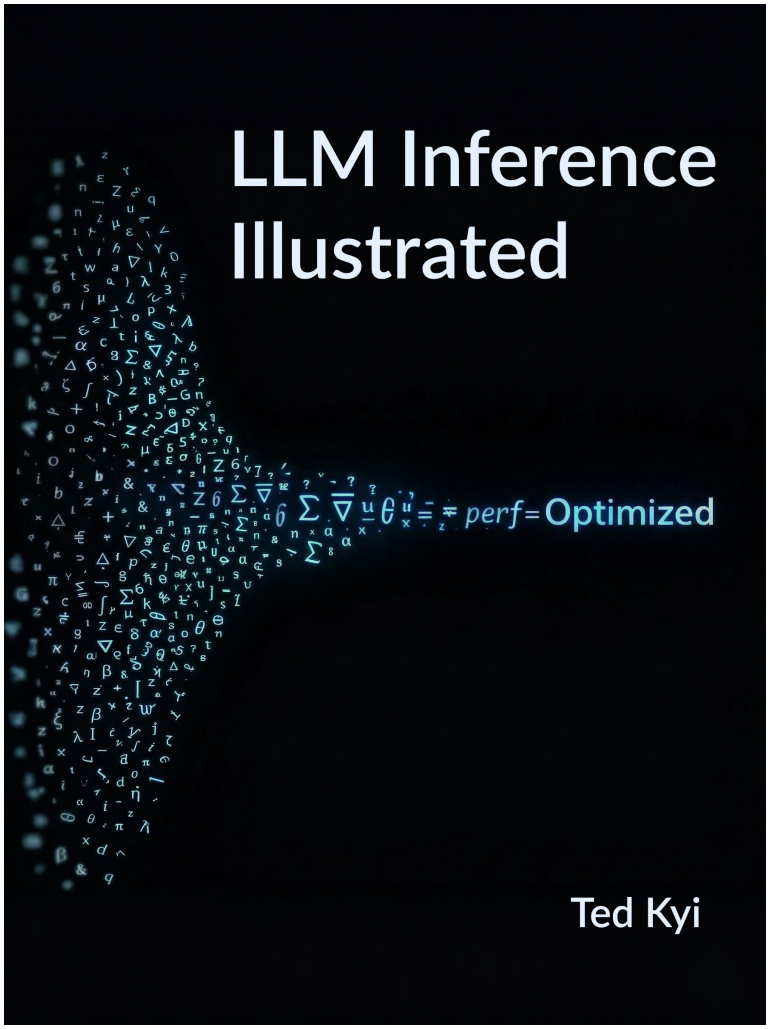
LLM Inference Illustrated

Ted Kyi

Version dated 2026-05-10

Table of contents

Cover



LLM Inference Illustrated

Typeset with Quarto.

Preface

A short book on autoregressive LLM inference and the techniques that make it fast, efficient, and scalable.

Who this book is for

This book is an intermediate guide for ML engineers (and people with a similar background) who are already familiar with neural networks, the transformer architecture, self-attention, and autoregressive decoder-only LLMs. This book evolved from research into efficient ways to do LLM inference for special cases such as RAG, Writing in the Margins, and beam search. After spending time learning about attention kernels, prefix caching, and systems such as vLLM and SGLang, I decided to share the knowledge I had gathered in one convenient place.

There are many detailed diagrams in the book, and it is best viewed on a high resolution screen.

What this book covers

This book discusses how LLM inference works, then reviews the techniques for optimizing and scaling LLM inference. It should not be the first resource for learning about the transformer, nor will it teach how to code LLMs. It is also not a comprehensive MLOps or system implementation book. I hope you find it useful for providing a good intuition and the necessary details about concepts like chunked prefill, ragged batching, tensor parallelism, and speculative decoding.

How this book was written

This book started as a detailed list of topics, techniques, and research papers. The idea to use data flow diagrams and the design of most diagrams were entirely my own.

Full disclosure: I used LLMs to suggest ways to organize the material and to write initial text based on my notes. I used Claude Code to speed up the implementation of the diagrams and to debug places where the appearance of text, formulas, or figures wasn't correct. If you are not interested in reading work that used AI assistance, that is your prerogative. For what it's worth, AI assistance shortened the process of getting the first release of this book from months to weeks. I probably would not have taken the time to write this book if it had required several months.

Acknowledgements and feedback

Thank you to everyone who provided feedback, starting with RC and RS.

I would appreciate your reporting any errata or making suggestions for how this book could be better. The easiest way is to open a GitHub issue from within the electronic book.

If you found this book useful, you can support it by sharing it with others, giving the GitHub repo a star, or buying me a coffee at ko-fi.com/tedkyi.

Conventions

Figures throughout this book use a consistent visual language for tensor shapes, dimensions, and components. This page collects those conventions in one place so you can decode any diagram at a glance.

Dimension names and abbreviations

Tensors in figures are labeled with their shapes using the single-letter abbreviations below. The full names are shown, along with abbreviations used elsewhere.

Table 1.: Standard dimension symbols used in figures and text.

Symbol	Full name	Example values (Llama 3 70B)
B	Batch size	1–64
D	Model dimension (d_{model})	8192
d_K	Head dimension, usually $d_K = D/H$	128
F	FFN intermediate dimension (d_{ff})	28672 (SwiGLU)
H	Number of attention heads (n_{heads})	64
H_{KV}	Number of KV heads (n_{kv})	8 (with GQA)
L	Number of layers	80
S	Sequence length	Prefill: prompt length (e.g., 2048); Decode: 1
V	Vocabulary size	128000

Tensor edge colors

Tensor rectangles use **colored edges** to indicate which dimension runs along that axis. Top/bottom edges represent one dimension; left/right edges represent another. This makes it possible to see at a glance how dimensions flow through a computation.

Table 2.: Edge colors encode tensor dimensions.

Edge	Dimension
	Model dimension (**D**)
	Head / Q-K dimension ($\mathbf{d_K}$)
	FFN intermediate dimension (**F**)
	Number of attention heads (**H**)
	Number of KV heads ($\mathbf{H}_{\mathrm{KV}}$)
	Number of layers (**L**)
	Sequence length (**S**)
	Vocabulary (**V**)
	Default / other

Tensor fill colors

The **fill color** of a tensor rectangle indicates its role in the computation.

Table 3.: Fill colors encode tensor roles.

Fill	Meaning
	Input / activations (neutral)
	Weight matrices
	Queries (**Q**)
	Keys (**K**)
	Values (**V**)
	Attention scores / weights
	KV cache entries

Component colors

Architectural block diagrams use a separate set of fill colors to distinguish transformer components.

Table 4.: Component colors in architectural block diagrams.

Color	Component
	Embedding layers
	Layer normalization / Add & Norm
	Self-attention blocks
	Feed-forward blocks
	Linear projections
	Softmax

Reading a tensor data flow diagram

To illustrate how these conventions combine, here is how a typical tensor rectangle should be read:

- The **top and bottom edge colors** tell you the dimension that runs horizontally, the number of columns.
- The **left and right edge colors** tell you the dimension that runs vertically, the number of rows.
- The **shape label** (e.g., **(S, D)**) confirms the dimensions explicitly.
- The **fill color** helps disambiguate what kind of tensor it is (query, key, weight, etc.).

When two tensors share an edge color on a matching side, it means those dimensions are compatible for the operation that connects them — a matrix multiply, a concatenation, or an elementwise operation. For matrix multiplcation, the number of columns on the lefthand tensor must match the number of rows on the righthand tensor. For elementwise operations, usually all dimensions must match, unless one of the dimensions is 1, and it is broadcast to match the other tensor. An sample diagram is explained in the Section 2.2 section.

1. Introduction

Before ChatGPT, most of the focus on language models was on how to train better, smarter models. Now that we are in a world where millions of people use **large language models (LLMs)** every day, much recent energy has gone into making the serving of these models more efficient and less expensive. This book summarizes many of the key advancements in LLM inference. We will attempt to provide clarity about each concept, but this is not an MLOps book, and we will not drill down into coding each technique. In this chapter, we start by understanding the components of LLM inference. In the remainder of this book, we will look at different bottlenecks and techniques for improving performance.

1.1. Overview

LLM inference is the process of generating a text response. You give the model some input, and it predicts what comes next, one word (technically, one token) at a time. Before we get into the technical details of how inference systems are built and optimized, let's walk through a simple example to build intuition for what's actually happening.

A simple example

Suppose we give an LLM the prompt **"The quick brown fox jumps"** — five input tokens. The model's job is to predict what tokens come next.

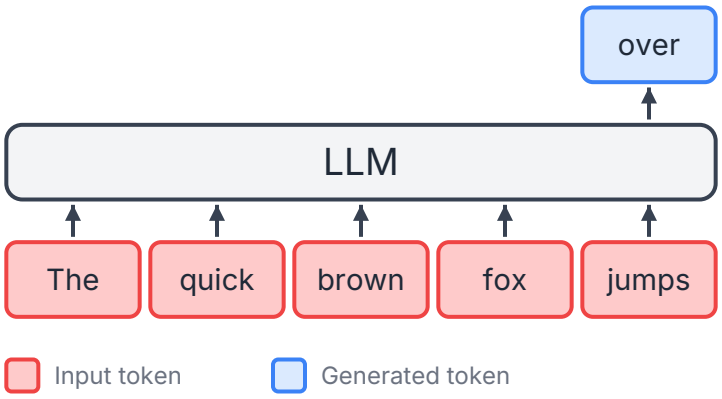


Figure 1.1.: First token predicted for the prompt "The quick brown fox jumps"

The text generation happens in a series of steps. First, the LLM runs a single **forward pass** of the neural network model using all of the input tokens. After processing all of these tokens together, it produces its prediction for the next token, **over**. This initial step is shown in Figure 1.1.

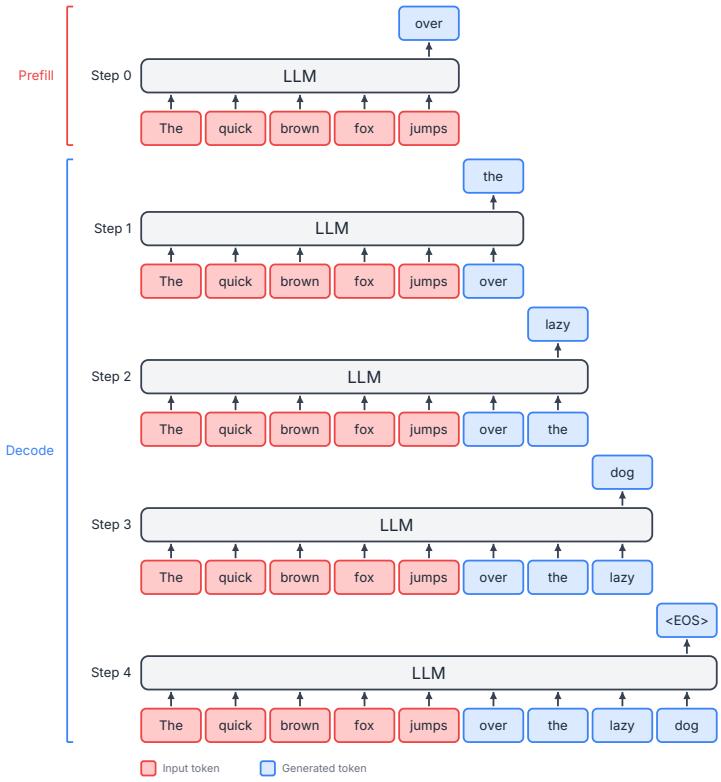


Figure 1.2.: An example LLM response for the prompt "The quick brown fox jumps"

Text generation continues in a series of steps, one token at a time, with the entire process shown in Figure 1.2:

1. In the next step, the predicted token **over** is appended to the end of the input text, and the model is fed **"The quick brown fox jumps over"**. In this step it predicts **the**.
2. Once again, the predicted token, which is now **the**, is appended to the end, the model is fed **"The quick brown fox jumps over the"**, and it predicts **lazy**.
3. Continuing this process, for the input **"The quick brown fox jumps over the lazy"**, the model predicts **dog**.
4. Then, for the input **"The quick brown fox jumps over the lazy dog"**, the model outputs a special token **<EOS>**. The special token signals the end of generation, and the completed response is returned to the user.

In our very first step, the LLM is given all of our input tokens. In many cases, this input prompt can be quite long. Because we have so much new data to process, this initial step uses different computing resources than later steps. This first step is called the **prefill**, and it has a high ratio of computation performed to volume of data used.

The remaining steps each add only one new token to the inputs that the LLM has seen before. This has a different computing profile from the prefill, with a much lower ratio of computation performed to volume of data used. This portion is called the **decode** phase, and when a lot of text is generated, it requires many decode steps to complete generation of the response.

This iterative token loop is the core process that LLMs such as ChatGPT and Claude use to respond to requests. In this book, we will dig deeper to understand the internals of this inference process, then explore techniques for improving it.

Why inference optimization matters

That simple loop — run the model, get a token, repeat — sounds straightforward. But the raw cost of running it naively is staggering. A 70-billion-parameter model stores its weights in about 140 GB of GPU memory (at 2 bytes per parameter). Every single decode step has to, at a minimum, read through all of those weights just to produce one token. Without any optimization, a single request to a 70B model might generate only 15-20 tokens per second on a high-end GPU, and frontier models, believed to be ten times larger, would run under 2 tokens per second. Run that for one user at a time with no batching, and you've got an extremely expensive system that produces tokens only marginally faster than a reader can consume them, while also leaving most of the GPU's computational capability unused.

The situation gets much better with the right techniques. Modern serving frameworks like vLLM and SGLang apply a combination of the optimizations covered in this book — smarter batching, memory management, kernel optimization, and more — and the results are dramatic. In one benchmark, an unoptimized serving setup on an A100 GPU achieved about 80 tokens per second for a 13B model. The same model served through vLLM achieved roughly 1,900 tokens per second — a 23x improvement (Anyscale 2024a). On newer hardware, optimized frameworks routinely serve models like Llama 3 8B at over 16,000 tokens per second on a single H100 GPU, with per-token latencies in the single-digit milliseconds.

These aren't gains from better hardware. They come from understanding the bottlenecks in the LLM inference pipeline and systematically eliminating them. That's what this book is about.

About this book

This book is written for ML engineers who are familiar with the transformer architecture behind modern LLMs and want to understand what else goes into serving them quickly. We assume you're familiar with neural networks, the transformer architecture, and self-attention at a conceptual level.

With that, let's get into the technical details. We'll start by breaking down the decoder-only transformer architecture in more detail and building a precise understanding of the data and operations involved.

2. The Decoder-Only Transformer

This chapter covers the architecture of the basic decoder-only transformer used by most LLMs such as ChatGPT and Llama. We will walk through it in enough detail to be able to trace a token through the full model, seeing the parameter and activation tensor dimensions. If you aren't at all familiar with the transformer, there are many good resources, and a few are listed in the Further Reading (Section 2.7) at the end of the chapter. If you're already comfortable with multi-head self-attention, the KV cache, and gated MLPs, you can skim through this material and move ahead to the next chapter. We recommend that you examine the Conventions tip at the beginning of Section 2.2 to understand the conventions used in tensor data flow diagrams before moving ahead to the rest of the book.

We'll use concrete numbers from the Llama family of models in this book. This is not because Llama is special, but because its architecture is representative of the core that modern LLMs share and its parameters are public. There are many newer models with better performance, but Llama has a simpler, more canonical architecture which is a good fit for this introduction. When we say "a 70B model," we mean something with roughly Llama 3 70B's dimensions: 80 layers, a model dimension of 8192, 64 attention heads, and a vocabulary of 128,000 tokens (Grattafiori et al. 2024).

2.1. From Encoder-Decoder to Decoder-Only

The original transformer architecture introduced in "Attention Is All You Need" (Vaswani et al. 2017) has two distinct halves: an **encoder** that processes the input sequence and a **decoder** that generates the output sequence. The encoder builds a rich representation of the input through layers of self-attention. The decoder then attends to that representation through **cross-attention** layers while also attending to its own previously generated tokens through **masked self-attention**.

This encoder-decoder design, shown in Figure 2.1, was built for sequence-to-sequence tasks like machine translation, where you have a clear separation between input (such as a sentence in French) and output (its English translation). The encoder on the left reads the full input and builds a bidirectional representation — each token can attend to every other token, past and future. The decoder on the right then generates the output one token at a time, attending both to the encoder's representation and to its own partial output.

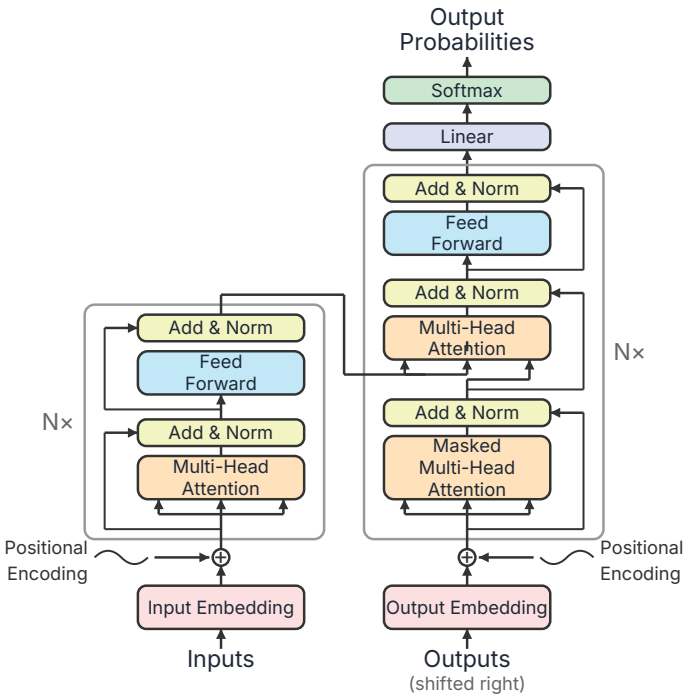


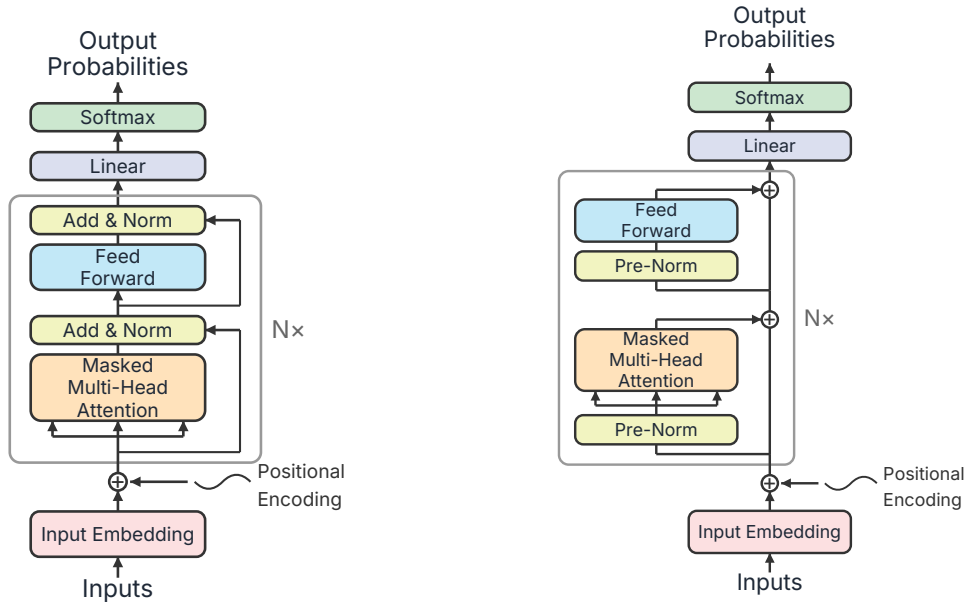
Figure 2.1.: The original transformer architecture

> **Note**
>
> It is a common convention in machine learning to draw architecture diagrams with the data flowing from the bottom to the top, with the layers stacked vertically. Under this convention, the input is at the bottom, and the prediction "heads" and output are at the top. This book follows the bottom-up flow convention.

The decoder-only transformer, used by GPT-2 (Radford et al. 2019), the original LLaMA (Touvron, Lavril, et al. 2023), and virtually all modern large language models in production today, takes this architecture and strips it down. The entire encoder is removed. The cross-attention layers in the middle of the decoder are removed. What's left is a single stack of transformer blocks, each containing masked self-attention and a **feed-forward network (FFN)**, preceded by a token embedding layer and followed by a linear language model head.

On the left of Figure 2.2 is a trimmed-down decoder-only version of the original transformer architecture. Researchers also found that training larger models was more stable when the normalization happened before each attention and FFN block, instead of after. This design is referred to as the **pre-norm** architecture. On the right side of Figure 2.2 is a pre-norm decoder-only model where the attention and FFN layers have been shifted to the side, making the residual connection path, known as the **residual stream**, more obvious. Throughout this book, we will use this straight-through arrangement of the pre-norm decoder-only LLM in our examples.

Why did this simpler architecture win? Three reasons stand out. First, **simplicity**: one stack



(a) Original decoder-only LLM

(b) Straight through view of pre-norm decoder-only LLM

Figure 2.2.: A decoder-only LLM, shown in two variants

of identical blocks is easier to implement, optimize, and scale than two interleaved stacks. Second, **favorable scaling behavior**: empirically, decoder-only models have shown strong performance scaling with increased parameters and data, as demonstrated by GPT-3, PaLM (Chowdhery et al. 2022), and the Llama series. Third, a **unified pretraining objective**: next-token prediction works on any text, requiring no paired input-output data. You can perform self-supervised training on the entire internet without needing aligned corpora or labeled data. Another key benefit is that the decoder-only model is faster to train than the original encoder-decoder transformer.

Layer repetition and weight sharing

Two aspects of the architecture are worth making very explicit, because they matter for understanding inference costs.

First, the "N×" in the architecture diagram means that the same block structure is repeated N times. Note that each layer has the same structure, shape, and size, but each layer has its own independent set of weights. For example, Layer 0's attention weights are completely separate from Layer 1's. A 70B model with 80 layers has 80 distinct sets of attention and feed-forward weights. During inference, the model must read each layer's weights from memory — there is no reuse across layers in the standard LLM.

A diagram making the N copies explicit is shown in Figure 2.3. This is what a decoder-only LLM really looks like, and if you review the PyTorch code for one, you will see this exact structure. The "N×" is shorthand, and it saves space in diagrams, but in actual GPU memory,

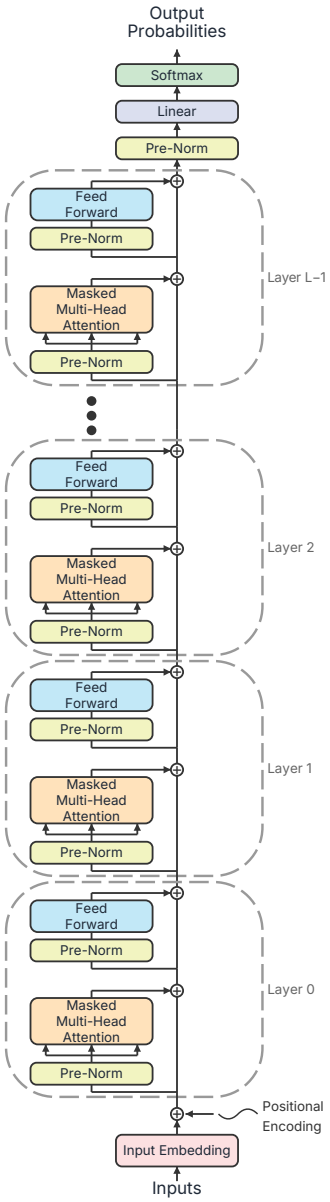


Figure 2.3.: A decoder-only LLM with explicit multiple layers

all of those many copies do exist and take up a lot of room. Please also note that each layer must be calculated before the next layer can be calculated. (If you are familiar with earlier work with RNNs and LSTMs, we have eliminated the sequential dependency in the sequence length direction, but we now have a sequential dependency in the layer direction.) We will also mention here that the input to the first layer is a vector of floating-point numbers of length d_{model}, which we will abbreviate **(D)**.

The second architectural understanding needed is that the model is applied identically across every token position in the sequence, sharing all weights. When we process a sequence of **S** tokens, the same weight matrices are applied to every token position. Figure 2.4 shows a first intuition of the model being copied repeatedly for each token position. Unlike the layers in the vertical direction that are real, the copies of each block in the same layer at different token positions are conceptual and share weights. For example, the FFN weights in layer 2 for token 0 are identical to the FFN weights in layer 2 for token 1. There is only one FFN for layer 2, and the horizontal copies are just a mental model.

One aspect of the data flow that is not yet captured in this diagram is that the attention calculation processes information across token positions. All of the other blocks, such as embedding or FFN, process each token position's input in the same way, regardless of what is in other token positions. Masked multi-head attention looks at information from the other token positions, so we add an orange line in Figure 2.5 to explicitly show that data flows from earlier tokens to all later tokens. Wiring up separate vertical stacks would also require connecting attention layers in this way. We are modeling this conceptually at this time, and we will see in Section 2.5 that a fundamental optimization will implement this orange path in a data store called the KV cache.

If we were to make a separate physical copy of the model for each token position, this would give us a correct answer, but it would consume massive amounts of memory for longer sequences. This idea of multiple copies of our model is a good mental model to understand data flow across token positions. However, this is not how model inference happens. Rather than making copies of the model, we simply take our vertical stack model and change the input from a 1D vector of shape **(D)** to a 2D tensor of shape **(S, D)**. Passing this 2D tensor into a single copy of the model yields a single tensor with outputs identical to what multiple copies of the model would have produced.

> **Note**
>
> For efficiency reasons, almost all deep learning training and inference happens in batches. This is true as well for LLM inference, and we will discuss batching in Section 5.1. For the sake of brevity, especially in diagrams, we will discuss non-batched data flow and omit the explicit mention of the batch dimension. For LLMs, the full batched shape of the input to the first layer is **(B, S, D)**, which we abbreviate to **(S, D)** for non-batched inputs. Wherever data tensor shapes are mentioned without a leading batch dimension, it is appropriate to think about batched processing adding a single dimension to the beginning of the shape. Weight tensors do not grow when processing data in batches. Rather, they are broadcast in the batch dimension. We hope this convention of using

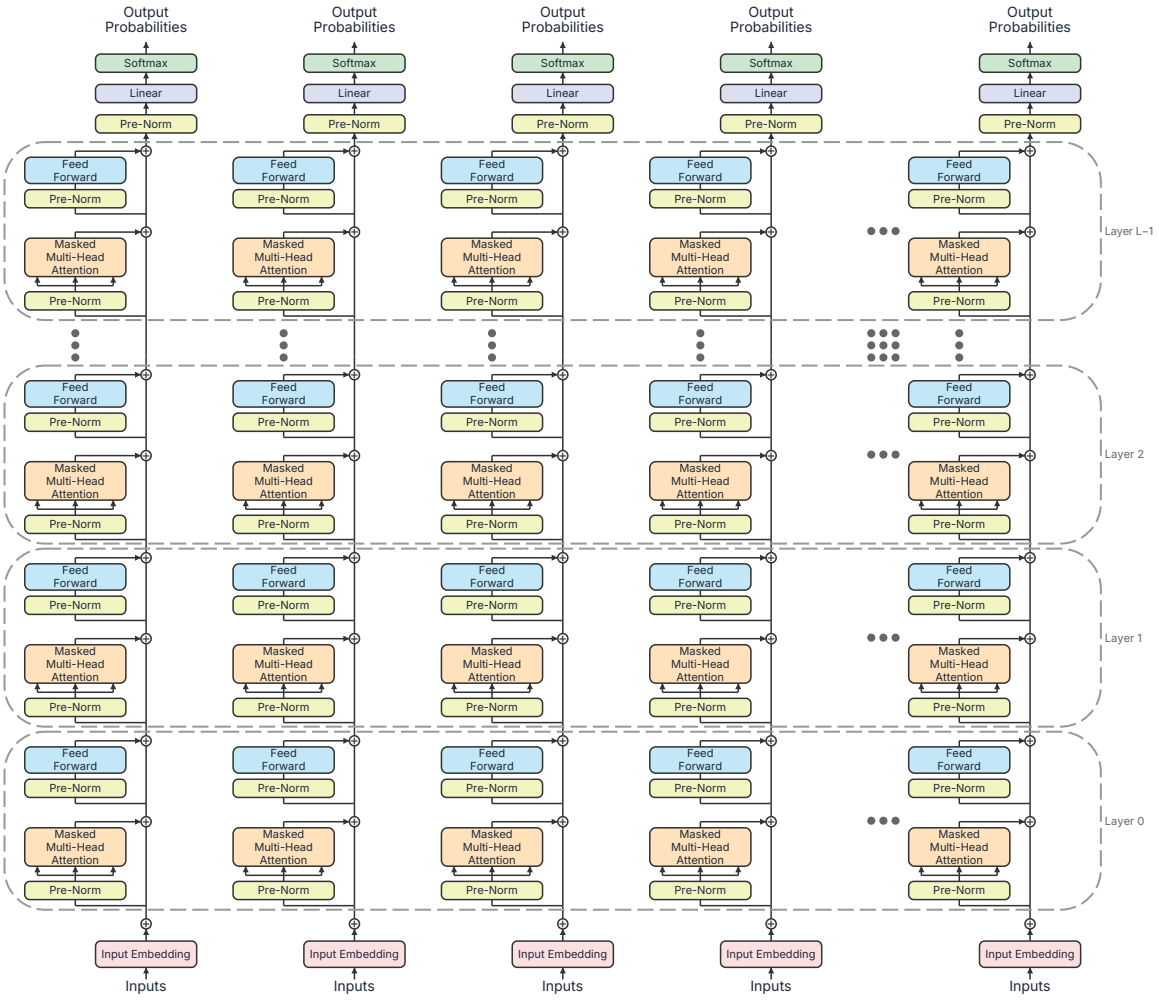


Figure 2.4.: Intuition of the LLM model being copied for each token position. Horizontal copies represent the virtual model processing each token.

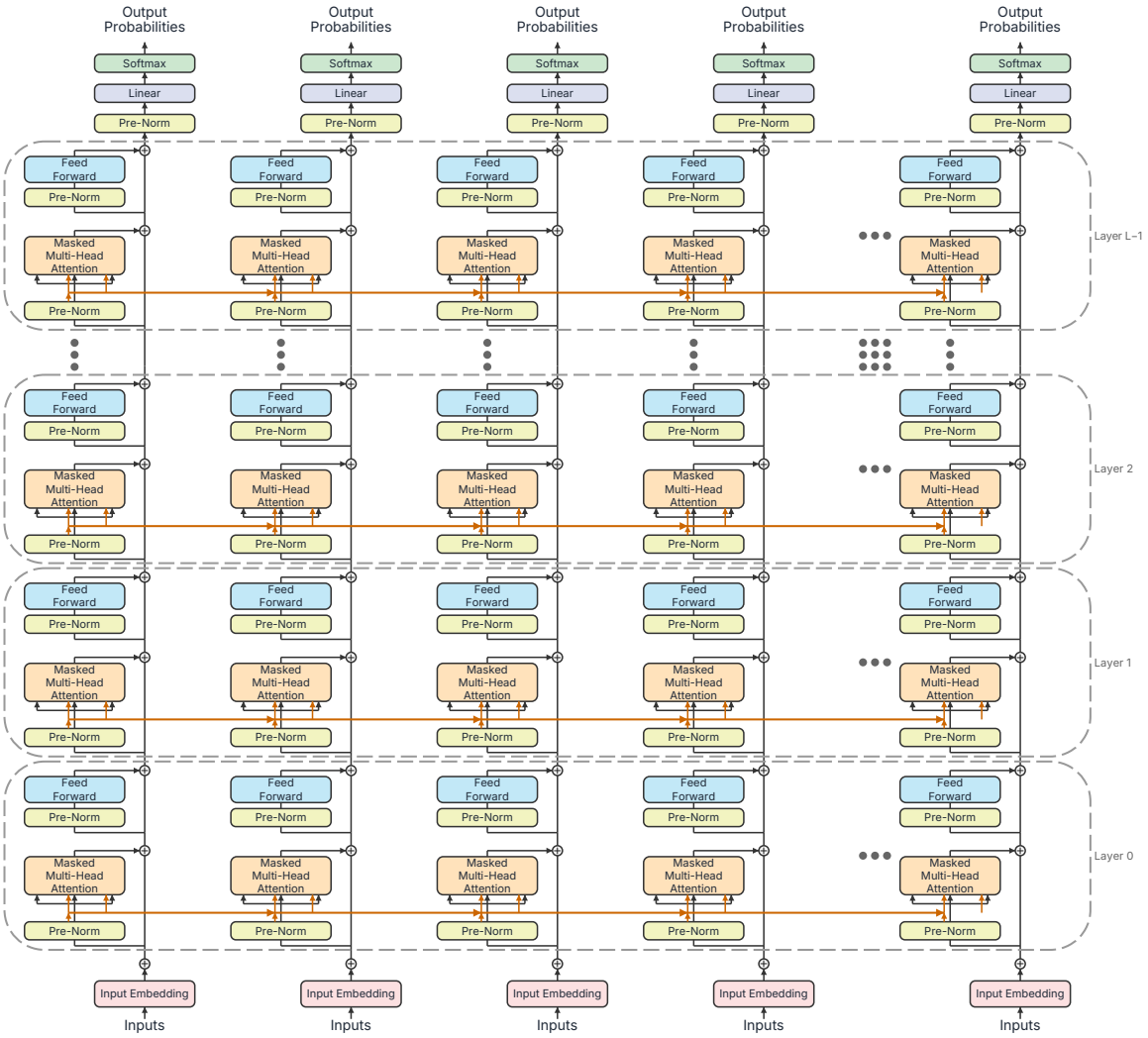


Figure 2.5.: Concept of LLM model being copied for each token position, with attention communication added.

non-batched shapes simplifies diagrams and eases intuition without any loss of generality for the batched use case.

2.2. Forward Pass Data Flow

To gain a deeper understanding of how inference works, we will examine what happens when we send data through a forward pass of the model. We will start at the big picture and work our way down into the individual components. Our diagrams will trace input tensors through all of the operations until we reach the output tensor, annotating every tensor's shape along the way. We call these **tensor data flow diagrams**, and they focus on the shapes of the tensors passed into operations more than the math of the calculation being performed. The focus on tensors is so we can understand the size and shape of the data, which directly affects transfer speeds and computation costs. The purpose of the data flow diagrams is to make properties of the tensors explicit so it is easier to have an intuition about the computations being performed. We still need to be familiar with the operands, especially to know how the inputs can be partitioned, and how much data must be exchanged between partitions.

Conventions

Before diving into the data flow, here are a few conventions used in the tensor data flow diagrams. Abbreviations and colors are listed in front matter in the **Conventions** section.

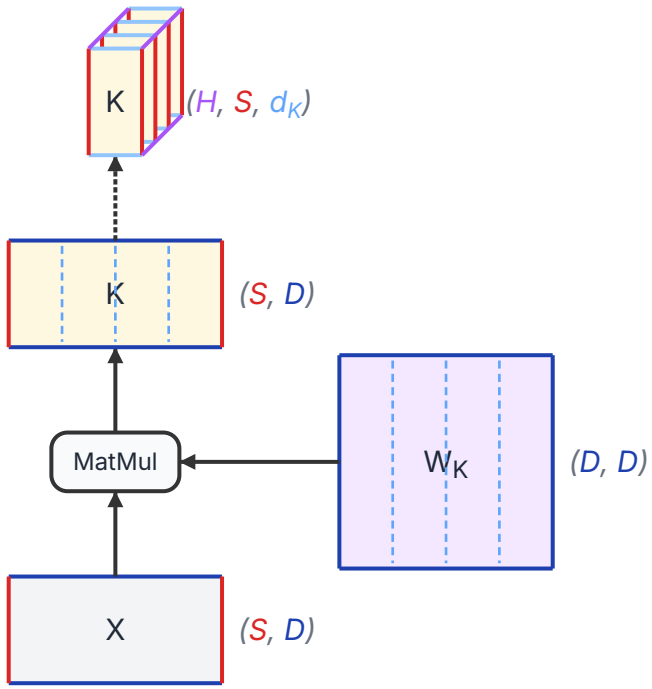


Figure 2.6.: A sample tensor data flow diagram, exhibiting many of the conventions

In this example tensor data flow diagram, the tensors are represented by rectangles, and the operand by a rounded rectangle.

Dimensions and edge color: Each tensor's shape is shown next to it on the right side, unless there isn't room. The first tensor on the bottom has shape (**S**, **D**), where **S** is the number of rows and **D** is the number of columns. Notice that the left and right edges are colored red, and every tensor with a dimension that is length **S** will have that dimension colored red. Similarly, the top and bottom edges are colored blue because the number of columns is **D**, and all other dimensions of length **D** will be the same color.

Fill color: The fill color of the bottom tensor is gray. For convenience, tensors holding similar semantic information will have similar fill colors. Weights are fixed model tensors, and they always have the same purple fill color.

3D tensors: The tensor at the top of the diagram is 3D with shape (**H**, **S**, d_K). Because it is 3D, it is shown with a "stacked" rectangle shape. The leading dimension, **H**, is the stacked dimension. The third dimension's purple edge color corresponds to this dimension.

Views: The middle data tensor and the top tensor are connected by an arrow with a dotted line. The dotted stroke indicates that there is a change of view, but no actual operation is performed.

The input

The input to our LLM model is a list of integer token IDs. Its shape is **(S)**, where **S** is the sequence length (number of tokens). As a reminder, in this book we are using the simpler notation of non-batched processing. (For batched processing, the input shape would be **(B, S)**.) The value of **S** varies. **S** starts as the length of the prompt, and it increases as generated tokens are appended. It is reasonable that **S** might be less than **D** at the beginning of generation, and then **S** grows to be greater than **D** by the end of generating a long response.

The full model

The complete forward pass stacks **L** decoder layers between the embedding layer and the language model head, as shown in Figure 2.7. This is functionally equivalent to Figure 2.3, except this tensor data flow diagram exposes the shapes of the weight and data tensors.

We see our input with shape **(S)** at the bottom of Figure 2.7. It is converted to a sequence of one-hot vectors, and the input is now shape **(S, V)**. This flows into a matrix multiply with the embedding weights, which are a giant matrix of shape **(V, D)**. We now have a dense input matrix of shape **(S, D)**. This is fed to Layer 0, whose output is the same shape, **(S, D)**, and the layer output is added to the existing values. As the data flows through all **L** layers, the shape never changes from **(S, D)** as the layer outputs continue to get added to the residual stream. After the last layer, the data passes through a final normalization layer, retaining its **(S, D)** shape. Finally, a Linear prediction head does a matrix multiply with a language modeling output weight matrix of shape **(D, V)** and it outputs the probability logits, which are shape **(S, V)**. These can be sharpened by passing through a SoftMax, and a sampling method must be performed to finalize the next token prediction.

