaces. At the outermost layer, vLLM exposes two primary interfaces designed for different use cases: the `LLM` Python class for offline batched inference, and an OpenAI-compatible API server for online serving. The `LLM` class provides a programmatic interface for users who want to run batched inference without deploying a persistent service. The API server, on the other hand, provides a RESTful interface compatible with OpenAI's API specification [71], enabling seamless integration with existing applications and tooling built around that ecosystem.

Both interfaces accept high-level inputs, such as raw text prompts or structured requests containing text interleaved with multimodal data (e.g., images, audio, or video), and return generated text completions. These interfaces abstract away all internal complexity, allowing users with no knowledge of vLLM internals to leverage its capabilities. Users need only specify the model name and their input prompts; all other details—such as memory management, batching, and scheduling—are handled automatically by the system.

Renderer. The `Renderer` serves as the bridge between user-facing interfaces and the inference engine, handling the transformation between human-readable inputs and LLM-processable token sequences. On the input path, the `Renderer` performs three key transformations:

1. A model-specific *chat template* formats conversational inputs according to the target model's expected schema. Different models have different conventions for representing

conversations, system prompts, and special tokens, and the chat template ensures that inputs are formatted correctly.

2. A *tokenizer* converts text into tokens that the model can process.

3. A *multimodal input processor* handles non-textual data such as images, audio, or video. This processor extracts the relevant features from the multimodal data and converts them into a format that can be consumed by the model.

On the output path, the `Renderer` performs the inverse transformation: it *detokenizes* generated tokens back into human-readable text for return to the user. The Renderer also handles streaming outputs, progressively sending generated text to users as tokens are produced rather than waiting for the entire response to complete.

Engine Client. The `Engine Client` operates on a token-in-token-out abstraction, serving as the interface between the front-end components and the inference engine. It forwards tokenized requests to the `Engine Core` and receives generated tokens in return, which it then passes back upstream to the Renderer.

This clean abstraction serves multiple purposes. First, it decouples the input/output processing from the core inference logic, allowing each to evolve independently. Second, it enables advanced users to build custom API servers directly atop the `Engine Client`, bypassing the built-in interfaces while still benefiting from vLLM's optimized inference backend. This is particularly valuable for users who need custom input processing, specialized output formatting, or integration with proprietary systems. Third, the abstraction facilitates testing and debugging by providing a well-defined boundary where inputs and outputs can be inspected.

Engine Core. The `Engine Core` is the central coordination layer of vLLM, orchestrating all aspects of request processing and resource management. It has three critical components:

1. The *Scheduler* determines the order and batching of requests to maximize throughput while respecting latency constraints. It makes decisions about which requests to process at each step, how many tokens to process for each request, and when to preempt requests to make room for others.

2. The *KV Cache Manager* efficiently allocates and manages GPU memory for KV caches using PagedAttention [46]. It tracks which KV blocks are allocated to which requests, handles block sharing for prefix caching, and ensures that memory is reclaimed when requests complete.

3. The *Executor* dispatches computation to distributed workers and collects their results. It handles the communication between the Engine Core and the worker processes, ensuring that inputs are delivered and outputs are collected efficiently.

Workers. At the execution layer, one or more `Worker` processes perform the actual model inference. Each worker encapsulates two key components:

Listing 4.1: vLLM's `LLM` Python API.

```python
from vllm import LLM

# Example prompts.
prompts = ["Hello, my name is", "The capital of France is"]

# Create an LLM with HuggingFace model name.
llm = LLM(model="Qwen/Qwen3-VL-8B-Instruct")

# Generate texts from the prompts.
outputs = llm.generate(prompts)
```

1. A *Model* instance that executes forward passes. The model takes tokens, positions, and attention metadata as input and produces logits as output.

2. A *Sampler* that selects output tokens based on the model's logits and the specified sampling parameters. The sampler supports various sampling parameters including temperature, top-p [36], top-k, and frequency penalties.

This architecture naturally supports distributed execution across multiple GPUs or nodes. Each GPU runs a separate worker process, and the workers coordinate through the Executor to process requests in parallel. This enables vLLM to scale to models that exceed the memory capacity of a single GPU through techniques such as tensor parallelism [89] and pipeline parallelism [39, 61].

Architectural Influence. vLLM's architecture clearly defines the responsibilities of each component and enforces a clean separation of concerns. This modular design has proven highly influential in the field and has become a reference architecture for many modern LLM inference engines such as TensorRT-LLM [68] and SGLang [115].

4.2 Interface

A well-designed interface is crucial for the adoption and usability of any software system. For an inference engine, the interface must balance simplicity (to lower the barrier to entry) with expressiveness (to enable advanced use cases). vLLM addresses this tension by providing two tiers of interfaces: a minimal end-user interface that abstracts away all complexity, and a detailed developer interface that enables customization and extension.

4.2.1 End User Interface

For ease of use, vLLM provides a clear and minimal interface for end users. As shown in Listing 4.1, the interface requires only the minimum necessary information: the model name and input prompts. With just these two pieces of information, vLLM is ready to perform inference. This stands in contrast to other inference engines such as TensorRT-LLM [68], which require extensive pre-processing steps to convert model weights and compile the inference engine before use.

Similarly, vLLM provides a command-line interface for online server deployment:

```
vllm serve <model-name>
```

This single command starts an OpenAI-compatible API server that can immediately begin serving requests. No configuration files, weight conversion scripts, or compilation steps are required.

The minimal API is made possible through several careful design decisions:

Native HuggingFace Integration. vLLM maintains tight integration with the HuggingFace ecosystem [104]. Since most open-source models are available through HuggingFace, vLLM leverages this ecosystem to understand model semantics—including model architecture, weight format, tokenizer, and chat templates—without requiring users to directly provide this information. When a user specifies a HuggingFace model name, vLLM automatically downloads the model weights and configuration, initializes the tokenizer, and sets up the chat template. This integration significantly reduces the burden on users and ensures compatibility with the vast majority of publicly available models.

Automatic Compilation and Memory Profiling. vLLM automatically performs compilation and memory profiling on the fly, eliminating any pre-processing steps. When a model is first loaded, vLLM uses `torch.compile` [6] to optimize the model's computational graph and generate efficient fused Triton [95] kernels. This compilation happens transparently to the user and the results are cached for subsequent runs.

Memory profiling is also automatic: vLLM runs the model with dummy inputs to measure peak memory usage, then calculates how much GPU memory remains available for the KV cache. This approach ensures that vLLM uses as much memory as possible for the KV cache (maximizing batch size and throughput) without exceeding available memory and causing out-of-memory errors. The profiling accounts for model weights, activation memory, and CUDA context overhead, providing an accurate estimate of available KV cache capacity.

Built-in Heuristics. vLLM includes built-in heuristics and robust algorithms that deliver good performance without extensive tuning. These heuristics allow users to achieve competitive performance immediately, while still providing configuration options for users who want to fine-tune behavior for their specific workloads.

4.2.2 Developer Interface

While the end-user interface prioritizes simplicity, vLLM also provides a well-defined developer interface that enables customization and extension. This interface achieves separation of concerns by clearly defining the responsibilities of each component and the contracts between them. We describe the three most important developer-facing interfaces below.

KV Cache Manager Interface. The KV Cache Manager is responsible for managing GPU memory for KV caches. As shown in Listing 4.2, it defines two main APIs: `get_computed_blocks` and `allocate_slots`.

Listing 4.2: vLLM's KV Cache Manager API.

```python
class KVCacheManager:

    # For prefix caching
    def get_computed_blocks(
        self,
        request: Request,
    ) -> KVBlocks | None

    # For on-demand block allocation
    def allocate_slots(
        self,
        request: Request,
        num_new_tokens: int,
        computed_blocks: KVBlocks | None,
    ) -> KVBlocks | None
```

When a new request arrives, the scheduler first calls `get_computed_blocks` to check for prefix cache hits. This method examines the request's input tokens and determines whether any prefix of the input matches previously computed KV cache blocks. If a cache hit is found, the method returns references to the cached blocks, allowing the scheduler to skip computation for the corresponding portion of the input. The KV Cache Manager also increments the reference count of the returned blocks to prevent them from being evicted while in use.

After checking for cache hits, the scheduler calls `allocate_slots` to allocate new blocks for the portion of the input that was not cached. This method reserves memory for storing the KV cache that will be computed during the forward pass. For continuing requests (those that have already started generating), the scheduler also calls `allocate_slots` at each step to secure memory for the newly generated token's KV cache.

This interface design provides several benefits. First, it separates the caching policy from the scheduling logic, allowing each to be modified independently. Second, it enables different implementations of prefix caching (e.g., using different hash algorithms or eviction policies) without changing the scheduler. Third, it allows third-party systems like LMCache [58] to implement their own KV cache management as a *KV Connector*, storing KV caches on external storage systems while maintaining compatibility with vLLM's scheduler.

Scheduler Interface. The scheduler is responsible for deciding which requests to execute at each step and how many tokens to process for each request. As shown in Listing 4.3, its primary entry point is the `schedule` method, which returns a `SchedulerOutput` describing the scheduling decision.

The key insight in the `SchedulerOutput` design is that it represents the scheduling deci-

Listing 4.3: vLLM's Scheduler API.

```python
class Scheduler:

    def add_request(self, request: Request) -> None

    def schedule(self) -> SchedulerOutput

    def update_from_output(self, sampled_tokens: list[int]) -> None


class SchedulerOutput:
    scheduled_req_ids: list[str]
    num_scheduled_tokens: list[int]
    num_computed_tokens: list[int]
```

sion in a highly generic form: a mapping from request IDs to the number of tokens scheduled. This abstraction is deliberately minimal, specifying *what* should be executed without prescribing *how* the execution should be performed. This design decision has several important implications:

1. **Decoupling**: The scheduling policy is completely decoupled from the execution backend. The scheduler does not need to know the details of how the model is executed, and the execution backend does not need to understand the scheduling policy.

2. **Unified optimization**: Various inference optimizations described in Section 4.4 can all be expressed through this single interface, enabling their combination without special-case handling.

The scheduler also provides `add_request` for submitting new requests and `update_from_output` for updating request state after each generation step. These methods complete the request lifecycle management, allowing the scheduler to track request progress and make informed scheduling decisions.

Attention Backend Interface. The landscape of attention mechanisms is highly fragmented. New variants appear frequently, including Sliding Window Attention [10], Multi-Head Latent Attention [56], and local chunked attention [3]. Furthermore, models may require specific variants such as attention sinks [1] or quantized KV caches. No single kernel library supports the union of all these features.

To address this fragmentation, vLLM introduces the `AttentionBackend` interface. This abstraction layer standardizes the interaction between vLLM and attention kernel implementations. The interface specifies:

1. Standard input tensors: query, key, and value tensors with well-defined shapes and memory layouts.

2. An associated `AttentionMetadata` class that carries implementation-specific auxiliary data. This metadata can include information such as sequence lengths, block tables, or any other data required by the specific attention implementation.