The most important observation to take away from Figure 2.7 is that the data along the residual stream is always shape **(S, D)**. When we drill into the decoder layer and its attention and feed forward components, our inputs and outputs will always be **(S, D)**. Most of the compute and memory access cost of a forward pass, especially for models with many layers, is associated with the decoder layers. There isn't much to be gained from modifying how the embedding or linear output projection are performed.

Positional encoding

One detail we've glossed over: how does the model know the order of the input tokens? The self-attention computation is permutation-equivariant, which means that if you shuffle the input tokens, the output gets shuffled the same way (ignoring the causal mask). For this reason, position information must be explicitly injected through a separate pathway.

The technical details and pros and cons of different position embedding techniques are beyond the scope of this section. Some approaches worth mentioning are:

Sinusoidal position embeddings: the position embeddings used in the original "Attention Is All You Need" paper. A special embedding table of shape **(S, D)** is added to the token

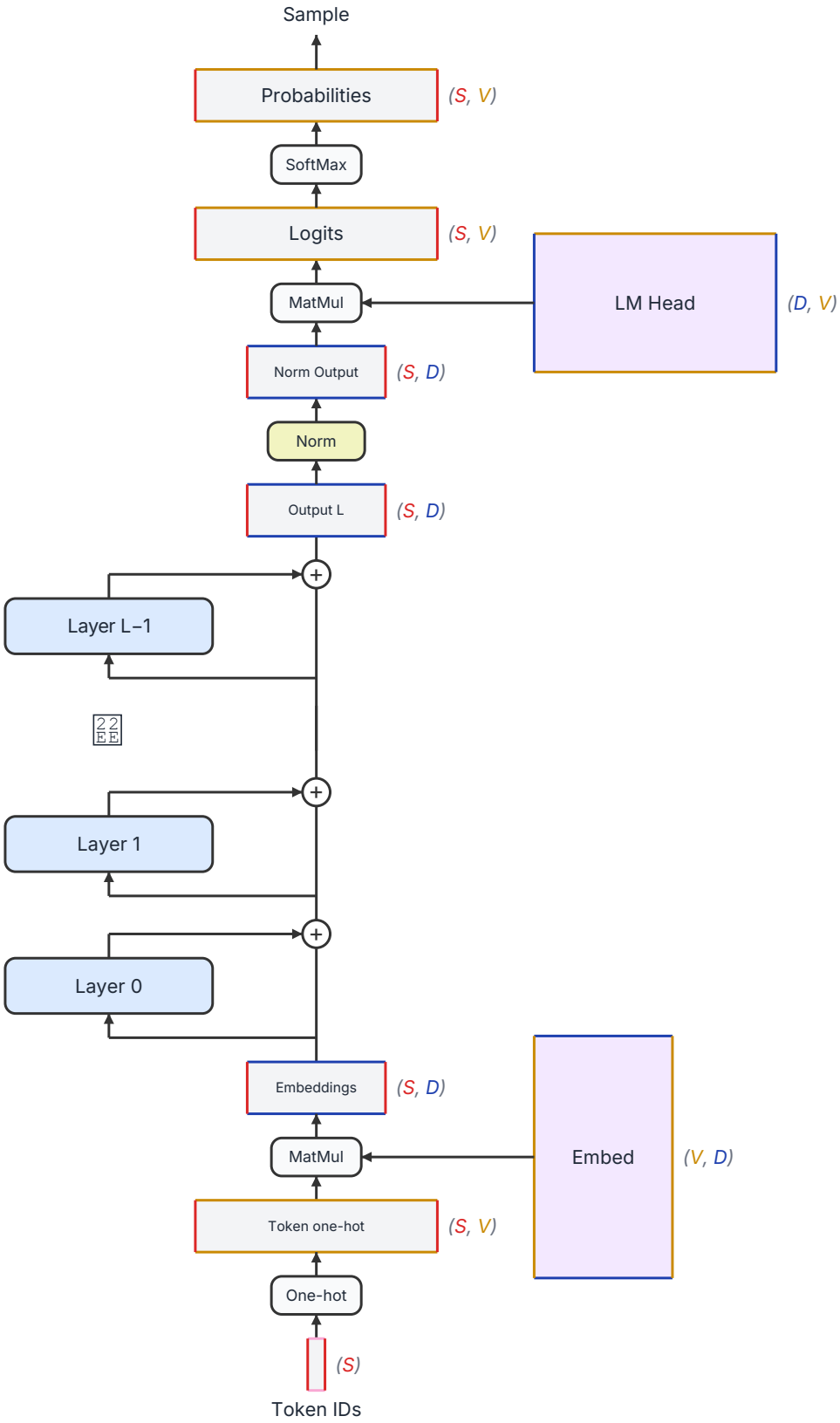


Figure 2.7.: Data flow of an entire decoder-only LLM. The details within each decoder layer are hidden for now.

embeddings before they are input to the first decoder layer. This is shown at the bottom of each figure from Figure 2.1 through Figure 2.3. In the embedding table, position 0 gets one vector, position 1 gets another, and so on. These are called sinusoidal because the values in each length **D** position vector are based on the trigonometric functions sine and cosine.

Learned absolute embeddings (GPT-2 (Radford et al. 2019)): a variant of the embedding table of shape **(S, D)** that is added to the token embeddings, except the values for each embedding vector are learned during training along with all the other model parameters.

Rotary Position Embeddings (RoPE) (Su et al. 2021): instead of adding position information to the input, RoPE applies a position-dependent rotation to the query and key vectors inside the attention computation. The rotation is defined such that the dot product between a query at position i and a key at position j depends only on the relative distance $i - j$. This is what most modern LLMs use, including Llama and its derivatives. RoPE enables better length generalization. Applying RoPE rotations to the query and key vectors is relatively inexpensive, so we omit its details from our diagrams and discussion.

Others: Another notable technique, called **ALiBi** (Press et al. 2021), adds a value dependent on the token position to the keys. This value increases linearly with the token position number. **NoPE** (Kazemnejad et al. 2023) is a newer technique that is gaining traction when mixed with other techniques like RoPE, usually on a layer-by-layer basis. In NoPE, no positional information is added. While it is surprising that having no position information would work, research has shown that NoPE can be highly effective when mixed with other approaches.

Weight tying

Some architectures share weights between the token embedding layer, whose parameter shape is **(V, D)**, and the LM head, whose parameter shape is **(D, V)** — which is the transpose of the embedding layer. When embedding tokens, the **(V, D)** weights are used, and when obtaining logits, the transpose of these weights is used. This is called **weight tying**. When the vocabulary is large, these matrices are very large. For Llama 3 70B, where **V** = 128,000 and **D** = 8192, each matrix has just over 1B parameters, so 2B combined. Tying them saves 1B parameters and ensures that the model's input and output representations are in the same space.

Weight tying doesn't affect model speed, but it does reduce the parameter count. The percentage savings is greater for small models, which is why we see weight tying used in some small LLMs. Because of the minimal impact on speed, we will keep things simple in this book and assume the weights are not tied.

2.3. The decoder block

Drilling into each layer in the model, each decoder block in a modern LLM uses the **pre-norm** formulation:

$$\hat{X} = X + \text{SelfAttention}(\text{Norm}(X))$$
$$y = \hat{X} + \text{FFN}(\text{Norm}(\hat{X}))$$

The data flow diagram for this is shown in Figure 2.8.

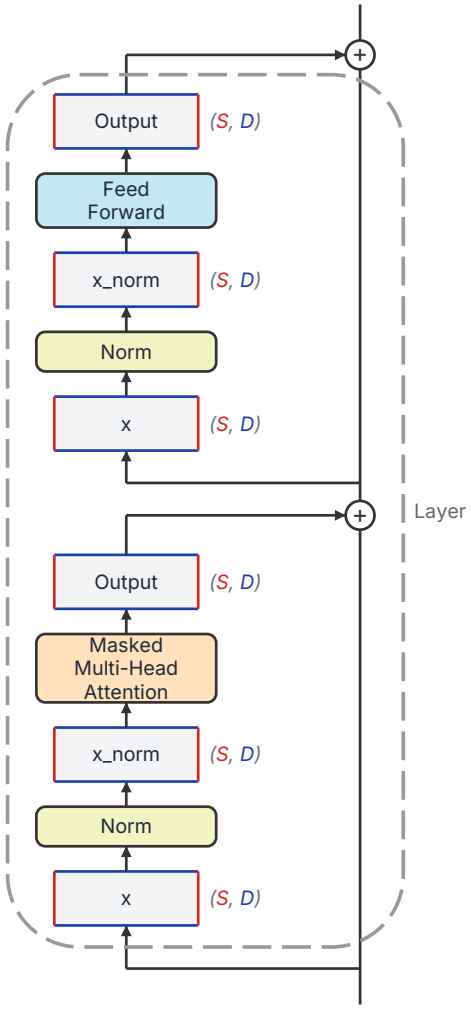


Figure 2.8.: Data flow of one decoder layer. The details within the attention and feed forward sub-layers are hidden for now.

The normalization in transformers, which originally was **LayerNorm** (Ba et al. 2016), is applied before each sub-layer (self-attention and FFN), and the sub-layer's output is added to its input via the residual connection. This is the pre-norm formulation, used by GPT-2 (Radford et al. 2019) and almost all modern LLMs. Another normalization layer which has replaced LayerNorm in popularity is **RMSNorm** (Zhang and Sennrich 2019). Both of these operations calculate the magnitude of each input vector across the model dimension **D**, then

scale the input vectors. The need to calculate across **D** will impact tensor parallelism, as we will see in Section 7.3.

The residual connections are critical for training deep networks. They allow gradients to flow through the network's layers without vanishing. For inference, they're simple addition operations that don't add meaningful compute cost.

Our data flow here is all of shape **(S, D)**, and there's not much computational overhead. There isn't much to optimize in what is shown here. We are now ready to drill into the sub-layers where the majority of the overhead lies. We will start with attention.

2.4. Self-Attention Mechanics

Self-attention is the mechanism that lets each token's representation incorporate information from other tokens in the sequence. This is what makes a transformer more than just a position-independent feature extractor. It lets the model understand that "bank" means something different in "river bank" versus "bank account." We'll walk through the computation step by step, tracking tensor shapes throughout. We will dissect the components and data flow of the attention layer in this section. If the reader is unfamiliar with attention, we recommend some of the resources in the Further Reading (Section 2.7) at the end of this chapter. Also, for a more detailed discussion of the data flow, side by side with sample code for attention, see Section A.1 in the Appendix.

The input

The input to a self-attention layer is a tensor of shape **(S, D)**, where **S** is the sequence length (number of tokens), and **D** is the model dimension (the width of each token's representation vector). As a reminder, in this book we are using the simpler notation of non-batched processing. For batched processing, the input shape would be **(B, S, D)**. As a concrete example, in Llama 3 70B, **D** = 8192.

Q, K, V projections

The first step of self-attention is to project the input into three separate representations: **queries** (Q), **keys** (K), and **values** (V). Each projection is a learned linear transformation:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

where X has shape **(S, D)** and W_Q, W_K, W_V each have shape **(D, D)**. The outputs Q, K, and V all have shape **(S, D)**, as shown in Figure 2.9. In this data flow diagram, **S** edges are drawn shorter than **D** edges. This will be accurate for Llama 3 70B when **S** < 8192. As the sequence gets longer and we have **S** > 8192, these dimensions will be longer, though this static diagram cannot show the change.

The intuition for Q, K, and V is roughly this: the query represents "what this token is looking for," the key represents "what this token offers to other tokens," and the value represents "the information this token will contribute when attended to." These are loose analogies, but they help build intuition for why we need three separate projections rather than just one or two.

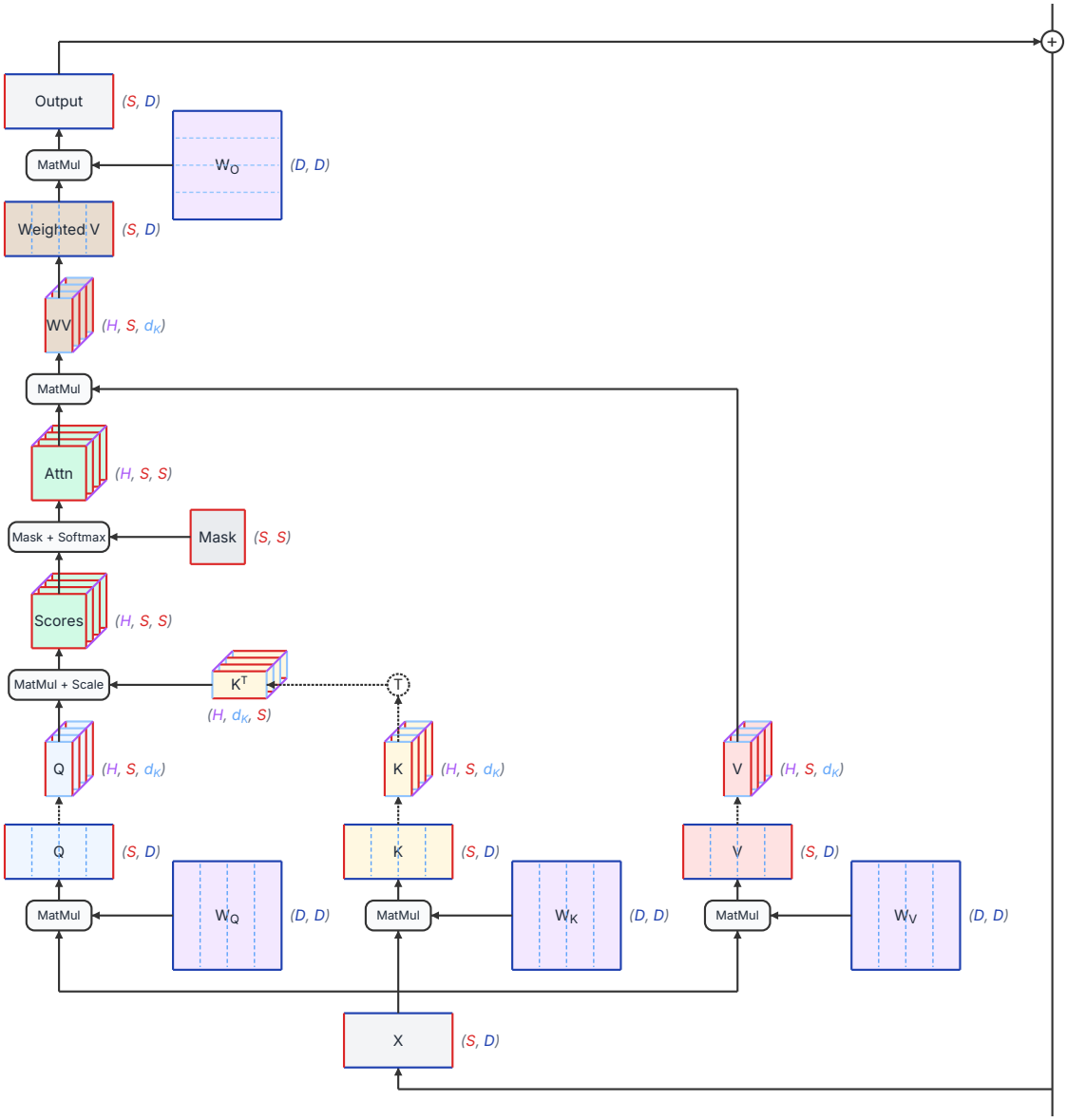


Figure 2.9.: Data flow of weights and activations in a multi-head attention layer using a basic, unoptimized implementation

Multi-head attention and the reshape

Rather than computing a single attention function over the full D-dimensional space, we split the computation into H parallel heads, each operating on a subspace of dimension $d_K = \mathrm{D}/H$.

For Llama 3 70B, **D** = 8192, **H** = 64 heads, and d_K = 128. This head dimension, d_K, of 128 is an extremely common choice in LLMs.

After the linear projections, we reshape Q, K, and V from (**S**, **D**) to (**H**, **S**, d_K). Mathematically, this is just reshaping the last dimension — breaking **D** into **H** groups of d_K — and transposing so that the head dimension comes before the sequence dimension. No additional computation is required. In Figure 2.9, dotted line arrows show the view operation that does the reshape. In addition, dashed lines on the weight matrices and Q, K, and V tensors show the conceptual division of those tensors into heads, even before the reshape. Note that these are divided into heads by groups of columns.

Why multiple heads? A single attention head computes a single attention pattern — one set of weights determining how much each position attends to every other position. Multiple heads allow the model to attend to different aspects of the input simultaneously. One head might focus on syntactic relationships, another on semantic similarity, and another on positional proximity. In practice, learned attention patterns are more complex than these examples, but the principle holds: multiple heads provide multiple "channels" for cross-position communication.

Attention score computation

With Q and K reshaped to (**H**, **S**, d_K), we compute the raw attention scores by taking the dot product of queries with keys:

$$\text{Scores} = \frac{QK^T}{\sqrt{d_K}}$$

The matrix multiplication QK^T produces a tensor of shape (**H**, **S**, **S**). For each head, we get a matrix with shape (**S**, **S**), where entry (i, j) is the raw attention score between the query at token position i and the key at token position j. The division by $\sqrt{d_K}$ prevents the dot products from growing too large as the dimension d_K increases, which would push SoftMax into regions with vanishing gradients.

This (**H**, **S**, **S**) attention matrix is the computational signature of standard self-attention. Its size is quadratic in sequence length. For a 4096-token prompt, each head produces a 4096×4096 matrix, and when **S** = 100,000, each head's attention matrix is 100,000×100,000 and has 10 billion entries. In Figure 2.9, the score tensor looks small for small values of **S**, but if drawn to scale, for large values of **S** it would be huge compared to other tensors. This quadratic scaling is why long-context inference is expensive and why we will see that techniques like FlashAttention (Dao et al. 2022) (Section 6.1) are so important. FlashAttention computes the exact same result but avoids materializing the full **S**×**S** matrix in GPU memory by tiling the computation to fit one piece at a time.

The causal mask

In a decoder-only model used for autoregressive generation, each token should only attend to itself and to tokens that came before it — never to future tokens. This is enforced by a **causal mask**: an upper-triangular matrix of $-\infty$ values added to the attention scores before SoftMax, as shown in Figure 2.10.

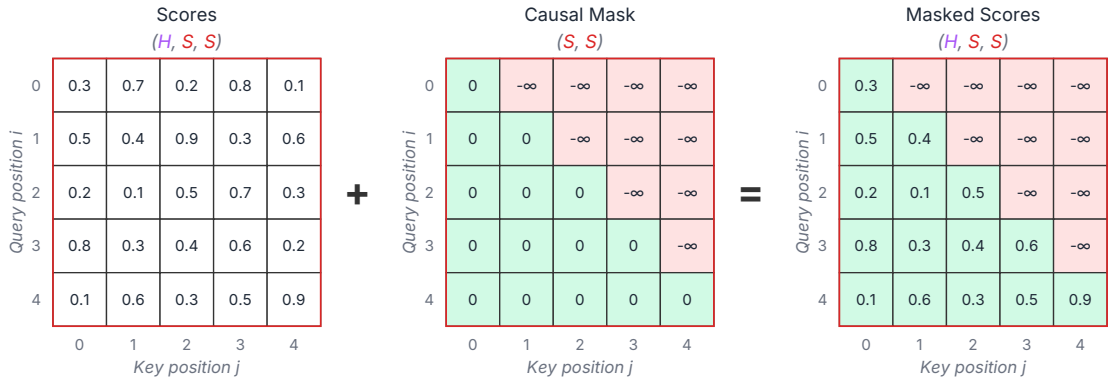


Figure 2.10.: How the causal mask is applied in self-attention, shown for only one attention head for $\mathbf{S} = 5$

Note that the mask is shape (**S**, **S**), so it has to be broadcast across the **H** dimension in order to be added to the scores tensor. After adding the mask, we apply SoftMax along the key dimension (the last dimension). Any number added to $-\infty$ results in $-\infty$, and these $-\infty$ values become zero after SoftMax. This prevents the masked-out positions from getting any attention weight, ensuring each position's attention weights sum to 1 over only the allowed (past and current) positions.

Weighted sum and output projection

The SoftMax output gives us the attention weights: a (**H**, **S**, **S**) tensor where each row sums to 1. We use these weights to compute a weighted sum of the value vectors:

$$\text{Weighted Values} = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_K}} + \text{mask}\right)V$$

The attention weights have shape (**H**, **S**, **S**) and V has shape (**H**, **S**, d_K), so the result has shape (**H**, **S**, d_K). We then reshape this back from (**H**, **S**, d_K) to (**S**, **D**) by concatenating the **H** heads, and apply a final output projection W_O of shape (**D**, **D**):

$$\text{Output}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H) \cdot W_O$$

The full self-attention block transforms an input of shape **(S, D)** to an output of the same shape **(S, D)**. The shape is preserved.

Parameter count

For standard multi-head attention with **D** = 8192, the parameter count per layer is:

- W_Q: D × D = 67.1M
- W_K: D × D = 67.1M
- W_V: D × D = 67.1M
- W_O: D × D = 67.1M
- **Total**: $4D^2 \approx 268$M parameters per attention layer

With 80 layers, the attention parameters alone account for about 21.5B of Llama 3 70B's parameter count. The rest is mostly in the feed-forward layers, which we'll cover in Section 2.6.

> **Note**
>
> **Grouped-Query Attention (GQA) and friends.** Modern models often use fewer key-value heads than query heads. Llama 3 70B uses GQA (Ainslie et al. 2023) with 64 query heads but only 8 KV heads, reducing W_K and W_V from **(D, D)** to **(D,** $8 \times d_K$**)** = (8192, 1024). This saves parameters and, more importantly, reduces the KV cache size by 8x. Multi-Query Attention (MQA) (Shazeer 2019) takes this further with a single KV head. Multi-head Latent Attention (MLA) (A. Liu et al. 2024a) compresses the KV cache through learned low-rank projections. These attention variants are covered in Section 4.4.

2.5. The KV Cache

The KV cache is one of the most important concepts in LLM inference. It's a simple idea with far-reaching consequences for memory management, scheduling, and system design. This section explains why the cache exists, what it stores, and how much memory it consumes.

Training vs. inference

During training, the full sequence is available upfront. The model processes all **S** tokens at once, computing Q, K, and V for every position simultaneously. The causal mask ensures that position i only attends to positions $0, 1, \ldots, i$, but all positions are computed in parallel. There's no need to cache anything — the K and V tensors for all positions are computed in one pass, used immediately, and never needed again.

During standard inference, tokens are generated one at a time. At decode step t, the model generates token t based on all previous tokens $0, 1, \ldots, t-1$. The new token's query needs to attend to the keys and values of all previous positions. This creates a choice:

The naive approach: at each step, feed all t tokens through the model from scratch. This recomputes K and V for positions 0 through $t-1$, even though they haven't changed since the last step. Each token t uses computation $O(t^2)$ for attention, and over $\mathbf{S}$ decode steps, the total computation for long sequences is $O(\mathrm{S}^3)$. We recompute increasingly longer sequences for each new token, and this is extremely wasteful.

The KV cache approach: store the K and V tensors from all token positions, accumulating one more of each with each step. At each decode step, compute K and V for *only* the new token, append them to the cache, and use the full cached K and V for the attention computation. This reduces the per-token computation to $O(t)$ and the total computation to $O(\mathrm{S}^2)$ across all decode steps. The naive and KV cache approaches are contrasted in Figure 2.11.

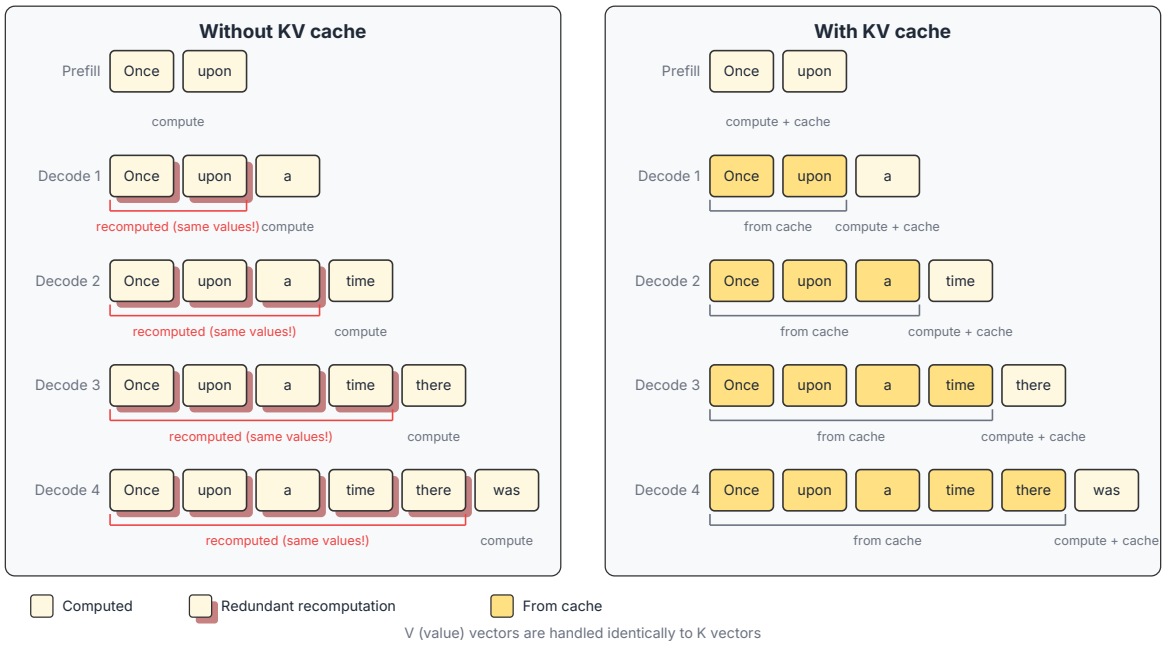


Figure 2.11.: The KV cache avoids wasted recomputation of keys and values that do not change between decode steps

When we do inference with a KV cache, the attention calculation becomes much less computationally expensive because our query tensor shrinks from $(\mathbf{S}, \mathbf{D})$ to $(1, \mathbf{D})$. You can see this effect by comparing the original data flow for attention in Figure 2.9 to the KV cache attention data flow in Figure 2.12.

Memory consumption

The KV cache stores keys and values for every layer, every KV head, every cached position, and every element of the head dimension. The total memory for a single request is:

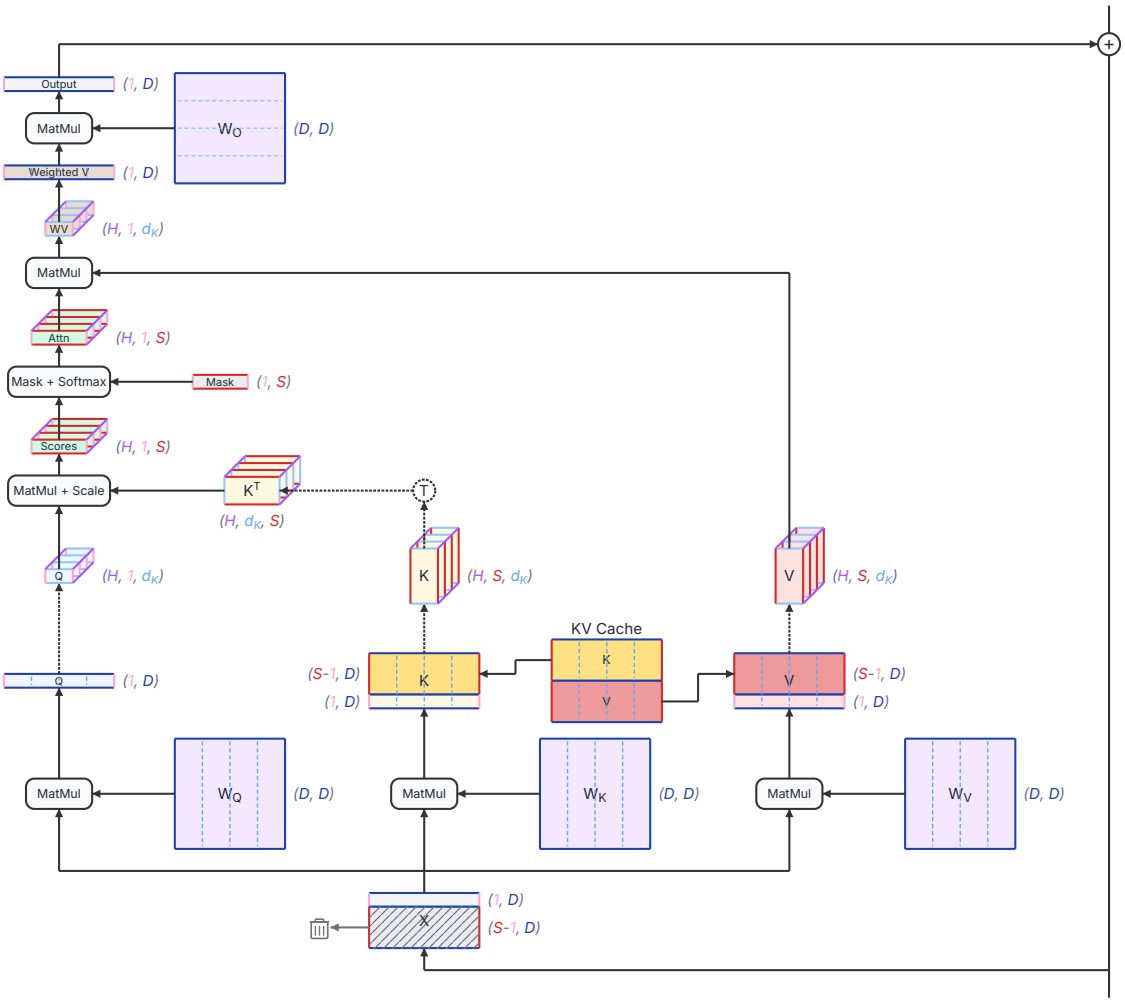


Figure 2.12.: Data flow for multi-head attention when using a KV cache

$$\text{KV cache memory} = 2 \times L \times H_{\text{KV}} \times S \times d_K \times b$$

where **L** is the number of layers, H_{KV} is the number of key-value heads (which is the same as **H** in a vanilla attention architecture), **S** is the current sequence length, d_K is the head dimension, and b is the number of bytes per element (2 for FP16/BF16).

For Llama 3 70B with GQA (8 KV heads), FP16 precision, and a 4096-token sequence:

$$2 \times 80 \times 8 \times 4096 \times 128 \times 2 \approx 1.34 \text{ GB}$$

That's 1.34 GB for a *single request*. If you're serving 40 concurrent requests, the KV cache alone consumes over 50 GB, which would be more than half of the memory on an 80 GB GPU that also needs to hold the model weights. This is why KV cache memory management is such a central concern in LLM serving systems.

The cache grows by a fixed amount at each decode step: $2 \times L \times H_{\text{KV}} \times d_K \times b$ bytes per token. For the Llama 70B numbers above, that's about 328 KB per token per request.

> **Note**
>
> **Forward references**: The KV cache drives many of the optimization techniques covered in the main chapters. Section 4.4 covers architectural techniques including reducing the number of KV heads, reducing the size of KV entries, and sharing KV cache entries across layers. Section 6.3 covers runtime techniques for managing the cache including PagedAttention (Kwon et al. 2023a) which eliminates memory fragmentation, quantization which reduces the bytes per element, and eviction policies which remove cache entries under memory pressure.

Attention alternatives

Standard multi-head attention dominates deployed LLMs today, but it's not the only option. **State-space models** like **Mamba** (Gu and Dao 2023) and **linear attention** variants such as **Kimi Linear** (Kimi Team et al. 2025) replace the **(H, S, S)** attention matrix with a fixed-size recurrent state. Instead of caching all past keys and values (which grows linearly with sequence length), they maintain a constant-size state that gets updated at each step. This eliminates the problem of KV caches growing large, giving a fundamentally different inference profile — constant memory regardless of sequence length, instead of memory that grows linearly with it.

These architectures are out of scope for this book. We focus on standard multi-head attention because it remains dominant in the models that practitioners actually deploy at scale. There are model variants that reduce the number of layers running multi-head attention, but so far there are no top-performing models which exclusively use attention alternatives. It's worth knowing that alternatives exist, especially since the KV cache is a major source of inference

complexity, but so far nothing less expensive has matched the capabilities of multi-head attention.

2.6. The Feed-Forward Network

Every transformer layer contains two main sub-layers: self-attention and a **feed-forward network (FFN)**, also called the **MLP** (multi-layer perceptron) block. While self-attention handles cross-position communication, the FFN operates on each position independently, transforming the representation at each position through nonlinear learned mappings. The FFN is where the majority of a model's parameters live.

Standard FFN

The original transformer (Vaswani et al. 2017) uses a two-layer FFN with a ReLU activation:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

We can see the data flow in Figure 2.13. For simplicity, we do not show the bias terms, which could be abstracted as part of the matrix multiplication. The input x has shape **(S, D)**. The first linear layer projects up from **D** to an intermediate dimension **F**, originally **F** = 4**D**. The activation function is applied element-wise. Then, the second linear layer projects down from **F** back to **D**. When **F** = 4**D**, each weight matrix has $4D^2$ parameters, for a total of $8D^2$ parameters.

Gated MLP (SwiGLU)

Modern LLMs like Llama and PaLM don't use the standard FFN. They use a **gated** variant of the MLP called **SwiGLU** (Shazeer 2020), which adds a third linear layer and uses a gating mechanism:

$$\text{FFN}_{\text{SwiGLU}}(x) = (\text{SiLU}(xW_{\text{gate}}) \odot xW_{\text{up}})W_{\text{down}}$$

where $\odot$ denotes element-wise multiplication and SiLU (also called Swish) is the activation function $\text{SiLU}(x) = x \cdot \sigma(x)$.

The flow for the gated MLP is shown in Figure 2.14. The up and gate projections run in parallel, both mapping from **D** to a higher intermediate dimension **F**. Their outputs are combined by element-wise multiplication. The intuition is that the gate values select which features from the up-projection pass through. When an element of the gate is 0, the corresponding up projection element is zeroed out. When an element of the gate is 1, the corresponding up projection element is passed through as is. The gated up projection result is then projected down back to **D**.

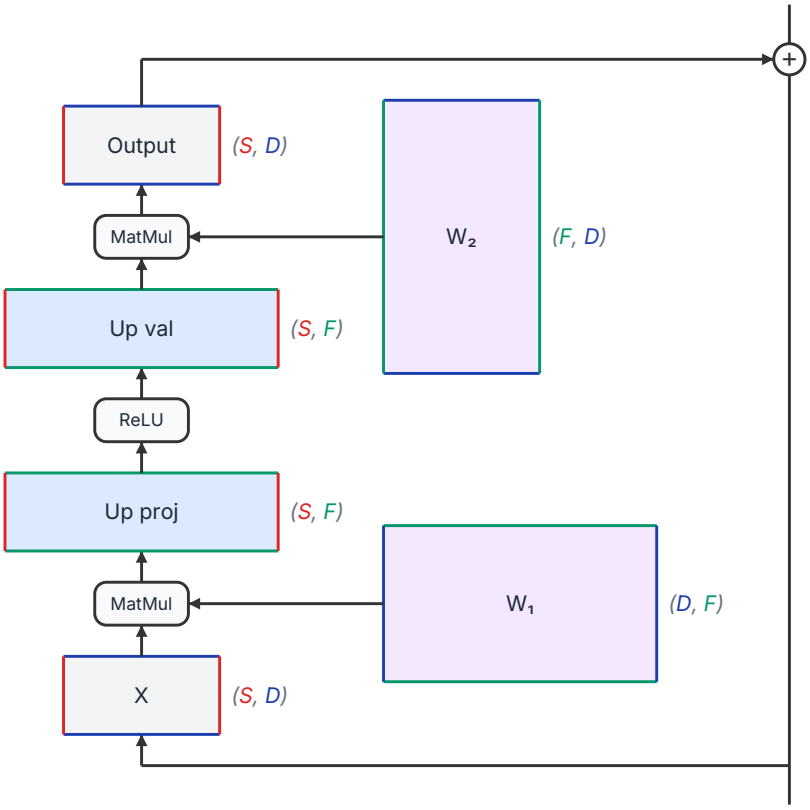


Figure 2.13.: Data flow of weights and activations in a standard MLP layer

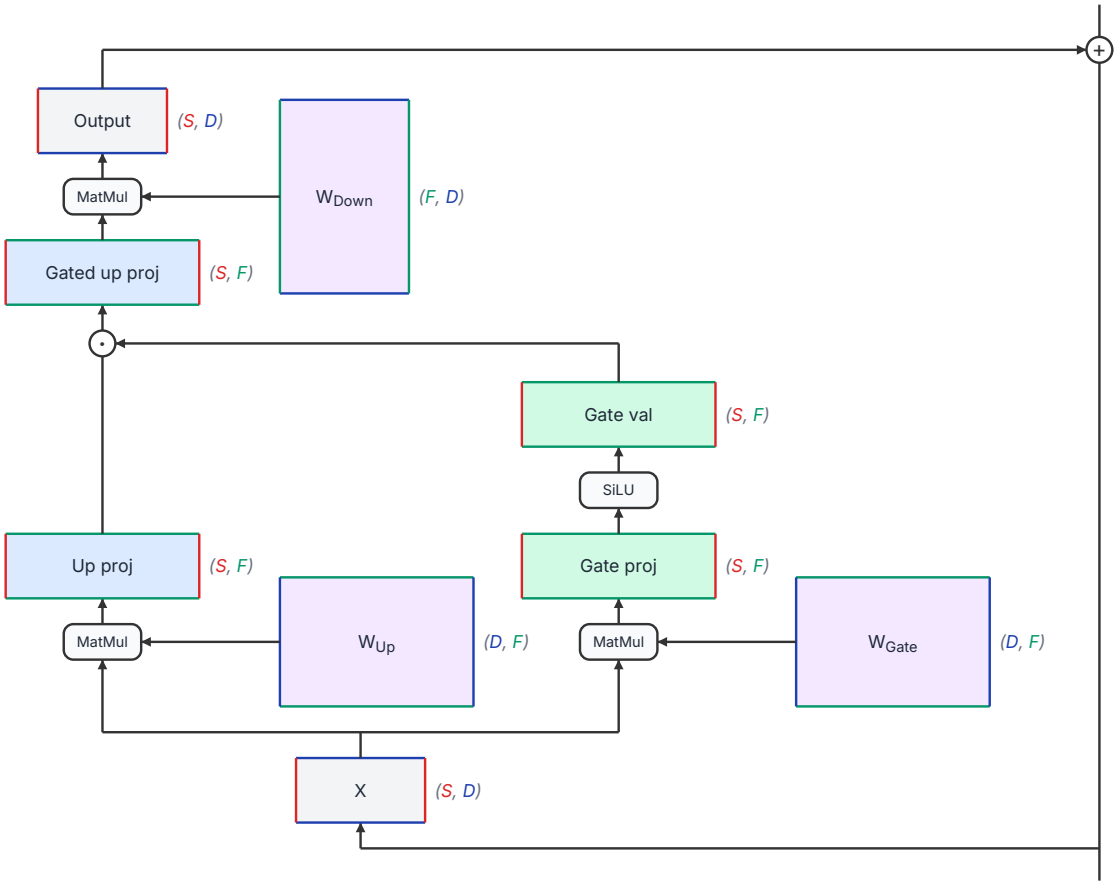


Figure 2.14.: Data flow of weights and activations in a gated MLP layer

Why go from two weight matrices to three? The gating mechanism has been shown empirically to improve model quality for a given parameter budget (Shazeer 2020). The intermediate dimension **F** is adjusted downward (from 4D to roughly $\frac{8}{3}$D) to keep the total parameter count approximately equal: $3 \times \mathrm{D} \times \frac{8}{3}\mathrm{D} = 8\mathrm{D}^2$, the same as the standard FFN's $2 \times \mathrm{D} \times 4\mathrm{D} = 8\mathrm{D}^2$ parameters. With **D** = 8192, the parameter count is $8\mathrm{D}^2 \approx 537$M parameters per FFN layer. The FFN layers have double the number of parameters as the attention layers, so the FFNs comprise about two-thirds of the parameters, and the attention layers have about one-third of the parameters in a vanilla, traditional LLM.

Mixture of Experts (MoE)

Standard LLMs have one MLP per layer, and they have no choice but to apply the same MLP to every token. **Mixture of Experts (MoE)** architectures replace the single MLP with multiple MLPs called **experts** and a lightweight **router** that selects which experts to use for each token. Because not all of the MLP weights are used for each token, this is often referred to as a **sparse MLP**. In contrast, the original MLP architecture is referred to as a **dense MLP** architecture.

Figure 2.15 shows the data flow for an MoE MLP layer. The router in MoE is usually a simple learned linear layer that produces a score for each expert. If there are **E** experts, the router has a weight matrix of shape **(D, E)**. There are different designs for how the experts are chosen, and one of the simplest methods is for the router to select the top k experts with the highest scores. These top-k experts are said to be **active**. Note that there is a router in every layer, so the choice of experts differs each layer — this is not at all like having **E** separate full models which are chosen at the beginning based on the input token. The science of MoE layers is outside the scope of this book, but we will highlight a recent trend. Early MoE LLMs such as **Mixtral** (Jiang et al. 2024) had only a few experts, such as 8, and set k to a very small value, such as 2. More recent LLMs such as **Qwen 3** (Yang et al. 2025) have hundreds of experts and set k to a higher number, such as 8. They also have the concept of always-on experts called **shared experts**. Figure 2.15 shows what the vanilla data flow looks like in an MoE configuration with 6 total experts, 2 of which are active.

The outputs of the selected experts are combined using the router's SoftMax weights.

The key trade-off of MoE for inference is clear: **memory footprint is large** (all expert weights must reside in GPU memory) but **per-token compute is modest** (only k of **E** experts are active and consuming FLOPs). Mixtral 8×7B (Jiang et al. 2024) has roughly 47B total parameters but only ~13B active per token. This means the model takes up as much GPU memory as a 47B dense model but computes each token about as fast as a 13B dense model.

For multi-GPU serving, MoE models naturally lend themselves to **expert parallelism** — placing different experts on different GPUs. But this introduces all-to-all communication as tokens are routed to the GPU holding their selected expert. Load balancing across experts also matters: if the router sends most tokens to the same few experts, some GPUs do much more work than others. We'll discuss expert parallelism in Section 7.5.

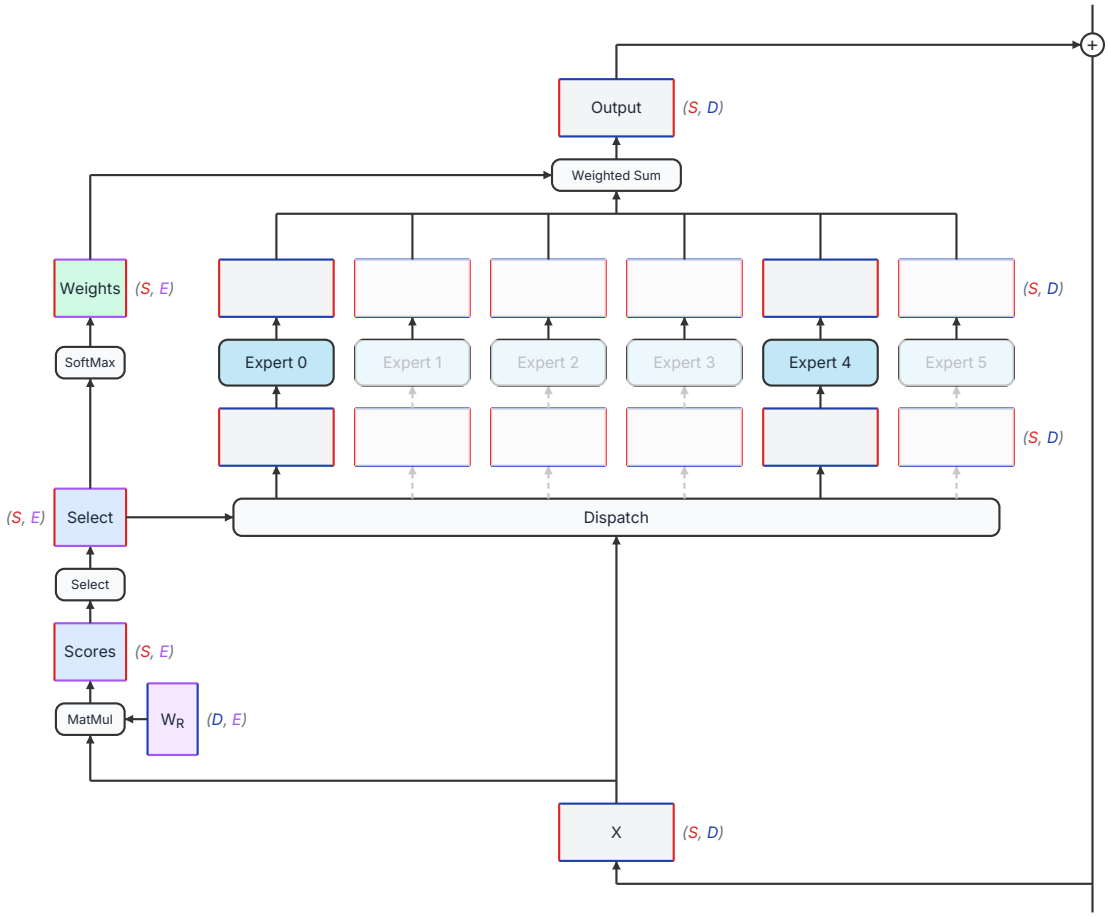


Figure 2.15.: Data flow of weights and activations in an MoE MLP layer. This figure depicts a model with 6 experts, 2 of which are active.

Why the FFN matters for inference

The MLP layers typically account for roughly two-thirds of a model's parameters. On one hand, all these parameters require a lot of resources. All of the active parameters must be loaded from GPU memory as computational inputs, so this costs bandwidth. These active parameters are used in matrix multiplication, requiring a lot of compute. On the other hand, modern GPUs are highly optimized for this use case of loading tensors and doing matrix multiplication. GPUs perform the FFN calculations quickly, and there's not much that can be done to speed them up other than not doing them.

We wrap up the data flow conversation here. With this background into the nuts and bolts of the data and operations inside the LLM, we can now shift our attention to the bigger picture of how the model forward pass is used in inference and how we can measure and assess inference performance.

2.7. Further Reading

The original transformer paper (Vaswani et al. 2017) is the essential starting point. Even though the decoder-only variant drops half the architecture, the core ideas — scaled dot-product attention, multi-head attention, positional encoding, and the feed-forward network — are all introduced there.

For intuitive explanations of the transformer architecture:

- Jay Alammar's "The Illustrated Transformer" (Alammar 2018) is considered a classic for explaining the original transformer architecture, with many beautiful diagrams.
- 3Blue1Brown's deep learning series (Sanderson 2024) provides what may be the clearest visual explanation of attention and the transformer architecture available anywhere. Grant Sanderson's animations build the mechanism step by step, making the matrix operations and information flow genuinely intuitive.
- "The Annotated Transformer" (Rush 2018) walks you through Python code to understand the original transformer architecture, and is incredibly well written.
- Peter Bloem's "Transformers from Scratch" (Bloem 2019) builds the architecture from first principles with clear diagrams.
- Lilian Weng's "The Transformer Family Version 2.0" (Weng 2023b) offers a more technical treatment, surveying transformer variants including the attention alternatives mentioned in Section 2.4.
- Andrej Karpathy's "Let's build GPT: from scratch, in code" (Karpathy 2023) walks through a complete implementation in about two hours.

For interactive visualizations that let you explore the architecture hands-on:

- Brendan Bycroft's "LLM Visualization" (Bycroft 2023) is a 3D interactive walkthrough of GPT-style inference that lets you trace data flow down to every add and multiply. It is an excellent companion to the tensor data flow diagrams in this chapter.

- "Transformer Explainer" (Cho et al. 2024) from Georgia Tech runs a live GPT-2 model directly in the browser. It uses Sankey-style diagrams showing how data flows through embeddings, attention heads, and transformer blocks, and lets you type in your own text and watch the model predict the next token in real time.

For the specific architectural choices used in LLMs, the Llama papers (Touvron, Lavril, et al. 2023; Touvron, Martin, et al. 2023; Grattafiori et al. 2024) are a good reference because they clearly document the model dimensions and design decisions (SwiGLU, RoPE, GQA). The PaLM paper (Chowdhery et al. 2022) provides a thorough analysis of scaling behavior that motivates the decoder-only design. For more modern LLMs, Sebastian Raschka does a brilliant job covering all of the different architectures. A good starting point for his writings is "LLM Architecture Gallery" (Raschka 2025).

The MQA (Shazeer 2019) and GQA (Ainslie et al. 2023) attention variants are covered in Section 4.4. FlashAttention (Dao et al. 2022) is covered in Section 6.1.

3. Measuring LLM Inference

Now that we are familiar with the decoder-only LLM and how data flows through it, let's go deeper into the autoregressive nature of standard text generation and discuss how to quantify its performance. We will develop an understanding of the stages of inference, how we measure inference, and the performance model that underlies our computing hardware, in particular GPUs.

3.1. The Inference Pipeline

From the time you send a prompt to an LLM until you get a response back, a lot happens between those two moments. The full journey of a single request breaks down into a few stages. Some happen once per request and some repeat for every output token. Understanding the characteristics of the stages will help you focus your attention on the ones that actually dominate wall-clock time. In this section, as with most of the book, we will assume the LLM is running on a computer with a GPU. The principles will be the same if it is running on CPU, TPU, or other hardware.

The stages of inference

Let's walk through the stages needed to process a user request. We will break this down into four stages that have distinct properties in terms of when they happen, where they happen, and what kinds of resources they consume. These stages are shown in Figure 3.1.

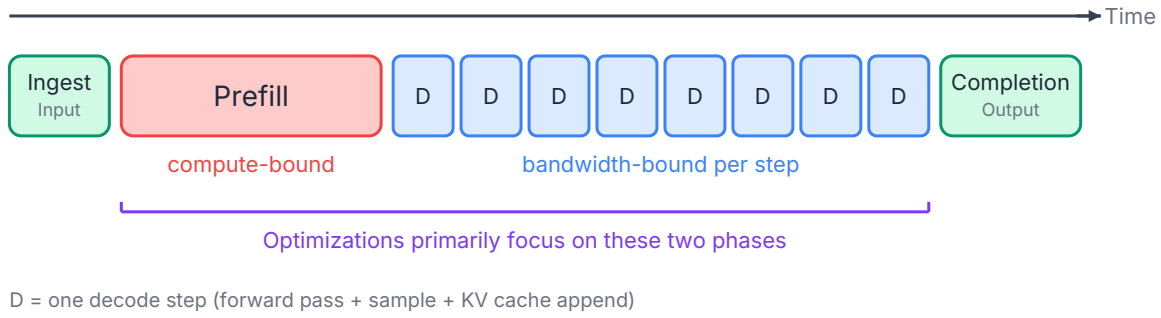


Figure 3.1.: The stages of LLM inference for a single request. Ingestion and response completion are fast. Prefill and decode dominate wall-clock time.

We will assume the model is already loaded into GPU main memory. For a 70-billion-parameter model stored in FP16 (2 bytes per parameter), the weights alone are about 140 GB. Loading that volume of data from an NVMe SSD at 7 GB/s takes about 20 seconds, and from a network file system, it could take longer. Because this is so slow, on typical inference software, the weights are loaded into GPU memory at startup and they stay there for the lifetime of the serving session.

Ingestion is the entry point for each request. The system takes the user's prompt and forms a new request. The user's prompt may be merged with other content, such as a system prompt. Next, this finalized raw text is tokenized — converted from characters into the integer token IDs that the model actually works with — and any additional validation or preprocessing is applied. This stage is fast compared to the resource-intensive work involving the neural network model that follows and is typically handled on the CPU. We are skipping many details about the steps within ingestion because what they have in common is that they usually happen once and are relatively fast. One exception, which can significantly harm user experience, is request queuing. In a production system, requests may wait in a queue until there are resources available to process them, and if the system has multiple priority levels, lower priority requests may wait longer. Queuing time can be a significant contributor to overall latency, even though it doesn't consume compute resources. The timing and granularity of scheduling request work can dramatically improve performance (up to the limits of system resources), and we will discuss scheduling in Chapter 5.

Prefill is where the heavy lifting begins. The model processes the entire set of input tokens in a single forward pass through the entire network — all layers, all attention heads, all FFN blocks. In particular, the attention layers calculate attention scores across all of the input tokens. Since the attention scores are a square whose sides are the number of tokens, the compute needed for attention grows quadratically with respect to the sequence length, and it becomes very expensive when there are many input tokens. Because all input tokens are known up front, the attention calculation can be done all at once using a large matrix-matrix multiplication between queries and keys. Despite how efficiently modern GPUs perform matrix multiplication, the size of the matrices involved means that prefill for all but the shortest prompts is **compute-bound** — the bottleneck is how fast the GPU can do the arithmetic, not how fast it can move data into the computational cores.

During prefill, each attention layer computes query, key, and value tensors for all of the input tokens. The keys and values will be needed again in the upcoming decoding stage. Rather than throwing these away and recomputing them during decoding, the system stores them in the **KV cache**, which we introduced in Section 2.5. Initializing the KV cache is the primary function of the prefill stage in modern LLM serving systems. The KV cache holds the key and value tensors for every attention head at every layer, for all tokens processed so far. The KV cache trades memory for compute: it uses GPU memory to store these tensors, but it eliminates the need to recompute them. The size of the KV cache created during prefill is:

$$\text{KV cache memory} = 2 \times L \times H_{\text{KV}} \times S \times d_K \times b$$

where the factor of 2 accounts for storing both keys and values, **L** is the number of layers, H_{KV} is the number of key-value heads (which may be less than the number of query heads,

as will be discussed in Section 4.4), S is the current sequence length, d_K is the dimension of each head, and b is the number of bytes per element (2 for FP16/BF16).

For a concrete example, consider a Llama-style 70B model with 80 layers, 8 KV heads per layer, head dimension of 128, and FP16 KV cache entries (2 bytes each). For a 4,096-token prompt:

$$\text{KV cache} = 2 \times 80 \times 8 \times 4096 \times 128 \times 2 \approx 1.34 \text{ GB}$$

That's 1.34 GB of GPU memory consumed by a single request's KV cache, just from the input tokens. This memory usage is one of the primary constraints on how many requests can be served concurrently, which we'll explore in Section 4.4, Section 6.3, and other sections.

After prefill completes, the system has two things: the KV cache populated with entries for all input tokens and the logits for the last input position's next token prediction, the latter of which are used to generate the first output token.

> **Note**
>
> For a detailed look at exactly how the key, value, and query projections are computed during the transformer's forward pass and how they relate to the KV cache, refer to sections 2.4 and 2.5.

Decode is the phase where the LLM's response is generated (except for the first token). The model typically produces output tokens one at a time, where each new token depends on all previously generated tokens. This sequential dependency comes from how the model defines the probability of an output sequence. Given input tokens x, the probability of generating output sequence $y = (y_1, y_2, \ldots, y_T)$ is decomposed using the chain rule of probability, where each output token's probability distribution is conditioned on both the prompt x and all of the prior output tokens:

$$p(y \mid x) = \prod_{t=1}^{T} p(y_t \mid y_{<t}, x)$$

The model estimates each conditional probability $p(y_t \mid y_{<t}, x)$ with a forward pass. Because each token's probability depends on all previously generated tokens, the model doesn't generate token y_t until it has committed to tokens y_1 through y_{t-1}. This gives us the familiar autoregressive loop:

- Run a forward pass for one new token, reading all active weights and the full KV cache
- Obtain the logit distribution over the vocabulary
- Sample a token
- Append new key-value entries to the KV cache
- Repeat until the model generates a stop token or hits a length limit

Each iteration of this basic loop is called a **decode step**. The forward pass in a decode step processes just one token (or one token per batch entry, in a batch), so the query-key operations are **matrix-vector multiplications** rather than the matrix-matrix multiplications of prefill. The keys are still a matrix of **S** vectors, but now they're being multiplied by a single query vector instead of a matrix of **S** vectors.

The one-token-at-a-time nature of typical decoding is the central challenge of LLM inference. Each decode step involves reading all the model weights and the full KV cache, but only to compute a single token's worth of output. The ratio of computation to data movement is very low, which makes decode steps typically **memory-bandwidth-bound**. Here, the bottleneck is how fast the GPU can read data from GPU memory, not how fast it can do math. A large portion of this book is about techniques to make this phase less painful.

It's worth noting that the left-to-right, one-token-at-a-time autoregressive factorization is not the *only* mathematically valid way to express the joint probability of a sequence. There are other approaches that calculate the same joint distribution differently. The left-to-right decomposition is a convenient modeling choice, but it imposes a strict sequential dependency at inference time. Several techniques in Section 6.5 — speculative decoding, multi-token prediction, and Medusa heads — work by finding ways to break or shortcut this sequential dependency while preserving the same output distribution.

At the end of each decode step, **sampling** selects the next token. After the forward pass has produced a logit vector over the vocabulary, one token ID is selected. Common strategies include:

- greedy decoding (always pick the highest-probability token),
- temperature sampling (scale logits before sampling randomly),
- top-k sampling (restrict to the k most probable tokens), and
- top-p / nucleus sampling (restrict sampling to the smallest set of tokens whose cumulative probability exceeds p)

Sampling adds negligible overhead relative to the preceding forward pass. The forward pass involves billions of multiply-accumulate operations across all the model's layers. Sampling merely involves a SoftMax over the vocabulary (typically 30K-200K entries) and a random draw, so sampling is not where all the time goes.

Completion wraps things up. The model has finished because it either generated a stop token, hit a maximum length limit, or has been interrupted by the serving system. The accumulated response is detokenized back into text, the KV cache memory is freed, and the result is returned to the user. In most production deployments, the response is **streamed** back to the user as tokens are generated, rather than waiting for the full response to complete. Streaming doesn't change what's happening on the GPU — each decode step produces the same token whether it's streamed immediately or batched for delivery later — but it reduces how long users have to wait to see parts of the LLM response. Whether batch or streaming, completion steps are like ingestion in that they are usually fast one-time operations that happen on the CPU.

Where the time goes

Of the four stages above, the vast majority of compute and memory bandwidth is consumed by prefill and decode. Ingestion and completion are fast operations on CPU that together account for a tiny fraction of request latency. Note that we are excluding queuing time from ingestion for now, treating that as a fixed cost associated with the workload. Queuing time can be long for a system under heavy load, and various techniques associated with scheduling are discussed in Chapter 5. Aside from scheduling, most of the techniques covered in this book target prefill and decode, because that's where the time goes and where the greatest opportunity for improvement lies.

Two fundamentally different regimes

While prefill and decode are both expensive, the contrast between the types of resources they need is worth emphasizing.

During prefill, we have a large set of tokens to process, so we can keep the GPU's compute units busy with very large matrix multiplication operations. During decode, we're typically generating one token at a time, so we need many forward passes which each require much smaller computations that usually involve reading a lot of data.

This means prefill and decode need different resources from the hardware. Prefill wants raw compute horsepower. Decode wants memory bandwidth. As we'll see in Section 3.4, the roofline model gives us a precise way to reason about where the boundary between these two bottlenecks falls for a given GPU or hardware accelerator.

Many of the optimizations covered in this book target just prefill or just decode, and some of the techniques span between the two of them. Understanding the fundamental difference in resource needs between prefill and decode will provide a good foundation for everything that follows.

3.2. Performance Metrics

Now that we have a mental model of what happens during inference, the natural next question is: how do we know if our system is doing it *well*? This section introduces the metrics you'll encounter in conversations about LLM serving. We'll group them by what they measure, as different concerns utilize different metrics.

User-facing latency metrics

The metrics that matter most in interactive applications are the ones users actually feel.

Time to First Token (TTFT) is the wall-clock time from when a request is submitted to when the first output token is produced. In a chat application, this is how long the user stares at a blank screen before text starts appearing. TTFT is primarily impacted by the

duration of the prefill phase. For the other phases contributing to TTFT, ingestion usually is fast, we don't need to decode many tokens to start streaming text to the user, and completion is also fast. Since the system has to process the entire input before it can begin generating any output, longer prompts mean longer prefill, which means higher TTFT.

Note that TTFT isn't as simple as the prefill duration plus fixed costs for the other phases. In systems under heavy load, the request may sit in a queue before being serviced. Also, in streaming deployments, the first token might be generated by the model but not immediately delivered to the user. Tokens can be buffered for various reasons, including network batching, detokenization requiring multiple tokens to resolve a word boundary, or framework-level batching of streamed chunks. So the TTFT the user experiences can be somewhat higher than the time the inference engine measures internally for prefill and generation of the first token.

Time Per Output Token (TPOT), also called **Inter-Token Latency (ITL)**, is the average time between successive output tokens during generation. This is what governs how fast text appears to stream after that first token shows up. TPOT is a function of the decode process. When each decode step produces one token, TPOT is essentially the time per decode step. For a fluent reading experience in interactive applications, you generally want TPOT below about 50-100 milliseconds (which equates to 10-20 tokens per second). Go much higher, and the output starts to feel sluggish.

TPOT is an average, and individual decode steps can vary. The KV cache grows with each token, so later decode steps read slightly more data than earlier ones. In practice, this variation is usually small relative to the total step time, but it can matter for very long sequences.

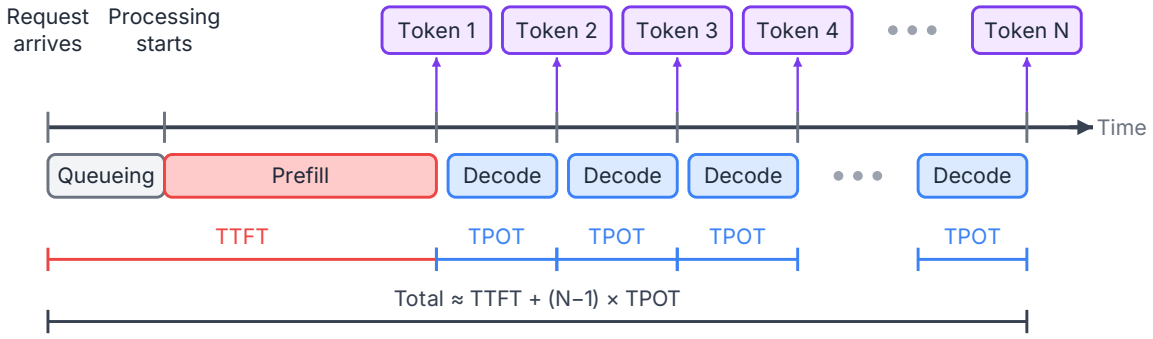


Figure 3.2.: Request timeline showing TTFT and TPOT durations

Together, TTFT and TPOT define the end-to-end latency of a request, as shown in Figure 3.2. The total time to generate a response of N output tokens is approximately $\text{TTFT} + (N - 1) \times \text{TPOT}$. For short outputs, TTFT dominates. For long outputs, TPOT dominates. This is why the workload shape — the ratio of input length to output length — matters so much for understanding metrics and performing optimizations, as we'll discuss in Section 3.3.

Throughput

While latency metrics describe the experience of a single request, **throughput** measures how much total work the system gets done. The most common unit is **tokens per second (TPS)** — specifically, output tokens per second combined across all requests currently being served. You'll also see **requests per second (RPS)** and **requests per minute (RPM)**, which measure how many complete requests the system finishes in a given time window. Note that a system processing one request that generates 100 tokens per second has 100 TPS, and a system processing 100 requests that each generate 1 token per second also has 100 TPS. RPS is highly dependent on the workload.

The latency–throughput tradeoff

Latency and throughput are in fundamental tension, and this tradeoff is at the heart of inference system configuration.

One key mechanism impacting latency and throughput is **batching**. If you process requests one at a time, each request gets the lowest possible latency while being serviced, since the hardware is entirely dedicated to that single request. But the GPU is badly underutilized, especially during decode, where a single request doesn't generate enough arithmetic to keep the compute units busy.

If instead you batch multiple requests together, the GPU can do more useful work per memory read. Reading the model weights once and applying them to 32 requests in a batch is far more efficient than reading them 32 separate times, once per request. With larger batches, throughput goes up, but now each individual request might have to wait before processing begins, because the system is pausing to accumulate enough requests for a full batch. Also, each decode step takes slightly longer because there's more work per step, so with larger batches, latency goes up a little.

This tradeoff is not just theoretical. In production serving, batch configuration significantly impacts scheduler performance, as shown in Figure 3.3. A scheduler that waits longer to fill batches achieves higher throughput but increases TTFT for waiting requests. A scheduler that dispatches immediately may keep TTFT lower but hurts throughput. Finding the right operating point depends on the application's requirements, which is where the next metric comes in.

Goodput

Raw throughput doesn't tell the whole story. **Goodput** is throughput counting only responses that actually meet the application's requirements — its service-level agreement (SLA). The SLA for a chat application might specify a maximum TTFT, a maximum TPOT, a maximum end-to-end latency, or some combination of these.

A system can have impressive throughput numbers but poor goodput. If you push batch sizes so high that 30% of requests exceed the latency budget, those requests don't count toward

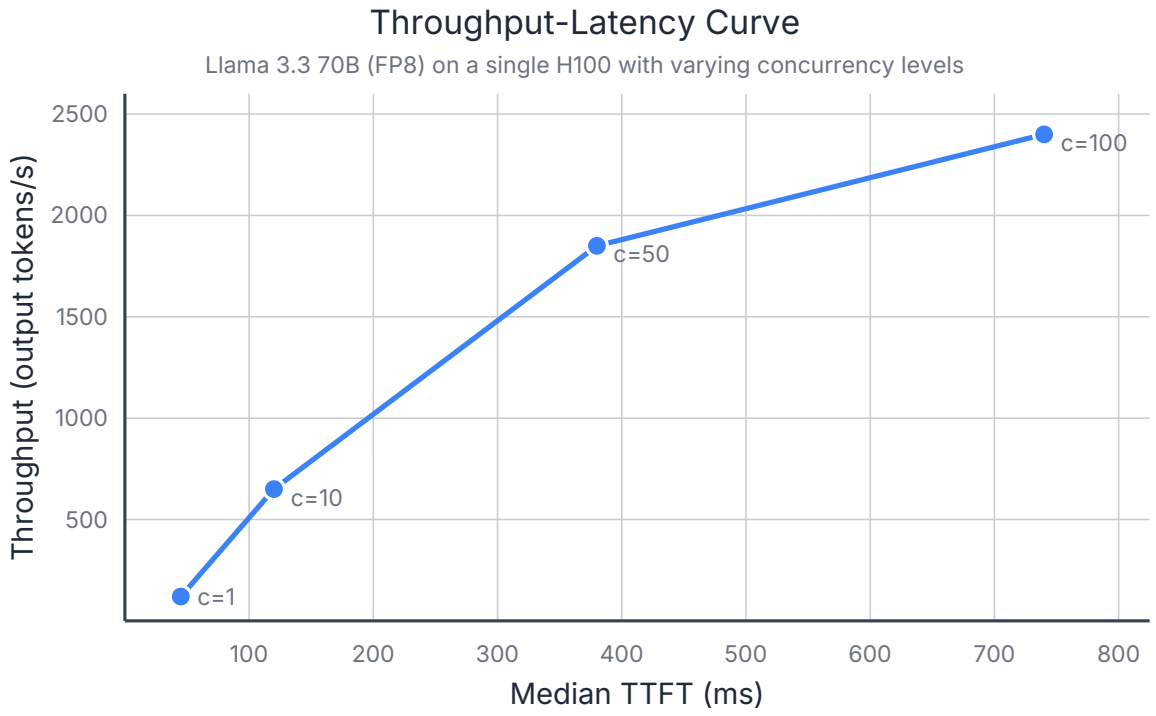


Figure 3.3.: Throughput-latency curve showing how throughput gains diminish as latency increases. Data from Spheron (2025), benchmarking Llama 3.3 70B (FP8) on a single H100 system running vLLM.

goodput, even though the system did the work. Similarly, if the system is so overloaded that it starts preempting or dropping requests, the raw TPS might look fine while the actual useful output is much lower. Goodput is the metric that connects basic system performance to whether users are actually getting a good experience.

Cost efficiency

At scale, inference cost is often a key optimization target. **Cost per token** measures how much it costs in dollars (or GPU-hours) to generate output tokens. It's roughly inversely proportional to throughput — if you double your throughput on the same hardware, you halve your cost per token. This means that most of the techniques covered in this book that improve throughput will reduce cost.

The picture gets more nuanced when you factor in different hardware devices (each with different cost and performance profiles), different precision formats, and the amortized cost of hardware across many workloads. But as a first approximation, optimizing for throughput is optimizing for cost.

While cost per token is often measured in aggregate, it can also be useful to break out cost between input and output tokens, or between prefill and decode. In fact, major providers usually charge separately for input and output tokens, reflecting the different costs for prefill and decode compute.

Hardware utilization metrics

The metrics above describe what the system achieves. The next two measure how efficiently it uses the hardware to get there. To diagnose whether you're maximizing the capabilities of your hardware, you need different metrics.

Model FLOPS Utilization (MFU) is the fraction of the GPU's peak floating-point throughput that is actually used for model computation. This metric was introduced in the PaLM paper (Chowdhery et al. 2022) and has become a standard efficiency metric. If your GPU can do 989 TFLOPS at FP16, and your model computation actually achieves 300 TFLOPS, your MFU is about 30%.

MFU is most useful for characterizing compute-bound workloads like prefill. A low MFU during prefill suggests that something is preventing the GPU from doing math at full speed — perhaps memory transfers, kernel launch overhead, or poorly shaped matrix multiplications. We'll see how to use the roofline model to reason about this precisely in Section 3.4.

Model Bandwidth Utilization (MBU) is the analogous metric for memory bandwidth. MBU is the fraction of peak memory bandwidth that is actually used for reading model data. During decode, the bottleneck is typically how fast you can stream model weights and KV cache entries from GPU main memory. If your GPU has 3.35 TB/s of memory bandwidth and your decode step actually achieves 2.5 TB/s, your MBU is about 75%.

MBU is a good diagnostic for decode activity. A low MBU during decode means the memory system is not being fully utilized — perhaps due to small batch sizes, kernel launch gaps between layers, or overhead from memory management. Together, MFU and MBU give you a quick read on whether your system is limited by compute efficiency, memory bandwidth efficiency, or something else entirely. Note that manufacturers often report peak compute and bandwidth numbers on very specific workloads that yield the highest values. You should not expect to get close to 100% MFU or MBU for real-world use cases.

Capacity metrics

These metrics describe the system's high-level resource utilization.

Concurrency is the average number of in-flight requests the system is serving simultaneously. This is primarily constrained by GPU memory: each active request needs space for its KV cache, and the KV cache grows with sequence length. If your GPU has 80 GB of HBM and your model weights consume 30 GB, you have roughly 50 GB left for KV caches (minus some overhead). How many requests fit in that budget depends on the model architecture, the precision of the KV cache, and how long the sequences are. The scheduler policy dictates whether to reserve memory conservatively or to pack it more aggressively and risk running out of memory.

GPU memory usage is the average amount of GPU memory that is being used. Usage is comprised of four components: model weights, KV cache, activations temporarily stored during the forward pass, and framework overhead (memory allocated by the CUDA runtime, the serving framework, and so on). The weights are fixed once the model is loaded. Activations are transient and relatively small. The KV cache is the variable that dominates, and it scales with both concurrency and sequence length.

Maximum sequence length supported is the longest sequence (counting both input and output tokens) that the system can handle. Note that even if the model was trained to support 128K tokens, your deployment might be limited to 32K if that's all the KV cache memory you can afford while maintaining your target concurrency. The serving framework typically sets this as a hard constraint at model load time.

Preemption metrics

When the system runs out of GPU memory to serve all active requests and their KV caches, something has to give. The scheduler may **preempt** a request, forcing it back to the waiting queue and freeing its KV cache entries to make more memory available. When the preempted request resumes, the system either recomputes its KV cache from scratch or swaps back in the entries it saved in CPU main memory at the time of preemption.

Three metrics diagnose preemption behavior: the **preemption rate** (how often preemption happens), **recomputation overhead** (the extra prefill work to rebuild evicted KV caches), and **swap latency** (the time to move KV cache data between GPU and CPU memory). These are important in high-load scenarios, because a system that preempts too aggressively

can waste significant resources on preemption activity. We will cover preemption policies and their tradeoffs in more detail when we discuss scheduling and memory management in Section 5.4.

Putting it all together

These metrics are not independent. Increasing batch size improves throughput and MBU but can hurt TTFT and TPOT. High concurrency improves throughput but can increase preemption rate. The art of inference optimization navigates these tradeoffs, trying to maximize user experience while minimizing cost.

3.3. Workload Patterns

Not all LLM requests look the same. A one-line classification question and a multi-page document summary put very different demands on the inference system, even if they're running on the same model with the same hardware. Before we get into the mechanics of how inference systems are built and optimized, it's worth stepping back and thinking about what kinds of work they're actually asked to do. The shape of the workload greatly impacts many metrics.

The input/output quadrants

The simplest way to characterize an LLM workload is by the length of the input and the length of the output. These two dimensions create four quadrants, shown in Figure 3.4, and each one has a distinct performance profile.

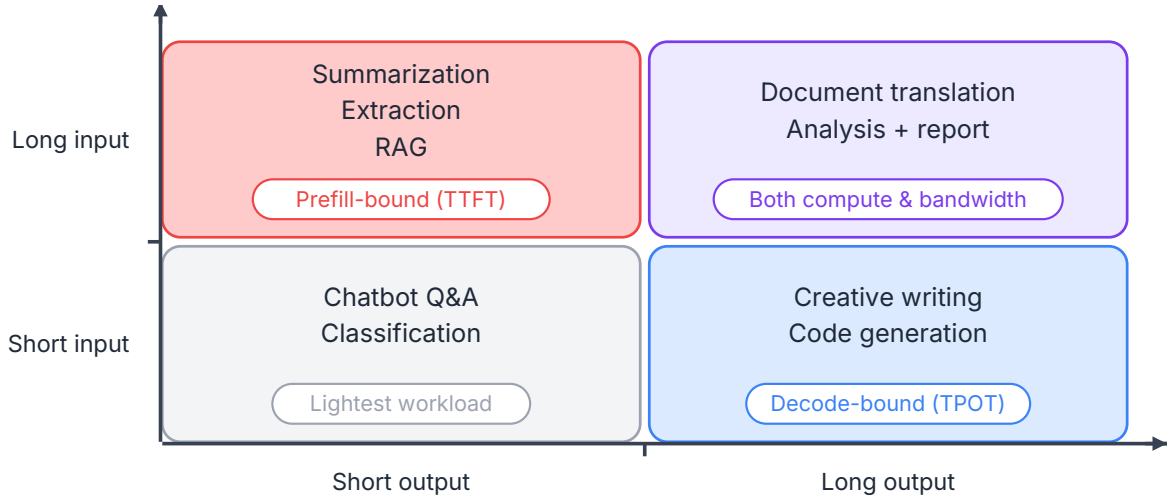


Figure 3.4.: The four workload quadrants defined by input and output length, with example scenarios

Short input, short output is the lightest quadrant. Think of a simple one-off chatbot exchange or a classification task: a sentence or two goes in, and a sentence or two comes out. Prefill is fast because there aren't many input tokens. Decode is fast because there aren't many output tokens to generate. The KV cache stays small. Individually, these requests are cheap. For these requests, serving systems will achieve the highest requests per minute, because these requests are so cheap.

Short input, long output is what you get with creative writing, open-ended code generation, or detailed explanations. Prefill is still quick, but decode dominates the request's lifetime because the model has to generate hundreds or thousands of tokens sequentially. The KV cache grows with each decode step, so memory pressure increases over the life of the request. For the user, what matters here is the sustained token generation rate — the TPOT we defined in the last section. A slow decode loop means a long wait for the full response.

Long input, short output is the pattern you see in summarization, information extraction, and retrieval-augmented generation (RAG). A large document or a long retrieved context goes in, and a relatively brief answer comes out. Prefill is expensive because the model has to process a large number of input tokens. But once prefill is done, decode finishes quickly. The KV cache is large from the start, but it doesn't grow much further. For the user, TTFT is the metric that matters most, since the wait is almost entirely in prefill.

Long input, long output is the most demanding quadrant. Document translation, detailed analysis with a lengthy report, and long multi-turn conversations all fall here. Prefill is expensive, decode is expensive, and the KV cache is large throughout. These requests consume the most GPU memory and occupy a batch slot for the most time. They're also the hardest to schedule efficiently alongside other requests, because they consume so many resources.

Why the quadrants matter

The quadrant a request falls into determines where the system's bottlenecks will show up. Long inputs mean expensive prefills and large initial KV caches. Long outputs mean many decode steps and growing KV caches. Sequence length directly affects how many requests can be batched together, because the KV cache for each active request takes up GPU memory. A batch of long-context requests might only fit a handful of concurrent requests, while a batch of short exchanges might fit many.