3. Standard output tensor: the shape and layout of the attention output tensor.

Through this interface, vLLM supports a pluggable backend system that currently integrates FlashAttention [23], FlashInfer [109], FlashMLA [24], and FlexAttention [25]. Each backend provides optimized kernels for specific hardware and model configurations. The abstraction ensures that vLLM can rapidly adopt the fastest available kernels for any given hardware or model architecture, without requiring changes to the rest of the system.

4.3 Model Authoring

Supporting new models rapidly and reliably is critical for any inference engine. The pace of innovation in LLM architectures is extraordinary, with new models being released on a near-weekly basis. An inference engine that cannot quickly adapt to new models will rapidly become obsolete.

Supporting new models typically involves a *porting* process. Open-source models are typically developed in the model vendor's private training framework, then converted to a standard format (usually HuggingFace Transformers [104]) for public release. This porting process is labor-intensive and error-prone. Subtle differences in numerics, attention implementations, normalization layers, and activation functions can lead to incorrect outputs that are difficult to diagnose. For example, different implementations of RMS normalization [111] may use different epsilon values or apply the normalization in slightly different ways, leading to numerical divergence.

The goal of vLLM's model authoring framework is to make this porting process easier and more robust to human errors. We achieve this through minimal interface requirements and automatic optimization.

4.3.1 Model Code

To minimize friction in the model porting process, vLLM allows substantial freedom in writing model code. In essence, there are only two requirements, as shown in Listing 4.4:

1. The highest-level `Model` class must implement a `forward` method that takes `input_ids`, `positions`, and `input_embeds` as input and returns the last hidden states as output. The `intermediate_tensors` argument is used for pipeline parallelism (discussed below).

2. Attention operations must be defined through vLLM's `Attention` API rather than standard PyTorch attention or any other implementation. This API includes hooks to gather all necessary information for vLLM's KV cache management.

Listing 4.4: vLLM's model code example.

Use vLLM's attention implementation

```python
from vllm.attention import Attention

class Layer(nn.Module):

    def __init__(self, ...):
        self.qkv_proj = nn.Linear(...)
        self.attn = Attention(...)
        ...

    def forward(self, x):
        qkv = self.qkv_proj(x)
        q, k, v = qkv.split(...)
        attn_output = self.attn(q, k, v)
        ...

class Model(nn.Module):

    def forward(
        self,
        input_ids: torch.Tensor,
        positions: torch.Tensor,
        input_embeds: torch.Tensor,
        intermediate_tensors: IntermediateTensors | None,
    ) -> torch.Tensor
```

With these minimal requirements, model developers have complete freedom in defining the rest of the model. They can use any PyTorch [78] operator, any custom CUDA kernel, and any control flow structure. This flexibility significantly reduces the burden of the porting process: instead of rewriting the model code entirely, developers can often plug in their existing implementation with only minor modifications to wrap the attention layer and adjust the forward signature.

By preserving the original model implementation, developers keep all the model details—numerics, activation functions, normalization variants—intact. This minimizes the risk of introducing bugs during porting. Indeed, [102] has demonstrated that vLLM's general model API even enables the unification of model code used for inference and training with TorchTitan [54], achieving bitwise equivalence between inference and training outputs. This level of reproducibility dramatically improves confidence in the correctness of model implementations.

Operation Fusion. One important aspect of model performance is ensuring that operations are fused appropriately. LLMs contain several memory-bound operations, including

RMS normalization [111], skip connections [35], and SiLU activations [85]. For optimal performance, these memory-bound operators should be fused with adjacent operators to reduce memory I/O costs.

However, operator fusion often compromises the simplicity and flexibility of model authoring. Fusion typically occurs across semantic boundaries: for example, one can fuse pre-LayerNorm with the skip connection from the previous layer. While this fusion improves performance, it breaks the clean layer abstraction, making the model code harder to read and maintain. Moreover, training frameworks may use different fusion strategies than inference frameworks, since training must also consider the impact of fusion on the backward pass. This divergence makes it difficult to maintain a single model implementation that works correctly in both contexts.

To address this challenge, vLLM leverages `torch.compile` [6] to automatically fuse operators. Rather than requiring manual fusion in the model code, vLLM allows the compiler to identify and fuse eligible operations. To enable compilation of the entire model and allow cross-layer fusion, vLLM provides supporting infrastructure. For example, vLLM encapsulates the `Attention` operator into a custom op that is opaque to the compiler, hiding its dynamic components (e.g., attention metadata) from the compiler's analysis. As a result, the rest of the model becomes a sequence of tensor operators that can be easily analyzed and optimized by the compiler.

This approach allows vLLM to keep model code in its original, readable form while still achieving high performance through automatic optimization.

Model Sharding. vLLM supports a comprehensive set of model parallelisms for distributed execution, including:

- **Tensor parallelism** [89]: Splits individual layers across multiple GPUs, with each GPU computing a portion of each layer's output.
- **Pipeline parallelism** [39, 61]: Assigns different layers to different GPUs, forming a pipeline where activations flow from one GPU to the next.
- **Expert parallelism** [47]: For Mixture-of-Experts [86] models, assigns different experts to different GPUs.
- **Context parallelism** [106, 12]: Splits the sequence dimension across GPUs, enabling processing of very long sequences.

For maximum flexibility, vLLM implements these parallelisms with minimal modifications to the model code. Tensor parallelism is implemented through the `ColumnParallelLinear` and `RowParallelLinear` APIs, following the approach established by fairseq [41]. These drop-in replacements for standard linear layers automatically partition weights and handle all-reduce communication. Pipeline parallelism is implemented by injecting pre- and post-hooks that send and receive `IntermediateTensors` (as shown in Listing 4.4) at pipeline stage boundaries. Context parallelism and expert parallelism are implemented transparently

Listing 4.5: vLLM's Renderer API.

```
class Renderer:

    def render_prompt(
        self,
        chat_conversation: list[ChatMessage] | str,
    ) -> tuple[list[int], list[MultimodalData]]
```

Content and Expert parallelism implemented like this

within the `Attention` and `FusedMoE` abstractions, respectively. These implementations encapsulate all the necessary communication and synchronization logic, so model authors need not be aware of the parallelism strategy.

This approach to model sharding minimizes the impact on model code while providing comprehensive support for distributed execution.

4.3.2 Renderer

Another important aspect of model authoring is defining how model inputs and outputs are parsed and formatted. Traditionally, inputs to LLMs were simple text strings that could be tokenized directly. However, modern models have become significantly more complex:

- Models often have specific *chat templates* that define how conversations should be formatted, including special tokens for system prompts, user messages, and assistant responses.
- Many models support *tool calling*, which requires specific formats for representing available tools and parsing tool invocations from model output.
- *Multimodal models* require specialized preprocessing for images, audio, and video, including tokenization of non-text inputs and injection of special placeholder tokens.

Different models may implement these features in incompatible ways, making it challenging to support the full diversity of models with a single, fixed preprocessing pipeline.

To accommodate this diversity, vLLM introduces the `Renderer` API, inspired by OpenAI Harmony [76]. As shown in Listing 4.5, the Renderer provides a single high-level `render_prompt` API that manages the end-to-end translation from user-level structured inputs to engine-level tokens.

Within the `render_prompt` method, model developers have complete freedom to implement their custom input processing logic. This may include:

- Traditional string-based multimodal processing rules, such as those used in Qwen-VL models [9], where special placeholder strings indicate where multimodal features should be inserted.

- Token-based tool parsing logic, such as that used in GPT-OSS [1], where tool definitions and invocations are represented as structured token sequences.
- Custom chat templates that differ from standard formats, enabling support for models with unique conversation structures.

Model developers can implement the Renderer using HuggingFace utilities or from scratch. This flexibility ensures that vLLM can support the full diversity of model input formats without requiring changes to the core engine.

4.4 Inference Optimizations

To deliver optimal throughput and latency, vLLM must integrate a wide range of inference optimization techniques that are being continually proposed by both academia and industry. These techniques include:

- **Chunked prefills** [2]: Split long prompts into chunks processed over multiple steps to bound latency and improve throughput.
- **Prefix caching** [115]: Identify and reuse KV caches from requests with shared prefixes to skip redundant computation and KV cache memory.
- **Prefill disaggregation** [116]: Separate prefill and decode onto different workers to avoid interference.
- **Speculative decoding** [49]: Use a small draft model to propose multiple tokens that are then verified in parallel by the target model, accelerating generation.

However, incorporating these diverse optimization techniques into a single serving engine is far from trivial. Each technique is often proposed and implemented within its own standalone context, with limited consideration for how it should interoperate with other optimizations. Without careful design, their implementations tend to become heavily fragmented, with separate code paths for each combination of features. This fragmentation increases overall system complexity and limits the ability to realize the *combined* benefits of multiple optimizations working together.

To address this challenge, vLLM introduces a **unified scheduling framework** designed to represent and compose different optimization techniques under a single coherent abstraction.

4.4.1 Limitations of the Traditional View

Traditionally, LLM inference has been conceptualized as two distinct phases: the *prefill* phase, which processes user prompts at once, and the *decode* phase, which generates new tokens one by one. The prefill phase has been characterized as a one-time, computationally heavy process that consumes all input tokens in the prompt in a single forward pass. The decode phase, on the other hand, has been characterized as a repeating, memory-bound process that takes a single token as input at each step.

While this two-phase view is conceptually intuitive, it has historically led to complex, special-case scheduling logic when combined with various optimizations. Consider the following scenarios:

- **Speculative decoding**: The model processes multiple tokens in each forward pass during decode. Does this belong to the "prefill" phase because it processes multiple tokens, or the "decode" phase because it is generating output?
- **Prefix caching**: A prefill input may get a prefix cache hit for all but one prompt token. When processing this single token, should it be considered "prefill" because it is the first step for this request, or "decode" because it processes a single token?

These scenarios reveal "grey areas" that cannot be easily categorized into the traditional prefill and decode phases. When inference optimizations are implemented on top of this rigid two-phase view, systems accumulate distinct code paths for each operational mode: prefill batches, micro-batches, decode loops, prefix cache hits, speculative verification, and so on, with each code path requiring its own tracking variables and heuristics. This fragmentation increases system complexity, limits the composability of multiple optimizations, and leads to poor user experience due to a confusing support matrix documenting which combinations of features work together and which do not.

4.4.2 Unified Representation

Our key insight is that the core complexity does not arise from the optimization techniques themselves, but rather from the lack of a common representation of request progress across heterogeneous inference operations. The traditional two-phase view treats prefill, decode, caching, preemption, and speculative execution as fundamentally different computational modes, each requiring its own code path and scheduling logic.

In contrast, we observe that all these behaviors can be expressed uniformly by tracking how far a request has progressed through its sequence of tokens—regardless of whether those tokens originate from the prompt, from cached computation, or from the model's own generation. This insight forms the basis of our unified representation.

The unified scheduling representation models the progress of each request using two scalar variables: `num_total_tokens` and `num_computed_tokens`. These variables encode the entire computational state of a request, and their evolution over time captures the full spectrum of behaviors required for LLM inference.