In practice, most production workloads are a mix of quadrants. A serving system might see a stream of short chatbot queries punctuated by occasional long summarization requests. The long requests consume disproportionate memory and compute, which can slow down the short requests sharing the same GPU. Managing this mix efficiently is one of the challenges of inference scheduling, which we'll explore in Chapter 5.

Batch vs. real-time inference

Beyond the shape of individual requests, the other major dimension of a workload is whether it runs in **batch** (offline) mode or **real-time** (online) mode. This distinction changes which

metrics you care about and how aggressively you can optimize.

In **batch inference**, you have a known set of requests to process and no one is waiting on the other end for an immediate response. Maybe the user is scoring a dataset, generating synthetic training data, or running evaluations. The metric that matters most is **throughput** — tokens per second across all requests — and its close relative, **cost per token**. Because there's no latency constraint, the goal is to take the time to plan the best way to minimize the total cost of processing these requests. You can fill batches to capacity, use cheaper or slower hardware, and let individual requests wait while the system finds the most efficient packing. Larger batches mean better GPU utilization, which means lower cost.

In **real-time inference**, a user (or another system) is waiting for a response, and how fast that response arrives directly affects the experience. Here, the metrics that matter most are **TTFT** and **TPOT**. A chatbot that takes five seconds to start responding feels broken, even if its overall throughput is excellent. This means the serving system has to balance resource utilization against latency. You can't hold requests in a queue too long to form larger batches, because that increases TTFT for every waiting request.

Some systems serve both kinds of traffic, and some offer different priority tiers — high-priority real-time requests that get scheduled immediately, and lower-priority requests that can wait to fill in the gaps. Batch inference is the easier workload to optimize for, so many of the optimizations in this book will focus on real-time latency issues. Latency metrics report the system speed, while goodput helps interpret system performance in a way that measures user experience.

Arrival patterns

The final piece of the workload picture is how requests arrive over time. Even in a real-time serving scenario, the demands on the system vary dramatically depending on the **arrival rate** and its **burstiness**.

If requests arrive at a steady, predictable rate, the system can maintain a stable batch size and keep GPU utilization relatively consistent. But real-world traffic is rarely that well-behaved. User-facing services see spikes — a sudden surge after a product launch, peak hours during the workday, or a viral moment that sends traffic through the roof. During a burst, the system has to either queue requests (increasing latency), drop them, or have enough spare capacity to absorb the spike.

The arrival pattern also affects how well the scheduling system can batch requests together. Requests that arrive close together in time can potentially share a batch. Requests that arrive in isolation may run in small, inefficient batches, wasting GPU compute. The more variable the arrival pattern, the harder it is to maintain consistently high utilization. We'll see in Section 5.1 how modern schedulers use techniques like continuous batching to adapt dynamically to changing arrival patterns, rather than waiting for a fixed batch to fill up.

In summary, understanding your workload — the mix of input/output lengths, the latency requirements, and the arrival patterns — is the first step toward choosing the right inference configuration. A system tuned for batch summarization of long documents looks very different

from one tuned for low-latency chatbot responses, even though they might be serving the same model.

3.4. The Hardware Model and Bottleneck Framework

In Section 3.1, we explained that prefill is compute-bound and decode is memory-bandwidth-bound. That's a good start, but to really be able to reason about optimizations more precisely, we need a more detailed picture of the hardware. This section covers just enough GPU architecture to build that picture, then introduces the roofline model as a framework for categorizing compute and memory bandwidth bottlenecks. This won't be a GPU deep dive or programming tutorial. We'll cover just enough to understand inference performance quantitatively.

> **Note**
>
> This book focuses on NVIDIA GPUs because they dominate the LLM inference landscape today. However, the bottleneck framework we're about to introduce applies equally to other hardware. For example, AMD's MI300X, Google's TPUs, and other accelerators all have different compute throughput ceilings and memory bandwidth limits, but the same approach to analyzing bottlenecks will work for those devices as well. We will briefly address alternative hardware in Chapter 7 when we discuss multi-device deployment.

GPU compute throughput

A modern GPU like the NVIDIA H100 is organized around **streaming multiprocessors (SMs)** that do the computing. The H100 SXM has 132 SMs, and each SM contains multiple types of execution units. For LLM inference, the most important of these are the **tensor cores**, the specialized hardware units designed to perform matrix multiplications very quickly. Tensor cores are what give modern GPUs their enormous throughput for the matrix multiplications that dominate neural network models. All of the SMs on a GPU are able to do work in parallel, which scales up their throughput.

The peak throughput of floating-point operations (assuming the SMs never have to wait for data) depends on the numerical precision you use. Lower precision means more operations per cycle, which translates directly into higher throughput. For the H100 SXM, a comparison of throughput for several data types is shown in Table 3.1.

Table 3.1.: Compute throughput figures for the NVIDIA H100 SXM. Source: NVIDIA (2023).

Precision	Peak throughput (TFLOPS)
TF32	494
FP16	989
BF16	989

Precision	Peak throughput (TFLOPS)
FP8	1,979
INT8	1,979

These numbers are theoretical peaks — the maximum the hardware can deliver if every tensor core is busy on every cycle. In practice, you never hit the peak, but it sets the ceiling. You get roughly double the throughput when you halve the precision. This is one reason model quantization (covered in Section 4.1) is such a useful optimization lever. When you reduce the precision of model weights and activations, you're saving memory and also increasing compute throughput.

The memory hierarchy

Raw compute throughput only matters if you can feed data to the tensor cores fast enough. This is where the memory hierarchy comes in. An NVIDIA GPU has four levels of memory, shown in Figure 3.5. The closest levels are the fastest but also the smallest, and as the levels get larger, they get farther and slower, trading off speed for capacity.

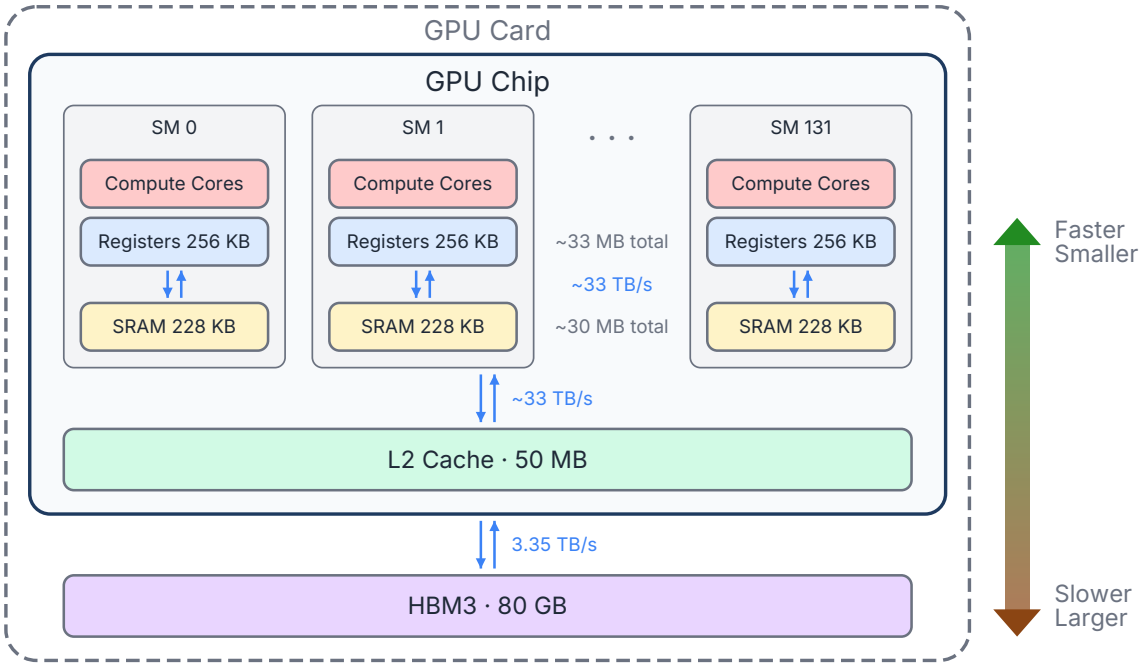


Figure 3.5.: GPU memory hierarchy showing registers, SRAM, L2 cache, and HBM with capacity and bandwidth at each level

Registers are the fastest storage, private to each thread. They're plentiful in aggregate (about 256 KB per SM on the H100, totaling roughly 33 MB across all SMs) but each thread running on an SM only sees a small portion. Data in registers is available essentially every clock cycle.

Shared memory and L1 cache (SRAM) sit on-chip, shared among threads within an SM. On the H100, each SM has 256 KB of combined shared memory and L1 cache, giving about 33 MB total across the chip. SRAM bandwidth is extremely high — on the order of tens of TB/s — but the capacity is small. This is the memory level that custom CUDA kernels like FlashAttention (Dao et al. 2022) exploit. By carefully tiling computations to fit in SRAM, they avoid repeated trips to the much slower main memory.

L2 Cache is a hardware cache that is not programmable. There is 50 MB of L2 cache total on the H100.

High Bandwidth Memory (HBM) is the GPU's main memory. This is the large off-chip memory where model weights, the KV cache, activations, and anything else that's not transient live. The H100 SXM has 80 GB of HBM3 with a peak bandwidth of approximately 3.35 TB/s. That sounds fast, and it is in absolute terms, but relative to the compute throughput, it's the bottleneck for most inference workloads. Every byte of model weights that needs to reach a tensor core has to be loaded via HBM's more limited bandwidth first.

The key insight about the GPU memory architecture is the enormous gap between what the compute units can consume and what HBM can deliver. The tensor cores can perform nearly 990 TFLOPS at FP16 (2 bytes per operand), but the HBM can only deliver 3.35 TB/s. We'll demonstrate how to compare these numbers in just a moment, when we discuss the roofline model.

CPU-GPU interaction and kernel launch overhead

There's one more piece of hardware context that matters for inference: the CPU-GPU relationship. The GPU is told what to do by the CPU. The CPU orchestrates inference by launching **kernels** — GPU functions that execute on the SMs. Each kernel launch involves the CPU sending a command to the GPU, which has a small but non-zero overhead, typically on the order of 5-10 microseconds.

For a single kernel launch, this overhead is negligible. But a single transformer layer can require multiple kernel launches (matrix multiplications, layer norm, activation functions, attention), and a large model may have 80 or more layers. During decode, where each step generates just one token, the actual computation per kernel can be very small. When you're launching hundreds of short kernels per decode step, the cumulative launch overhead starts to matter.

CUDA graphs address this problem. A CUDA graph captures a fixed sequence of kernel launches into a single object that can be replayed with a single launch command. Instead of the CPU issuing multiple individual kernel launches, the CPU replays one graph. This amortizes the launch overhead and can noticeably reduce latency for decode steps. We'll revisit CUDA graphs in more detail in Chapter 6, where we'll also discuss their limitations — primarily that the captured graph must be static, which creates complications when sequence lengths or batch sizes change dynamically.

The bottleneck framework

With the hardware picture in place, we can now talk about bottlenecks precisely. GPU-based workloads have three potential bottleneck sources:

1. **Compute**: the GPU's arithmetic units can't perform operations as fast as the rest of the system.
2. **Memory bandwidth**: the GPU can't move data from HBM to the compute units as fast as the rest of the system.
3. **Inter-device communication**: data can't move between GPUs (or between CPU and GPU) as fast as the rest of the system.

For single-GPU inference, compute and memory bandwidth are the two relevant bottlenecks. Communication becomes critical when you distribute a model across multiple GPUs, which we'll discuss in Chapter 7. For now, we'll focus on the first two bottlenecks.

The roofline model

The **roofline model**, introduced by Williams et al. (2009), gives us a simple visual framework for understanding which bottleneck applies to a given workload. The core idea is that every compute workload has a property called **arithmetic intensity**, which is the ratio of compute operations to bytes of data moved, measured in FLOPs per byte (FLOP/byte). Arithmetic intensity tells you how much work the computational unit does for each byte being read from memory.

The roofline model says that a kernel's achievable performance is capped by whichever ceiling it hits first:

- If arithmetic intensity is low (the kernel reads a lot of data but does relatively little math per byte), the kernel is **memory-bandwidth-bound**. Performance is limited by how fast data can be read from HBM, regardless of how much compute is available.
- If arithmetic intensity is high (the kernel does a lot of math per byte read), the kernel is **compute-bound**. Performance is limited by the GPU's peak FLOPS, regardless of how much bandwidth is available.

The crossover point — where the compute ceiling and the bandwidth ceiling intersect — is called the **ridge point**. For a given GPU, the ridge point is simply the ratio of peak compute throughput to peak memory bandwidth.

Table 3.2.: Key hardware parameters for the NVIDIA H100 SXM.

Parameter	Value
Peak FP16 throughput	989.4 TFLOPS
HBM3 bandwidth	3.35 TB/s
Ridge point (FP16)	~295 FLOP/byte

Table 3.2 summarizes the key hardware parameters for the H100 SXM that we need for the roofline model. For the H100 SXM at FP16, we calculate:

$$\text{Ridge point} = \frac{\text{Peak FLOPS}}{\text{Peak bandwidth}} = \frac{989.4 \times 10^{12}}{3.35 \times 10^{12}} \approx 295 \text{ FLOP/byte}$$

This means that any kernel with an arithmetic intensity below about 295 FLOP/byte is memory-bandwidth-bound on the H100 at FP16 precision. Any kernel above that threshold is compute-bound. The regions we've discussed are shown on the roofline plot for the H100 SXM in Figure 3.6. On the left, the attainable FLOPS equals the bandwidth times the arithmetic intensity, so the slope of the blue line is 3.35×10^{12}. On the right, the attainable FLOPS is capped at the peak of 989.4×10^{12}.

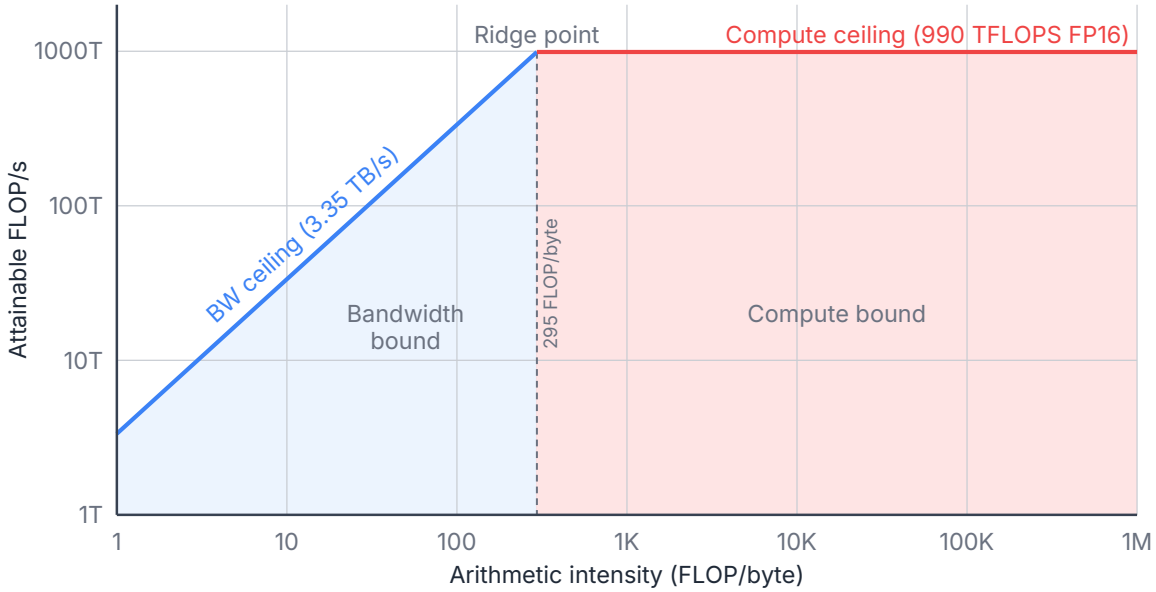


Figure 3.6.: Roofline plot for the H100 SXM at FP16

If we were calculating the roofline plot for FP8, the peak bandwidth remains the same, but the peak throughput is about double at 1,979 TFLOPS. The blue line would be extended farther up to the right before it hit our higher 1,979 TFLOPS compute ceiling, so our ridge point would be farther right, at 591 FLOP/byte. When you increase compute throughput through lower precision, the bar for being compute-bound gets higher. This means that switching to FP8 can actually make some workloads that were compute-bound at FP16 become memory-bandwidth-bound at FP8. For this reason, workloads don't always run twice as fast when switching from FP16 to FP8. We will revisit this in Section 4.1 when we discuss model quantization.

3.5. Putting Numbers on a Baseline

With the roofline model in hand, let's look at real numbers for prefill and decode. We'll use the NVIDIA H100 SXM as our reference hardware and the 70 billion parameter Llama 3 model as our reference model. The goal is to establish a **baseline** for inference without any optimizations. We will start by looking at the model parameters and the computations with them.

Prefill is compute-bound

During prefill, the model processes **S** input tokens in one forward pass. A large, fixed cost is the matrix multiplications in the attention projections and MLP layers. A widely used approximation for the total FLOPs of a forward pass is:

$$\text{FLOPs}_{\text{prefill}} \approx 2 \times P \times S$$

where P is the number of model parameters and **S** is the number of input tokens. The factor of 2 comes from the fact that each parameter participates in one multiply and one add (a multiply-accumulate operation counts as 2 FLOPs). This approximation covers the linear layers and is reasonably accurate for large models where attention's quadratic component is small relative to the total.

> **Note**
>
> This $2 \times P \times S$ approximation counts the FLOPs for the linear projections (QKV projections, output projection, and MLP layers). It omits the quadratic attention score computation ($O(S^2 \times d_K \times H \times L)$) and smaller operations like layer norms and activations. For very long sequences, the S^2 in the attention cost becomes significant and adds even more to the arithmetic intensity.

For our 70B model with $S = 4{,}096$ input tokens:

$$\text{FLOPs}_{\text{prefill}} \approx 2 \times 70 \times 10^9 \times 4{,}096 = 5.73 \times 10^{14} \text{ FLOPs} \approx 573 \text{ TFLOP}$$

Now let's compute the arithmetic intensity. The data that must be read from HBM during prefill is primarily the model weights (the input activations for the first layer come from the embedding lookup and are small by comparison). In FP16, the weights occupy:

$$\text{Weight bytes} = 70 \times 10^9 \times 2 = 140 \text{ GB}$$

The arithmetic intensity is:

$$\text{AI}_{\text{prefill}} = \frac{573 \times 10^{12}}{140 \times 10^9} \approx 4{,}093 \text{ FLOP/byte}$$

Recall from Section 3.4 that the H100 SXM ridge point at FP16 is approximately 295 FLOP/byte. Our prefill arithmetic intensity of ~4,093 FLOP/byte is more than **13x above the ridge point**. Prefill is firmly in compute-bound territory.

What does this mean for execution time? Since we're compute-bound, the time is limited by the GPU's peak compute throughput:

$$t_{\text{prefill}} \geq \frac{573 \times 10^{12}}{989.4 \times 10^{12}} \approx 0.58 \text{ seconds}$$

In practice, you won't achieve 100% of peak throughput due to overhead, memory access patterns that aren't perfectly optimized, and the attention computation. A realistic MFU of 50-70% would put the actual prefill time at roughly 0.8-1.2 seconds for this workload.

Notice what happens as the prompt gets shorter. With $S = 256$ tokens:

$$\text{AI}_{\text{prefill}} = \frac{2 \times 70 \times 10^9 \times 256}{140 \times 10^9} = 256 \text{ FLOP/byte}$$

That's just below the H100's ridge point of 295. Very short prompts could push prefill toward the memory-bandwidth-bound side, though in practice long system prompts tend to prevent requests from ever being this short.

Decode is memory-bandwidth-bound

Now consider a single decode step with batch size 1. The model runs a forward pass for just one new token. The approximate FLOPs are:

$$\text{FLOPs}_{\text{decode}} \approx 2 \times P \times 1 = 2 \times 70 \times 10^9 = 140 \times 10^9 \text{ FLOPs} = 140 \text{ GFLOP}$$

But the data movement is nearly the same as prefill — the GPU still has to read all the model weights from HBM:

$$\text{Weight bytes} = 140 \text{ GB}$$

The arithmetic intensity calculated using only the weight reads is:

$$\text{AI}_{\text{decode}} = \frac{140 \times 10^9}{140 \times 10^9} = 1 \text{ FLOP/byte}$$

One FLOP per byte. The H100's ridge point is 295 FLOP/byte. We are **295x below the ridge point**. Decode is deeply, profoundly memory-bandwidth-bound.

This is the central quantitative insight of LLM inference. During decode, the GPU reads 140 GB of weight data to perform just 140 GFLOP of computation. The tensor cores are capable

of nearly 990 TFLOPS, but they're starved for data. The bottleneck in this scenario is how fast the GPU can stream weights from HBM.

The minimum time for a single decode step is bounded by the time to read the weights:

$$t_{\text{decode}} \geq \frac{140 \text{ GB}}{3.35 \text{ TB/s}} \approx 41.8 \text{ ms}$$

At one decode step per output token, that's a theoretical maximum of about 24 tokens per second for a single request. In practice, accounting for KV cache reads and other overhead, you might see 15-20 tokens per second for a batch-1, 70B FP16 model on an H100 — comfortably in the range where text streaming feels smooth, but not exactly fast.

> **Note**
>
> This is why quantization has such a dramatic effect on decode speed. If you quantize the weights from FP16 (2 bytes) to INT4 (0.5 bytes), the weight data drops from 140 GB to 35 GB. The minimum decode step time drops to 35/3,350 ≈ 10.4 ms, roughly 4x faster. The arithmetic intensity goes from 1 to 4 FLOP/byte — still far below the INT4 ridge point, so you're still bandwidth-bound, but you've reduced by 4x the bandwidth needed. So, not only does quantization help fit models into GPU memory, it also speeds up inference. We'll cover model quantization in detail in Section 4.1.

The KV cache adds to reads and writes

The analysis above only accounts for reading the model weights. But during decode, the attention computation also reads the full KV cache — all the keys and values from all previous tokens, across all layers.

For our 70B model with 80 layers, 8 KV heads, head dimension 128, and FP16 entries, the KV cache after processing a total of **S** tokens (input + output so far) is:

$$\text{KV cache size} = 2 \times 80 \times 8 \times 128 \times S \times 2 = 327{,}680 \times S \text{ bytes}$$

After a 4,096-token prefill and 512 tokens of decode ($S = 4{,}608$):

$$\text{KV cache} \approx 1.51 \text{ GB}$$

This adds to the data that must be read from HBM at every decode step. The total bytes read per decode step becomes:

$$\text{Total bytes} = \underbrace{140 \text{ GB}}_{\text{weights}} + \underbrace{1.51 \text{ GB}}_{\text{KV cache}} \approx 141.5 \text{ GB}$$

For a 70B model, this KV cache is a relatively small addition — about 1% of the total data read. The relative increase in memory reads for a smaller model is larger. For a longer sequence, such as $S = 100{,}000$, the KV cache would be ~33 GB. At these lengths, KV cache bandwidth becomes a significant factor in decode time.

The KV cache also grows by one entry per layer per KV head per decode step. Each step must write:

$$\Delta\text{KV cache} = 2 \times L \times H_{\text{KV}} \times d_K \times b_{\text{KV}}$$

For our 70B model: $2 \times 80 \times 8 \times 128 \times 2 = 327{,}680$ bytes ≈ 320 KB per decode step. This is a small amount of data to write, but the cumulative effect is what matters: the KV cache grows monotonically, and every subsequent decode step has to read the entire thing.

Batching changes the picture

So far, we've analyzed decode with batch size 1 — a single request in isolation. This is the worst case for GPU utilization. The key insight is that **batching multiple requests together improves arithmetic intensity** by amortizing the weight reads across more tokens.

When you batch B requests together, each decode step processes B tokens instead of 1. The model weights are read once and applied to all B tokens. The FLOPs scale linearly with B:

$$\text{FLOPs}_{\text{batch}} \approx 2 \times P \times B$$

But the weight data read stays constant at $P \times b$ bytes (where b is bytes per weight). The arithmetic intensity for the weight multiplications becomes:

$$\text{AI}_{\text{batch}} = \frac{2 \times P \times B}{P \times b} = \frac{2B}{b}$$

For FP16 weights ($b = 2$), we end up with $\text{AI}_{\text{batch}} = B$ FLOP/byte.

Table 3.3.: Arithmetic intensity of decode steps as a function of batch size for FP16 weights on H100 SXM. The ridge point is ~295 FLOP/byte.

Batch size	Arithmetic intensity (FLOP/byte)	Regime on H100
1	1	Bandwidth-bound
8	8	Bandwidth-bound
32	32	Bandwidth-bound
128	128	Bandwidth-bound
295	295	Ridge point
512	512	Compute-bound

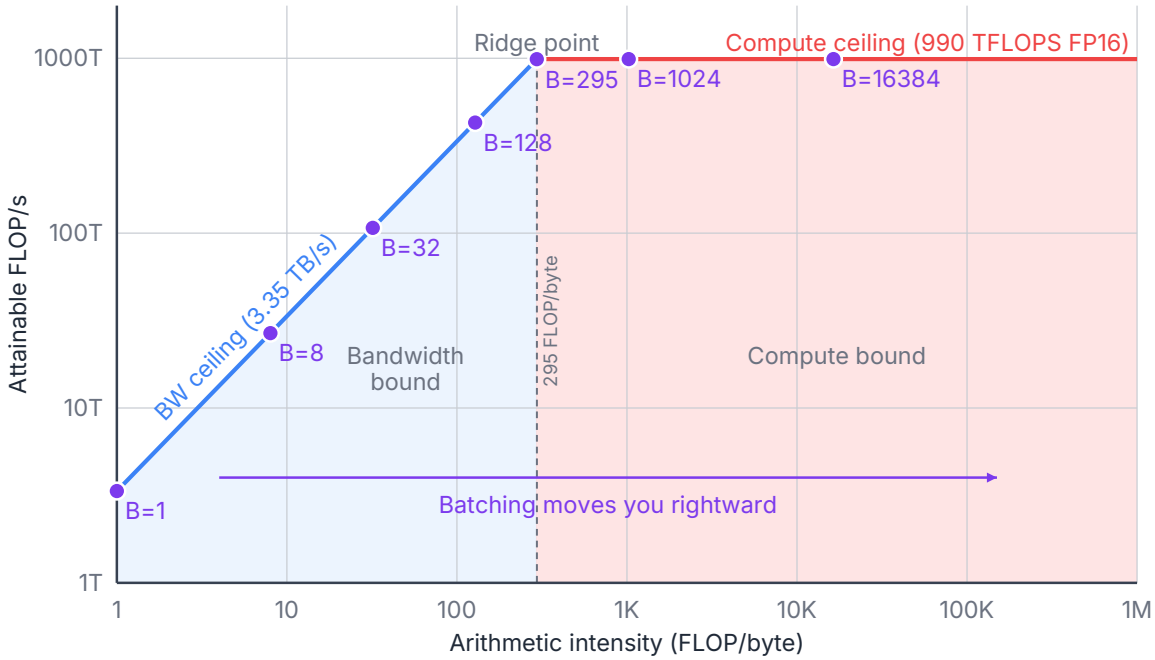


Figure 3.7.: Roofline plot showing decode operating points at different batch sizes. Each point sits on the roofline at arithmetic intensity equal to the batch size (for FP16 weights). Batching moves the operating point rightward, from deeply bandwidth-bound toward the compute ceiling.

This is the fundamental reason batching helps decode throughput. With batch size 1, you're reading 140 GB to do around 140 GFLOP of work. With batch size 128, you're reading the same 140 GB but doing 18 TFLOP of work. You're moving rightward on the roofline, getting more useful computation per byte read.

The throughput gain is substantial. At batch size 1, the decode throughput is bounded by bandwidth at about 24 tokens/s. At batch size 128, you're still bandwidth-bound but now generating 128 tokens per step, so the throughput is approximately $128 \times 24 \approx 3{,}072$ tokens/s across all requests. Each individual request still sees about 24 tokens/s (the step time is the same because we're reading the same weight data), but the system throughput has increased 128x.

> **Note**
>
> This first-order analysis ignores the KV cache reads, which do scale with batch size, since each request has its own KV cache. At large batch sizes with long sequences, the KV cache bandwidth can become a significant additional cost. It also ignores the memory capacity constraint: each active request's KV cache occupies HBM, so the maximum batch size is limited by how many KV caches fit in memory alongside the model weights.

To push the arithmetic intensity above the ridge point and become compute-bound, you'd need a batch size of about 295 for FP16 weights on the H100. In practice, memory capacity limits usually prevent batch sizes this large for big models. A 70B model with 140 GB of weights leaves only ~40 GB for KV caches on an 80 GB GPU. If each request's KV cache is 1.5 GB, you can fit about 26 concurrent requests, far short of the 295 needed to hit the ridge point. This is why decode for large models remains memory-bandwidth-bound even with batching, and why the optimizations for reducing KV cache memory usage in Section 6.3 and elsewhere are so important — they enable larger batch sizes, which improves throughput.

Back-of-envelope summary

Let's collect the key numbers for our reference configuration: a 70B parameter model in FP16 on an H100 SXM (80 GB HBM, 3.35 TB/s bandwidth, 989.4 TFLOPS FP16).

Table 3.4.: Back-of-envelope performance for a 70B FP16 model on H100 SXM. Prefill throughput is computed as S/t_{prefill}. Decode throughput is B/t_{decode}. These are theoretical lower bounds on step time; real-world performance will be somewhat lower.

Quantity	Prefill (S=4096)	Decode (B=1)	Decode (B=32)
FLOPs per step	573 TFLOP	140 GFLOP	4.48 TFLOP
Data read (weights)	140 GB	140 GB	140 GB
Arithmetic intensity	~4,093 FLOP/byte	~1 FLOP/byte	~32 FLOP/byte
Bottleneck	Compute	Bandwidth	Bandwidth
Min step time	~0.58 s (compute)	~41.8 ms (bandwidth)	~41.8 ms (bandwidth)

Quantity	Prefill (S=4096)	Decode (B=1)	Decode (B=32)
Tokens per step	4,096 (all input)	1	32
Throughput	~7,000 tok/s	~24 tok/s	~766 tok/s

A few things stand out from this table:

1. **Prefill processes tokens roughly 300x faster than batch-1 decode**, in terms of tokens per second. This is the direct consequence of the arithmetic intensity gap.
2. **Batching is the single most important lever for decode throughput.** Going from batch size 1 to 32 increases system throughput by 32x with essentially no increase in step time, because we're bandwidth-bound either way. The same weight data is read, just applied to more tokens. Unfortunately, total memory capacity limits how large we can push batch size.
3. **The step time for decode barely changes with batch size** (as long as we remain bandwidth-bound). This means that batching increases system throughput without significantly increasing per-request latency. This is a rare case where we can improve one metric, throughput, without a tradeoff in another metric, latency.
4. **Weight memory dominates the data read for large models.** The KV cache is a relatively small fraction of the total data read per step, though it becomes more significant for smaller models and very long sequences.

These numbers are the baseline that every optimization in subsequent chapters maps onto. When we discuss model quantization in Section 4.1, we'll see how reducing bytes per weight directly reduces the bandwidth denominator. When we discuss batching strategies in Section 5.1, we'll see how continuous batching and chunked prefill allow the system to maintain high batch sizes. When we discuss KV cache optimizations in Section 6.3, we'll see how reducing KV cache memory enables larger batches. And when we discuss speculative decoding in Section 6.5, we'll see how generating multiple tokens per decode step changes the numerator.

3.6. Map of the optimizations

The next four chapters in this book each focus on a specific class of bottleneck and the techniques that address it:

Chapter	Bottleneck class	Core lever
Chapter 4 – Model Design	Model size and compute	Reduce cost per token via decreasing model size
Chapter 5 – Scheduling and Batching	Scheduling and queuing	Reduce wasted GPU resources through smarter batching and memory management

Chapter	Bottleneck class	Core lever
Chapter 6 – Request-Level Optimizations	Request compute and memory; sequential decoding	Optimize individual requests; break the sequential decoding constraint
Chapter 7 – Multi-Device Inference	Hardware capacity	Distribute across multiple devices when one GPU isn't enough

This bottleneck framework gives you a way to evaluate each optimization you encounter: what bottleneck does it address, and does that bottleneck actually apply to your workload? An optimization that reduces compute won't help a memory-bandwidth-bound decode step much. A technique that reduces memory traffic won't help a compute-bound prefill. Keeping the roofline in mind as you read the rest of this book will help you understand not just *what* each technique does, but *why* it works and *when* it helps.

The book concludes in Chapter 8 with a discussion of production LLM serving systems.

3.7. Further Reading

The Anyscale blog post on reproducible performance metrics (Kadous et al. 2023) is a practical guide to LLM inference metrics, including many of the ones introduced in this chapter, such as TTFT, TPOT, throughput, and goodput.

The NVIDIA Hopper architecture blog post (NVIDIA 2022) provides detailed block diagrams of the H100's SM and full-chip layout, including the memory hierarchy, tensor core organization, and the Tensor Memory Accelerator. Gu and Dao (2024) from Hazy Research gives a practitioner-oriented walkthrough of GPU memory levels with concrete bandwidth numbers at each tier, which is useful context for understanding why kernel design choices matter so much for inference performance. For a detailed empirical look at the H100's L2 cache behavior and measured bandwidth at different access patterns, Chips and Cheese (2023) is an informative deep dive.

For GPU hardware specifics, the NVIDIA H100 datasheet (NVIDIA 2023) and the CUDA Programming Guide (NVIDIA 2024a) are the authoritative references. The datasheet gives the peak throughput and bandwidth numbers used throughout this book. The programming guide covers the memory hierarchy, kernel launch mechanics, and CUDA graphs in much more detail than we do here.

The back-of-envelope analysis in this chapter follows the approach laid out by Pope et al. (2022), who provide a thorough accounting of FLOPs, memory bandwidth, and arithmetic intensity for each layer of a transformer during inference. Their paper covers both prefill and decode on TPU hardware, but the methodology translates directly to GPUs. They also cover the effects of multi-query attention, different parallelism strategies, and how the roofline picture changes as you scale model size.

For an accessible walkthrough of counting parameters and FLOPs in transformer models, Kipply's "Transformer Inference Arithmetic" (Chen 2022) is an excellent companion to this chapter. It works through the same kinds of calculations we've done here — parameter counts, memory footprints, FLOPs per layer — with clear explanations of where each number comes from.

For a broader survey of LLM inference techniques and how they fit together, Yuan et al. (2024) covers the full landscape from model compression through serving system design, with the distinguishing feature of using roofline analysis to explain why each technique helps. Their framework makes clear why LLM decoding is memory-bound and what each optimization actually changes about the bottleneck. It's a useful map of the territory before diving into the detailed treatments in the chapters that follow.

The roofline model was introduced by Williams et al. (2009) and remains one of the clearest frameworks for reasoning about hardware bottlenecks. Their original paper uses CPU examples, but the model applies directly to GPUs — the key insight is the same: compare arithmetic intensity against the ridge point to determine whether you're compute-bound or bandwidth-bound. For an accessible, visual walkthrough of the roofline model in a GPU context, the Modal GPU Glossary (Modal 2024) covers roofline analysis, arithmetic intensity, and memory bandwidth with clear explanations of how each concept maps to modern GPU hardware.

A comprehensive, in-depth treatment of hardware performance is in "How to Scale Your Model" (Austin et al. 2025). This book primarily focuses on TPUs, with one specific chapter on GPUs. All of the concepts apply to whatever hardware you have.

4. Model Design Choices

Now that we have an understanding of the inference pipeline and where the bottlenecks live, starting with this chapter, we shift to doing something about them.

The optimizations in this chapter are about the model itself, separate from request handling. They change what the hardware has to do per token by reducing the model's compute and memory footprint at the architecture and weight level. Many of these are choices made by model designers prior to training. A few can be applied post-training to an existing model. Either way, the result is the same: fewer bytes to move, fewer FLOPs to compute, or both.

We'll keep the coverage here lighter than in subsequent chapters. Much of the time, you don't get a choice about the model to be used. Because of this, much of the focus of this book is on what you can do when you're given a model to serve. But understanding these techniques is still helpful, whether you have any say in the model design or not.

4.1. Model Quantization

One can make a strong argument that **quantization** is the single most impactful model-level optimization for LLM inference. The reason is straightforward, and it connects directly to the bottleneck framework from Section 3.4.

During decode, the biggest overall bottleneck is memory bandwidth. As we've discussed, each decode step reads the entire model's weights from HBM just to produce one token. If your model has 70 billion parameters stored in FP16 (2 bytes each), that's 140 GB of weight data read per token. On an H100 with 3.35 TB/s of bandwidth, reading 140 GB takes about 42 milliseconds. That's a hard floor on your per-token latency, ignoring everything else. If you quantize those same weights to INT4 (0.5 bytes each), you're reading 35 GB instead, and the floor drops to about 10 milliseconds. That's a 4x improvement in your theoretical best-case decode speed, and we didn't need to change a single line of serving code.

A further insight is that quantization doesn't just reduce the amount of GPU memory needed to store the model weights (which is a pretty big benefit). Quantization also reduces the bandwidth demand that dominates decode latency, and it can speed up computation too. Let's walk through the main approaches to quantization. We will assume 16-bit weights in FP16 or BF16 are our baseline.

> **Note**
>
> Floating point arithmetic used to be performed at 32 bits (FP32) for both training and inference, but the last few years have seen a shift to 16-bit floating point (FP16 or BF16). The dynamic range of FP16 and BF16 is still quite high, which made the transition from FP32 relatively easy, even for training, where automatic mixed precision (AMP) techniques can be used to keep the model stable. More recently, the focus has shifted to quantization below 16 bits, which requires more careful techniques to avoid quality degradation.

Lower precision datatypes

There are a number of datatypes supported, especially in the latest generation of GPUs. While higher-precision datatypes, such as FP32, are needed for training, most models can run inference using FP16 without any noticeable difference. Datatypes that require fewer than 16 bits to store can save even more memory. Table 4.1 lists commonly supported datatypes.

Table 4.1.: Common datatype precisions supported. The Layout column shows the number of bits allocated for the sign ("s"), exponent ("e"), and mantissa ("m").

Type	Bits	Layout	Key Properties
FP32	32	1s + 8e + 23m	Standard training precision; baseline
FP16	16	1s + 5e + 10m	2x compression; narrower dynamic range (max ~65,504)
BF16	16	1s + 8e + 7m	Same dynamic range as FP32, less mantissa precision; preferred for training stability
FP8 E4M3	8	1s + 4e + 3m	More precision; used for weights/activations in forward pass
FP8 E5M2	8	1s + 5e + 2m	More dynamic range; used for gradients
INT8	8	signed integer	Range [-128, 127]; 4x compression vs FP32
INT4	4	signed integer	Range [-8, 7]; 8x compression; requires sophisticated algorithms

Type	Bits	Layout	Key Properties
NF4	4	non-uniform	16 levels at normal distribution quantiles; information-theoretically optimal for normally distributed weights (QLoRA)
FP4	4	1s + 2e + 1m	Less common than NF4 for LLM weights

Older GPUs support 32-bit and 16-bit floating point, as well as 8-bit integer matrix multiplication. FP8 is a newer format supported on H100 and later GPUs. With the popularity of quantization, newer generations of GPUs are supporting a wider range of smaller datatypes.

Weight-only quantization

The simplest form of quantization leaves the activations in their original precision (typically FP16 or BF16) and only quantizes the weights. This is called **weight-only quantization**, and it's the most common starting point. In weight-only quantization, the weights are stored in a smaller datatype, then dequantized back to FP16 on the fly before each matrix multiplication, so the compute still happens at FP16 precision. The benefit is purely about reducing how much data needs to be read from HBM and reducing TPOT.

When quantizing below 16-bit precision, the storage space and memory read savings are usually slightly reduced because we also need to store conversion parameters. If we have both an offset and a scaling parameter, then when x is the higher precision value and $\tilde{x}$ is the quantized value, the conversion is $x = (\tilde{x} - \text{offset}) \times \text{scaling}$. This dequantization step also adds a small amount of compute overhead to model inference.

LLM inference tolerates quantization well. While the lower precision datatypes could cause cascading rounding errors, in practice the errors tend to cancel out. Quantizing from FP16 to INT8, known as **W8**, cuts the weight memory in half compared to FP16. The quality impact is usually negligible for large models — a 70B model at W8 will behave almost identically to its FP16 counterpart on most benchmarks. If we drop to **W4** quantization, cutting weight memory by 4x compared to FP16, this is where quality tradeoffs start to become more visible, especially for smaller models. A 7B model at W4 may show degradation on some tasks, while a 70B model at W4 often remains remarkably capable. The general pattern is that larger models are more robust to aggressive quantization.

Weight-activation quantization

Weight-only quantization reduces memory bandwidth demand but doesn't speed up the actual matrix multiplication, since the compute still runs at FP16. **Weight-activation**

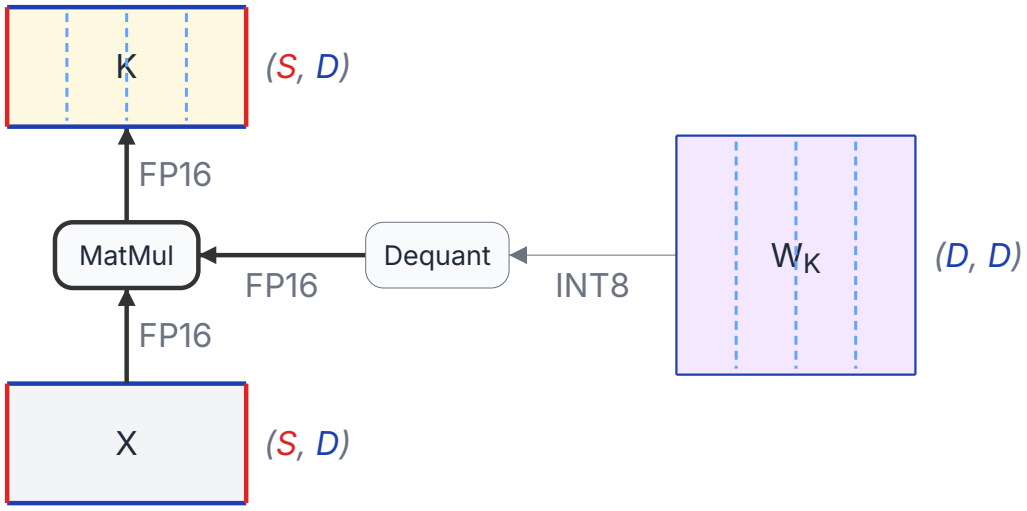


Figure 4.1.: Sample weight-only quantization of a FP16 model into INT8 weights, which still performs FP16 calculations. A dequantization step is needed before each weight is used, but impacts to accuracy are minimized by maintining the normal precision of calculations.

quantization takes the next step: quantize both weights *and* activations so that the matrix multiplications themselves can run on lower-precision hardware units. Faster, lower-precision compute cores increase compute throughput. In this way, TTFT can be reduced for prefill, especially for long prompts.

For example, weight-activation quantization using INT8, known as **W8A8**, quantizes both weights and activations to INT8, allowing the use of INT8 tensor cores. On the H100, INT8 tensor cores deliver roughly 2x the throughput of FP16 tensor cores. W8A8 reduces memory traffic for bandwidth-bound operations, and it also increases the compute ceiling for compute-bound operations. As with weight-only quantization, weight-activation quantization introduces some additional storage and compute overhead associated with the conversion parameters.

Granularity

Early attempts to quantize below 16-bit floating point took a simple approach that used the same datatype and conversion factor for all values in each weight matrix. In this scenario, one very large value distorts the conversion scale for the entire weight matrix. This increases the rounding errors caused by quantization and can also cause small weights to get rounded to zero, which can cause even more problems. To mitigate the risk of outlier values, quantization can be performed at more fine-grained granularities:

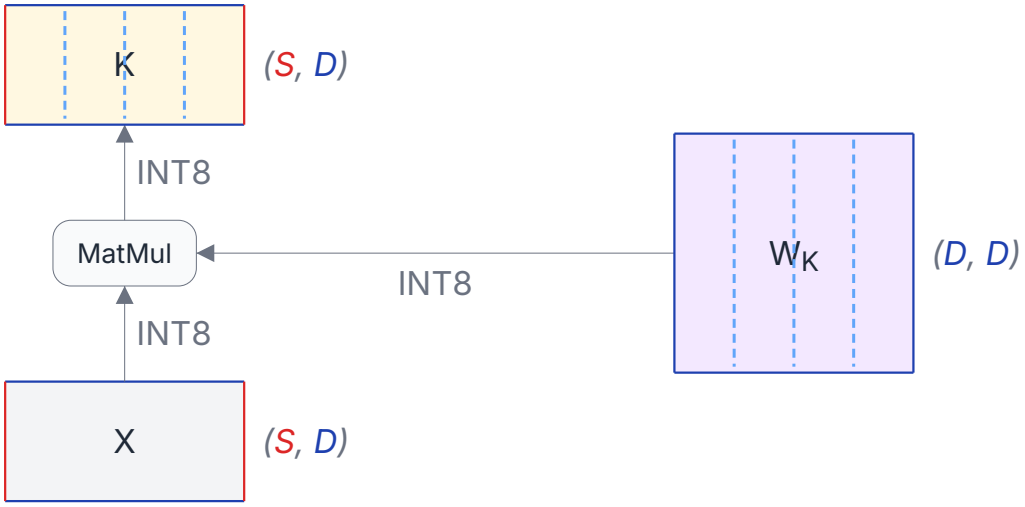


Figure 4.2.: Sample weight-activation quantization of a FP16 model into INT8 weights and calculations. No dequantization is needed, but there are greater risks of accuracy degradation, overflow, and underflow. Note that slightly more work is performed with each operand to manage the scaling parameters (not shown).

- Per channel(s)
- Per block, potentially in a hierarchy of super-blocks and sub-blocks
- Per token

Fine-grained quantization seeks to group similar values together, and is highly effective. It does, however, come at a cost of additional memory and compute overhead. For example, dividing a 2,048 by 2,048 weight matrix into groups of 32 values will require 131,072 scaling parameters and dequantization calculations. To reduce the storage of the scaling factors, they can themselves be quantized at a cost of additional dequantization overhead.

When quantization is applied

Post-Training Quantization (PTQ) is performed after the model is fully trained. This is the dominant approach since it can be applied to any model from any source, without requiring any special training. Simple PTQ only examines the weights, and more advanced PTQ techniques require processing a small set of calibration data through the model so activation information can also be analyzed.

Quantization-Aware Training (QAT) modifies the model training procedure to minimize the rounding errors caused by quantization. While effective, the extra work required has limited adoption rates.

Techniques

One of the practical challenges of quantization, especially weight-activation quantization, is **activation outliers**. Large LLMs tend to develop a small number of activation channels with values that are orders of magnitude larger than the rest. If you try to quantize activations using a single scale factor per tensor, these outliers force a wide quantization range that wastes precision on the many small values. Addressing activation outliers is an area of active research, and we will mention a few post-training quantization methods and how they try to handle these activation outliers.

GPTQ (Frantar et al. 2022) uses approximate second-order information to quantize weights to 4 bits (or even 3 bits) with minimal accuracy loss. It works by quantizing weights one layer at a time and adjusting the remaining weights to compensate for the error introduced by each quantization step. GPTQ can be applied to a pre-trained model without a large training set. It only needs a small calibration set.

AWQ (Activation-aware Weight Quantization) (Lin et al. 2023) takes a different approach. It observes that a small fraction of weights are disproportionately important — specifically, the weights that correspond to large activation magnitudes. Rather than treating all weights equally, AWQ protects these salient weights by scaling them up before quantization and scaling the activations down correspondingly. This keeps the important weights at higher effective precision while allowing aggressive quantization elsewhere.

SmoothQuant (Xiao et al. 2023) addresses activation outliers by migrating the quantization difficulty from activations to weights. The idea is to apply a per-channel scaling transformation that divides the activation by a smoothing factor and multiplies the corresponding weight by the same factor. This makes the activation distribution more uniform — and therefore easier to quantize — at the cost of making the weight distribution slightly less uniform. Since weights are fixed and can be quantized offline with more care, this is a favorable tradeoff.

Connecting quantization to MBU and MFU

In Section 3.2, we introduced Model Bandwidth Utilization (MBU) and Model FLOPs Utilization (MFU) as key metrics for decode efficiency. Quantization improves the theoretical decode speed by reducing the bytes that need to be read per token. But it doesn't automatically improve decode speed linearly. If your serving system has overhead that prevents it from saturating HBM bandwidth, that overhead will prevent the total time from dropping proportionally to the reduction in memory traffic. For example, if transferring half the number of bytes requires 70% of the time, achieved bandwidth has decreased, so MBU will decrease. Similarly, if moving to a lower precision doubles the peak FLOPS of the GPU, but your overhead limits you to a 70% increase in FLOPS, your MFU will drop. So, quantization could improve TTFT and TPOT but potentially hurt MBU and MFU. It's important to remember that latency and efficiency metrics are measuring different things.

In practice, quantized models can achieve higher MBU because the smaller data footprint means fewer cache misses, better memory access patterns, and the possibility of fitting the entire model in a single GPU's memory rather than splitting across devices. These secondary

effects can make the real-world speedup even better than the raw byte reduction and fixed overhead would suggest.

> **Note**
>
> Quantization affects the roofline model in an important way. When you use FP8 or INT8 tensor cores, you raise the compute ceiling (from ~990 TFLOPS to ~1,979 TFLOPS on the H100) while the bandwidth ceiling stays constant. This shifts the ridge point higher, which means some workloads that were compute-bound at FP16 can become memory-bandwidth-bound at FP8. Keep this in mind when choosing quantization strategies for prefill-heavy workloads.

4.2. Knowledge Distillation

Knowledge distillation is a training technique where a smaller **student** model is trained to mimic the behavior of a larger **teacher** model. In the original distillation formulation (Hinton et al. 2015), the student learns not just from the ground-truth labels, but from the teacher's full output distribution — the probability the teacher assigns to every token in the vocabulary. These soft targets carry richer information than hard labels, which is why a distilled student can often outperform a model of the same size trained from scratch. Newer techniques vary the approach by training on sample outputs from the larger model or using reinforcement learning to impart some of the larger model's capability into the smaller model.

From an inference perspective, the result is simple: you get a smaller model. Fewer parameters means less data to read from HBM per decode step, a smaller KV cache (assuming the model also has fewer layers or a smaller hidden dimension), and less memory consumed overall. The inference system doesn't need to know or care that the model was distilled — it just sees a model with fewer parameters that performs better than you'd expect for its size. Running a smaller model usually improves all metrics, with decreases in TTFT and TPOT, and increases in concurrency.

The tradeoff is that distillation requires significant training compute, and you need the larger, more capable teacher model. This may not be an option for a frontier model, but distillation is popular in research and in use for smaller LLMs.

4.3. Pruning

Pruning removes weights or structures from a model that contribute little to its output. The intent is similar to quantization: fewer parameters means less data to move, which speeds up bandwidth-bound decode, resulting in a lower TPOT. But while quantization keeps all the parameters and reduces their precision, pruning eliminates parameters entirely. There are side effects to pruning, however, that make it challenging.

Unstructured pruning

Unstructured pruning zeroes out individual weights based on some importance criterion (typically magnitude). A pruned model might have 50% of its weights set to zero, theoretically cutting compute and memory in half. The problem is that these zeros are scattered throughout the weight matrices in an irregular pattern. These irregular patterns reduce the ability to load data efficiently and parallelize computations, which can decrease MFU on the GPU dramatically. If enough of the weights can be pruned, good speedups can be achieved for CPU inference, which doesn't parallelize compute as much as a GPU. Standard GPU hardware and matrix multiplication kernels still process the full dense matrices, so they can't exploit this sparsity efficiently. And sparse matrix multiplications are much slower than their dense counterparts, so they need extremely high levels of sparsity before they are faster than full dense matrix multiplication. This is why unstructured pruning rarely sees speedups, despite the reduction in parameters and FLOPs. To get real speedups from pruning, you need specialized sparse compute support, such as NVIDIA's 2:4 structured sparsity format on Ampere and later GPUs. It is more difficult to prune weights into this structured sparsity pattern, so structured sparsity isn't common either.

Structured pruning

Structured pruning takes a coarser approach: instead of zeroing individual weights, it removes entire structural units, such as attention heads, entire layers, or neurons in MLP blocks. The result is a smaller dense model that runs on standard hardware with no special sparse computation support required.

Removing an attention head reduces the KV cache proportionally. Removing an entire layer reduces both parameters and the depth of the forward pass. These are straightforward wins for inference, and the cost model from Section 3.5 applies directly — fewer parameters means proportionally less time reading weights during decode.

The challenge with structured pruning is identifying which structures can be removed with minimal quality impact. Attention heads in the later layers of a model often contribute less than early or middle layers, but this varies by model and task. Structured pruning typically requires some retraining or fine-tuning after removal to recover quality, which limits its use as a purely post-training optimization.

In practice, pruning is less widely adopted for LLM inference than quantization because of the performance and text quality challenges. Quantization offers a more predictable quality-performance tradeoff and is easier to apply post-training.

4.4. Efficient Attention Architectures

The sections above are optimizations you can apply to an existing model. This section covers architecture choices made during model design that affect inference efficiency from the start.

FFN layers account for roughly two-thirds of a typical transformer's parameters, but they operate independently on each token. Their cost scales only linearly with sequence length, and they are straightforward to parallelize. In addition, GPUs are highly optimized for dense matrix multiplications, so FFNs can be executed very efficiently. The MoE architecture discussed in Section 2.6 is one architectural approach that reduces the FFN inference cost by activating only a subset of FFN experts per token.

Attention is the more interesting bottleneck. Its KV cache grows with sequence length, creating a per-request memory cost that directly limits concurrency. For long sequences, the quadratic cost of full attention also becomes significant during prefill. This makes attention the primary target for architectural optimization.

Reducing KV cache size

The KV cache is one of the largest consumers of GPU memory during inference, and unlike model weights, it scales with the number of active requests and the sequence length of each. Reducing the size of the KV cache per request lets you serve more requests in the same memory budget, increasing concurrency and throughput. We'll cover KV cache management techniques in Section 6.3; here we focus on the architectural choices that determine how large the cache is in the first place.

Multi-Head Attention (MHA) is the original design from the transformer paper (Vaswani et al. 2017). Each attention head has its own key and value projections, so the KV cache stores separate K and V vectors for every head, every layer, and every token in the sequence. For a model using FP16 with 64 heads and a head dimension of 128, each token adds $2 \times 64 \times 128 \times 2 = 32{,}768$ bytes to the KV cache per layer. This is the most expensive option. The top row of Figure 4.3 shows how queries align with keys and values in MHA.

Multi-Query Attention (MQA) (Shazeer 2019) goes to the opposite extreme: all attention heads share a *single* set of key and value projections, as shown on the third row of Figure 4.3. Each head still has its own query projection, so the heads can attend to different aspects of the input, but the KV cache only stores one K and one V vector per layer per token. This reduces the KV cache by a factor equal to the number of heads — from 64 sets of K/V down to 1 in the example above. The negative impact on the quality of text generation is the main reason why MQA is not widely adopted.

Grouped-Query Attention (GQA) (Ainslie et al. 2023) is a middle ground between MHA and MQA. Instead of one shared KV set for all heads, GQA uses $\mathbf{G}$ groups, where each group of heads shares one set of keys and values. With 32 heads and $\mathbf{G} = 8$ groups, the KV cache is reduced by 4x compared to MHA. GQA has become the dominant choice in recent models — LLaMA 3, Mistral, and Gemma all use it. It offers much of MQA's memory savings with less text quality risk. A GQA example with 8 query heads and 4 groups is shown on the second row of Figure 4.3.

Multi-head Latent Attention (MLA) (A. Liu et al. 2024a) is DeepSeek's approach to KV cache compression. Rather than storing full key and value vectors, MLA compresses them into a low-rank **latent vector** at each layer. At attention time, the full keys and values are

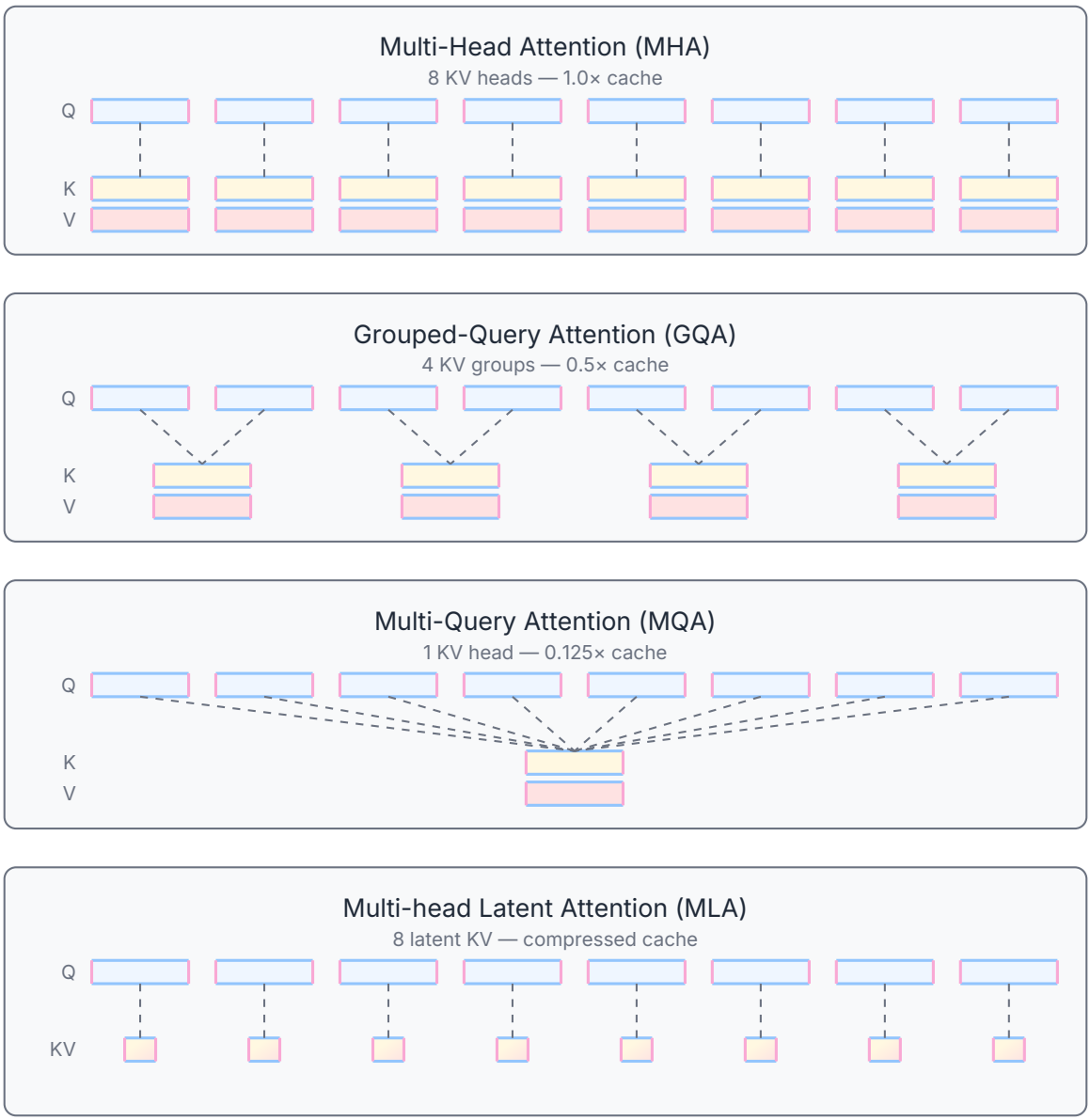


Figure 4.3.: Techniques for reducing the KV cache size of a given layer. GQA and MQA share keys and values with multiple query heads. MLA compresses key and value information together to reduce size.

reconstructed from the latent representation. The compression ratio depends on the latent dimension, but MLA can achieve significant cache reductions — comparable to or better than GQA — while preserving model quality. The tradeoff is additional compute to decompress the latent vectors during attention, but this compute is typically small relative to the bandwidth savings, and there are tricks to reduce the compute. MLA is depicted on the bottom row of Figure 4.3.

Cross-Layer Attention (CLA) (Brandon et al. 2024) takes a different angle: rather than sharing KV across heads within a layer, it shares KV across adjacent layers. If every pair of consecutive layers shares the same KV cache, the total cache size is halved. Figure 4.4 compares standard attention with CLA, where the K and V vectors are shared between pairs of layers. CLA can be combined with GQA or MQA for even greater reductions.

Reducing attention compute

Beyond the KV cache, the attention computation itself can be a bottleneck, particularly during prefill for long sequences, since the cost is quadratic in sequence length.

Sliding window attention (Beltagy et al. 2020) restricts each token to attend only to a fixed-size window of nearby tokens rather than the full sequence. This reduces the attention computation from $O(S^2)$ to $O(S \times w)$, where S is the sequence length and w is the window size. Figure 4.5 illustrates the difference between full attention and sliding window attention with a window size of 4. For prefill, the smaller attention computation improves TTFT. For decode, sliding window attention also bounds the KV cache size per layer to the window size, regardless of the total sequence length. The reduction in the amount of KV that needs to be transferred can improve TPOT, but the main benefit is that the smaller KV cache increases concurrency and throughput. Several recent models — including Mistral — use sliding window attention in some or all layers, often combined with a few full-attention layers to preserve the model's ability to attend to distant tokens.

> **Note**
>
> This book focuses on the dominant transformer architecture with softmax attention. Alternative approaches like Mamba and other state-space models, as well as linear attention variants, replace the attention mechanism entirely with recurrent-style computations that have constant-size state rather than a growing KV cache. These are an active area of research, but the vast majority of deployed LLMs today still use softmax attention. Because sliding window attention and attention alternatives hurt generation quality, competitive models today retain the use of full softmax attention in some of their layers. For now, these KV cache optimizations are still needed.

Parallelizing computation

Parallel attention and MLP blocks is a simple architectural trick used in some models, including PaLM (Chowdhery et al. 2022). Instead of computing attention and MLP

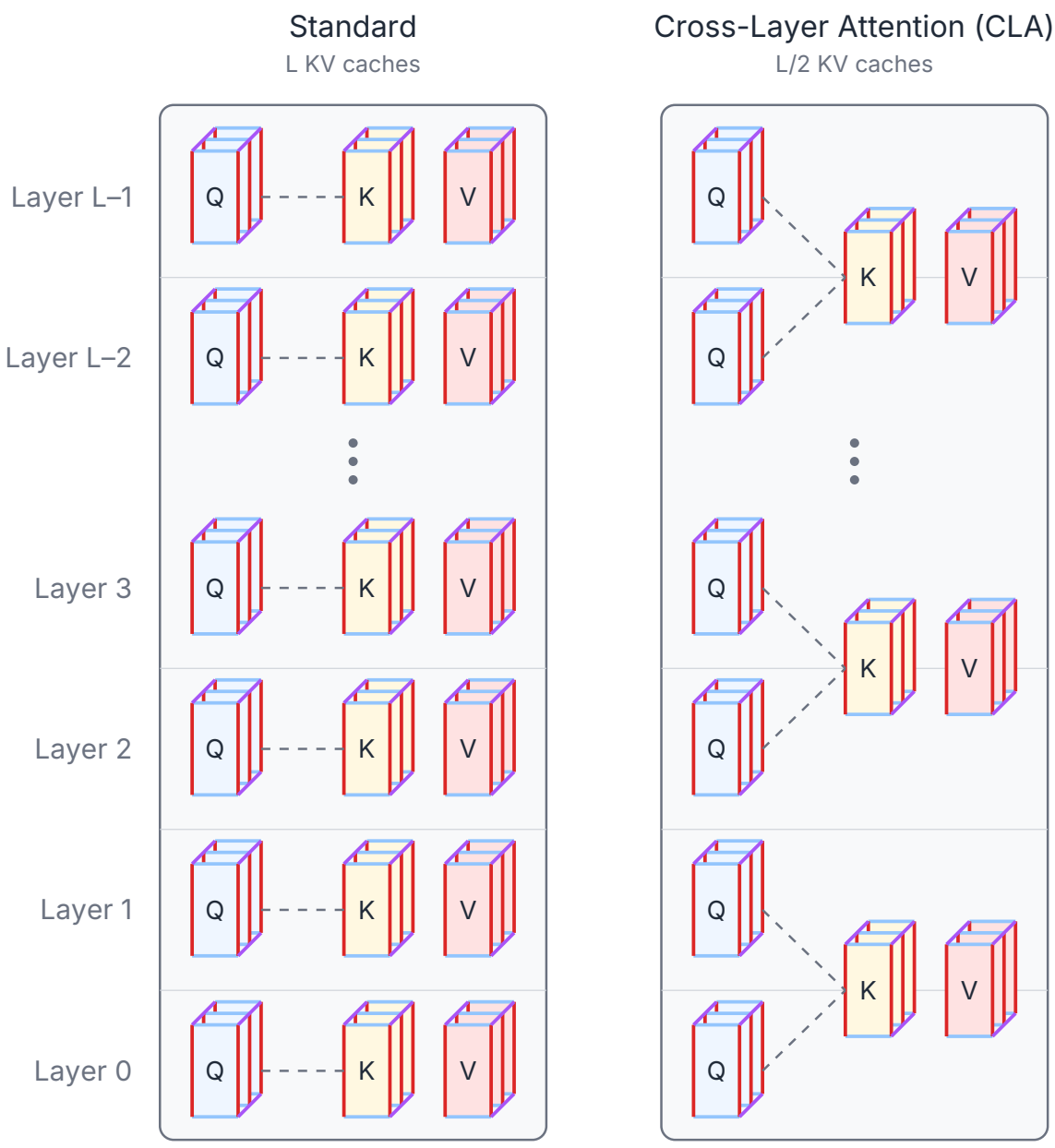


Figure 4.4.: Cross-layer attention with a factor of 2 KV sharing between layers

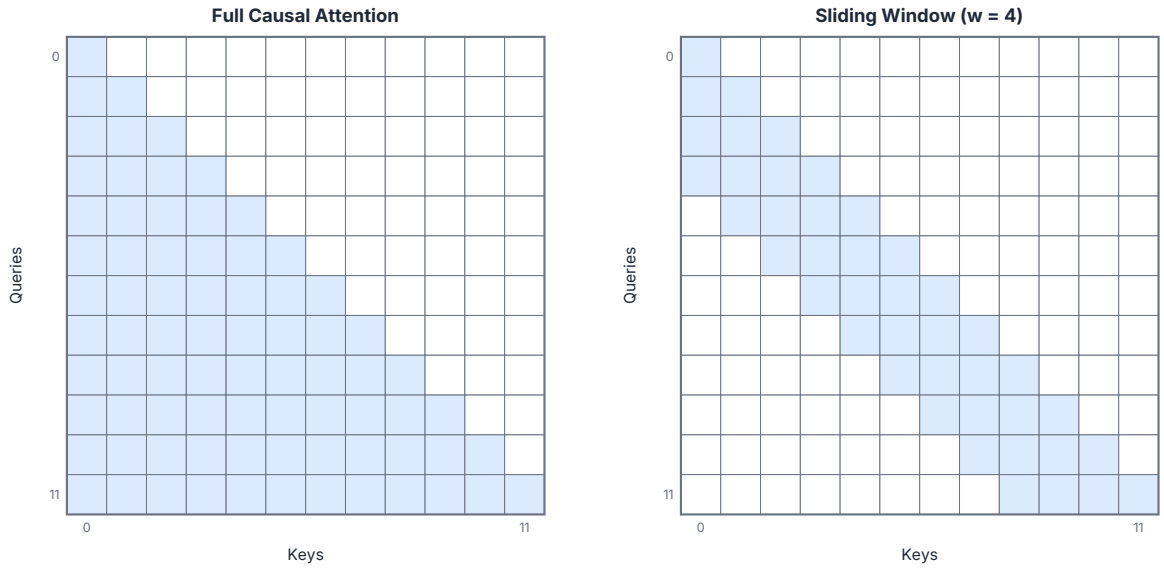


Figure 4.5.: Full causal attention (left) vs. sliding window attention with window size w=4 (right). Each blue shaded cell indicates a query-key pair included in the attention computation.

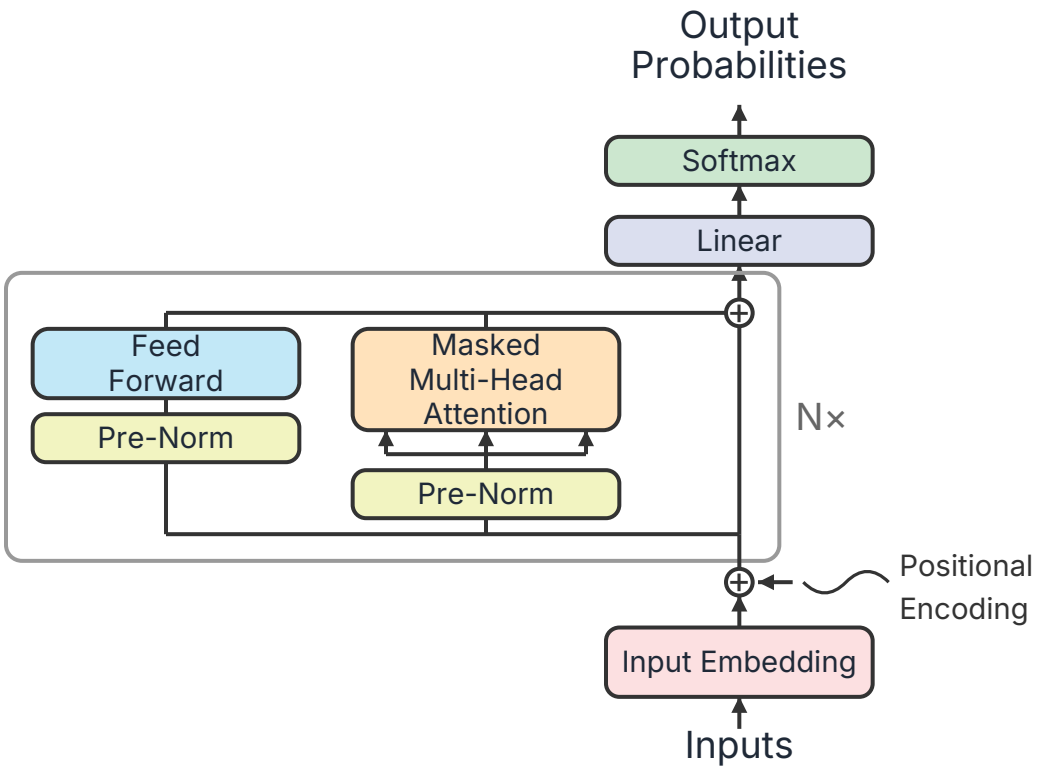


Figure 4.6.: Parallel attention and MLP blocks

sequentially within each transformer block, they are computed in parallel and their outputs are summed. This doesn't reduce total FLOPs, but it reduces the serial dependency within each layer, which can improve MFU and TPOT — especially when the attention and MLP computations can be overlapped. A schematic of this parallel design is shown in Figure 4.6.

4.5. Tokenization and Vocabulary Effects

One model-level lever that's easy to overlook is the **tokenizer** and its vocabulary size. The tokenizer determines how many tokens a given piece of text is broken into, and that token count directly drives the number of decode steps, each of which requires a full forward pass through the model.

A larger vocabulary means each token covers more text on average. With a larger vocabulary, more words and subwords that would otherwise be split into multiple tokens get their own single token. Fewer tokens means fewer decode steps, which means faster end-to-end generation. For latency-sensitive applications, this is a meaningful win. Going from a 32K vocabulary to a 128K vocabulary might reduce the average token count for a given text by 10-20%, which translates directly to 10-20% fewer decode steps.

The tradeoff is that a larger vocabulary increases the size of two model components: the **embedding table** (which maps token IDs to vectors at the input) and the **LM head** (the output projection that produces logits over the vocabulary). For a model with a hidden dimension of 4,096 and a vocabulary of 128K tokens, the LM head alone has $4{,}096 \times 128{,}000 \approx 500$ million parameters. At FP16, that's about 1 GB, which is not negligible, but typically a small fraction of the total model size for large LLMs. The compute cost of the final softmax over a larger vocabulary also increases, though this is usually a minor contributor to overall latency.

For most practical scenarios, the efficiency gains from fewer decode steps outweigh the increased memory footprint and minor TPOT increase from the larger embedding and LM head. This is one reason why the trend in recent LLMs has been toward larger vocabularies.[1]

4.6. Summary

The techniques in this chapter all reduce how much work the hardware has to do for each token. Model quantization reduces the bytes-per-parameter, reducing TPOT, and weight-activation quantization can also improve TTFT. Distillation and pruning produce smaller models with fewer total parameters, primarily affecting TPOT. Efficient attention architectures reduce the KV cache footprint and, in some cases, the attention compute itself. This primarily improves concurrency and throughput, but it can also reduce TTFT for prefill. Even vocabulary design affects inference cost by changing how many decode steps are needed.

[1] Larger vocabularies also improve multilingual performance for languages such as Chinese, which require very different tokens from English.

These are the model-level knobs. Once they're set — once you have a quantized, architecturally efficient model ready to serve — the next set of optimizations operates at the system level: how to schedule requests, batch them together, and manage memory. That's where we turn next in Chapter 5.

4.7. Further Reading

For intuitive explanations of quantization:

- Maarten Grootendorst's "A Visual Guide to Quantization" (Grootendorst 2024) walks through symmetric and asymmetric quantization, per-tensor vs. per-channel granularity, and formats like GGUF with over 50 custom diagrams. It's one of the best visual introductions to the topic.
- Tim Dettmers' blog post "LLM.int8() and Emergent Features" (Dettmers 2022) explains *why* naive INT8 quantization breaks down at scale: large transformers develop outlier activation channels that are orders of magnitude larger than the rest. The post provides the intuition behind the mixed-precision decomposition that addresses this, which is useful background for understanding SmoothQuant and similar techniques. Dettmers is also the author of the `bitsandbytes` library (Dettmers et al. 2021), the most widely used quantization library in the Hugging Face ecosystem.
- The Hugging Face Quantization Concept Guide (Hugging Face 2024b) is a practical reference for choosing between quantization backends (GPTQ, AWQ, bitsandbytes, and others) in the Hugging Face ecosystem, with guidance on when each approach is appropriate.

For knowledge distillation applied to LLMs, Snorkel AI's "LLM Distillation Demystified" (Casey 2024) is a practitioner-oriented guide covering distillation approaches and when to use them. For a more comprehensive treatment, Xu et al. (2024) surveys the full taxonomy from white-box distillation (where student models train on the teacher's logits) to black-box distillation (where they learn from the teacher's generated outputs), along with domain-specific and skill-specific distillation techniques.

On pruning, Shaw and Goin's "SparseGPT: Remove 100 Billion Parameters for Free" (Shaw and Goin 2023) is an accessible introduction to how one-shot pruning works in practice, showing that OPT-175B can be pruned to 50% sparsity on a single GPU in about four hours. For the technical details behind this, the SparseGPT paper (Frantar and Alistarh 2023) describes how approximate second-order information guides the pruning decisions, and how the pruning pattern can be constrained to NVIDIA's 2:4 structured format for hardware acceleration. Wanda (M. Sun et al. 2024) simplifies things further: it prunes based on the product of weight magnitude and input activation norm, achieving competitive results without any retraining and at a fraction of SparseGPT's computational cost. For the hardware side of sparsity, NVIDIA's blog post on accelerating inference with Ampere Sparse Tensor Cores (NVIDIA 2021) explains the 2:4 structured sparsity format, the train-prune-fine-tune workflow, and the performance gains achievable through TensorRT.

On efficient attention architectures, Sebastian Raschka's "A Visual Guide to Attention Variants in Modern LLMs" (Raschka 2026) covers MHA, GQA, MLA, sliding window attention, and several newer variants side by side with clear diagrams, making it a good companion to the brief treatment of these architectures in this chapter. For a deeper understanding of Multi-head Latent Attention specifically, Eryk Banatt's "Understanding Multi-Head Latent Attention" (Banatt 2025) is a detailed technical walkthrough of the low-rank KV compression, the decoupled rotary position embedding trick, and how MLA compares to GQA in practice.

On tokenization, Tao et al. (2024) establishes scaling laws relating optimal vocabulary size to compute budget, showing that most current LLMs have undersized vocabularies — Llama 2's 32K vocabulary should have been roughly 7x larger for its model size, according to their analysis. This provides theoretical grounding for the vocabulary-size discussion in this chapter.

For broader surveys that cover multiple topics from this chapter:

- Lilian Weng's "Large Transformer Model Inference Optimization" (Weng 2023a) is one of the most widely cited blog posts on the topic, covering quantization, pruning, distillation, and architectural optimizations in a single thorough reference.
- Wan et al. (2024) is the most comprehensive academic survey of LLM efficiency techniques, covering quantization, pruning, distillation, and efficient architectures from model-centric, data-centric, and framework-centric perspectives. It is accompanied by an actively maintained GitHub repository that tracks new papers.
- NVIDIA's "Mastering LLM Techniques: Inference Optimization" (Verma and Vaidya 2023) provides a practitioner-oriented overview that covers quantization, KV cache optimization, batching, and model parallelism in a single article, making it a useful starting point for someone new to the field.