Definition of `num_total_tokens`. The variable `num_total_tokens` is an integer representing the full length of the token sequence associated with a request, i.e., the prompt length plus the output length:

$$\texttt{num\_total\_tokens} = |\texttt{prompt\_token\_ids}| + |\texttt{output\_token\_ids}|.$$

For a new request, `num_total_tokens` is initialized to the prompt length, since `output_token_ids` is empty. As the model generates new output tokens, this value is incremented accordingly. This variable represents the "target"—how many tokens exist in the complete sequence.

Definition of `num_computed_tokens`. The variable `num_computed_tokens` tracks how many of the `num_total_tokens` have been processed by the model. Equivalently, it represents the number of tokens whose KV cache has been computed and stored:

$$0 \leq \texttt{num\_computed\_tokens} \leq \texttt{num\_total\_tokens}.$$

For a new request without prefix cache hits, this value starts at 0. It grows over time as tokens are scheduled and processed by the model. This variable represents the "progress"—how far we have come toward processing the complete sequence.

Scheduling Transition Rule. With these two variables, we can formally define the state update rules that govern scheduling. At each scheduling step, the scheduler selects a number of tokens Δ to process for each request, subject to the constraint:

$$0 < \Delta \leq \texttt{num\_total\_tokens} - \texttt{num\_computed\_tokens}.$$

That is, we can process any positive number of tokens up to the number remaining. After processing, the system updates:

$$\texttt{num\_computed\_tokens} \leftarrow \texttt{num\_computed\_tokens} + \Delta.$$

If the model generates new tokens (which occurs when `num_computed_tokens` is equal to `num_total_tokens` after the update), their count $L_{\text{generated}}$ is added to the total:

$$\texttt{num\_total\_tokens} \leftarrow \texttt{num\_total\_tokens} + L_{\text{generated}}.$$

These simple update rules are sufficient to express all scheduling behaviors, as summarized in Table 4.1.

4.4.3 Modeling Prefill and Decode

A key strength of the unified representation is its ability to express the traditional prefill and decode phases as special cases of the same general framework.

Prefill as Bulk Progress. For a new request with prompt length L_{prompt}, the variables are initialized as:

$$\texttt{num\_computed\_tokens} = 0, \qquad \texttt{num\_total\_tokens} = L_{\text{prompt}}.$$

Traditional prefill corresponds to scheduling:

$$\Delta = L_{\text{prompt}},$$

a single step that processes the entire prompt at once. After this step, the model generates the first output token, so the system updates:

$$\texttt{num\_total\_tokens} \leftarrow \texttt{num\_total\_tokens} + 1.$$

Table 4.1: Unified representation of various scheduling optimizations. All behaviors are expressed through updates to `num_computed_tokens` and `num_total_tokens`.

Scheduling	**State Update**
Prefill	$\Delta = L_{\text{prompt}}$
Decode	$\Delta = 1$
Preemption	$\texttt{num\_computed\_tokens} \leftarrow 0$
Chunked Prefills	$\Delta < L_{\text{prompt}}$
Streaming Prefills	$\texttt{num\_total\_tokens} \leftarrow \texttt{num\_total\_tokens} + L_{\text{new}}$
Prefix Caching	$\texttt{num\_computed\_tokens} \leftarrow L_{\text{cached}}$
Prefill Disaggregation	$\texttt{num\_computed\_tokens} \leftarrow L_{\text{prompt}}$, $\texttt{num\_total\_tokens} \leftarrow L_{\text{prompt}} + 1$
Speculative Decoding	$\texttt{num\_computed\_tokens} \leftarrow \texttt{num\_computed\_tokens} + \Delta - L_{\text{rejected}}$, $\texttt{num\_total\_tokens} \leftarrow \texttt{num\_total\_tokens} + L_{\text{draft}} - L_{\text{rejected}} + 1$

No explicit "prefill mode" is necessary; prefill behavior emerges naturally from the initial conditions and a bulk scheduling step.

Decode as Incremental Progress. After prefill completes, the state becomes:

$$\underline{\texttt{num\_computed\_tokens} = L_{\text{prompt}}, \qquad \texttt{num\_total\_tokens} = L_{\text{prompt}} + 1.}$$

In this state, the constraint $\Delta \leq \texttt{num\_total\_tokens} - \texttt{num\_computed\_tokens} = 1$ means we can only schedule a single token. During decoding, the system repeatedly:

1. Processes one token: $\Delta = 1$.
2. Increments `num_computed_tokens` by 1.
3. Appends a new generated token: $\texttt{num\_total\_tokens} \leftarrow \texttt{num\_total\_tokens} + 1$.

Decode thus appears as incremental progress through the sequence, with both variables growing by one token after each step.

Preemption as Reset. Request preemption, introduced in Section 3.4.5, frees a request's KV cache to provide space for other requests. In the unified framework, preemption is simply:

$$\underline{\texttt{num\_computed\_tokens} \leftarrow 0,}$$

while `num_total_tokens` remains unchanged.

When the request resumes, we can schedule up to:

$$\underline{\Delta \leq \texttt{num\_total\_tokens} = L_{\text{prompt}} + L_{\text{output}}}$$

tokens to recover its KV cache. This means we can efficiently recompute the KV cache for both prompt and previously-generated output tokens in a single forward pass, rather than regenerating the output tokens one by one.

4.4.4 Capturing Advanced Optimizations

Although advanced inference optimizations modify request states in more complex ways, they can all be expressed as simple updates to `num_computed_tokens` and `num_total_tokens`. We now describe how each major optimization technique maps to this unified representation.

Chunked Prefills. Chunked prefills [2] is a scheduling technique that splits prompt inputs into chunks and schedules them over multiple steps, instead of processing the entire prompt in a single step. This technique provides two benefits:

1. By limiting chunks to a fixed size, it bounds the latency of individual scheduling steps, reducing tail latency.

2. By batching prefill chunks together with decode inputs from other requests, it increases GPU compute utilization and improves overall throughput.

In the unified framework, chunked prefills is simply represented as choosing a smaller Δ:

$$\Delta < L_{\text{prompt}}, \qquad \texttt{num\_computed\_tokens} \leftarrow \texttt{num\_computed\_tokens} + \Delta,$$

while `num_total_tokens` remains fixed at L_{prompt} since no token is generated until the entire prompt is processed. The scheduler can choose any Δ (up to the remaining tokens) and can repeat this over multiple steps until `num_computed_tokens` reaches `num_total_tokens`. No special "chunked prefill mode" is needed; the unified representation directly supports partial progress through the sequence.

Automatically enabled in V1

Streaming Prefills. Input streaming is a technique that delivers prefill inputs to the inference server incrementally, rather than providing the entire input at once. This approach is particularly valuable for inputs that are generated in temporal order, such as audio and video streams. For example, when a user is speaking, input streaming allows the inference server to begin processing (prefilling) the audio as it arrives, without waiting for the utterance to complete. This pipelining of input reception with computation reduces time-to-first-token (TTFT) latency, as the model can begin computing the KV cache before the full input is available.

The unified framework naturally accommodates input streaming through a simple update rule. When new input tokens arrive, the system increments:

$$\texttt{num\_total\_tokens} \leftarrow \texttt{num\_total\_tokens} + L_{\text{new}},$$

where L_{new} is the number of newly received tokens. Since `num_computed_tokens` remains unchanged while `num_total_tokens` increases, the difference between these variables grows by L_{new}. By the scheduling transition rule, this makes the new tokens eligible for scheduling in subsequent steps—precisely the desired behavior.

Prefix Caching. Prefix caching [115] opportunistically reuses KV caches from previous requests when a new request shares the same prefix. For the prefix that hits the cache, the scheduler can skip computation and directly use the cached KV values through the block tables in PagedAttention [46].

The skipping logic in prefix caching is elegantly expressed in the unified framework. If a new request's cached prefix length is L_{cached}, the scheduler initializes and schedules:

$$\texttt{num\_computed\_tokens} \leftarrow L_{\text{cached}}, \qquad \Delta = L_{\text{prompt}} - L_{\text{cached}}.$$

The initialization of `num_computed_tokens` to L_{cached} reflects that the first L_{cached} tokens are already "computed" (their KV cache is available), and the scheduling decision processes only the remaining tokens.

Uncached prompt can be chunked

Furthermore, the unified framework naturally enables combining chunked prefills with prefix caching: the scheduler can choose $\Delta < L_{\text{prompt}} - L_{\text{cached}}$ to process the uncached portion of the prompt in multiple chunks.

Prefill Disaggregation. Prefill disaggregation [116] decouples prefill and decode phases by processing them on separate, dedicated workers. When a new request arrives, a prefill worker processes its prompt, generates the KV cache, and samples the first output token. The request and its KV cache are then transferred to a decode worker, which continues generation. This separation alleviates prefill-decode interference, achieving stable performance for both phases at the cost of KV cache transfer latency.

From the scheduling perspective, prefill disaggregation can be viewed as a variant of prefix caching where the "cache" is located on a remote GPU rather than locally. When a request is transferred from the prefill worker to a decode worker, its state is initialized as:

$$\texttt{num\_computed\_tokens} \leftarrow L_{\text{prompt}}, \qquad \texttt{num\_total\_tokens} \leftarrow L_{\text{prompt}} + 1.$$

Note that `num_total_tokens` exceeds the prompt length by one because the prefill worker has already sampled the first output token. With these initial values, the decode worker naturally continues performing decode by incrementing both variables as usual.

The unified framework even supports combining local prefix caching with prefill disaggregation. If the decode worker happens to have cached KV for L_{cached} tokens of the prompt, it only needs to receive the KV cache for the remaining $L_{\text{prompt}} - L_{\text{cached}}$ tokens from the prefill worker.

For prefill workers, the unified framework applies identically to standalone prefill—they benefit from chunked prefills and local prefix caching. The only difference is that generation stops after sampling the first output token, and the KV cache is transferred rather than retained locally.

Speculative Decoding. Speculative decoding [49] accelerates generation by using a small, fast *draft model* to predict the next several tokens, then verifying these predictions with the target model in parallel. The draft model generates a sequence of L_{draft} candidate tokens. The target model then processes all draft tokens in a single forward pass, computing the probability distribution it would assign to each position. Finally, a rejection sampling procedure determines which draft tokens to accept based on comparing the draft and target model distributions.

In the unified framework, speculative decoding is represented as generating multiple tokens at each step:

$$\texttt{num\_total\_tokens} \leftarrow \texttt{num\_total\_tokens} + L_{\text{draft}} + 1,$$

where L_{draft} is the number of draft tokens and the +1 accounts for the token sampled by the target model. This allows:

$$\Delta \leq L_{\text{draft}} + 1,$$

in contrast to standard decode where Δ is constrained to 1.

However, we must account for rejected draft tokens. After verification, some draft tokens may be rejected, requiring a correction to the state. The complete update rules are:

$$\texttt{num\_total\_tokens} \leftarrow \texttt{num\_total\_tokens} + L_{\text{draft}} - L_{\text{rejected}} + 1,$$

$$\texttt{num\_computed\_tokens} \leftarrow \texttt{num\_computed\_tokens} + \Delta - L_{\text{rejected}},$$

where $L_{\text{rejected}} \in [0, L_{\text{draft}}]$ is the number of rejected tokens. The subtraction of L_{rejected} corrects for having tentatively advanced the state before knowing the verification outcome.