5. Scheduling Bottlenecks

The previous chapter focused on making the model itself smaller and faster, such as fewer parameters, lower precision, or more efficient attention mechanisms. Those techniques change what gets computed. This chapter drills into how requests flow through the system. Even with a well-optimized model, a naive scheduler can leave the GPU sitting idle while requests wait in line, waste compute on padding tokens that contribute nothing, or let a single long prefill block dozens of shorter requests from making progress.

The techniques here — batching strategies, prefill management, and request scheduling — act on the serving system layer. They're about keeping the GPU busy doing useful work as much of the time as possible, regardless of what model is being used.

5.1. Batching Strategies

Why batching matters

In Section 3.4, we established that decode is deeply memory-bandwidth-bound. Each decode step reads the entire set of model weights from HBM to produce just one token per request. On an H100 SXM with a 70B parameter model in FP16, that's roughly 140 GB of weight data read through 3.35 TB/s of bandwidth, requiring about 42 ms just to stream the weights, and all of that work produces a single token per request.

Batching changes this equation. When you process **B** requests in a single decode step, you still read the weights once, but you apply them to **B** token vectors instead of one. The compute scales linearly with **B**, but the data movement stays roughly constant (dominated by the weight read). This directly increases the **arithmetic intensity** — the ratio of FLOPs to bytes moved — pushing decode toward the compute-bound side of the roofline.

Let's put concrete numbers on this. For a single linear layer with weight matrix of shape **(D, D)** in FP16:

- **Data moved**: $2D^2$ bytes (reading the weight matrix)
- **Compute**: $2BD^2$ FLOPs (B matrix-vector multiplications)
- **Arithmetic intensity**: $2BD^2/2D^2 = B$ FLOP/byte

Recall Figure 3.7 shows how increasing batch size moves decode further right toward the compute-bound region, improving throughput.

With batch size 1, the arithmetic intensity is about 1 FLOP/byte, far below the H100's FP16 ridge point of ~295 FLOP/byte. The GPU's tensor cores are almost entirely idle, waiting

for data to arrive from HBM. With batch size 64, the intensity rises to 64 FLOP/byte. Still below the ridge point, but now the GPU is doing 64x more useful work per byte read. With batch size 256, we're approaching the ridge point and the GPU's compute units are finally busy. We have successfully increased MFU, at a roughly constant TPOT.

The throughput gain is dramatic. A system decoding one request at a time on an H100 might produce ~20 tokens/s. The same system decoding a batch of 64 requests might produce ~1500 tokens/s total — roughly the same latency per step, but 50x the throughput. This is why batching is the simplest, most important lever for inference cost efficiency.

The memory cost of batching

There's a catch, and it's a big one. Each request in the batch needs its own **KV cache**, with the stored key and value tensors from every layer, accumulated across every token processed so far. As we'll cover in detail in Section 6.3, the KV cache for a single request can consume several gigabytes for long sequences with large models. The large amount of memory needed for the KV cache sets a **concurrency ceiling** on how many requests can be processed in the same batch.

Here's a concrete example. If your model weights occupy 30 GB on an 80 GB GPU, you have roughly 50 GB left for KV caches (minus activation memory and framework overhead). How many requests fit depends on the model architecture, the KV cache precision, and the sequence lengths involved. A batch of 64 requests, each with 2,048 tokens of context, might consume far more than 50 GB of KV cache, at which point you simply can't fit that many requests.

This is the central batching tradeoff: **larger batches improve GPU utilization and throughput, but each additional request in the batch requires more KV cache memory**. We want the improvements to MFU, concurrency, and throughput from larger batches, but we are limited by the amount of HBM available. HBM is an expensive and limited resource, so you can't easily throw more memory at the problem. If we push the batch size too high, we may run out of GPU memory. Keep it too low and we're leaving performance on the table. The memory management techniques in Section 6.3 and the preemption policies we'll discuss at the end of this chapter focus on this tradeoff.

Example workload

To compare the batching strategies that follow, we'll use the same small workload in Table 5.1: five requests arriving at staggered times, each needing 2 prefill steps and a varying number of decode steps.

Table 5.1.: Example workload used in the batching diagrams below.

Request	Arrival time	Decode steps
A	0	4

Request	Arrival time	Decode steps
B	1	6
C	1	3
D	4	4
E	5	3

This is a toy example. In practice, prefill time depends on input length and can vary enormously across requests, queuing delays can stretch to seconds under heavy load, and decode runs for hundreds or thousands of steps. We've flattened all of that to keep the diagrams readable. The point is to show how each strategy handles staggered arrivals and variable-length outputs, not to depict realistic timescales.

Static batching

The simplest batching strategy is **static batching**: collect a fixed number of requests, process them all together, and don't start the next batch until every request in the current batch has finished generating.

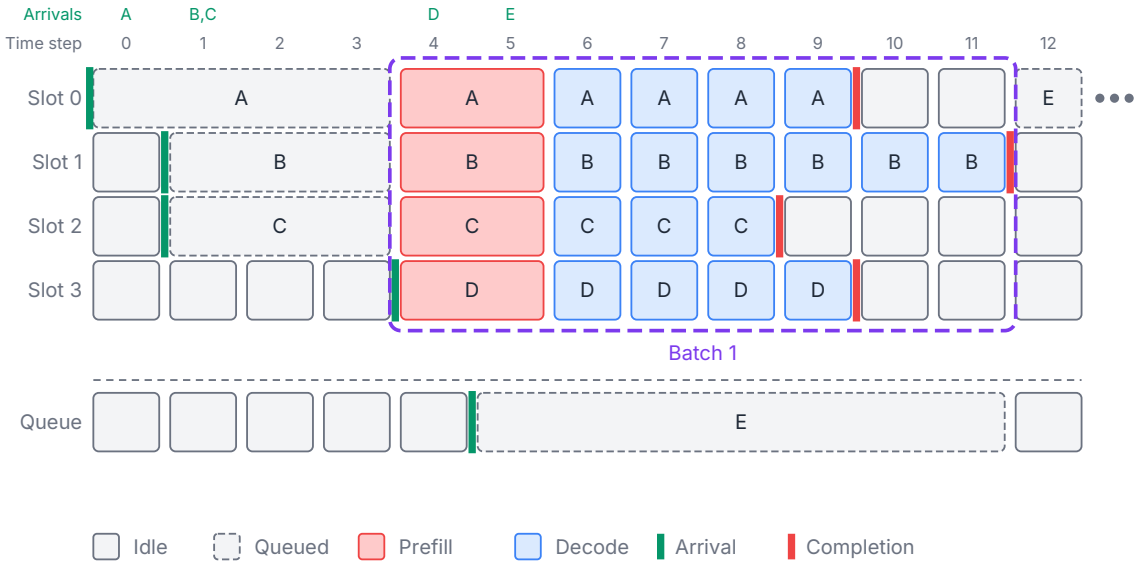


Figure 5.1.: Static batching for the example workload

Figure 5.1 shows the results of our workload with a static batch size of 4 requests. Requests A through D start work at time step 4 and finish at time step 11. At time step 12, Request E is still waiting for three more requests to arrive, and could finish much later. There are obvious problems with these results. First, the GPU sits idle for the first four time steps even though we have requests in hand, because we don't have a full batch yet. Next, Requests A, C, and D finish earlier but their batch slots sit empty. The GPU does no useful work for those slots

while continuing to work only on Request B until it completes. Meanwhile, Request E is stuck in the queue even though there are free batch slots starting at time step 9. This problem where a single slow request holds up the entire batch is known as **head-of-line blocking**. Imagine how much worse this problem would be if B were a very long request generating thousands of tokens. Head-of-line blocking hurts almost all latency and utilization metrics, and it can really destroy goodput.

There are also issues with prefill, which are not so obvious from our simplified diagram. If requests have different input lengths and prefill is done with all of them at the same time (as shown here with A, B, C, and D), then the shorter inputs need to be padded to match the longest one, wasting compute on dummy tokens during prefill.

Static batching works fine for offline workloads where all inputs and outputs have uniform length — say, scoring a dataset of fixed-length classification examples. Having the entire dataset available at the beginning, instead of dealing with unknown arrival times, makes scheduling much easier. For anything with variable-length inputs, variable-length outputs, and stochastic arrival times, which is the common case in online LLM generation, it wastes too many GPU cycles.

Dynamic batching

Dynamic batching improves on static batching by allowing the batch size to vary. Instead of fixing the batch size up front, the scheduler collects requests from a queue and forms a batch based on what's available, subject to a maximum batch size and a timeout. If the batch size is 32 and only 5 requests are waiting, once the timeout elapses, a batch with 5 requests will start processing, rather than waiting for a full batch of 32. This reduces queuing delay and TTFT, and dynamic batching allows the system to adapt to variable arrival rates.

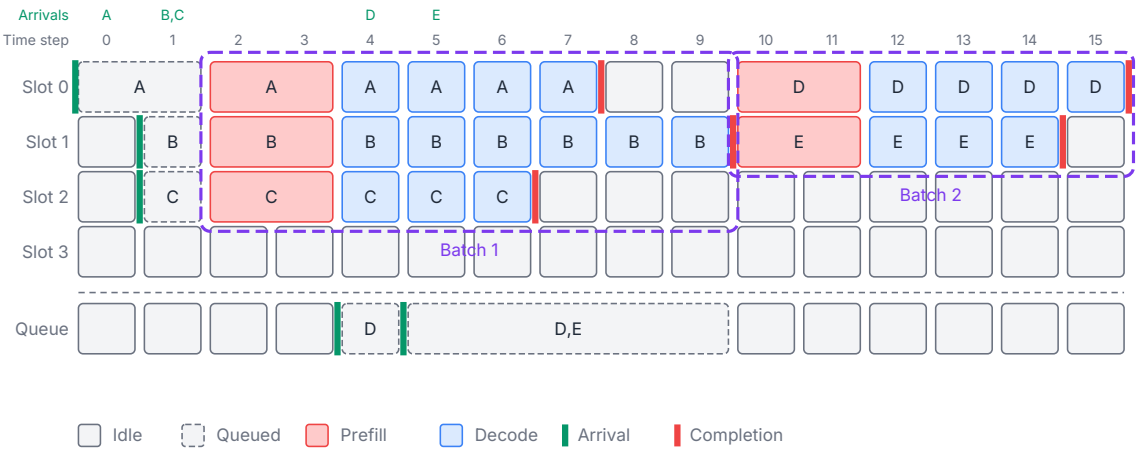


Figure 5.2.: Dynamic batching for the example workload

Figure 5.2 shows the results of dynamic batching on the sample workload in Table 5.1 for a maximum batch size of 4 and a timeout of 2 time steps. Under dynamic batching, the first

batch starts at time step 2 and finishes requests A, B, and C by time step 9. Requests D and E start immediately after Batch 1 finishes. These two requests start at time step 10 and finish at time step 15. Compared to static batching, we have reduced the amount of waiting before Batch 1 starts. But it still has the same fundamental problem as static batching during decode: the batch doesn't complete until the last request finishes generating. Short-output requests leave batch slots that sit idle while they wait for long-output requests to wrap up, so a really long Request B still wastes a lot of GPU capacity, lowering MFU. In our example, requests D and E stay queued until time step 10 even though the GPU has capacity starting at time step 7.

Continuous batching

One breakthrough that modern serving systems are built on is **continuous batching**, also called **in-flight batching**, introduced by Orca (Yu et al. 2022). The key insight is to operate at the granularity of individual decode iterations rather than complete requests.

In continuous batching, the scheduler makes a decision at every decode step: which requests should be in this step's batch? When a request finishes generating (by hitting a stop token or a length limit), its slot is immediately freed and can be filled by a new request from the queue, without waiting for the rest of the batch to finish. This eliminates head-of-line blocking, and significantly improves MFU.

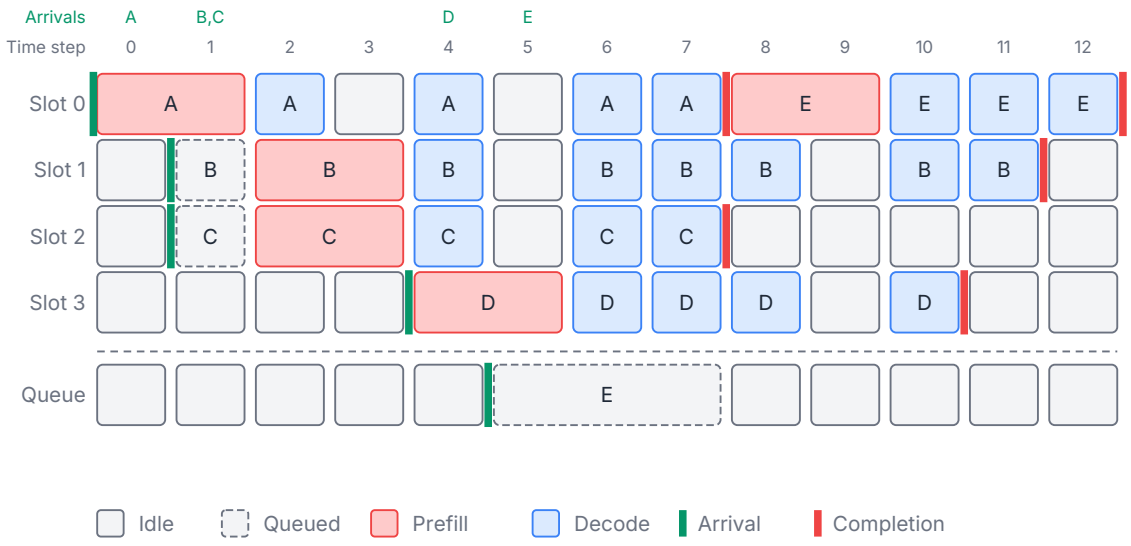


Figure 5.3.: Continuous batching for the example workload

On our workload from Table 5.1, we see in Figure 5.3 that continuous batching completes all five requests by time step 12. Requests A and D each start as soon as they arrive. Request E does have to wait, but it starts processing as soon as the first batch slot is available, at time step 8. Continuous batching has eliminated unnecessary GPU idle time waiting for batches to be full (great for TTFT), and it has eliminated the problem of waiting for every request in a

batch to complete before new requests can be added (great for MFU). It has a weakness in that long prefills still block the entire GPU, harming TPOT. In our example workload, we see this happening at time steps 1, 3, 5, and 9. At time step 1, new prefills can't start until Request A's prefill finishes. At time steps 3, 5, and 9, additional decode steps can't start until the prefills for Requests B, C, D, and E finish. (Note that for static and dynamic batching, we showed all requests having equal prefill times, but in reality, there would also be a version of this problem where long prefills block the GPU from starting new work.)

The throughput improvement from continuous batching can be substantial. Under workloads with high variance in output length, continuous batching can improve MFU and throughput by 2-3x or more compared to static batching, simply by keeping batch slots occupied with useful work. This is why continuous batching is the foundational scheduling technique in serving frameworks like vLLM (Kwon et al. 2023a) and SGLang (Zheng et al. 2023).

> **Note**
>
> Continuous batching requires the serving system to manage per-request state (KV cache, position counters, stop conditions) independently for each slot, and to handle the bookkeeping of inserting and removing requests mid-generation. This is more complex than static batching but is well worth the implementation effort for any production system.

Continuous batching has eliminated most of the GPU idle time, but we've seen it still has a problem with long prefills causing other requests to wait. In this simple implementation of continuous batching, where the prefill for each input is processed all at once, we don't have the ability to start new prefill or decode requests in the middle of a long-running prefill. And it's worthwhile to remember that the prefill of a long input can take dozens or hundreds of times longer than a decode step (not just twice as long, as we've depicted).

This is less of a GPU utilization problem, which would show in MFU, than it is a problem with increasing TPOT and overall request completion latency. In the next section, we will unpack more of the issues with prefill and decode, and ultimately solve this issue of holes in GPU utilization.

5.2. Prefill Strategies

The prefill–decode tension

In Section 3.5, we quantified the nature of prefill being compute-bound and decode being memory-bandwidth-bound. This difference creates a scheduling headache when the system has to do both at the same time.

Consider a serving system running continuous batching with 30 requests in the decode phase, but with capacity for more decoding. A new request arrives with a 4,096-token prompt. The system needs to prefill this request before it can start decoding. But the prefill for

a 4,096-token prompt is a large, compute-intensive operation that can take hundreds of milliseconds, during which the 30 decode requests are blocked from making progress.

From those 30 requests' perspective, waiting while this prefill happens is terrible. Each one is generating text for a user, and they all just stalled for the duration of someone else's prefill. Their TPOT spikes, the streaming output freezes, and the user experience degrades. This is the **prefill–decode tension**: serving a new request's prefill comes at the cost of stalling existing requests' decode steps.

The tension gets worse as prompt lengths increase. With the trend toward longer contexts — 32K, 128K, or even longer — a single prefill can take multiple seconds, creating an unacceptable stall for all active decode requests.

Chunked prefill

Chunked prefill, introduced by Sarathi (Agrawal et al. 2023), resolves this tension by splitting long prefills into smaller chunks and combining each chunk with a decode step.

Instead of processing all 4,096 input tokens in one large forward pass, the system breaks the prefill into smaller chunks, such as 8 chunks of 512 tokens each. At each iteration, the scheduler runs one prefill chunk alongside the batch of decode tokens. The decode requests make progress every iteration, and the new request's prefill completes over 8 iterations instead of blocking everything during one long prefill operation. Capping the duration of each prefill chunk to be on the same order as a decode step eliminates the scheduling holes that hurt TPOT.

With chunked prefill, the differing resource profiles of prefill and decode work in our favor. The prefill chunk is compute-heavy and the decode step is bandwidth-heavy, so combining them in the same iteration gives the GPU a mix of both types of work. In the best case, the memory transfer of the decode step is overlapped with the compute of the prefill chunk, so the GPU doesn't take much longer to complete the combined iteration than it would to do the prefill chunk alone. We can increase MFU for the prefill chunk without adding much to TPOT.

The chunk size is a tuning parameter. Smaller chunks mean smoother decode progress but an increase in the number of iterations needed to complete the prefill, adding overhead from additional kernel launches and attention computations across chunk boundaries. Larger chunks reduce this overhead but create longer pauses in decode progress and increase TPOT. In practice, chunk sizes of 256–2,048 tokens are common, depending on the model size, the capability of the GPU (or other hardware accelerator), and the latency sensitivity of the workload. In this range, the decode memory latency and the prefill compute time are well balanced.

For our sample workload from Table 5.1, we can now improve upon continuous batching by breaking up our prefills (which we simplified to all be exactly twice as long as a decode step) into separate chunks. Figure 5.4 shows that Requests A, B, and D all complete sooner than with vanilla continuous batching. In reality, if prefill were dozens of times slower than decode, the savings would be even greater.

Figure 5.4.: Chunked prefill

Chunked prefill combined with continuous batching improves all aspects of GPU utilization, so both MFU and MBU increase. The net effect is almost all inference metrics improving simultaneously — in particular, TTFT, TPOT, and throughput all improve.

> **Note**
>
> Chunked prefill builds upon continuous batching in an intuitive way. The scheduler treats each prefill chunk as a partial request that occupies a "prefill slot" for one iteration. When the last prefill chunk completes, the request transitions to the decode phase and joins the regular decode batch. The system never has to choose between serving new requests and making progress on existing ones — it does both simultaneously.

Disaggregated prefill

Chunked prefill is effective, but it's still a compromise because prefill and decode share the same GPU and compete for some of the same resources. **Disaggregated prefill** (Zhong et al. 2024) takes a more radical approach: run prefill and decode on entirely separate hardware instances.

The idea is straightforward. Prefill is compute-bound, so prefill instances should be optimized for compute (i.e., a high compute ceiling). Decode is memory-bandwidth-bound, so decode instances should be optimized for memory bandwidth and capacity. By disaggregating the two phases onto separate hardware, each can be independently optimized and scaled without interfering with the other.

The workflow looks like this:

1. A request arrives and is routed to a **prefill instance**
2. The prefill instance processes the full input and generates the KV cache
3. The KV cache is transferred over a fast interconnect (such as NVLink or InfiniBand) to a **decode instance**
4. The decode instance generates tokens autoregressively using the transferred KV cache

A comparison of co-located and disaggregated prefill on a sample workload is shown in Figure 5.5.

The main cost is the KV cache transfer. For a large model with a long prompt, the KV cache can be several gigabytes, and this data needs to move between machines before decoding can begin. The transfer latency adds directly to TTFT, so disaggregated prefill needs a fast interconnect to be practical. With NVLink or high-speed InfiniBand, this transfer can be completed in tens of milliseconds, which is often acceptable.

The benefits go beyond eliminating interference. Disaggregated prefill lets you scale prefill and decode capacity independently. If your workload has long prompts but short outputs, you might need more prefill capacity and less decode capacity. If you're serving a chatbot with short prompts but long responses, the ratio flips. With disaggregated serving, you allocate hardware to match the workload rather than being stuck with a fixed ratio. This allows you to manage TTFT and TPOT SLAs more effectively, maximizing goodput for a given hardware budget.

Disaggregated prefill is a more complex deployment architecture, and the KV cache transfer adds latency and requires fast networking. For many deployments, chunked prefill provides enough balance between the phases. But at large scale, where the workload is diverse enough and the infrastructure supports fast interconnects, disaggregation can provide additional efficiency gains. These benefits will only increase with new hardware accelerators specifically optimized for prefill workloads or optimized for decoding workloads.

5.3. Ragged batching

Even with continuous batching, there's still a source of waste: when requests in a batch have different sequence lengths, the shorter sequences need padding to match the longest one in the batch for the attention computation. **Ragged batching** eliminates this by concatenating all the tokens from different requests into a single flat sequence, with appropriate attention masks to prevent cross-request attention.

In standard batching, all requests must be the same length, so we pad shorter sequences with special tokens to match the longest one, and we wind up with an input batch of shape $\mathbf{(B, S_{max})}$ where $\mathbf{B}$ is the batch size and S_{max} is the length of the longest sequence in the batch. This results in all $\mathbf{B}$ requests performing attention over S_{max} tokens, even if some requests have much shorter sequences.

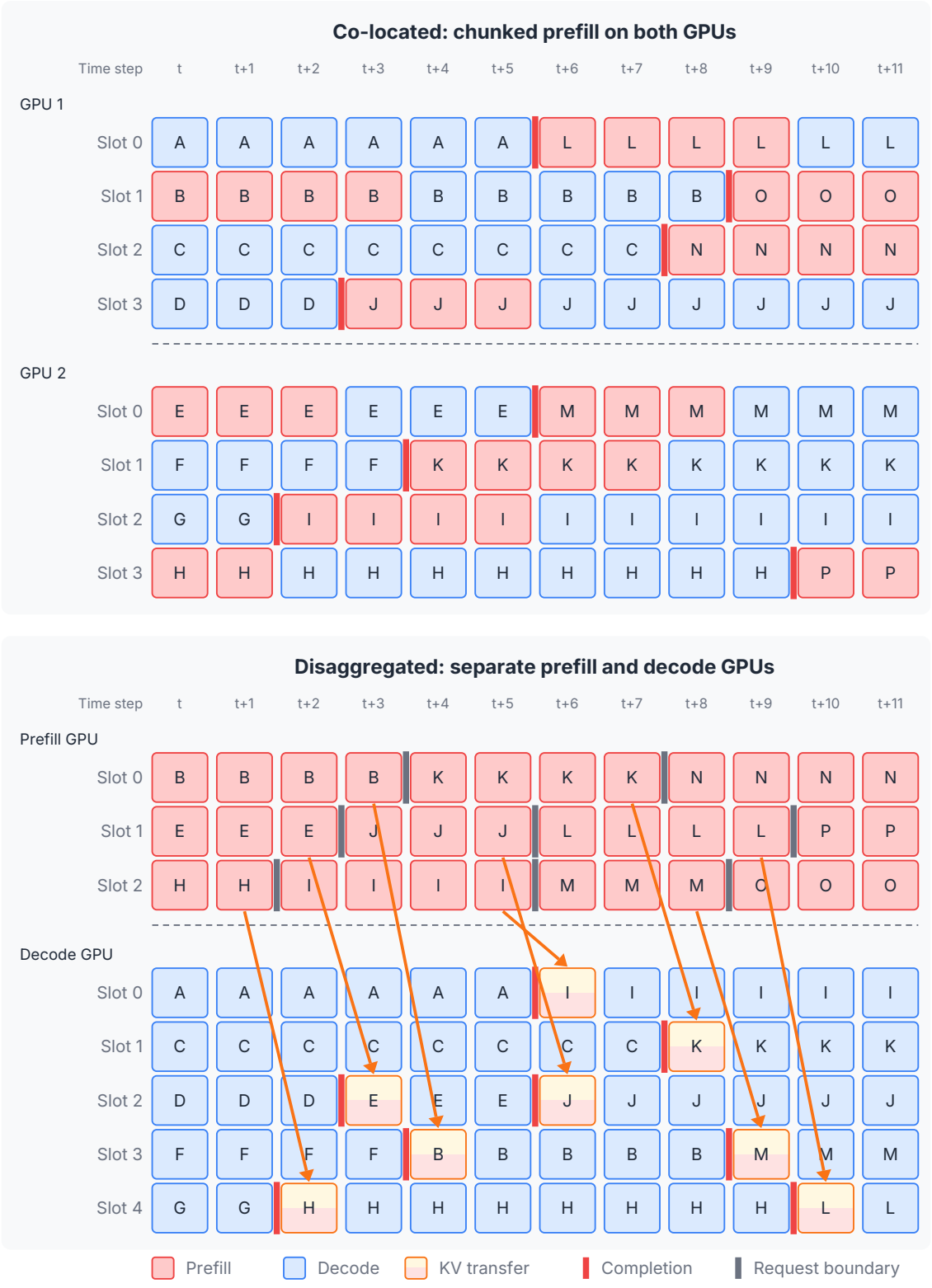


Figure 5.5.: Disaggregated prefill

Instead of padding, ragged batching creates a single packed tensor of shape $(1, S_{total})$, where S_{total} is the sum of all actual token counts. An index array tracks where each request's tokens begin and end, and a special attention mask ensures that tokens from one request can't attend to tokens from another request. To maximize the efficiency of the attention computation, a mask-aware attention kernel such as FlexAttention (Dong et al. 2024) is used. Attention kernels will be discussed in Section 6.1, but the key point here is that the kernel needs to be designed to handle the irregular memory access patterns and skip unnecessary computations. A comparison of simple padded batching and ragged batching is shown in Figure 5.6 and Figure 5.7.

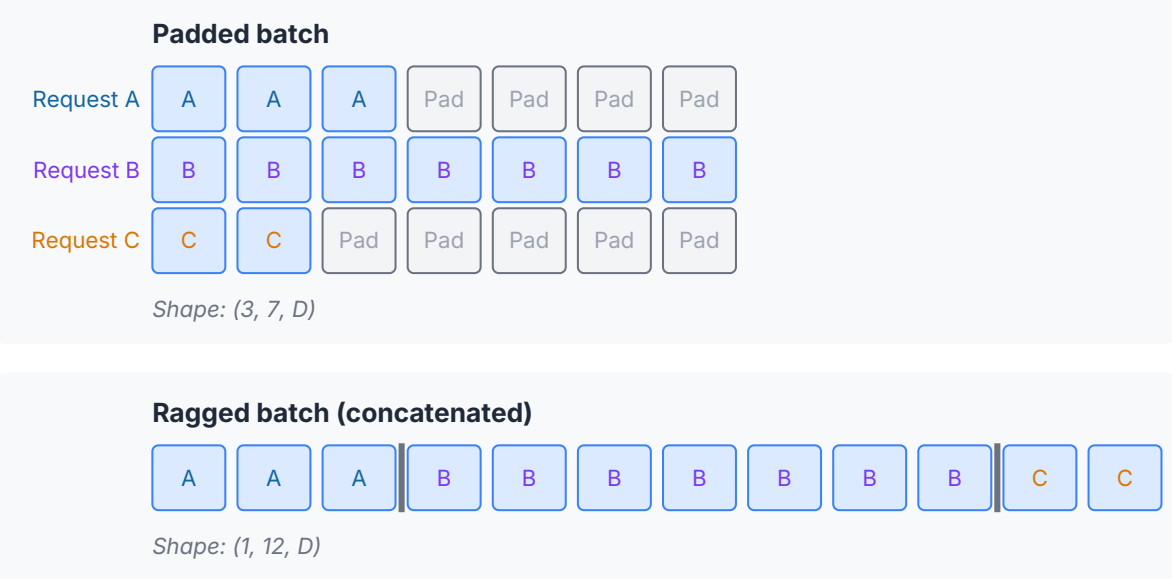


Figure 5.6.: Ragged batching compared to simple batching with padding

The benefits of ragged batching are most pronounced when the input lengths across requests vary widely. A batch with one 2,048-token prompt and three 128-token prompts would waste enormous compute on padding without ragged batching. With ragged batching, the compute is roughly proportional to the actual number of tokens: $2{,}048 + 3 \times 128 = 2{,}432$ tokens instead of $4 \times 2{,}048 = 8{,}192$ tokens. This example shows how ragged batching supports high concurrency without wasteful padding, maximizing the useful work and keeping TTFT low.

The benefits of ragged batching combine particularly well with prefix sharing, which we will discuss in Section 6.4. Figure 5.8 briefly shows how sharing the prefix in ragged batching can save huge amounts of compute with long shared prefixes, such as a long system prompt.

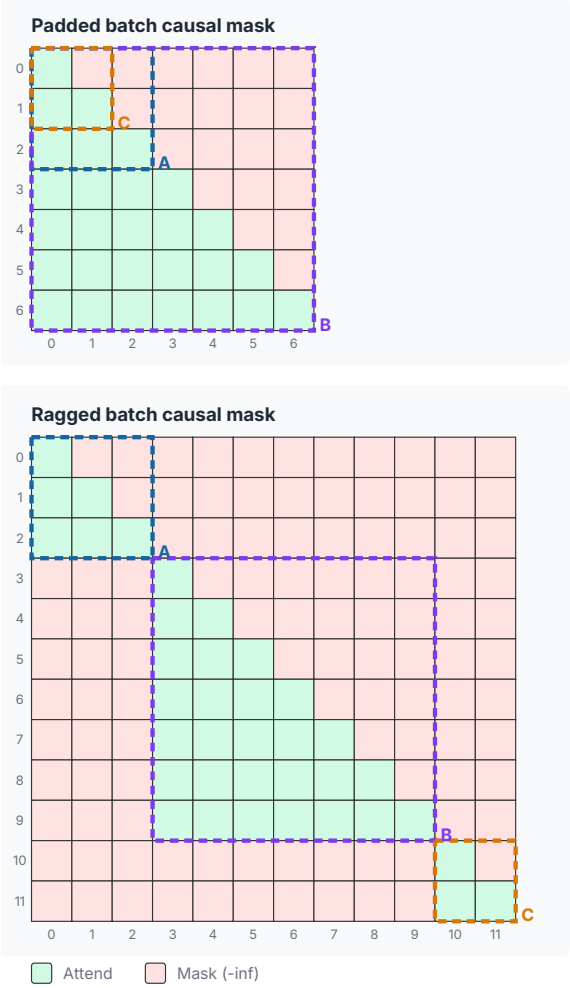


Figure 5.7.: Causal mask for ragged batching. The dashed squares show the portions applicable to each of the requests.

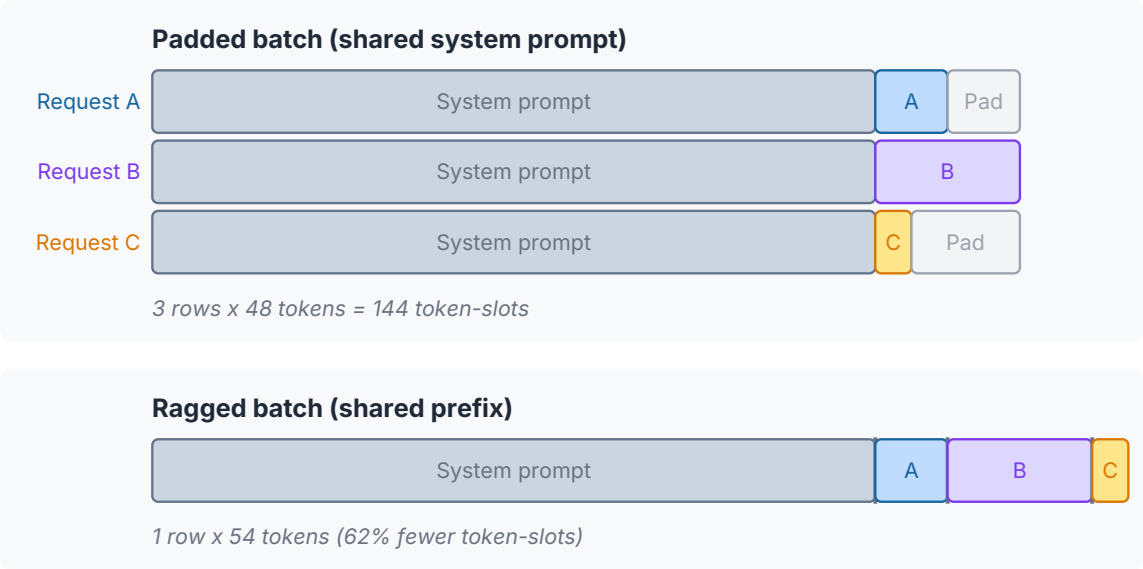


Figure 5.8.: Ragged batching example with a shared prefix

5.4. Request Scheduling and Prioritization

Granularity of scheduling

We have seen in the previous section that continuous batching shifts the scheduling granularity from the batch level to the iteration level. While it introduces some additional overhead, the benefits are substantial, and the overhead lies mostly on the CPU side, allowing it to be overlapped with all of the GPU work. Modern serving systems are built on this iteration-level scheduling, and it allows very fine-grained control over which requests are processed at each step.

Priority-based scheduling

Not all requests are equally urgent. A real-time chatbot response that a user is waiting on matters more — at least from a latency perspective — than a background batch job scoring a dataset. **Priority-based scheduling** lets the serving system express this difference.

In a priority-aware scheduler, each incoming request is assigned a priority level. High-priority requests are placed at the front of the scheduling queue, while lower-priority requests wait until there's spare capacity. The scheduler decides at each iteration which requests to include in the batch, and high-priority requests get scheduled first. With iteration-level scheduling, the scheduler can even preempt lower-priority requests mid-generation, in order to make room for higher-priority requests that arrive.

Priority-based scheduling enables a common production pattern: a system serves both real-time and batch traffic on the same hardware. Real-time requests get low TTFT because they

jump the queue. Batch requests fill in the gaps when real-time traffic is light, keeping the GPU busy during off-peak periods. The result is better overall utilization without sacrificing the latency SLA for interactive users.

SLA-aware scheduling takes this a step further. Instead of simple priority levels, the scheduler tracks each request's deadline — for example, "this request must produce its first token within 500 ms." The scheduler then makes admission decisions based on whether it can meet the deadline. If admitting a new request would cause an existing request to miss its deadline, the scheduler may defer the new request or take other action.

> **Note**
>
> The details of how serving frameworks like vLLM and SGLang implement their iteration-level schedulers — deciding which mix of prefill chunks and decode tokens to process each step — are discussed further in Chapter 8. The scheduler is the heart of a modern serving system, and tuning it well has a large impact on goodput.

Preemption

What happens when the system runs out of KV cache memory? With continuous batching, the scheduler is constantly admitting new requests into the batch. Each new request needs KV cache memory, and the existing requests' KV caches grow with each decode step. If you're lucky, requests finish before you run out of memory, freeing up space for new requests. If you're not so lucky, eventually something has to give.

Preemption is the mechanism for handling this. When KV cache memory is exhausted or if capacity for a new high-priority request needs to be created, the scheduler can preempt one or more existing requests, suspending them and freeing their KV cache memory to make room. When resources become available later, the preempted requests can resume.

There are two strategies for handling the KV cache of a preempted request:

Swap evicts the preempted request's KV cache blocks from GPU memory to CPU memory. When the request resumes, the blocks are swapped back. This preserves the work already done, and the request picks up exactly where it left off without any recomputation. The cost is the time and the HBM bandwidth utilized to transfer data over the PCIe bus between the GPU and CPU. For a KV cache of several hundred megabytes, a swap to CPU memory at PCIe Gen5 speeds (~64 GB/s) takes a few milliseconds. This may be acceptable, but the PCIe bandwidth is shared with other transfers, so the transfer could take longer. In the extreme, high amounts of swapping can degrade decode performance for all requests, even the ones that aren't preempted, harming TPOT and goodput.

Recomputation takes a simpler but more expensive approach: discard the preempted request's KV cache entirely. When the request resumes, the system reruns prefill from scratch to rebuild the KV cache. This frees GPU memory immediately, requires no CPU memory, and doesn't spend time transferring data back and forth between GPU and CPU, but it wastes the compute that went into the original prefill and the calculation of all the KV cache entries

accumulated during decode so far. For a request with a long prompt, recomputation can be very costly. The extra recomputation won't show up in MFU, but it will increase TTFT. In the extreme, high amounts of recomputation can cause almost all metrics to degrade.

Table 5.2.: Comparison of preemption strategies.

Strategy	GPU memory freed	CPU memory needed	Resume cost	Best when
Swap	Yes	Yes	Transfer latency	Plenty of CPU memory, long contexts
Recomputation	Yes	No	Full prefill/decode re-run	Limited CPU memory, rare preemption

The choice between swap and recomputation depends on the situation. If the system has ample CPU memory and preemptions are brief, swapping is better, since you avoid redoing work. If CPU memory is tight or the preempted request would sit idle for a long time, recomputation may be simpler and more predictable. Some systems use a hybrid approach: swap when possible, fall back to recomputation when CPU memory is full.

OOM handling and graceful degradation

Ideally, preemption should be a rare event that happens only when the system is under heavy load and the scheduler can't fit all active requests into memory. But in adverse conditions, such as a sudden burst of long-context requests, preemption alone may not be optimal. At this point, the system may want an additional strategy. Options include:

- **Reducing batch size**: admit fewer concurrent requests, prioritizing the ones already in progress. New requests queue up and wait. Reducing concurrency can protect TPOT and goodput by avoiding repeated preemptions and their associated overhead.
- **Rejecting new requests**: return an error and let the client retry later. This provides immediate feedback for rejected requests, instead of letting them queue for an unacceptably long time and damage goodput.
- **CPU offloading**: proactively keep some fraction of KV cache blocks on CPU memory, streaming them to the GPU as needed. This trades PCIe bandwidth for GPU memory capacity. While this is not ideal for performance, in small doses it can keep the system running smoothly by avoiding preemptions and their impact on performance.

The details of how production systems handle these failure modes — and how to configure policies that balance throughput, latency, and reliability — are discussed further in Section 8.6.

Putting scheduling together

The scheduling techniques in this chapter form a stack. At the bottom, the batching strategy determines how requests are grouped for each forward pass. Continuous batching provides the foundation, keeping batch slots full at all times. Ragged batching eliminates padding waste within each batch. On top of that, chunked or disaggregated prefill manages the resource needs for both prefill and decode. And the priority and preemption policies govern which requests get resources in order to best serve the overall workload.

None of these techniques change the model or its weights. They don't reduce the number of FLOPs needed to process a token or shrink the KV cache per token. What they do is ensure that the FLOPs the GPU performs are useful FLOPs — not wasted on padding, not blocked by long prefills, not sitting idle while a long request holds up the batch. Primarily, these scheduling techniques increase MFU and concurrency, which in turn improves latency and throughput. In the next chapter, we'll turn to techniques that operate on individual requests: optimizing the attention computation, managing the KV cache efficiently, and breaking the one-token-at-a-time constraint of autoregressive decoding.

5.5. Further Reading

For accessible introductions to batching strategies, two sources are notable. The Anyscale blog post (Anyscale 2024a) is the most widely-shared visual explainer of continuous batching. It walks through static vs. continuous batching with clear diagrams and shows where the 23x throughput claim comes from. The original vLLM blog post (Kwon et al. 2023b) introduces PagedAttention with accessible visual explanations of KV cache memory fragmentation and how OS-style paging solves it — a good companion to the academic paper.

Several papers extend the scheduling ideas in this chapter in important directions. Sarathi-Serve (Agrawal et al. 2024) builds on the original Sarathi chunked prefill work, adding stall-free scheduling that eliminates decode stalls even in pipeline-parallel deployments — an important consideration for multi-GPU serving that we don't cover here. For a more accessible treatment of chunked prefill with diagrams and benchmarks, NVIDIA's blog post on chunked prefill in TensorRT-LLM (NVIDIA 2024f) walks through the concept and its impact on GPU utilization. FastServe (Bai et al. 2023) proposes a skip-join multi-level feedback queue scheduler that uses input length information to assign requests to priority queues, a more principled approach to preemptive scheduling than the simple priority levels discussed in this chapter. DeepSpeed-FastGen (Holmes et al. 2024) introduces Dynamic SplitFuse, an approach closely related to chunked prefill that decomposes prefill requests and composes them with decode tokens to maintain consistent forward pass sizes. The DeepSpeed team's blog post (DeepSpeed Team 2023) explains the approach with clear diagrams contrasting SplitFuse with other batching strategies.

On disaggregated serving, Splitwise (Patel et al. 2024) independently explored prefill-decode disaggregation around the same time as DistServe, with additional analysis of how to balance capacity between prefill and decode pools as workload mix changes. Mooncake (Qin et al. 2024) describes Moonshot AI's production disaggregated system, which treats the KV cache

as the first-class scheduling resource and manages it through a distributed cache pool — a concrete example of how disaggregation works at scale with real traffic. For a retrospective on how disaggregation went from research prototype to industry standard across nearly every major serving framework, "Disaggregated Inference: 18 Months Later" (Chen et al. 2025) traces the adoption arc and explores emerging directions like attention-FFN disaggregation.

For scheduling beyond single-instance serving, Llumnix (B. Sun et al. 2024) introduces live migration of requests and their KV caches between model instances, enabling dynamic rescheduling, load balancing, and memory defragmentation across a serving cluster. It's a useful reference for anyone thinking about scheduling at the cluster level rather than the single-GPU level.

6. Request Compute, Memory, and Latency Bottlenecks

The previous chapter tackled scheduling, addressing how to keep the GPU busy by managing how requests are handled. This chapter examines the efficiency with which individual requests are handled. The first two sections address memory-bandwidth bottlenecks at the hardware execution level. The middle sections cover efficient use and reuse of the KV cache. And the remaining sections target the sequential decoding constraint, exploring techniques that can generate multiple tokens per step.

6.1. Memory-Aware Attention Kernels

Why standard attention wastes bandwidth

To understand why attention is expensive in practice, you need to think about where data lives during computation. Recall from Section 3.4 that the GPU has small, fast, on-chip registers and SRAM, and a large, slow off-chip HBM. The straightforward implementation of self-attention is depicted in Figure 6.1 and the memory access is as follows:

1. Read the Q and K tensors, each shape (**H**, **S**, d_K) — reminder, this would be (**B**, **H**, **S**, d_K) with batched processing, but we're avoiding the clutter of repeatedly listing the batch dimension.
2. Compute the score matrix Scores $= QK^T$ and write Scores, shaped (**H**, **S**, **S**), to HBM.
3. Read Scores shaped (**H**, **S**, **S**) back from HBM.
4. Apply SoftMax to get the attention weights and write Attn, also shaped (**H**, **S**, **S**), to HBM.
5. Read Attn shaped (**H**, **S**, **S**) back from HBM, read the V tensor shaped (**H**, **S**, d_K), and compute the weighted values Weighted Values $=$ Attn$V$.
6. Write the weighted values, shaped (**H**, **S**, d_K), to HBM

The read of Q and K at the beginning needs to happen in some form, and the writing of the weighted values also needs to happen at the end. But the writing and reading back of the (**H**, **S**, **S**) Scores and Attn tensors in the middle wastes bandwidth, especially as **S** gets large. For a sequence of length 4,096, the score matrix alone is $4096 \times 4096 = 16.7$ million entries per head. The actual computations of the matrix multiplications and softmax are necessary, fixed costs. The round-trip to HBM for these intermediate results is slow and wastes HBM memory. This problem scales quadratically with sequence length — double the sequence length and you quadruple the HBM usage and intermediate memory traffic. The latency associated with

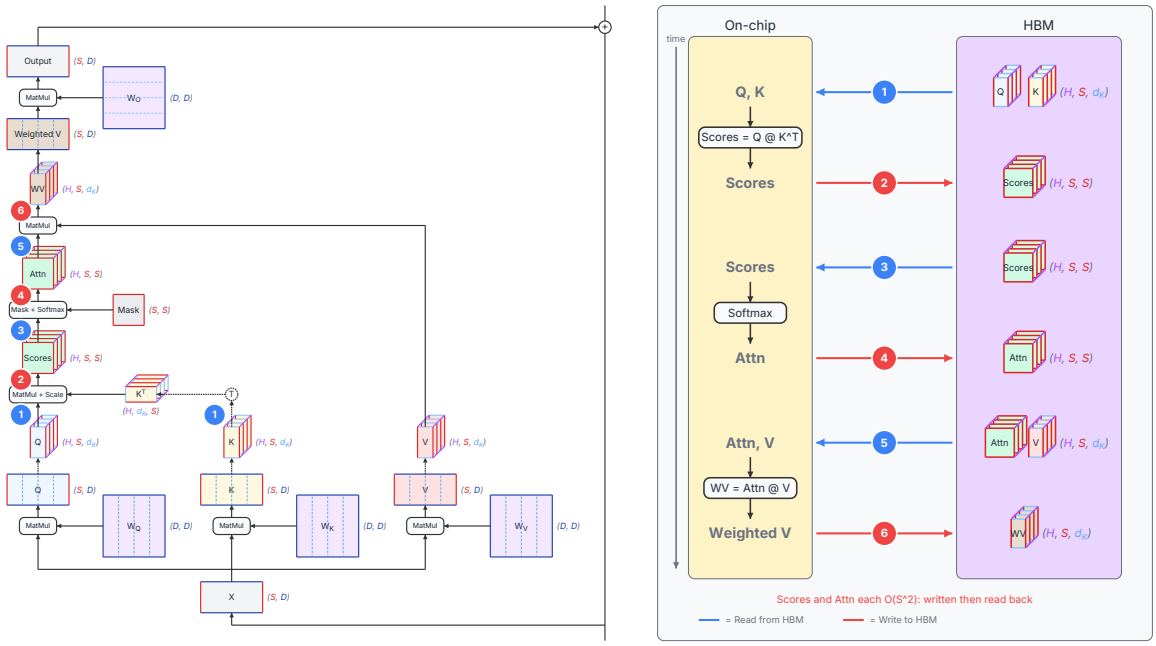


Figure 6.1.: Memory access pattern for a straightforward implementation of self-attention

increased HBM traffic reduces MFU and increases TTFT. The quadratic memory usage is also why this implementation of attention causes GPU out-of-memory errors for long sequences.

It's worth noting that even a single matmul like Scores $= QK^T$ doesn't happen as one monolithic operation. The Q and K matrices are far too large to fit in SRAM, so the GPU kernel **tiles** the computation. Consider Q with shape (**S**, d_K) and K^T with shape (d_K, **S**). A standard tiled matmul loops over row-blocks of Q (groups of query positions) and column-blocks of K^T (groups of key positions). For each pair, the kernel loads a block of Q rows and a block of K^T columns into SRAM, computes their partial product, and writes the result into the corresponding tile of Scores in HBM. Since d_K is small (typically 64 or 128), the inner dimension often fits in a single pass. But the outer dimension doesn't: each column-block of K^T may be re-read for every row-block of Q. This means the total HBM reads exceed the theoretical minimum of reading each matrix exactly once — the GPU re-reads portions of Q and K multiple times to tile through the computation. But that extra read traffic is modest and scales with **S**, not S^2. The real waste in this simple attention implementation is the round-trips between operations: writing the (**H**, **S**, **S**) Scores matrix to HBM just so softmax can read it back, then writing the attention weights just so the next matmul can read them.

FlashAttention

FlashAttention (Dao et al. 2022) solves this memory problem by never materializing the full attention matrix in HBM. If we had enough SRAM, we could store these attention matrices there, and we would avoid the round trip to HBM, but we don't. The core idea with FlashAttention is once again **tiling**: break the Q, K, and V matrices into small blocks that

fit in SRAM, then compute the attention output block by block, keeping all intermediate results on-chip. The key change is combining enough steps into a single kernel that processes one block of K and V against all blocks of Q before moving on. Once a given portion of the weighted values is fully computed for a given block of K and V, this portion of the attention scores and weights are not needed anymore, so there's no need to write them to HBM, and they are discarded.

The algorithm works in two nested loops. The outer loop iterates over blocks of K and V. The inner loop iterates over blocks of Q. For each pair of blocks, the kernel computes the local attention scores, applies a local softmax, and accumulates the weighted sum, all in SRAM. The tricky part is that softmax requires the maximum and sum over the full row of scores, not just the current block. FlashAttention handles this with an **online softmax** technique that maintains running statistics and corrects the accumulated output as new blocks are processed.

The result is depicted in Figure 6.2. For large values of S, HBM traffic drops from $O(\mathrm{S}^2)$ to $O(\mathrm{S} \cdot d_K)$ for the attention matrix, where d_K is the head dimension (typically 64 or 128). For a 4K sequence with $d_K = 128$, that's a reduction from 16.7 million elements to about 524K elements, roughly a 32x reduction in memory traffic for the attention intermediates.

Here's a counterintuitive detail: FlashAttention actually performs *more* FLOPs than standard attention. The online softmax rescaling adds extra multiply-accumulate operations. But wall-clock time drops substantially because the bottleneck was never the arithmetic — it was the HBM traffic. By trading a small increase in compute for a large decrease in memory movement, FlashAttention increases arithmetic intensity and MFU, moving to a much better spot on the roofline. (FlashAttention also has optimizations for model training that are outside the scope of our inference discussion.)

FlashAttention's tiling is most effective during prefill, where the sequence length $\mathbf{S}$ (the full prompt) is large and the $O(\mathrm{S}^2)$ memory traffic savings and decrease to TTFT are dramatic. During decode, however, the query dimension is just 1 (the new token). FlashAttention parallelizes over batch size and query length, so with a single-token query, most of the GPU's streaming multiprocessors sit idle — there simply aren't enough blocks of work to fill the machine.

This gap motivated a family of dedicated decode-phase attention kernels. **Flash-Decoding** (Dao et al. 2023) added a new parallelization dimension: splitting the KV sequence across thread blocks, with a lightweight reduction kernel to rescale and combine partial results. This keeps the on-chip memory benefits of FlashAttention while fully utilizing the GPU even at small batch sizes, delivering up to 8x speedups for long contexts. **FlashDecoding++** (Hong et al. 2023) further improved on this with asynchronous softmax (avoiding a global synchronization barrier) and better handling of the flat GEMM shapes that arise during decode. More recently, **FlashMLA** (DeepSeek-AI 2025) provides optimized decode kernels for DeepSeek's Multi-head Latent Attention, and **FlashInfer** (Ye et al. 2025) offers a unified kernel library covering prefill, decode, and append with support for paged KV caches and attention variants like GQA.

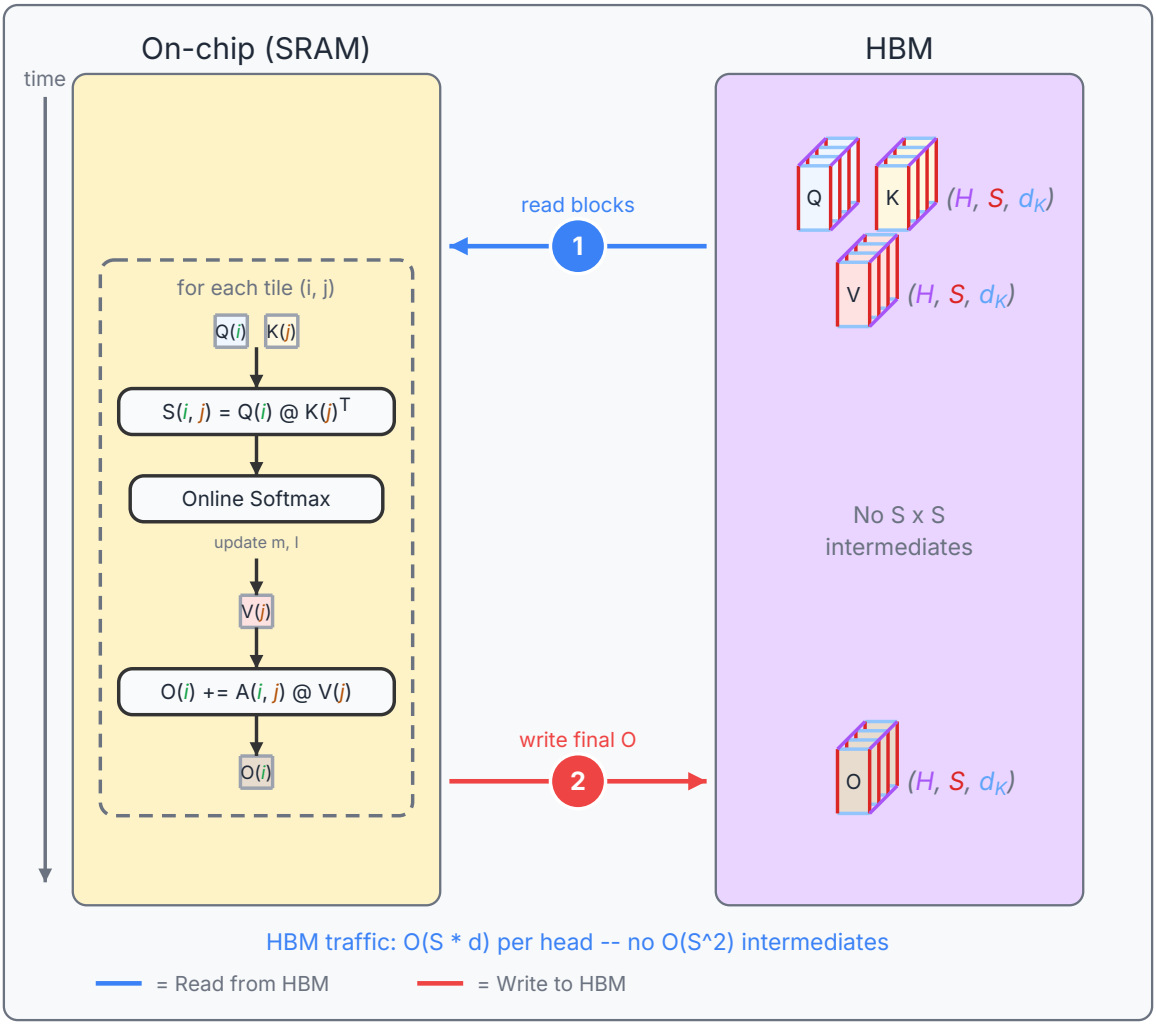


Figure 6.2.: FlashAttention keeps all intermediate results on-chip — only two HBM transfers instead of six

FlashAttention v1, v2, v3, and v4

The original FlashAttention (Dao et al. 2022) established the tiling approach. Subsequent versions improved the implementation substantially:

FlashAttention-2 (Dao 2023) restructured the algorithm to improve GPU parallelism. The key change was swapping the inner and outer loops – iterating over Q blocks in the outer loop and K/V blocks in the inner loop. This reduced the amount of shared memory communication between warps (groups of 32 threads that execute together on the GPU) and improved occupancy on modern GPUs. The result was roughly a 2x speedup over v1.

FlashAttention-3 (Shah et al. 2024) targeted the H100 specifically, exploiting its new hardware features. It introduced **warp specialization** – dedicating some warps to data loading while others perform computation, overlapping memory access and arithmetic within the same kernel. It also leveraged the H100's FP8 tensor cores and hardware-assisted asynchronous memory operations. These changes pushed FlashAttention closer to the theoretical peak throughput on H100 hardware. **FlashAttention-4** (Dao et al. 2025) continues this hardware-specific approach for the Blackwell B200, achieving up to 1,605 TFLOPS and 71% hardware utilization.

The progression across versions illustrates a broader principle: writing a correct tiled attention kernel is highly beneficial, and extracting maximum performance from a specific GPU architecture can lead to further gains. Many other attention kernels have been written. One worth mentioning is **FlexAttention** (Dong et al. 2024), which provides a flexible framework in PyTorch for implementing various attention algorithms (dense, sparse, low-rank) without having to implement new kernels from scratch for every use case.

> **Note**
>
> FlashAttention handles the tiling of attention computation within a single GPU. When the KV sequence is very long, you may also want to parallelize the attention computation *across* the sequence dimension using multiple GPU cores or even multiple GPUs. This is the role of FlashDecoding, which we'll cover in Section 7.6.

6.2. Kernel Fusion and Compute Graph Optimization

FlashAttention is a specific instance of a more general optimization strategy: reducing the number of times data makes a round trip to HBM by combining operations into fewer GPU kernels. This section covers the general techniques.

Kernel fusion

Every time the GPU finishes one kernel and starts the next, there's an implicit data handoff through HBM. The output of the first kernel gets written to HBM, and the input of the

second kernel gets read from HBM. If those two kernels operate on the same data, this round trip is pure waste.

Kernel fusion combines multiple operations into a single GPU kernel so that intermediate results stay in registers or SRAM. Common fusion opportunities in transformer inference include:

- **Fused QKV projection**: instead of three separate matrix multiplications for Q, K, and V, fuse them into a single kernel that reads the input activations once and produces all three outputs
- **Fused softmax + mask**: apply the causal mask and softmax in one pass, avoiding a separate write-then-read for the masked scores
- **Fused add + LayerNorm**: combine the residual addition and layer normalization, keeping the intermediate sum in registers
- **Fused RoPE + attention**: apply rotary position embeddings to Q and K within the attention kernel itself

Each fusion eliminates one HBM round trip. For a single transformer layer, there might be 10 or more such opportunities. Across 80+ layers in a large model, these small savings add up to a meaningful reduction in total memory traffic and kernel launch overhead, increasing arithmetic intensity and decreasing TPOT.

CUDA graphs

We introduced CUDA graphs briefly in Section 3.4. Here's the full picture.

During decode, the GPU executes the same sequence of kernels at every step — the same transformer layers in the same order with the same shapes (for a given batch size). But the CPU doesn't know that. At each step, it launches each kernel individually, waits for the GPU to be ready, sends the launch command, and moves on to the next kernel. Each launch takes roughly 5–10 microseconds, which doesn't sound like much, but a single decode step through a large model might involve hundreds of kernel launches. At 5 μs each, that's hundreds of microseconds per token of pure overhead — time added to TPOT that the GPU spends idle waiting for the CPU to tell it what to do next.

A **CUDA graph** captures a fixed sequence of kernel launches into a single replayable object. You run the decode step once in "capture mode," which records every kernel launch, memory copy, and synchronization point. Then on subsequent steps, you replay the entire graph with a single command. The CPU issues one launch instead of many, and the GPU executes the full sequence without waiting.

The main limitation is that CUDA graphs are static. The captured graph assumes fixed tensor shapes and memory addresses. If the batch size changes, or a request finishes and leaves the batch, the graph is invalidated and must be recaptured. Serving frameworks handle this with a small pool of pre-captured graphs for common batch sizes and sequence lengths, falling back to eager execution for uncommon shapes.

Computation graph compilation

Rather than manually fusing kernels, you can let a compiler do it. **Computation graph compilers** analyze the full model graph, identify fusion opportunities, optimize memory layouts, and emit optimized GPU code. This more efficient code improves both TTFT and TPOT, but doesn't directly support higher concurrency.

`torch.compile` in PyTorch traces the model's computation graph and applies a series of optimizations including operator fusion, memory planning, and code generation through backends like Triton (Tillet et al. 2019). For inference workloads, it can deliver significant speedups with minimal code changes.

TensorRT-LLM (NVIDIA 2024g) takes a more aggressive approach. It converts the model into NVIDIA's TensorRT format, which applies extensive graph-level optimizations — layer fusion, precision calibration (auto-selecting FP16 or FP8 for each operation), kernel auto-tuning, and memory-efficient execution planning. The compilation step takes longer, but the resulting engine is heavily optimized for the specific model and GPU.

The tradeoff is flexibility versus performance. Eager execution (no compilation) is the most flexible but the slowest. `torch.compile` offers a middle ground. TensorRT-LLM produces the fastest code but requires a more involved build step and is harder to modify. Most production serving frameworks use some form of graph compilation, often TensorRT-LLM for NVIDIA hardware.

6.3. KV Cache Engineering

The KV cache is one of the biggest memory consumers during inference, and managing it well has a direct impact on concurrency levels and maximum sequence lengths. This section covers the core techniques for reducing KV cache memory pressure.

KV cache memory formula

For a model with $\mathbf{L}$ layers, H_{KV} KV heads per layer, and head dimension d_K, each token adds the following to the KV cache:

$$\text{bytes per token} = 2 \times \mathrm{L} \times H_{KV} \times d_K \times b$$

The factor of 2 accounts for both keys and values. b is the bytes per element (2 for FP16/BF16, 1 for FP8/INT8).

For a concrete example, consider a 70B parameter model with $\mathrm{L} = 80$ layers, $H_{KV} = 8$ GQA heads (see Section 4.4), $d_K = 128$, stored in FP16:

$$\text{bytes per token} = 2 \times 80 \times 8 \times 128 \times 2 = 32,768 \text{ bytes}$$

On an 80 GB GPU that's already holding 35 GB of model weights (a 70B model in INT4), you're consuming another MB for about every 30 tokens per request. The techniques in this section – paging, quantization, compression, and eviction – deal with how we can fit more requests into the same memory budget.

PagedAttention

In a naive implementation, you allocate a contiguous block of GPU memory for each request's KV cache sized for the maximum possible sequence length. This wastes memory in three ways, illustrated in Figure 6.3:

- **reserved** slots are those held for tokens that will be, but haven't been generated yet,
- **internal fragmentation** is slots allocated that will never be used because the sequence won't reach the maximum length, and
- **external fragmentation** is gaps between allocations that are too small to be useful (which arise if requests have varying lengths)

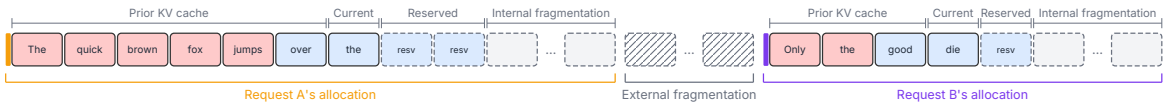


Figure 6.3.: Memory waste with contiguous KV cache allocation. Each request pre-allocates for the maximum sequence length, leaving most of the memory unused.

This naive approach is good in that it avoids out of memory errors for requests that get the initial allocation, it guarantees contiguous memory for each request's KV cache, and it's simple to implement. But the **PagedAttention** paper (Kwon et al. 2023a) found that combined wasted memory could be as high as 80%.

PagedAttention, which was introduced in vLLM, borrows the virtual memory model from operating systems to solve the wasted memory problem. Instead of one big contiguous allocation per request, the KV cache is divided into fixed-size **pages** (also called blocks), typically holding 16 or 32 tokens each. A request's KV cache is a linked list of pages that can be scattered anywhere in GPU memory. The same requests from Figure 6.3 are shown in Figure 6.4 with a paged memory model.

The benefits are significant:

- **Near-zero fragmentation**: no external fragmentation, and the only internal fragmentation is the partial last page of each request. With 16-token pages, average waste is 8 tokens per request — negligible compared to the old approach of reserving memory for the maximum sequence length. The near-zero memory waste allows dramatically higher concurrency and longer sequence lengths within the same memory budget.

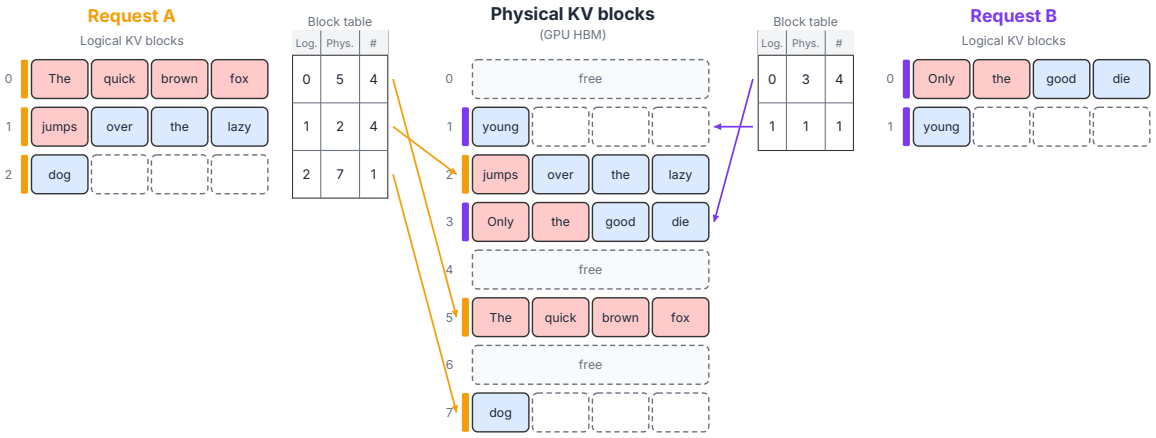


Figure 6.4.: Paged attention

- **Dynamic allocation**: pages are allocated on demand as the sequence grows, so a short request doesn't tie up memory it will never use, and a long generation doesn't hold reserved memory for long periods.
- **Sharing**: two requests that share a common prefix (same system prompt, for instance) can point to the same physical pages for the shared portion, storing the data once. The combination of paged attention and ragged batching (discussed in Section 5.3) is a beautiful solution that reduces both KV cache memory usage and memory reads, benefiting TPOT. Figure 5.8 shows the elegance and large savings when you combine the two, and is worth reviewing again here.
- **Fine-grained preemption**: if the scheduler needs to free memory (Section 5.4), pages for an evicted request can be reallocated individually, rather than killing the entire request. vLLM uses this optimistic strategy with recomputation, finding that when the request is resumed, most of the needed pages are often still in memory, avoiding much of the recomputation cost.

PagedAttention has become the standard approach for KV cache memory management. Nearly every major serving framework, including vLLM, SGLang, and TensorRT-LLM, uses some variant of it.

KV cache quantization

The KV cache formula above assumed 2 bytes per key/value using FP16 storage. If you store keys and values in INT8 or FP8 instead, you immediately halve the memory per token. For our 70B model example, that drops per-request KV cache memory from 1.28 GB to 640 MB at 4K sequence length.

KV cache quantization is separate from model weight quantization (Section 4.1). The model weights can be in one precision while the KV cache is stored in another. In practice, many deployments that run model weights in INT4 keep the KV cache in FP8 or INT8, because these different components have different sensitivity to precision loss.

The quality impact of KV cache quantization is generally small. Keys and values are intermediate activations, not learned parameters, and small quantization errors in individual KV entries tend to average out across the many entries involved in an attention computation. The tradeoff is straightforward: reduced memory for a small and often negligible reduction in output quality.

Another quantization approach, which garnered a lot of attention recently, is **TurboQuant** (Zandieh et al. 2025). Rather than simply choosing a new datatype, they use techniques like random rotations to support quantization to lower bit depths. They report success squeezing entries down to 3.5 bits without hurting LLM performance. Interest in KV cache quantization techniques is high because bandwidth-bound nature of decoding means that reductions in KV cache size provide close to linear speedups to TPOT and throughput.

KV cache compression

Rather than quantizing individual elements, you can compress the KV cache using a learned low-rank representation. The idea is to store a compressed version of the keys and values that uses fewer dimensions, then reconstruct the full representation when needed for attention.

Multi-Head Latent Attention (MLA), introduced in DeepSeek-V2 (A. Liu et al. 2024a), is the most prominent example. As was mentioned in Section 4.4, instead of caching separate key and value tensors for each head, MLA compresses them into a single low-rank latent representation. This can reduce KV cache size by 5–10x compared to standard multi-head attention, at the cost of additional compute to decompress during attention (and there are operation fusion techniques to mostly eliminate the overhead). The key difference from the attention architecture changes covered in Section 4.4 (like GQA and MQA) is that MLA achieves compression without reducing the number of effective attention heads – it maintains full representational capacity while storing less data.

Memento (Microsoft Research 2026) is a research model from Microsoft that includes the LLM in the process of learning a compressed KV representation. Rather than utilizing a separate compression process, Memento trains the LLM itself to participate in the compaction process for long reasoning chains. First, the model performs reasoning in chunks. At the conclusion of each chunk, the model produces a short, compressed summary of the key information from that portion of the reasoning. This summary is kept, and the raw text from the reasoning chunk is discarded. Since long reasoning chains make up the majority of generated tokens in many applications, this approach can reduce the KV cache size by 2-3x and double throughput. This is just one example of the active research in KV cache compression and compaction.

Selective KV cache and token eviction

Not all tokens in the context are equally important for generating the next token. In many attention patterns, a small fraction of tokens receive the vast majority of attention weight, while most tokens contribute very little. If you can identify tokens that never receive significant attention, you can remove them from the KV cache without impacting the attention

mechanism. This observation motivates **selective KV cache** strategies that drop or compress low-importance entries.

Keyformer (Adnan et al. 2024) identifies important tokens based on accumulated attention scores. Tokens that consistently receive high attention weight across layers and heads are retained, while low-scoring tokens are evicted. The retained set is refreshed periodically as the model generates more tokens and attention patterns shift.

Native Sparse Attention (NSA) (Yuan et al. 2025) from DeepSeek takes a different approach, designing the sparsity pattern into the model architecture itself. Rather than using a learned or heuristic eviction policy at inference time, NSA defines structured sparse attention patterns that combine compressed coarse-grained tokens, selected high-relevance tokens, and a sliding window of recent tokens. This approach avoids the quality risk of post-hoc eviction because the model is trained to work with the sparse pattern.

The general tradeoff with token eviction is that it reduces memory but introduces risk. If the eviction policy drops a token that turns out to be important for a later generation step, the output quality degrades. Conservative eviction (keeping more tokens) is safer but saves less memory.

6.4. Prompt and Prefix Caching

The shared prefix opportunity

In many real-world deployments, a large fraction of requests share the same prefix. A chatbot application might prepend a 2,000-token system prompt to every user message. A RAG pipeline might prefix retrieved context that is identical across multiple queries. A coding assistant might include the same repository-level context for many completion requests.

Without caching, the system runs prefill on those 2,000 shared tokens for every single request, computing the same key-value pairs over and over. If you're serving 100 requests per second, that's potentially 200,000 tokens of redundant prefill per second. The wasted compute shows up directly as higher TTFT and lower throughput.

RadixAttention

RadixAttention (Zheng et al. 2023), introduced in the SGLang framework, stores KV cache entries in a **radix tree** keyed by token sequences. A radix tree is a prefix-compressed trie: shared prefixes are stored once, and branches diverge only where token sequences differ. An example radix tree is shown in Figure 6.5.

When a new request arrives, the system walks the radix tree matching its token sequence. If the first 2,048 tokens match a cached prefix, those KV entries are reused directly — no prefill needed for them. The system only runs prefill on the remaining tokens that diverge from the cached prefix. For a request with a 2,000-token system prompt and a 200-token user message, this cuts prefill work by roughly 90%.

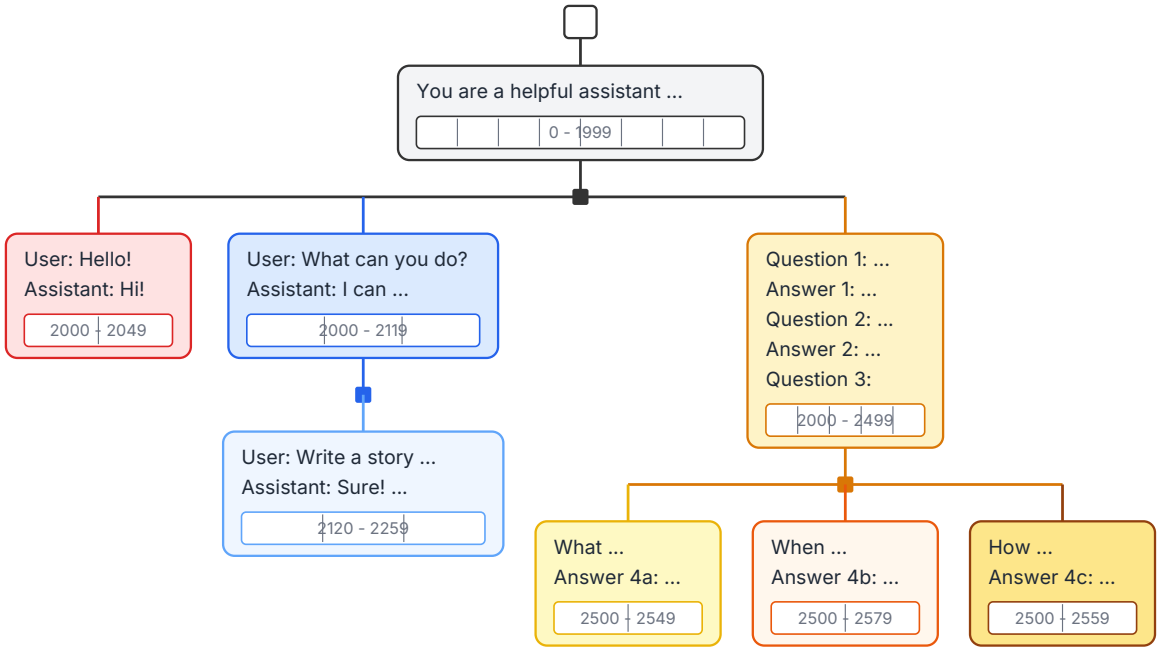


Figure 6.5.: Radix attention: an example radix tree with several different prompts and responses sharing different common prefixes.

The radix tree structure enables fine-grained prefix matching that goes beyond simple exact-prefix caching. Two requests that share a system prompt but have different few-shot examples will still reuse the system prompt portion. A multi-turn conversation reuses all KV entries from previous turns.

Prompt caching in production APIs

The radix tree approach is what happens inside the inference engine. At the API level, prompt caching is often exposed as an explicit feature. When prefix caching is enabled, a provider hashes the prefix tokens and looks up cached KV entries. If found, the cached entries are reused and the customer pays reduced cost for the cached portion.

This is a win for both sides: the user gets faster TTFT and lower cost, and the provider avoids redundant compute. The effectiveness depends on how much prefix overlap exists in the workload. Applications with long, stable system prompts benefit enormously, while applications with unique prompts for every request see little benefit.

Implementation dependencies

Prefix caching can be implemented very efficiently with the paged KV cache from Section 6.3. In the PagedAttention model, the physical pages for the shared prefix are simply referenced by multiple requests through their page tables. The pages themselves are stored once in GPU

memory, and a reference count tracks how many requests are using each page. When the last request using a shared prefix completes, the pages can be freed or kept for future reuse based on an eviction policy (typically LRU).

Without paged memory, sharing would require contiguous memory layouts to be identical across requests, which is impractical. PagedAttention makes prefix sharing a natural consequence of the memory model.

6.5. Speculative Decoding and Multi-Token Prediction

Everything up to this point has optimized the cost of individual prefill or decode steps. But there's a more fundamental constraint we haven't attacked yet: the autoregressive loop itself. Standard decoding produces one token per forward pass, and each token depends on the previous one. For a 500-token response, that's 500 sequential forward passes through the entire model. Even if each pass is perfectly optimized, you're still bound by the serial dependency chain.

This section covers techniques that break or amortize that sequential constraint, generating multiple tokens per model invocation.

The core idea of speculative decoding

Speculative decoding (Leviathan et al. 2022; C. Chen et al. 2023) is based on a simple but powerful observation: verifying a sequence of multiple tokens is cheaper than generating them one at a time.

Here's the setup. You have a large **target model** — the main model you actually want to generate from — and a small, fast **draft model**. The draft model generates k candidate tokens autoregressively (cheap, because the draft model is small). Then the target model processes all k candidates in a single forward pass, checking whether it would have generated the same tokens. Accepted tokens are kept. The first rejected token is resampled from the target model's distribution. Draft tokens after the first rejection cannot be used because they have diverged from the target model's distribution, so they are discarded. Then the cycle repeats.

In the best case, all k draft tokens are accepted, and you get $k + 1$ tokens from one target model forward pass. In the worst case, the first draft token is rejected, and you get just 1 token from that target model forward pass. If you're averaging more than one token per target forward pass, TPOT is improved. A comparison of standard decoding and speculative decoding is shown in Figure 6.6.

The reason speculative decoding works is the asymmetry between generation and verification. Generating k tokens with the target model requires k sequential forward passes. But *verifying* k tokens requires just one forward pass, because you can feed all k tokens as input and check them all in parallel, in the same way prefill processes multiple tokens at once. Blockwise Parallel Decoding (Stern et al. 2018) is an early precursor to this idea.