☆ The unified framework naturally enables combining speculative decoding with other optimizations. For example, if the draft sequence is very long, the scheduler can process and verify draft tokens in chunks across multiple steps.

4.4.5 Limitations

While the unified framework provides a flexible foundation for implementing and combining inference optimizations, it has certain limitations.

Dynamic and Parallel Decoding. The framework assumes that output tokens are generated as a linear sequence, which does not naturally accommodate dynamic or parallel decoding algorithms. For example, beam search [103] dynamically creates and prunes multiple parallel beams, with the number of active sequences changing at each step. Similarly, some draft models for speculative decoding [60, 52] propose a tree of draft tokens to maximize the number of accepted tokens. Such dynamic, branching structures are difficult to represent using two scalar variables.

Overhead for Simple Cases. The unified representation provides value when the model implementation is flexible enough to handle arbitrary scheduling decisions. However, for less mature models or hardware platforms that do not support all optimization techniques, the unified framework adds cognitive overhead by making simple scenarios appear more complex than necessary. For example, if a platform does not support chunked prefills, the simplistic traditional prefill-decode view from Section 4.4.1 may be clearer. In such cases, the unified framework's generality can obscure what is fundamentally straightforward.

Algorithm 1 FIFO Scheduling with Token Budget

```
Require: FIFO-ordered request queue $Q$
Require: Token budget $B$
Ensure: Scheduled request set $S$ with token counts
 1: $S \leftarrow [\,]$                                          ▷ Initialize empty schedule
 2: for each request $r$ in $Q$ do
 3:     $\Delta_{\max} \leftarrow r.$num_total_tokens $- r.$num_computed_tokens    ▷ Max schedulable
 4:     $\Delta \leftarrow \min(\Delta_{\max}, B)$                      ▷ Respect budget
 5:     if $\Delta > 0$ then
 6:         append $(r, \Delta)$ to $S$                ▷ Schedule request with token count
 7:         $B \leftarrow B - \Delta$                              ▷ Consume budget
 8:     end if
 9:     if $B = 0$ then
10:         break                                          ▷ Budget exhausted
11:     end if
12: end for
13: return $S$
```

4.5 Scheduling Policy

While Section 4.4 provides a unified framework to represent different optimization techniques, it does not specify how many tokens should be scheduled for each request at each step. The unified framework establishes only the *bounds*—the maximum number of tokens that can be scheduled for a request. The *policy* determines concrete scheduling decisions within these bounds.

This section describes scheduling policies that determine which requests to execute, in what order, and how many tokens to process for each.

4.5.1 First-In First-Out Scheduling

vLLM's simplest scheduling policy is First-In First-Out (FIFO), described in Algorithm 1. Under FIFO, requests are processed in the order they arrive. The scheduler maintains a queue of requests ordered by arrival time. At each step, it iterates through the queue from earliest to latest, computes the maximum number of tokens that could be scheduled for each request ($\Delta_{\max}$), and takes the minimum of this value and the remaining token budget. This process continues until the budget is exhausted.

The token budget B represents the maximum total number of tokens that can be processed in a single forward pass. Scheduling more tokens per step increases throughput by improving arithmetic intensity. However, larger batches also increase per-step latency and peak GPU memory usage for intermediate activations. The token budget thus provides a tunable bound on per-step latency and peak memory usage, allowing users to choose their preferred tradeoff with throughput. A typical value is 8,192 tokens, though this is easily configurable to adapt to specific workloads and hardware constraints.

FIFO scheduling has the advantage of simplicity and predictability: requests complete in the order they arrive, and the implementation is straightforward. However, as is well known from the operating systems literature, FIFO has several significant limitations in the context of LLM inference.

Head-of-Line Blocking. In LLM inference, request prompts have highly variable lengths. A request with a very long prompt may require substantial time to complete its prefill phase, during which it consumes a large portion of the token budget. Under FIFO, this long request blocks all subsequent requests in the queue, even if those requests could be processed quickly. For example, consider a queue containing a request with a 100,000-token prompt followed by a request with a 100-token prompt. Under FIFO, the short request must wait for the long prefill to complete, potentially experiencing latency many times greater than its own processing time. This head-of-line blocking significantly degrades tail latency.

No Prioritization. FIFO treats all requests equally, regardless of their importance or urgency. In many deployment scenarios, different requests have different priority levels. For example, interactive chat requests may require lower latency than batch processing jobs. FIFO provides no mechanism to express or enforce such priority distinctions.

No Cache Awareness. FIFO does not consider how much requests can benefit from prefix cache hits. For example, when a request can achieve a prefix cache hit (e.g., in a multi-turn conversation scenario), scheduling it immediately would leverage the cached KV blocks before they are evicted so that the computation for the prefix can be skipped. FIFO, however, ignores this opportunity entirely.

Ad-hoc Mitigations. vLLM has implemented several ad-hoc heuristics to address the limitations of FIFO. For example, it introduced a `long_prefill_threshold` parameter that limits the number of tokens scheduled for any single request, partially mitigating head-of-line blocking. However, these ad-hoc changes are fragile, difficult to reason about, and challenging to maintain. They address symptoms rather than the underlying limitations of the FIFO model, motivating the need for a more principled scheduling framework.

4.5.2 Shortest Remaining Token First Scheduling

To address the limitations of FIFO scheduling, we propose Shortest Remaining Token First (SRTF) scheduling, inspired by Shortest Remaining Time First scheduling in traditional operating systems. The core idea is to prioritize requests that are closest to the generation of a new token, measured by the number of tokens remaining to be processed.

As described in Algorithm 2, SRTF maintains two request queues: a waiting queue W containing requests that have not yet been scheduled, and a running queue R containing requests that have been scheduled at least once and therefore retain their KV cache. At each step, SRTF selects the request with the minimum number of remaining tokens from both queues until the token budget is exhausted.

A subtle but important detail is that SRTF includes an inner loop to refresh the waiting requests' `num_computed_tokens` and Δ values. This refresh is necessary because scheduling one request can affect the prefix cache hits of other waiting requests. For example, if two requests share the same prompt and one is scheduled in a previous iteration, the remaining

Algorithm 2 Shortest Remaining Token First Scheduling

```
Require: Waiting request queue $W$
Require: Running request queue $R$
Require: Token budget $B$
Ensure: Scheduled request set $S$ with token counts
 1: $S \leftarrow [\,]$                                          ▷ Initialize empty list
 2: $X \leftarrow W \cup R$
 3: while $B > 0$ and $X$ is not empty do
 4:     for each request $r$ in $W$ do
 5:         $r$.num_computed_tokens ← |get_computed_blocks(r)|    ▷ Check prefix cache hit
 6:         $\Delta_r \leftarrow r$.num_total_tokens − $r$.num_computed_tokens    ▷ Update $\Delta_r$
 7:     end for
 8:     $r^* \leftarrow \arg\min_{r \in X} \Delta_r$         ▷ Select request with minimum remaining tokens
 9:     $\Delta \leftarrow \min(\Delta_{r^*}, B)$
10:     pop $r^*$ from $X$ and append $(r^*, \Delta)$ to $S$       ▷ Schedule request
11:     $B \leftarrow B - \Delta$
12:     if $r^* \in W$ then
13:         move $r^*$ from $W$ to $R$                      ▷ Move from waiting to running
14:     end if
15: end while
16: return $S$
```

tokens Δ for the other request may drop substantially (or even to zero) due to the newly available cached KV blocks.

SRTF exhibits several desirable properties and addresses the limitations of FIFO:

Natural Decode Prioritization. SRTF naturally prioritizes requests in the decode phase over those in the prefill phase. This behavior emerges directly from the structure of the problem: decode requests have $\Delta = 1$ by definition, since only one token remains to be computed at each step, whereas prefill requests have $\Delta > 1$. Because SRTF always selects the request with minimum Δ, decode requests automatically take precedence. This prioritization aligns well with user expectations—once a request begins generating tokens, users expect continuous output rather than interruptions caused by new prefill work arriving in the system.

Optimal Time-to-First-Token. SRTF achieves optimal average time-to-first-token (TTFT) by eliminating head-of-line blocking. Rather than forcing short prefill requests to wait behind longer ones, SRTF schedules them in order of remaining work. This strategy mirrors Shortest Remaining Time First scheduling in classical operating systems, which is provably optimal for minimizing average job completion time; the same optimality argument applies to TTFT in the LLM inference setting. Furthermore, SRTF interacts naturally with chunked prefills [2]. Because the policy considers only remaining tokens rather than original prompt length, a long prompt that has been partially processed in previous steps is treated

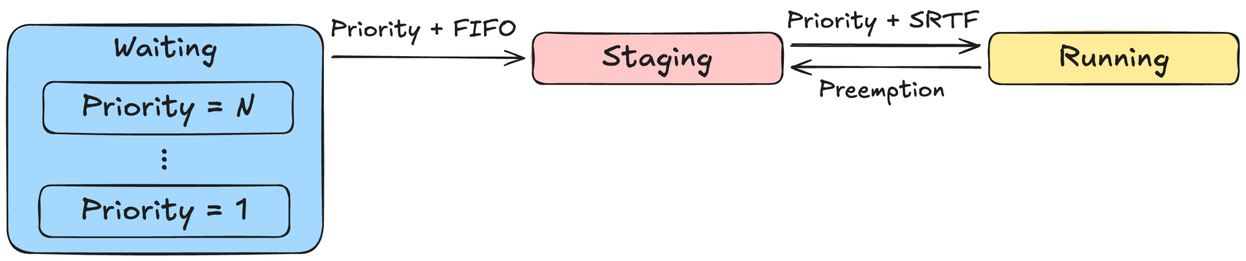


Figure 4.2: Two-level scheduling with First-In First-Out and Shortest Remaining Token First scheduling policies.

as a "short" prefill once few tokens remain, ensuring that nearly-complete prefills are not deprioritized.

Improved Cache Hit Rates. SRTF implicitly improves prefix cache utilization through its selection criterion. Consider two requests with identical original prompt lengths: if one has more prefix cache hits than the other, it will have fewer remaining tokens and thus a smaller Δ. SRTF will therefore schedule the cache-hitting request first, ensuring that cached KV blocks are leveraged promptly. This implicit preference is particularly valuable because cached blocks may be evicted under memory pressure; by scheduling cache-hitting requests sooner, SRTF maximizes the probability that cached data remains available when needed.

However, SRTF also has significant drawbacks:

Starvation. Like Shortest Job First scheduling in operating systems, SRTF can starve requests with long prompts. If short requests arrive continuously, a long request may wait indefinitely, never receiving service.

Scheduling Overhead. SRTF includes an inner loop that updates `num_computed_tokens` for all requests in the waiting queue. This operation can be expensive because the waiting queue can contain an arbitrary number of requests, and each cache hit check involves comparing prefix hashes—a non-trivial computation. The resulting CPU overhead can become a performance bottleneck.

4.5.3 Two-Level Scheduling

To retain the benefits of SRTF while mitigating its drawbacks, we propose **two-level scheduling** that combines SRTF with FIFO. As illustrated in Figure 4.2, the key idea is to introduce a small intermediate queue called the *staging queue*.

The two-level scheduler operates as follows:

1. **First level (FIFO):** Requests move from the waiting queue to the staging queue in FIFO order. Importantly, this transfer occurs in batches: a batch of requests move from waiting to staging *only when* the staging queue becomes empty (i.e., all staged requests have been scheduled).

2. **Second level (SRTF):** Within the staging and running queues, SRTF determines which requests to schedule at each step.

Two-level scheduling effectively addresses the problems of pure SRTF:

Bounded Scheduling Overhead. By limiting the scope of SRTF to the small staging queue (rather than the entire waiting queue), the expensive cache-hit checking and Δ computation are performed over a bounded number of requests. The larger waiting queue uses only simple FIFO ordering, which requires no per-request computation.

Starvation Prevention. Two-level scheduling prevents starvation through two mechanisms. First, FIFO ordering in the first level guarantees that every request eventually moves from waiting to staging. Second, the batch transfer requirement ensures that once a request enters the staging queue, it will be scheduled before any new requests can enter staging. Together, these properties bound the maximum wait time for any request.

Preserved SRTF Benefits. Within the staging queue, requests still benefit from SRTF's optimality properties. Short prefills are prioritized, cache-aware scheduling improves hit rates, and decode requests maintain priority over prefill work.

The staging queue size provides a tunable parameter that trades off between SRTF's optimality (larger staging queue) and bounded overhead (smaller staging queue). In practice, a staging queue of 8 requests provides a good balance for typical workloads.

Two-level scheduling can also be combined with priority scheduling by maintaining separate waiting queues for different priority levels, and by considering priority before applying SRTF. This extensibility allows the framework to support diverse deployment requirements while maintaining its core performance properties.

4.6 CPU Overheads

Another critical factor in LLM inference performance, which is becoming increasingly challenging, is minimizing CPU overheads. Mainstream models are expected to generate more than 100 tokens per second on modern hardware [5]. This means each generation step typically takes less than 10 milliseconds. In other words, even a 1 millisecond CPU overhead could degrade performance by 10% or more.

This poses a significant engineering challenge for vLLM for several reasons:

1. **Python implementation**: vLLM is primarily implemented in Python. Any mis-engineering—unnecessary object allocations, inefficient data structures, or suboptimal control flow—can easily lead to multi-millisecond overheads, causing severe performance degradation.

2. **Complex bookkeeping**: The various inference optimizations described in Section 4.4 and the sophisticated scheduling policies in Section 4.5 require extensive bookkeeping of request states. This bookkeeping adds CPU work that must be performed at each step.

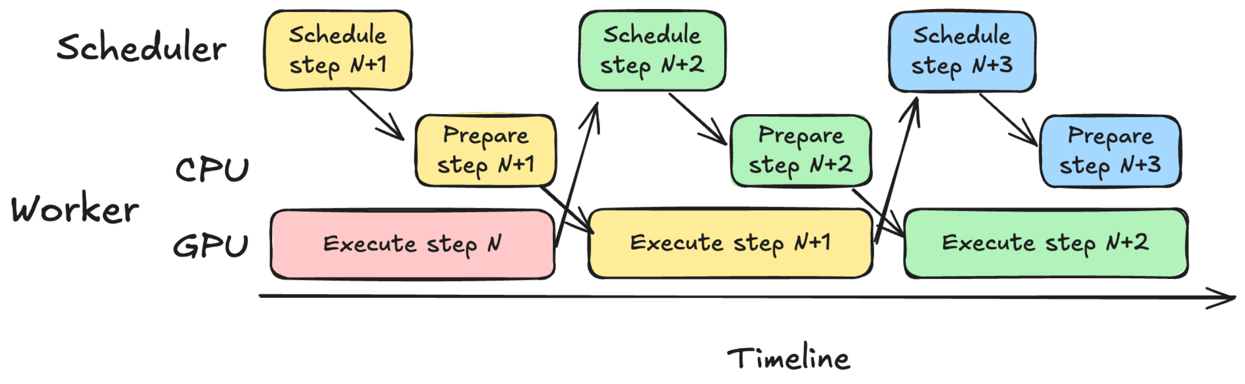


Figure 4.3: vLLM's Asynchronous Scheduling.

3. **Widening GPU-CPU gap**: GPUs are improving at a faster rate than CPUs. While CPU overheads may be acceptable on current hardware, they will become increasingly problematic as next-generation GPUs further reduce per-step latency.

The CPU overheads in vLLM can come primarily from three sources:

1. **Scheduler and KV cache manager**: Making scheduling decisions and managing memory allocation (Section 4.6.1).

2. **Input preparation**: Constructing the tensors and metadata required for each forward pass (Section 4.6.2).

3. **Model Execution**: Running the PyTorch model in eager mode (Section 4.6.3).

Combined, these overheads can consume a majority of the step time—often more than 50% in latency-sensitive scenarios. The following subsections describe how vLLM defeats these overheads through various optimizations.

4.6.1 Asynchronous Scheduling

Traditionally, vLLM adopted a synchronous scheduling model where, at every step, the worker waits for the scheduler to make a decision, then prepares model inputs, and finally executes the model on the GPU. This sequential approach means that the scheduler is on the critical path of model execution and any CPU overhead in scheduling directly adds to the end-to-end latency of each step. As GPUs have become faster, this synchronous model has become a critical performance bottleneck:

- The scheduler must iterate through potentially thousands of requests in the queue at each step. Given Python's interpretation overhead, even a simple loop over this many requests can take milliseconds.

- The KV cache manager allocates new blocks on demand for each request using PagedAttention. This includes updating block status, finding available blocks, and allocating them for all requests in the batch—operations that scale with batch size.
- When new requests are scheduled, the KV cache manager checks for prefix cache hits (through the `get_computed_blocks` API). This may involve hashing input contents and searching cache data structures, adding further overhead.

To address this bottleneck, vLLM implements **asynchronous scheduling**, inspired by NanoFlow [118]. The key idea is to overlap scheduling work with model execution on the GPU. The scheduler operates one step ahead: while the worker executes step N on the GPU, the scheduler makes the scheduling decision for step $N+1$. The outputs of step N are only reflected in the scheduling decision for step $N+2$.

As illustrated in Figure 4.3, this approach hides the scheduler overhead behind GPU computation, minimizing the time the GPU spends stalled waiting for CPU work to complete. The GPU can proceed to the next step immediately after completing the current one, without waiting for scheduling decisions.

Asynchronous scheduling is possible because scheduling decisions typically do not depend on model outputs. The variables `num_computed_tokens` and `num_total_tokens` can be updated based on the scheduling decision alone, without needing to see the actual generated tokens. However, there are some edge cases that require special handling:

Prefix Caching Interaction. With synchronous scheduling, vLLM updates the hash of newly computed KV cache immediately after each step. This enables maximum cache sharing: if request X with prompt $[A, B, C]$ generates token D at step N, and request Y with prompt $[A, B, C, D]$ arrives at step $N+1$, then Y can share the entire KV cache for $[A, B, C, D]$.

With asynchronous scheduling, we schedule Y before receiving D from step N. Therefore, Y can only hit the cache for $[A, B, C]$ but not D. That is, asynchronous scheduling delays the registration of output KV cache by one step, potentially reducing cache hit rates for prompts that end with recently-generated tokens. In practice, this edge case is rare, and the performance benefits of asynchronous scheduling far outweigh this minor reduction in cache effectiveness.

Speculative Decoding Interaction. With speculative decoding, the number of rejected tokens L_{rejected} is not known until verification completes. This creates uncertainty in the exact values of `num_computed_tokens` and `num_total_tokens`.

However, this does not prevent asynchronous scheduling. The key insight is that while the absolute values of the state variables depend on L_{rejected}, the scheduling decision only needs to know Δ, the maximum number of tokens that can be scheduled, which is $L_{\text{draft}} + 1$ regardless of how many were rejected in the previous step. As long as L_{draft} is known in advance—which is the case for most speculative decoding methods like Eagle [52]—asynchronous scheduling remains applicable.

4.6.2 Input Preparation

At every step, vLLM workers must prepare the inputs for the model forward pass. These inputs fall into three categories:

1. Standard LLM inputs: tokens, position IDs, and multimodal embeddings.
2. Attention metadata: block tables (for PagedAttention), sequence lengths, etc.
3. Per-request sampling parameters: temperature, top-k, top-p, etc.

While conceptually simple, preparing these inputs for every forward pass can consume significant CPU cycles if not carefully engineered. Three primary factors contribute to input preparation overhead:

1. **Scale of batched requests.** In high-throughput scenarios, thousands of requests can be batched together in a single forward pass. Naively iterating over each request using a Python loop to construct input tensors easily incurs millisecond-level overhead. For example, if processing 1,000 requests requires even 1 microsecond per request for bookkeeping, the total overhead reaches 1 millisecond—a significant fraction of the 10 millisecond step budget on modern GPUs.
2. **PyTorch dispatch overhead.** Input preparation typically involves many small PyTorch operations: indexing into tensors, concatenating arrays, slicing, and creating new tensors. Each PyTorch operation incurs substantial dispatch overhead as it traverses the framework's dispatcher stack, validates inputs, and allocates output tensors. Profiling shows that even a simple index operation (`tensor[i]`) can take tens of microseconds due to this overhead. When hundreds of such operations occur in tight loops—as is common during input preparation—the overhead accumulates to milliseconds.
3. **Implicit CPU-GPU synchronization.** Certain PyTorch operations can inadvertently trigger CPU-GPU synchronization, forcing the CPU to block until all pending GPU operations complete. A single synchronization point can stall the CPU for the entire duration of GPU computation, completely negating any benefits from pipelining or asynchronous execution.

To address these challenges, vLLM employs three complementary optimization techniques:

1. **GPU-side input preparation.** Rather than preparing inputs on the CPU, vLLM offloads most input preparation logic to the GPU using carefully written Triton kernels. This approach provides two key benefits. First, it exploits GPU parallelism: while a CPU must process requests sequentially, a GPU can prepare inputs for thousands of requests simultaneously using thousands of parallel threads. Second, it completely

bypasses Python interpreter overhead and PyTorch dispatch—the GPU kernel executes native code without any framework involvement. As a result, vLLM can prepare inputs for thousands of requests in microseconds rather than milliseconds.

2. **Unified Virtual Addressing (UVA).** Input preparation is complicated by data locality: some data (e.g., new scheduling decisions) originates on the CPU, while other data (e.g., existing block tables) resides on the GPU. Traditionally, this would require explicit memory copies between CPU and GPU, adding latency and programming complexity.

   vLLM leverages CUDA's Unified Virtual Addressing (UVA) [62] to streamline this data movement. UVA enables GPU kernels to access pre-registered host (CPU) memory pointers directly, as if they were GPU pointers. The CUDA runtime handles the underlying data transfer transparently. This eliminates explicit copy operations from the critical path and allows GPU input preparation kernels to seamlessly access data regardless of its physical location.

3. **Incremental state updates.** vLLM workers maintain persistent state across steps and apply only incremental updates rather than rebuilding inputs from scratch. This optimization exploits a key property of LLM serving workloads: at each step, only a small fraction of requests change state. Most requests in the batch are continuing generation and require only minor updates (e.g., appending one new block ID). Only newly arrived requests require full initialization, and only completed requests require cleanup.

   By maintaining persistent data structures and updating only the changed entries, vLLM avoids redundant work proportional to the total batch size. The per-step overhead becomes proportional to the number of *changed* requests rather than the *total* requests.

Case Study: Block Table Management. To illustrate how these techniques work together, we describe vLLM's management of block tables for PagedAttention. Efficient block table management is essential because this tensor can be large: with a batch size of 1,000 requests and maximum sequence length of 1,000,000 tokens (at 16 tokens per block), the block table contains over 62,500,000 entries and 250 MB in size.

Figure 4.4 illustrates vLLM's approach, which combines all three optimization techniques:

1. **Persistent GPU buffer.** vLLM maintains a persistent block table buffer on the GPU that stores block IDs for all active requests. This buffer persists across steps, avoiding the need to reconstruct it from scratch at each iteration. The buffer is pre-allocated to accommodate the maximum expected batch size, eliminating dynamic allocation overhead during serving.

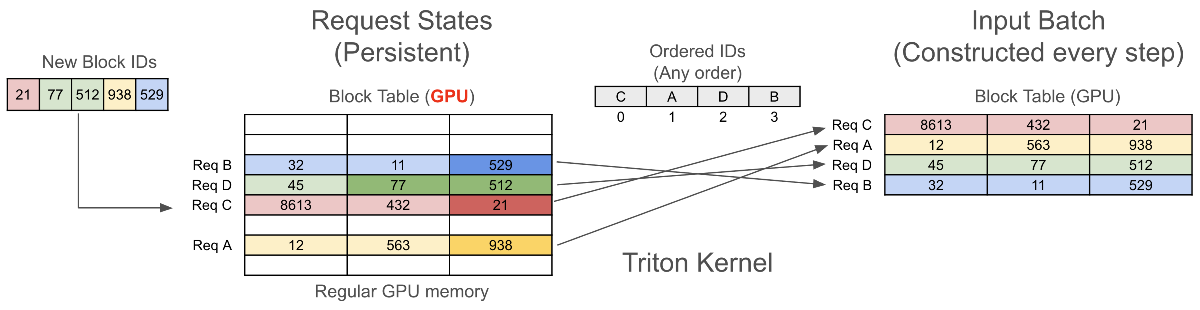


Figure 4.4: Block table management in vLLM.

2. **Incremental updates via UVA.** At each step, the scheduler determines which new blocks have been allocated (for new requests or for continuing requests that need additional blocks). Rather than sending the entire block table, the scheduler sends only the *delta*—the list of new block IDs and their target locations. These updates are written to pinned CPU memory that has been registered with CUDA for UVA access. A lightweight GPU kernel then reads these updates directly from CPU memory and applies them to the persistent buffer.

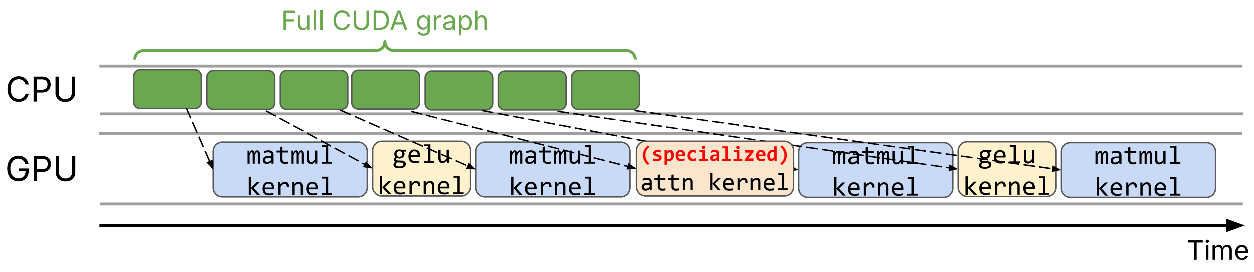
3. **GPU-side gather for input preparation.** When preparing inputs for a forward pass, the attention kernels need block tables arranged according to a certain request order (which may differ from the storage order in the persistent buffer). A GPU kernel performs this gather operation entirely on the GPU: it reads the scheduled request IDs, looks up the corresponding rows in the persistent buffer, and assembles them into the input tensor. No CPU involvement is required for this step.

The combination of these techniques achieves near-zero CPU overhead for block table management. The CPU's only responsibility is to write the small delta of new block IDs to pinned memory; all other operations execute on the GPU in parallel with other work. This approach scales efficiently to large batch sizes and enables vLLM to fully utilize modern high-throughput GPUs.

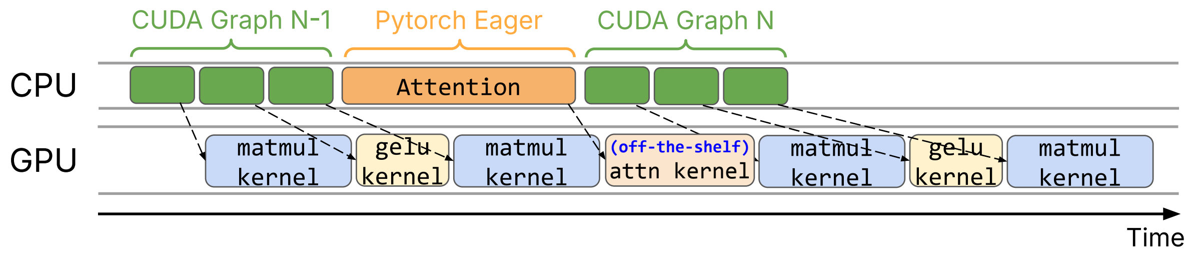
4.6.3 CUDA Graphs

vLLM implements its models in PyTorch, which provides excellent ease of use and extensibility but involves substantial CPU-side overhead during execution. Each PyTorch operation requires Python interpreter execution to process the operator call, framework dispatch logic to route the operation to the appropriate backend, kernel launch preparation, including setting up kernel parameters and allocating output memory.

Profiling reveals that this overhead can consume up a majority of end-to-end inference latency, particularly for small batch sizes where GPU computation completes quickly relative to CPU dispatch time.



(a) Full CUDA graph.



(b) Piecewise CUDA graphs.

Figure 4.5: Comparison of full and piecewise CUDA graph approaches.

CUDA graphs [64] address this overhead by recording a sequence of GPU operations during a *capture* phase, then *replaying* the entire sequence with a single CPU-side launch. The captured graph encodes the complete execution trace—kernel configurations, memory addresses, and inter-kernel dependencies—enabling the GPU to execute the entire model forward pass without returning control to the CPU between operations. This effectively bypasses PyTorch's eager execution overhead.

However, CUDA graphs impose strict constraints:

- **Fixed shapes**: All tensor shapes must remain identical between capture and replay. Graphs cannot handle varying batch sizes or sequence lengths.
- **No CPU operations**: No CPU-side operations may occur within the captured region. This precludes any dynamic control flow or Python logic.
- **Static memory**: Memory addresses used during capture must remain valid during replay. Dynamic allocation within the graph is not supported.

These constraints often conflict with the dynamic nature of LLM inference. Batch sizes vary as requests arrive and complete. Sequence lengths differ across requests and grow during generation. Attention implementations may need to make dynamic decisions based on sequence characteristics.

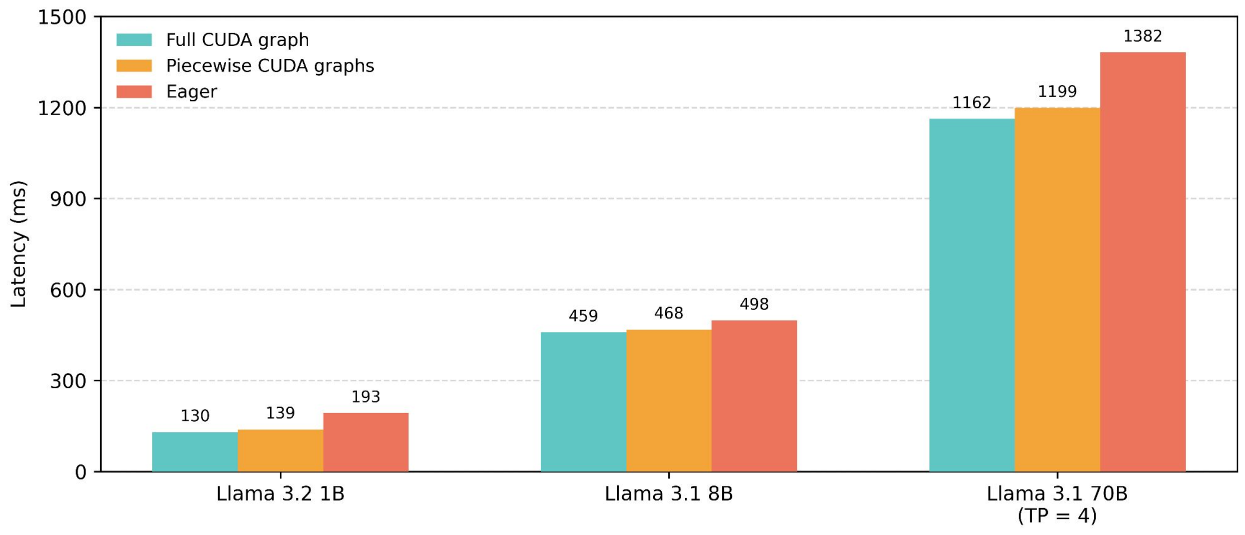


Figure 4.6: Performance comparison of execution modes on H100 GPUs. Batch size is 1, input length is 4000 tokens, and output length is 50 tokens.

Previous systems faced a difficult choice: abandon CUDA graphs entirely and accept eager execution overhead, or develop highly specialized GPU kernels that internalize all decision logic to enable full graph capture. The former sacrifices latency benefits; the latter imposes severe engineering burden and precludes the use of off-the-shelf optimized kernels.

vLLM resolves this tension by introducing **Piecewise CUDA Graphs**. The key insight is that dynamism in Transformer models is not uniformly distributed. The attention mechanism exhibits dynamic behavior—varying sequence lengths, dynamic scheduling decisions, and adaptive kernel selection. In contrast, the surrounding operations—linear projections and feed-forward networks—operate in a stateless, token-wise manner that is fully amenable to graph capture.

Piecewise CUDA graphs partition the model into regions based on the execution characteristics:

1. **Graph-captured regions** encompass stateless operations with fixed shapes and no runtime decisions. These include query/key/value projections, output projections, and the feed-forward network (MLP) and MoE layers.

2. **Eager-executed regions** contain the attention computation, where dynamic scheduling, varying sequence lengths, and adaptive kernel selection preclude static capture.

The execution flow alternates between these regions: a CUDA graph performs all the operations before attention, control returns to PyTorch for eager attention execution, then a second CUDA graph handles the post-attention MLP and the pre-attention projections in the subsequent layer. This pattern repeats for each Transformer layer.

Graph partitioning is implemented using `torch.compile` [6] with custom graph break annotations. The compiler generates separate graph regions that can be independently captured, with explicit boundaries at the attention operations.

Figure 4.6 demonstrates that piecewise CUDA graphs achieve performance comparable to full CUDA graphs while being significantly faster than eager-mode PyTorch in latency-sensitive scenarios. The results show that piecewise graphs successfully eliminate most CPU overhead while preserving the flexibility needed for dynamic attention implementations.

4.7 Conclusion

This chapter presented vLLM, a comprehensive inference engine that addresses the full spectrum of challenges in practical LLM deployment. We described vLLM's design from multiple perspectives: its goals and guiding philosophies, its layered architecture and clean separation of concerns, its minimal yet powerful interfaces for both end users and developers, and its flexible model authoring framework.

A central contribution of this chapter is the unified scheduling framework, which represents diverse inference optimizations—including chunked prefills, prefix caching, prefill disaggregation, and speculative decoding—through a common abstraction based on two scalar variables tracking request progress. This unified representation enables principled composition of multiple optimizations without the fragmentation and special-case handling that plagued earlier systems.

We also described vLLM's techniques for minimizing CPU overhead, which is becoming increasingly critical as GPU performance continues to improve. Asynchronous scheduling, GPU-offloaded input preparation, and CUDA graphs together ensure that vLLM can fully utilize modern GPU hardware without being bottlenecked by CPU-side processing.

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

Large Language Models (LLMs) have fundamentally reshaped the landscape of artificial intelligence, achieving capabilities that seemed like science fiction only a few years ago. Yet as these models have grown in scale and ubiquity, the infrastructure required to serve them efficiently has struggled to keep pace. The central thesis of this dissertation is that efficient LLM inference is not simply a problem of model architecture or hardware acceleration, but fundamentally a *systems* challenge.

We identified that the primary bottleneck in LLM serving is not computation but memory management. The autoregressive nature of decoding, combined with the enormous size of KV caches, leads to memory fragmentation, low utilization, and reduced throughput on modern hardware. To address this challenge, we designed and implemented vLLM, an open-source end-to-end inference engine.

Our contributions span three core dimensions:

5.1.1 Algorithmic Innovation: PagedAttention

We introduced **PagedAttention** (Chapter 3), a novel attention algorithm inspired by virtual memory management. Recognizing that the requirement for contiguous KV cache allocation was the primary source of fragmentation, we decoupled the logical structure of the KV cache from its physical layout. PagedAttention partitions KV data into fixed-size blocks that may reside in non-contiguous memory regions.

This abstraction enables near-zero fragmentation and flexible memory sharing. Complex decoding scenarios—such as beam search and parallel sampling—can share KV blocks for common prefixes, substantially reducing memory usage.

5.1.2 Systems Architecture: The vLLM Engine

Building on PagedAttention, we developed the **vLLM engine** (Chapter 4), a high-throughput, distributed inference system organized around a modular architecture that separates the user interface, scheduling logic, and execution backend.

Key innovations include:

- **Unified Scheduling:** We introduced a unified representation of request progress using two scalar quantities, `num_computed_tokens` and `num_total_tokens`. This abstraction enabled the integration of diverse optimization techniques—chunked prefill, prefix caching, speculative decoding—within a single coherent framework.
- **Scheduling Policies:** We proposed Shortest Remaining Token First (SRTF) scheduling to mitigate head-of-line blocking inherent in FIFO systems. Combined with a multi-level queue, SRTF achieves state-of-the-art time-to-first-token (TTFT) performance without starvation.
- **Zero-Overhead Execution:** To address the widening performance gap between CPU and GPU, we implemented asynchronous scheduling and piecewise CUDA graph execution, removing CPU overhead from the critical path and enabling vLLM to fully utilize modern accelerators.

5.1.3 Open-Source Impact

Beyond its technical innovations, vLLM has become a cornerstone of the open-source AI ecosystem. By adhering to principles of generality, flexibility, and openness, vLLM supports dense Transformers, Mixture-of-Experts architectures, and multimodal systems. The adoption of vLLM's architectural patterns by other inference frameworks further illustrates its community-wide influence.

5.2 Future Directions

Although vLLM addresses many of the core challenges in LLM inference, the field is evolving quickly. As models grow larger, contexts become unbounded, and applications increasingly exhibit autonomous or agentic behaviors, new research opportunities arise.

5.2.1 Stateful Inference for Agentic Workflows

Agentic workloads involve tool usage, iterative loops, and complex dependencies that extend beyond the traditional request–response paradigm. Future inference systems may require explicit support for *stateful inference*, allowing agents to maintain persistent context across long interactions without retransmitting histories.

Promising directions include:

- Cross-replica and cross-cluster KV sharing, where prefill work performed on one engine can be reused by others via shared storage or KV cache connectors.
- Integration with hierarchical or external storage systems (e.g., object stores or specialized KV stores) to spill KV blocks transparently, effectively extending context beyond the memory capacity of a single node.
- Designing cache-consistency models, eviction policies, and security constraints tailored to distributed multi-tenant inference environments.

These challenges, though non-trivial, promise substantial reductions in redundant computation at scale.

5.2.2 Unifying Training and Inference

There is increasing interest in unifying training and inference infrastructure. Recent work [102] demonstrates that vLLM's model authoring interface can integrate with training frameworks such as TorchTitan [54] to achieve bitwise consistency between training and inference. Extending this unification could simplify the path from research prototypes to production systems and enable sophisticated on-policy training loops where models continuously learn from their own inference behavior.

Furthermore, this unification can be particularly important for **continual learning**, where models are trained during deployment. A unified infrastructure collapses might enable online continual learning where inference requests become real-time training signals and replay buffers are populated from live traffic rather than static datasets. Bitwise consistency ensures quantities like log-probabilities computed during training exactly match inference outputs, preventing numerical discrepancies that may corrupt gradient estimates in online learning loops. Combined with parameter-efficient methods like LoRA [37], this this could offer a path toward self-improving systems that continuously learn while preserving base capabilities.

5.2.3 Reliability, Debugging, and Safety

As LLMs enter safety-critical settings, reliability and observability become central concerns. Potential extensions to vLLM include:

- Deterministic or replayable execution modes for debugging.
- Fine-grained tracing of KV usage, scheduling decisions, and per-token latencies for integration with monitoring pipelines.
- Interfaces for model-level safety mechanisms—including content filters, verification models, and tool-use constraints—that interact with scheduling and execution.

Developing these capabilities will require new abstractions that expose internal state without compromising modularity.

5.3 Concluding Remarks

The rapid advancement of LLMs has widened the gap between the computational demands of modern AI and the limitations of contemporary hardware. This dissertation demonstrates that principled systems research provides the bridge across this gap.

vLLM establishes a new standard for efficient inference. Yet it is not a static artifact—it is a foundation. As we move toward trillion-parameter models, effectively infinite context windows, and increasingly autonomous agents, the principles articulated in this dissertation—efficient memory management, flexible abstractions, and unified scheduling—will remain essential.

We hope that vLLM continues to catalyze innovation, enabling the next generation of intelligent systems to operate at the scale, speed, and reliability that modern applications demand.

Bibliography

[1] Sandhini Agarwal et al. “gpt-oss-120b & gpt-oss-20b model card”. In: *arXiv preprint arXiv:2508.10925* (2025).

[2] Amey Agrawal et al. “Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}”. In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pp. 117–134.

[3] Meta AI. *The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation.* 2025. URL: `https://ai.meta.com/blog/llama-4-multimodal-intelligence/`.

[4] Reza Yazdani Aminabadi et al. “DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale”. In: *arXiv preprint arXiv:2207.00032* (2022).

[5] Artificial Analysis. *Comparison of Models: Intelligence, Performance Price Analysis.* URL: `https://artificialanalysis.ai/models`.

[6] Jason Ansel et al. “Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.* 2024, pp. 929–947.

[7] Anthropic. *Introducing Claude Opus 4.5.* Nov. 2025. URL: `https://www.anthropic.com/news/claude-opus-4-5`.

[8] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

[9] Shuai Bai et al. “Qwen2. 5-vl technical report”. In: *arXiv preprint arXiv:2502.13923* (2025).

[10] Iz Beltagy, Matthew E Peters, and Arman Cohan. “Longformer: The long-document transformer”. In: *arXiv preprint arXiv:2004.05150* (2020).

[11] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. “A neural probabilistic language model”. In: *Advances in neural information processing systems* 13 (2000).

[12] Nidhi Bhatia et al. “Helix Parallelism: Rethinking Sharding Strategies for Interactive Multi-Million-Token LLM Decoding”. In: *arXiv preprint arXiv:2507.07120* (2025).

[13] Ond rej Bojar et al. "Findings of the 2016 Conference on Machine Translation". In: *Proceedings of the First Conference on Machine Translation*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 131–198. URL: `http://www.aclweb.org/anthology/W/W16/W16-2301`.

[14] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[15] Marc Brysbaert. "How many words do we read per minute? A review and meta-analysis of reading rate". In: *Journal of memory and language* 109 (2019), p. 104047.

[16] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[17] Tianqi Chen et al. "Training deep nets with sublinear memory cost". In: *arXiv preprint arXiv:1604.06174* (2016).

[18] Wei-Lin Chiang et al. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023. URL: `https://lmsys.org/blog/2023-03-30-vicuna/`.

[19] Aakanksha Chowdhery et al. "Palm: Scaling language modeling with pathways". In: *arXiv preprint arXiv:2204.02311* (2022).

[20] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 613–627.

[21] Daniel Crankshaw et al. "InferLine: latency-aware provisioning and scaling for prediction serving pipelines". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 477–491.

[22] Weihao Cui et al. "DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 183–198.

[23] Tri Dao et al. "Flashattention: Fast and memory-efficient exact attention with io-awareness". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 16344–16359.

[24] DeepSeek. *FlashMLA: Efficient Multi-head Latent Attention Kernels*. URL: `https://github.com/deepseek-ai/FlashMLA`.

[25] Juechu Dong et al. "Flex attention: A programming model for generating optimized attention kernels". In: *arXiv preprint arXiv:2412.05496* (2024).

[26] Jiarui Fang et al. "TurboTransformers: an efficient GPU serving system for transformer models". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 389–402.

[27] FastAPI. *FastAPI*. `https://github.com/tiangolo/fastapi`. 2023.

[28] Pin Gao et al. “Low latency rnn inference with cellular batching”. In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–15.

[29] Amir Gholami et al. “Ai and memory wall”. In: *RiseLab Medium Post* 1 (2021), p. 6.

[30] Github. 2022. URL: `https://github.com/features/copilot`.

[31] Google. 2023. URL: `https://bard.google.com/`.

[32] Google. *A new era of intelligence with Gemini 3*. Nov. 2025. URL: `https://blog.google/products/gemini/gemini-3`.

[33] Arpan Gujarati et al. “Serving {DNNs} like Clockwork: Performance Predictability from the Bottom Up”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 443–462.

[34] Mingcong Han et al. “Microsecond-scale Preemption for Concurrent {GPU-accelerated}{DNN} Inferences”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 539–558.

[35] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[36] Ari Holtzman et al. “The curious case of neural text degeneration”. In: *arXiv preprint arXiv:1904.09751* (2019).

[37] Edward J Hu et al. “Lora: Low-rank adaptation of large language models.” In: *ICLR* 1.2 (2022), p. 3.

[38] Chien-Chin Huang, Gu Jin, and Jinyang Li. “Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1341–1355.

[39] Yanping Huang et al. “Gpipe: Efficient training of giant neural networks using pipeline parallelism”. In: *Advances in neural information processing systems* 32 (2019).

[40] Aaron Hurst et al. “Gpt-4o system card”. In: *arXiv preprint arXiv:2410.21276* (2024).

[41] Meta Inc. *fairseq: Facebook AI Research Sequence-to-Sequence Toolkit written in Python.* `https://github.com/facebookresearch/fairseq`.

[42] Paras Jain et al. “Checkmate: Breaking the memory wall with optimal tensor rematerialization”. In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 497–511.

[43] Norm Jouppi et al. “Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings”. In: *Proceedings of the 50th annual international symposium on computer architecture*. 2023, pp. 1–14.

[44] Jared Kaplan et al. “Scaling laws for neural language models”. In: *arXiv preprint arXiv:2001.08361* (2020).

[45] Tom Kilburn et al. “One-level storage system”. In: *IRE Transactions on Electronic Computers* 2 (1962), pp. 223–235.

[46] Woosuk Kwon et al. “Efficient memory management for large language model serving with pagedattention”. In: *Proceedings of the 29th symposium on operating systems principles.* 2023, pp. 611–626.

[47] Dmitry Lepikhin et al. “Gshard: Scaling giant models with conditional computation and automatic sharding”. In: *arXiv preprint arXiv:2006.16668* (2020).

[48] Brian Lester, Rami Al-Rfou, and Noah Constant. “The power of scale for parameter-efficient prompt tuning”. In: *arXiv preprint arXiv:2104.08691* (2021).

[49] Yaniv Leviathan, Matan Kalman, and Yossi Matias. “Fast inference from transformers via speculative decoding”. In: *International Conference on Machine Learning.* PMLR. 2023, pp. 19274–19286.

[50] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[51] Xiang Lisa Li and Percy Liang. “Prefix-tuning: Optimizing continuous prompts for generation”. In: *arXiv preprint arXiv:2101.00190* (2021).

[52] Yuhui Li et al. “Eagle-2: Faster inference of language models with dynamic draft trees”. In: *arXiv preprint arXiv:2406.16858* (2024).

[53] Zhuohan Li et al. “AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving”. In: *arXiv preprint arXiv:2302.11665* (2023).

[54] Wanchao Liang et al. “TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training”. In: *arXiv preprint arXiv:2410.06511* (2024).

[55] Bin Lin et al. “Video-llava: Learning united visual representation by alignment before projection”. In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing.* 2024, pp. 5971–5984.

[56] Aixin Liu et al. “Deepseek-v3 technical report”. In: *arXiv preprint arXiv:2412.19437* (2024).

[57] Haotian Liu et al. “Visual instruction tuning”. In: *Advances in neural information processing systems* 36 (2023), pp. 34892–34916.

[58] LMCache. *LMCache: Supercharge Your LLM with the Fastest KV Cache Layer.* URL: `https://github.com/LMCache/LMCache`.

[59] Lingxiao Ma et al. “Rammer: Enabling holistic deep learning compiler optimizations with rtasks”. In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation.* 2020, pp. 881–897.

[60] Xupeng Miao et al. “Specinfer: Accelerating large language model serving with tree-based speculative inference and verification”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024, pp. 932–949.

[61] Deepak Narayanan et al. “PipeDream: Generalized pipeline parallelism for DNN training”. In: *Proceedings of the 27th ACM symposium on operating systems principles*. 2019, pp. 1–15.

[62] NVIDIA. *CUDA Driver API*. URL: `https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__UNIFIED.html`.

[63] NVIDIA. *FasterTransformer*. `https://github.com/NVIDIA/FasterTransformer`. 2023.

[64] NVIDIA. *Getting Started with CUDA Graphs*. 2019. URL: `https://developer.nvidia.com/blog/cuda-graphs/`.

[65] NVIDIA. *NCCL: The NVIDIA Collective Communication Library*. `https://developer.nvidia.com/nccl`. 2023.

[66] NVIDIA. *NVIDIA Blackwell Architecture*. URL: `https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/`.

[67] NVIDIA. *NVIDIA Hopper Architecture*. URL: `https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/`.

[68] NVIDIA. *TensorRT-LLM*. URL: `https://github.com/NVIDIA/TensorRT-LLM`.

[69] NVIDIA. *Triton Inference Server*. `https://developer.nvidia.com/nvidia-triton-inference-server`.

[70] Christopher Olston et al. “Tensorflow-serving: Flexible, high-performance ml serving”. In: *arXiv preprint arXiv:1712.06139* (2017).

[71] OpenAI. 2020. URL: `https://openai.com/blog/openai-api`.

[72] OpenAI. 2022. URL: `https://openai.com/blog/chatgpt`.

[73] OpenAI. 2023. URL: `https://openai.com/blog/custom-instructions-for-chatgpt`.

[74] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: `2303.08774 [cs.CL]`.

[75] OpenAI. *Introducing GPT-5*. Aug. 2025. URL: `https://openai.com/index/introducing-gpt-5/`.

[76] OpenAI. *OpenAI Harmony: OpenAI’s response format for its open-weight model series gpt-oss*. URL: `https://github.com/openai/harmony`.

[77] LMSYS ORG. *Chatbot Arena Leaderboard Week 8: Introducing MT-Bench and Vicuna-33B*. https://lmsys.org/blog/2023-06-22-leaderboard/. 2023.

[78] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[79] Shishir G Patil et al. “POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 17573–17583.

[80] Reiner Pope et al. “Efficiently Scaling Transformer Inference”. In: *arXiv preprint arXiv:2211.05102* (2022).

[81] Jie Ren et al. “ZeRO-Offload: Democratizing Billion-Scale Model Training.” In: *USENIX Annual Technical Conference*. 2021, pp. 551–564.

[82] Reuters. 2023. URL: `https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/`.

[83] Timo Schick et al. “Toolformer: Language models can teach themselves to use tools”. In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 68539–68551.

[84] Amazon Web Services. 2023. URL: `https://aws.amazon.com/bedrock/`.

[85] Noam Shazeer. “Glu variants improve transformer”. In: *arXiv preprint arXiv:2002.05202* (2020).

[86] Noam Shazeer et al. “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer”. In: *arXiv preprint arXiv:1701.06538* (2017).

[87] Haichen Shen et al. “Nexus: A GPU cluster engine for accelerating DNN-based video analysis”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 322–337.

[88] Ying Sheng et al. “High-throughput Generative Inference of Large Language Models with a Single GPU”. In: *arXiv preprint arXiv:2303.06865* (2023).

[89] Mohammad Shoeybi et al. “Megatron-lm: Training multi-billion parameter language models using model parallelism”. In: *arXiv preprint arXiv:1909.08053* (2019).

[90] Benoit Steiner et al. “OLLA: Optimizing the Lifetime and Location of Arrays to Reduce the Memory Usage of Neural Networks”. In: (2022). DOI: `10.48550/arXiv.2210.12924`.

[91] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems* 27 (2014).

[92] Rohan Taori et al. *Stanford Alpaca: An Instruction-following LLaMA model.* `https://github.com/tatsu-lab/stanford_alpaca`. 2023.

[93] ShareGPT Team. 2023. URL: `https://sharegpt.com/`.

[94] vLLM Team. *vLLM: A high-throughput and memory-efficient inference and serving engine for LLMs.* URL: `https://github.com/vllm-project/vllm`.

[95] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2019, pp. 10–19.

[96] Hugo Touvron et al. "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023).

[97] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[98] Jing Wang et al. "Pacman: An Efficient Compaction Approach for {Log-Structured}{Key-Value} Store on Persistent Memory". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 773–788.

[99] Linnan Wang et al. "Superneurons: Dynamic GPU memory management for training deep neural networks". In: *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 2018, pp. 41–53.

[100] Xiaohui Wang et al. "LightSeq: A High Performance Inference Library for Transformers". In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*. 2021, pp. 113–120.

[101] Yizhong Wang et al. "Self-Instruct: Aligning Language Model with Self Generated Instructions". In: *arXiv preprint arXiv:2212.10560* (2022).

[102] Bram Wasti et al. *No More Train-Inference Mismatch: Bitwise Consistent On-Policy Reinforcement Learning with vLLM and TorchTitan*. 2025. URL: `https://blog.vllm.ai/2025/11/10/bitwise-consistent-train-inference.html`.

[103] Sam Wiseman and Alexander M Rush. "Sequence-to-sequence learning as beam-search optimization". In: *arXiv preprint arXiv:1606.02960* (2016).

[104] Thomas Wolf et al. "Transformers: State-of-the-art natural language processing". In: *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 2020, pp. 38–45.

[105] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[106] Amy Yang et al. "Context parallelism for scalable million-token inference". In: *arXiv preprint arXiv:2411.01783* (2024).

[107] An Yang et al. "Qwen3 technical report". In: *arXiv preprint arXiv:2505.09388* (2025).

[108] Shunyu Yao et al. "React: Synergizing reasoning and acting in language models". In: *The eleventh international conference on learning representations*. 2022.

[109] Zihao Ye et al. "Flashinfer: Efficient and customizable attention engine for llm inference serving". In: *arXiv preprint arXiv:2501.01005* (2025).

[110] Gyeong-In Yu et al. "Orca: A Distributed Serving System for {Transformer-Based} Generative Models". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 521–538.

[111] Biao Zhang and Rico Sennrich. "Root mean square layer normalization". In: *Advances in Neural Information Processing Systems* 32 (2019).

[112] Hong Zhang et al. "SHEPHERD: Serving DNNs in the Wild". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 787–808. ISBN: 978-1-939133-33-5. URL: `https://www.usenix.org/conference/nsdi23/presentation/zhang-hong`.

[113] Susan Zhang et al. "Opt: Open pre-trained transformer language models". In: *arXiv preprint arXiv:2205.01068* (2022).

[114] Lianmin Zheng et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 559–578.

[115] Lianmin Zheng et al. "Sglang: Efficient execution of structured language model programs". In: *Advances in neural information processing systems* 37 (2024), pp. 62557–62583.

[116] Yinmin Zhong et al. "{DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pp. 193–210.

[117] Zhe Zhou et al. "PetS: A Unified Framework for Parameter-Efficient Transformers Serving". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 489–504.

[118] Kan Zhu et al. "{NanoFlow}: Towards optimal large language model serving throughput". In: *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 2025, pp. 749–765.