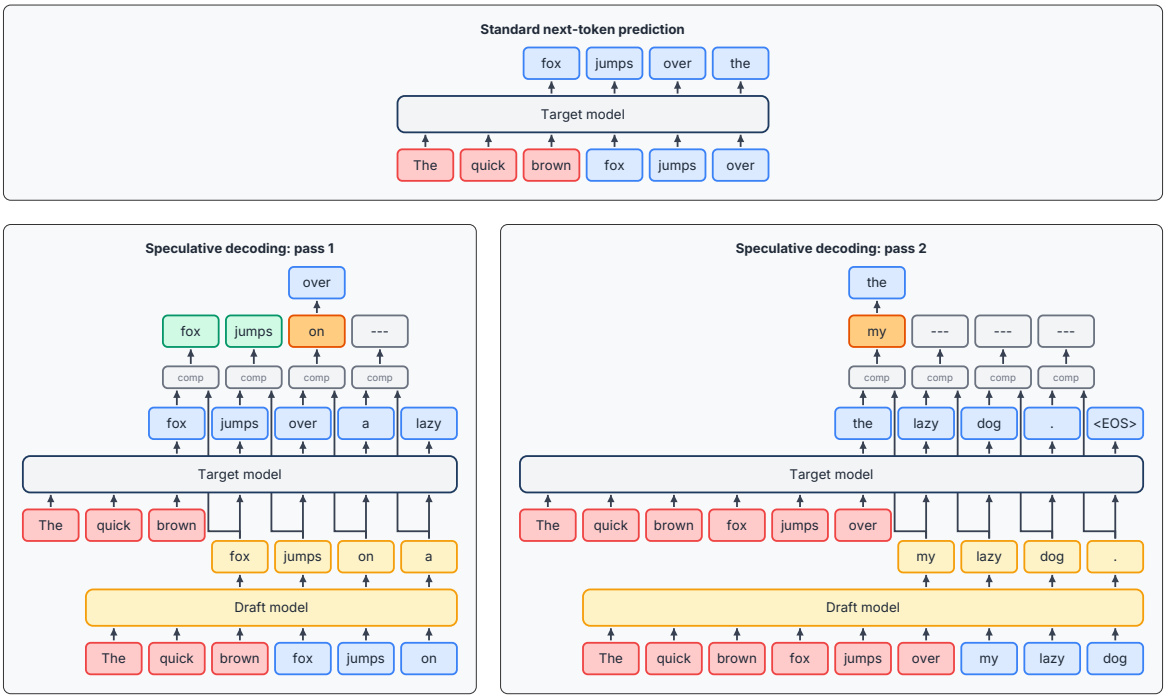


Figure 6.6.: Speculative decoding showing a sequence of four tokens that are generated in only two forward passes of the target model. In the first pass, two tokens from the draft model are accepted, and resampling generates a third token. In the second pass, the worst case scenario is illustrated. No tokens are accepted, which means that only the one resampled token is kept from this forward pass.

Acceptance rate and expected speedup

The **acceptance rate** α is the probability that the target model accepts a draft token. It depends on how well the draft model approximates the target model's distribution for the current context and task. Typical acceptance rates range from 0.6 to 0.9, depending on the draft model quality and the generation temperature.

The expected number of tokens generated per verification cycle is:

$$E[\text{tokens per cycle}] = \frac{1 - \alpha^{k+1}}{1 - \alpha}$$

where k is the number of draft tokens. For $\alpha = 0.8$ and $k = 4$, this gives about 3.4 tokens per cycle on average.

The speedup to TPOT depends on the relative cost of the draft and target models. If the draft model runs at c times the speed of the target model (so a draft forward pass takes $1/c$ the time of a target forward pass), the wall-clock speedup is approximately:

$$\text{speedup} \approx \frac{E[\text{tokens per cycle}]}{1 + k/c}$$

The denominator accounts for the cost of the k draft passes (each costing $1/c$ of a target pass) plus the one verification pass. For $c = 10$ (the draft model is 10x faster), $k = 4$, and $\alpha = 0.8$, the speedup is roughly $3.4/1.4 \approx 2.4$x.

A crucial property of speculative decoding is that it produces *exactly* the same distribution as the target model. This is not a heuristic or an approximation. The acceptance/rejection mechanism is designed so that the output distribution is identical to what you'd get from running the target model autoregressively. The speculative decoding paper goes into the details of how rejection sampling is used to ensure the exact statistical distribution is preserved.

Tree-based speculative decoding

Instead of generating a single chain of k draft tokens, the draft model can produce a **tree of candidates**. At each position, the draft model proposes multiple alternatives, branching into different possible continuations.

The target model verifies the entire tree in a single forward pass using a carefully constructed attention mask. The longest accepted path through the tree becomes the output. Because the tree explores multiple branches, it has a higher chance of finding a long accepted sequence than a single chain does, especially when individual token acceptance rates are moderate.

The cost is that the tree contains more total tokens than a single chain, so the verification forward pass does more work. But since the verification step is typically memory-bandwidth-bound (it's essentially a prefill over the tree tokens), the marginal cost of additional tree tokens is often small.

Inference with reference

Inference with reference (Yang et al. 2023) is a specialized form of speculative decoding where the "draft" comes from an existing text, which could be a retrieved document, a cached previous response, or template text. If the target model's likely output overlaps significantly with the reference text, you can propose long spans of reference tokens as candidates and verify them in a single forward pass.

This works especially well for tasks like text editing, abstractive summarization (where the output may quote the input), or regenerating a cached response with minor modifications. The acceptance rate can be very high — often above 0.95 — making this one of the most efficient speculative decoding variants when it applies.

Speculative speculative decoding

Standard speculative decoding has a subtle inefficiency: the draft model must wait for the target model to complete verification before it can start drafting the next batch of tokens. This is because the draft model needs to know which tokens were accepted so it can start with the correct prefix.

Speculative speculative decoding (Kumar et al. 2026) eliminates this sequential dependency. The draft model speculatively starts generating the next batch of candidates *before* the target model finishes verifying the current batch. It does this by assuming all current draft tokens will be accepted and conditioning on that optimistic prefix. If some tokens are rejected, the speculative draft work is discarded and redone, but when acceptance rates are high, the early generation pays off, and the draft and target models can operate in a pipelined fashion with better hardware utilization.

When speculative decoding helps vs. hurts

Speculative decoding is not a universal win. Several factors determine whether it helps:

Temperature: at low temperatures (greedy or near-greedy sampling), the draft model's predictions are more likely to match the target model's sampled tokens, giving higher acceptance rates. At high temperatures, the target model's distribution is more spread out, making it harder for the draft to guess correctly. Speculative decoding works best for deterministic tasks like code generation and factual Q&A.

Draft model quality: a better draft model means higher acceptance rates and more tokens per cycle. But usually, better draft models are also larger and slower, which cuts into the speedup. There's a sweet spot where the draft model is good enough to maintain high acceptance rates but small enough to be much faster than the target.

Batch size: this is the most important practical consideration. Speculative decoding's benefit comes from replacing sequential target model forward passes with one parallel verification pass. But in high-throughput serving with large batch sizes, the target model's forward passes may have high MFU already, meaning that the extra verification tokens can slow down the

forward passes. Adding a draft model on top of a high-MFU pipeline can drive an increase in TPOT, offsetting gains from speculative decoding, and in the extreme, it can actually hurt throughput. Speculative decoding shines in low-batch-size, latency-sensitive scenarios.

Multi-token prediction

Speculative decoding uses a separate draft model to propose candidates. **Multi-token prediction (MTP)** methods take a different approach to speculative decoding: they generate multiple future tokens using the base model itself, without any auxiliary model.

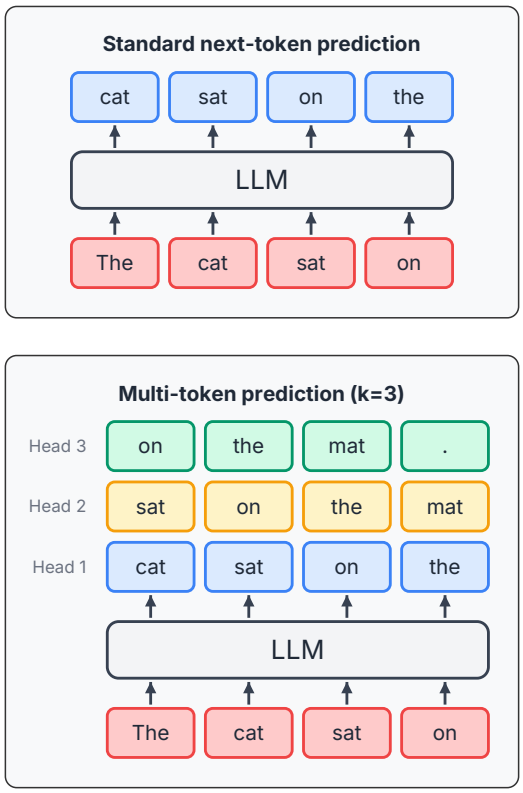


Figure 6.7.: Multi-token prediction with $k = 3$ predictions for each token position

Multi-token prediction (Gloeckle et al. 2024) modifies the model architecture to predict multiple future tokens at each position. In addition to the standard next-token prediction head, the model has auxiliary heads that predict the token 2, 3, or k steps ahead. During training, all heads are trained simultaneously with a shared loss. A comparison of standard decoding and multi-token prediction is shown in Figure 6.7.

At inference time, the auxiliary heads generate candidate tokens for positions beyond the next token. These candidates are then verified by treating them as draft tokens in a speculative decoding framework. When the candidates are accepted, the model has effectively generated multiple tokens from a single forward pass through the main body of the network.

DeepSeek-V3 uses MTP with two prediction heads, and Meta's Llama architecture has explored similar approaches. The key advantage of MTP over external draft models (covered above) is that the draft candidates come from the model itself, so they tend to have high acceptance rates. One cost is a modest increase in model size, which is usually negligible. The primary barrier to widespread adoption is the added training complexity from the auxiliary heads.

Medusa

Medusa (Cai et al. 2024) allows you to do MTP with a pre-trained base model that wasn't trained for MTP. It adds multiple lightweight decoding heads to an ordinary base model. The base model has the standard one head that predicts the next token. Medusa adds k additional heads, where head i predicts the token $i+1$ positions ahead. These heads are small (typically just a single linear layer or a small MLP) and can be inexpensively trained on top of the frozen base model.

At inference time, all $k+1$ heads produce predictions simultaneously from the same hidden state. The candidates from all heads are organized into a tree structure and verified using the tree-based attention approach described earlier. The longest accepted path through the tree gives the output.

Medusa has a practical advantage over standard speculative decoding with a separate draft model, since there's no need to run a separate model. The additional compute for the lightweight heads is small compared to the main model's forward pass. The disadvantage is that the heads need to be trained for each specific base model, adding a fine-tuning process before the model can be used.

6.6. Parallel Decoding

Lookahead decoding

Lookahead decoding (Fu et al. 2024) reformulates autoregressive generation as a fixed-point problem. Instead of generating tokens one at a time, it maintains a window of $\mathbf{W}$ future token positions, initialized with guesses. Then it iteratively refines all positions in parallel using **Jacobi iteration**: at each step, every position computes what its token should be given the current values of all other positions. When the sequence converges — that is, when a contiguous run of positions doesn't change between iterations — those tokens are accepted.

The intuition is that many tokens in a sequence are relatively predictable given their context. Common phrases, syntactic patterns, and function words often converge quickly. The tokens that take many iterations to converge are the "surprising" ones — the genuinely creative or informative tokens.

Lookahead decoding requires no draft model, no extra parameters, and no changes to the base model. The tradeoff is that each Jacobi iteration involves a forward pass over $\mathbf{W}$ token

positions, which is more expensive than a single-token decode step. The method wins when enough tokens converge quickly to offset the cost of processing the wider window.

Tree-based verification as a unifying framework

Looking across the techniques in these last two sections, a common pattern emerges. Speculative decoding with a draft tree, multi-token prediction, and even lookahead decoding all follow the same high-level structure:

1. **Propose**: generate multiple candidate tokens using some cheap mechanism (draft model, auxiliary heads, Jacobi iteration, or reference text)
2. **Organize**: arrange candidates into a tree (or chain) of possible continuations
3. **Verify**: run the target model on the full tree in a single forward pass, using a custom causal mask to check all candidates in parallel
4. **Accept**: take the longest prefix (or longest path through the tree) where the target model agrees with the proposals

The main differences between techniques lie in step 1 — how candidates are generated. The verification and acceptance steps are similar across all methods. This unifying view makes it easier to reason about the tradeoffs: the quality and cost of the proposal mechanism determine the acceptance rate and overhead, while the verification step is a single forward pass whose cost depends on the total number of candidate tokens. And all this works because the one-token-at-a-time autoregressive process is not a fundamental constraint, just a convenient way to structure generation.

This unifying view also explains why these techniques interact with batch size the same way. The verification step is essentially a small prefill — it processes multiple tokens in parallel. At large batch sizes, the GPU is already busy with many requests, so the marginal cost of the extra verification tokens is higher relative to the benefit. At small batch sizes, the GPU has spare capacity, and the extra tokens come nearly for free.

The techniques in this chapter operate at the level of individual requests — optimizing attention kernels, managing KV cache memory, and breaking the sequential decoding constraint. In the next chapter, we will consider what happens when a single GPU isn't enough: how to distribute the model across multiple devices while keeping communication overhead manageable.

6.7. Further Reading

Attention kernels. Aleksa Gordić's "ELI5: FlashAttention" (Gordić 2023) builds up the FlashAttention algorithm from first principles, starting with vanilla attention and addressing its inefficiencies one by one. It's one of the most widely recommended explainers for building intuition about why tiling and IO-awareness matter. For readers who want to go deeper into

hardware-specific optimizations, Tri Dao's blog post accompanying FlashAttention-3 (Dao 2024) walks through how asynchronous memory transfers and FP8 computation work together on Hopper GPUs.

Kernel-level optimizations. NVIDIA's "Mastering LLM Techniques: Inference Optimization" (Verma and Vaidya 2023) is a broad overview covering kernel fusion, quantization, and parallelism strategies, and is one of the most-referenced introductions to the topic. For a more focused look at how CUDA graphs reduce kernel-launch overhead in LLM decode loops specifically, NVIDIA's blog post on optimizing llama.cpp (NVIDIA 2024e) walks through the implementation and reports concrete latency improvements.

KV cache quantization and compression. The chapter covers PagedAttention for memory management and touches on KV cache quantization and token eviction. For readers interested in going deeper on these topics, several papers push the frontier of KV cache compression. KIVI (Z. Liu et al. 2024) shows that keys and values have different distribution properties and should be quantized differently — per-channel for keys, per-token for values — achieving 2-bit KV cache quantization with minimal quality loss. H2O (Zhang et al. 2023) formalizes the observation that a small fraction of tokens dominate attention scores and proposes a "heavy-hitter" eviction policy that retains these tokens plus recent tokens, with theoretical guarantees on approximation quality. SnapKV (Li et al. 2024) takes a different approach, using an observation window at the end of the prompt to identify which KV positions each attention head consistently focuses on, then compressing the cache before generation begins. This is a fast-moving area; for a continuously updated survey of KV cache optimization techniques — compression, merging, quantization, budget allocation, and cross-layer sharing — the "Awesome-LLM-KV-Cache" repository (Cai 2024) is a useful index.

Prefix caching. The LMSYS blog post introducing SGLang and RadixAttention (LMSYS 2024a) is the best visual explanation of how a radix tree enables automatic, fine-grained KV cache reuse across requests. It illustrates four common sharing patterns in LLM workloads and walks through the tree operations step by step with an LRU eviction policy.

Speculative decoding. For an accessible introduction, the PyTorch blog's "A Hitchhiker's Guide to Speculative Decoding" (PyTorch Team 2024) walks through draft-verify mechanics with diagrams and practical implementation guidance. For a retrospective from the original authors on how speculative decoding has been deployed in production at Google — including AI Overviews in Search — see Leviathan et al. (2024). For a comprehensive academic treatment, Xia et al. (2024) surveys the full landscape of speculative decoding methods, organizing them by drafter selection strategy and verification approach, with comparative benchmarks.

7. Scaling Across Hardware

Everything we've covered so far in this book operates on a single GPU. That works fine for smaller models, but the largest LLMs today — with hundreds of billions of parameters or more — simply don't fit in the memory of a single device. Even when a model does fit, you may want to spread the work across multiple GPUs. This chapter is about how to do that.

The parallelism techniques used for inference are mostly the same ones developed for training. The good news is that they're easier to implement for inference, because they only have to deal with the forward pass — no activations to store for backpropagation, no gradient calculations, and no optimizer state. The core challenge here is that every time we split work across devices, we introduce communication overhead between devices, and that communication can be much slower than communication within one device.

We'll start by understanding the communication fabric, then work through each parallelism strategy, and finish by analyzing when communication becomes the bottleneck.

7.1. Interconnects and Multi-GPU Topology

Before we can reason about the cost of distributing work, we need to know how fast data moves between devices. The answer varies enormously depending on which devices are talking to each other. Figure 7.1 shows the different communication paths between GPUs in a cluster of two nodes, each with eight H100 GPUs, and the typical bandwidths for each path.

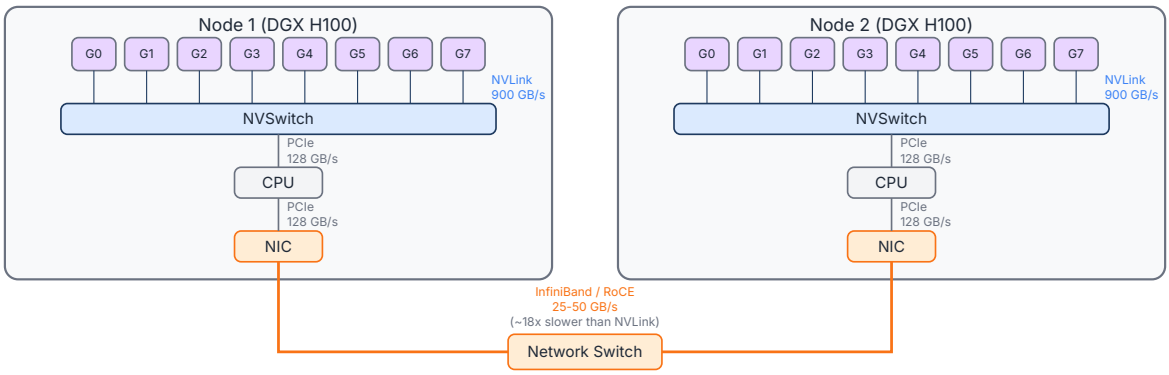


Figure 7.1.: Intra-node and inter-node connections

Intra-node: NVLink and NVSwitch

Within a single server node, NVIDIA GPUs communicate over **NVLink**, a high-bandwidth point-to-point interconnect. On the H100, each GPU has 18 NVLink connections providing 900 GB/s of bidirectional bandwidth (NVIDIA 2023). That's roughly 7x the bandwidth of PCIe Gen5.

In an 8-GPU node like the NVIDIA DGX H100, an **NVSwitch** fabric connects all GPUs in a full-mesh topology, so any GPU can communicate with any other GPU at the full NVLink bandwidth without going through intermediate hops. This is critical for collective operations like all-reduce, where every GPU needs to exchange data with every other GPU.

PCIe

PCIe (Peripheral Component Interconnect Express) is the general-purpose interconnect between the CPU and all attached devices, including GPUs. PCIe Gen5 x16 provides about 128 GB/s of bidirectional bandwidth, which we've noted is roughly 7x slower than NVLink. PCIe is used for CPU-GPU data transfers (loading input tokens, returning results) and as a fallback communication path between GPUs when NVLink isn't available.

For inference, PCIe bandwidth matters in two scenarios: KV cache offloading to CPU memory (discussed in Section 5.4), and consumer/workstation GPU setups that lack NVLink entirely. In those configurations, inter-GPU communication is bottlenecked at PCIe speeds, which severely limits the parallelism strategies that are practical.

Multi-node networking

When you need more GPUs than fit in a single node, communication moves to the network. The two dominant technologies are **InfiniBand** and **RDMA over Converged Ethernet (RoCE)**.

NVIDIA's HDR InfiniBand provides 200 Gb/s (25 GB/s) per port, and NDR doubles that to 400 Gb/s (50 GB/s) per port. With multiple ports bonded together, a node can achieve aggregate network bandwidth in the hundreds of GB/s – but this is still far below the 900 GB/s available over NVLink within the node. Network latency is also significantly higher: a few microseconds for InfiniBand versus roughly 100 nanoseconds for NVLink.

This bandwidth gap is the fundamental reason why multi-node inference is hard. Strategies that require frequent all-to-all communication, like tensor parallelism, work beautifully within a node over NVLink but become impractical across nodes.

Bandwidth comparison

Table 7.1.: Interconnect bandwidth and latency comparison for common GPU infrastructure.

Interconnect	Bandwidth (bidirectional)	Typical latency	Scope
NVLink (H100)	900 GB/s per GPU	~100 ns	Intra-node
PCIe Gen5 x16	128 GB/s	~1 s	Intra-node
InfiniBand NDR	50 GB/s per port	~1-2 s	Multi-node
InfiniBand HDR	25 GB/s per port	~1-2 s	Multi-node
RoCE v2	25-50 GB/s per port	~2-5 s	Multi-node

Beyond NVIDIA

While this book focuses on NVIDIA GPUs, it's worth noting the competitive landscape. **AMD's MI300X** offers 192 GB of HBM3 memory — significantly more than the H100's 80 GB — connected via AMD's Infinity Fabric with bandwidth comparable to NVLink. **Google's TPU v4 and v5** use a custom 3D torus interconnect (ICI) that provides high-bandwidth, low-latency communication between chips without needing an explicit switch fabric. **AWS Inferentia** and **Trainium** chips use NeuronLink for inter-chip communication. **Cerebras** takes a radically different approach with wafer-scale integration, eliminating inter-chip communication entirely within a single wafer. Each of these platforms has different communication characteristics, but the fundamental tradeoffs between computation, memory, and communication apply universally.

7.2. Parallelism Overview and Data Parallelism

There are several ways to distribute inference across multiple devices. Each strategy partitions a different dimension of the problem:

- **Data parallelism (DP)**: spreads the batch dimension across devices. The simple approach replicates the entire model on different devices, and then splits requests across replicas, with no communication between them.
- **Tensor parallelism (TP)**: spreads the model dimension across devices. This involves splitting individual layer parameters and calculations among the devices.
- **Pipeline parallelism (PP)**: assigns different layers to different devices.
- **Expert parallelism (EP)**: assigns different MoE experts to different devices.
- **Context/sequence parallelism (CP/SP)**: spreads the sequence dimension across devices. Care is required to maintain dependencies between tokens.

In practice, modern frameworks support, and production systems combine, multiple parallelism strategies. A common pattern is TP within a node (where NVLink provides the bandwidth for frequent all-reduce operations) and PP across nodes (where only point-to-point activations need to cross the network).

Data Parallelism

Simple **Data parallelism** is the easiest approach: place a complete copy of the model on each GPU and route different requests to different replicas. There's no communication between replicas during inference. Since there are no data dependencies between different requests, each replica processes its requests independently.

Without any communication overhead, DP scales throughput linearly with the number of replicas by increasing concurrency. If one GPU can serve 50 tokens per second, eight GPUs can serve 400 tokens per second. However, DP does nothing for single-request latency. TTFT, TPOT, MFU, and MBU remain unchanged as throughput increases. When running DP as the only parallelism strategy, it requires each device to have enough memory to hold the full model. For a 70B parameter model in FP16, that's 140 GB of weights, already beyond the capacity of a single H100 with 80 GB of HBM3.

When the model fits on a single device, DP is the go-to strategy for scaling throughput. When it doesn't, we need the techniques that follow.

7.3. Tensor Parallelism (TP)

Tensor parallelism splits data tensors and weight matrices across devices, so each GPU computes a slice of each layer. This is the most communication-intensive parallelism strategy, but it directly reduces per-device memory and enables serving models that don't fit on a single GPU.

Column-parallel and row-parallel linear layers

The core idea is to partition each linear layer's weight matrix along either its columns or its rows. For the following, assume we have $\mathbf{T}$ GPUs and a weight matrix W of shape $(d_{\mathbf{in}}, d_{\mathbf{out}})$. We will ignore the batch and sequence dimensions for simplicity, but the same principles apply when those are included. By splitting W into $\mathbf{T}$ parts, we can distribute the storage requirement and computations involving W across the GPUs. How we split W determines the communication patterns.

In a **column-parallel** linear layer, the weight matrix W is split along the output dimension into $\mathbf{T}$ parts, each with $d_{\text{out}}/\mathbf{T}$ columns. Each GPU holds a subset of the weights W_i with shape $(d_{\mathbf{in}}, d_{\mathbf{out}}/\mathbf{T})$. Each GPU also maintains a replica of the residual stream and uses the full input activation x of shape $(d_{\mathbf{in}})$ to compute xW_i locally, producing a partial output slice of shape $(d_{\mathbf{out}}/\mathbf{T})$. No communication is needed yet — each GPU has an independent slice of the result.

Figure 7.2 shows an example of a column-parallel linear layer with 2-way tensor parallelism, with $\mathbf{T} = 2$. The darker shaded portion of the weight matrix W_{Up} shows the half that is stored on each GPU. When multiplied by the full input x, each GPU produces a half-sized output, represented by the darker shaded portion of the output tensor.

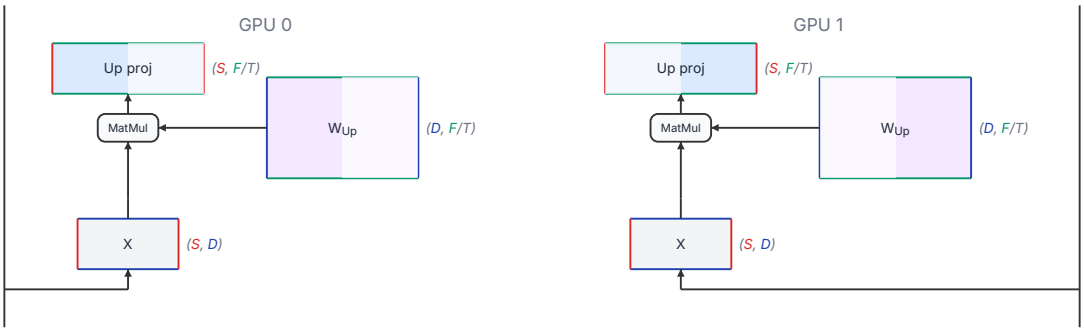


Figure 7.2.: Column-parallel linear layer calculation for 2-way tensor parallelism with a W_{Up} matrix. Only the darker shaded portions of weight and data tensors are stored on each GPU, approximately reducing the memory requirement in half.

In a **row-parallel** linear layer, the weight matrix is split along the input dimension into $\mathbf{T}$ parts, each with $d_{\text{in}}/\mathbf{T}$ rows. Each GPU holds a subset of the weights W_i with shape $(d_{\text{in}}/\mathbf{T}, d_{\text{out}})$. Once again ignoring the batch and sequence dimensions, each GPU receives only its slice of the input x_i, which is of shape $(d_{\text{in}}/\mathbf{T})$. Then, each GPU computes $x_i W_i$, producing a partial result. This partial result is the correct shape of (d_{out}), but it's only a fraction of the correct value because it was computed with only part of the input. To get the correct output, we need an **all-reduce** operation to sum the partial results from all $\mathbf{T}$ GPUs.

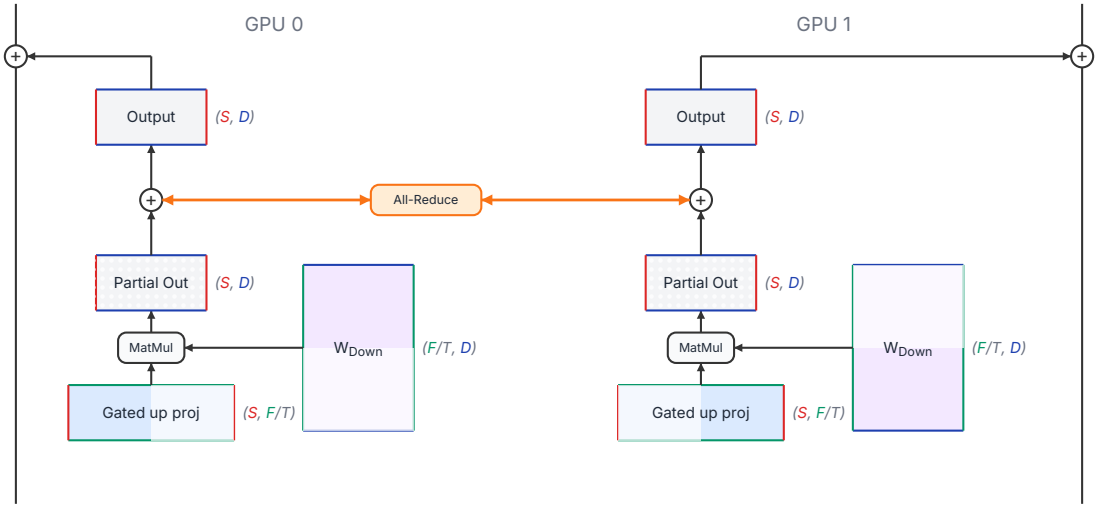


Figure 7.3.: Row-parallel linear layer calculation for 2-way tensor parallelism with a W_{Down} matrix. Only the darker shaded portions of weight and data tensors are stored on each GPU, approximately reducing the memory requirement in half.

Figure 7.3 shows an example of a row-parallel linear layer with tensor parallelism $\mathbf{T} = 2$. The darker shaded portion of the weight matrix W_{Down} shows the half that is stored on each GPU. When multiplied by the slice of the input called "Gated up proj," each GPU produces a partial output represented by the dotted and dashed tensor. To get the full output, we need to sum the partial outputs from both GPUs using an all-reduce operation, which is a collective

communication primitive that efficiently sums tensors across multiple devices. When the FFN output is summed with the residual stream, each GPU maintains its own local copy of the full residual stream, so no further communication is needed for the residual connection.

The standard pattern with tensor parallelism is to pair column-parallel and row-parallel layers so that only one all-reduce is needed per pair. In a transformer block:

1. The MLP up-projection and the QKV projection use column-parallel splits
2. The MLP down-projection and the attention output projection use row-parallel splits
3. An all-reduce follows each row-parallel layer

This means each transformer layer (with both attention and MLP) requires **two all-reduce operations**, one after the attention block and one after the MLP block.

Gated MLP under TP

We start with the simpler FFN block and show attention next. Figure 7.4 shows what 2-way tensor parallelism looks like for the gated MLP, which is now the more common form of FFN. With each GPU maintaining a local copy of the residual stream, one all-reduce is the only inter-GPU communication requred for the forward pass through this sub-layer.

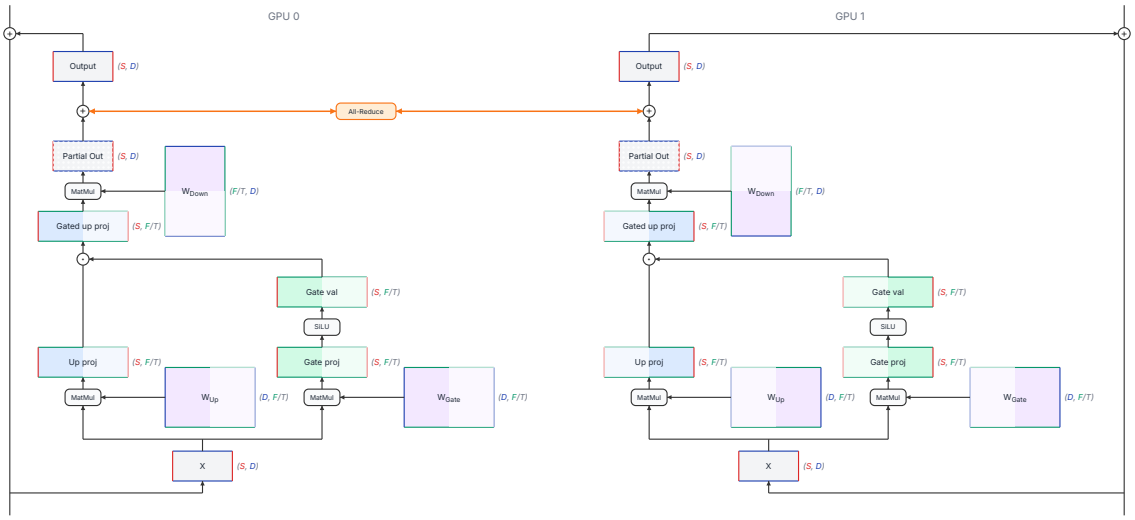


Figure 7.4.: 2-way tensor parallel gated MLP data flow. Only the darker shaded portions of weight and data tensors are stored on each GPU, approximately reducing the memory requirement in half.

Multi-head attention under TP

Attention is naturally suited to tensor parallelism because the heads are independent. With $\mathbf{H}$ attention heads split across $\mathbf{T}$ GPUs, each GPU handles $\mathbf{H}/\mathbf{T}$ heads. The Q, K, and V projection matrices are split column-parallel by head, each GPU computes attention for its

local heads, and the output projection is row-parallel with an all-reduce to combine results. Once again, one all-reduce is needed for the forward pass, and each GPU maintains its own local copy of the residual stream. This effect is depicted in Figure 7.5 with 2-way tensor parallelism for the attention layer.

The **KV cache** is also sharded: each GPU only stores the K and V tensors for its local heads. With GQA or MQA (discussed in Section 4.4), there are fewer KV heads than query heads, so the KV cache per device is even smaller. For example, with 8 KV head groups split across 8 GPUs with TP=8, each GPU holds just one KV head group — the minimum possible.

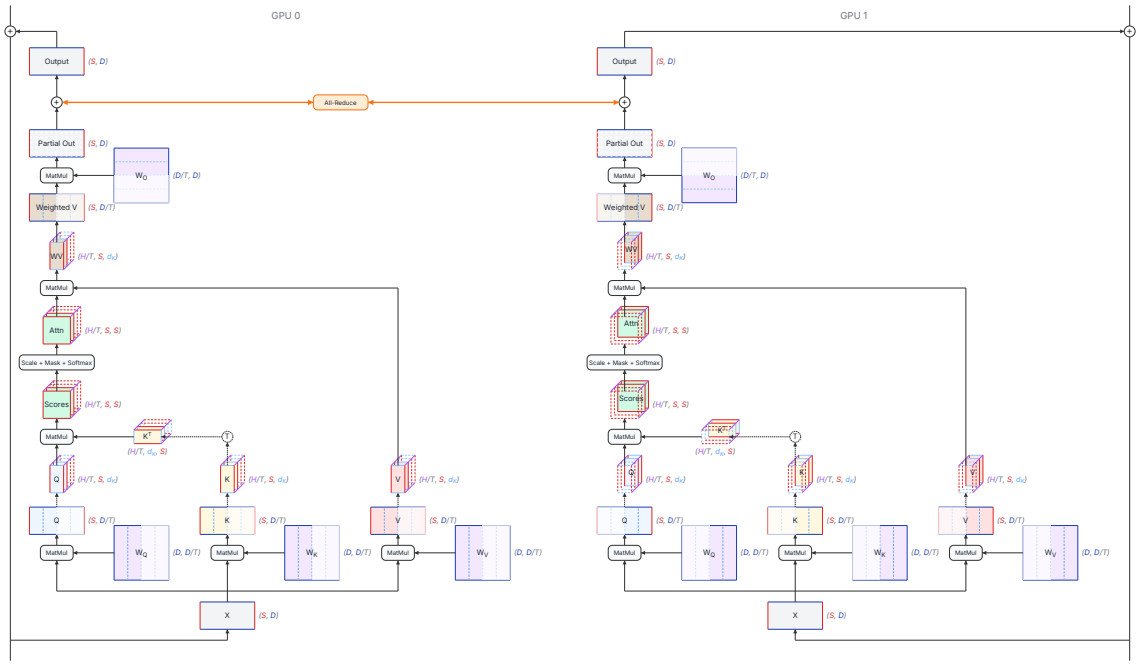


Figure 7.5.: 2-way tensor parallel attention data flow. Only the darker shaded portions of weight and data tensors are stored on each GPU. In addition, only half of the KV cache heads (not shown) are stored on each GPU. This cuts the memory requirement in half.

Communication cost

The all-reduce operation sums tensors across all **T** GPUs. Using the ring all-reduce algorithm, the communication volume per all-reduce is approximately:

$$V_{\text{all-reduce}} = 2 \times \frac{T-1}{T} \times D \times \text{bytes\_per\_element}$$

where the factor of 2 accounts for the reduce-scatter and all-gather phases. With two all-reduces per transformer layer and **L** layers, the total communication per forward pass is $2 \times$ **L** all-reduce operations.

For a 70B model with **D** = 8192 and TP=8 on H100 NVLink (900 GB/s), each all-reduce for a single-token decode step moves roughly 2 × 8192 × 2 bytes 32 KB. That completes in well under a microsecond, which is small but adds directly to TPOT. For prefill with thousands of tokens, or with larger batch sizes, the communication volume scales linearly with sequence length and batch size, and it can start to show up in TTFT.

The key insight is that TP requires **high-bandwidth, low-latency** communication every layer, making it very slow across nodes where network bandwidth is 10-30x lower than NVLink.

7.4. Pipeline Parallelism (PP)

Pipeline parallelism takes a different approach: instead of splitting each layer across devices, it assigns entire layers to different devices. If you have 80 transformer layers and 4 GPUs, each GPU gets 20 consecutive layers. This is depicted in Figure 7.6.

How it works

The forward pass flows through the pipeline stages sequentially. GPU 0 processes the input through layers 0-19 and sends the activation tensor to GPU 1, which runs layers 20-39, and so on. The communication pattern is simple point-to-point transfers between adjacent stages — no all-reduce needed.

With **P** pipeline stages, the activation tensor passed between stages has shape **(S, D)**, and it only happens **P** – 1 times, resulting in a much smaller volume of communication (so less TPOT overhead) than the 2 × **L** all-reduce operations in TP. This low communication volume makes PP better-suited for **multi-node** deployment, where network bandwidth is more limited.

The pipeline bubble

The downside of PP is the **pipeline bubble**. When processing a single request, only one stage is active at a time. The other stages sit idle, waiting for their turn. With **P** pipeline stages, the bubble fraction is roughly $(P-1)/P$. With 4 stages, 75% of the hardware is idle at any given moment.

Micro-batching is the standard solution. Instead of sending the entire batch through the pipeline, break it into smaller micro-batches. While Stage 1 processes micro-batch 2, Stage 0 can start processing micro-batch 3. With enough micro-batches in flight, all stages stay busy and the bubble shrinks.

For inference, the bubble problem is less severe than in training for two reasons. First, during decode, each step generates just one token per request, so the pipeline latency per step is short. Second, with continuous batching (Section 5.1), there are usually many concurrent requests to keep the pipeline full.

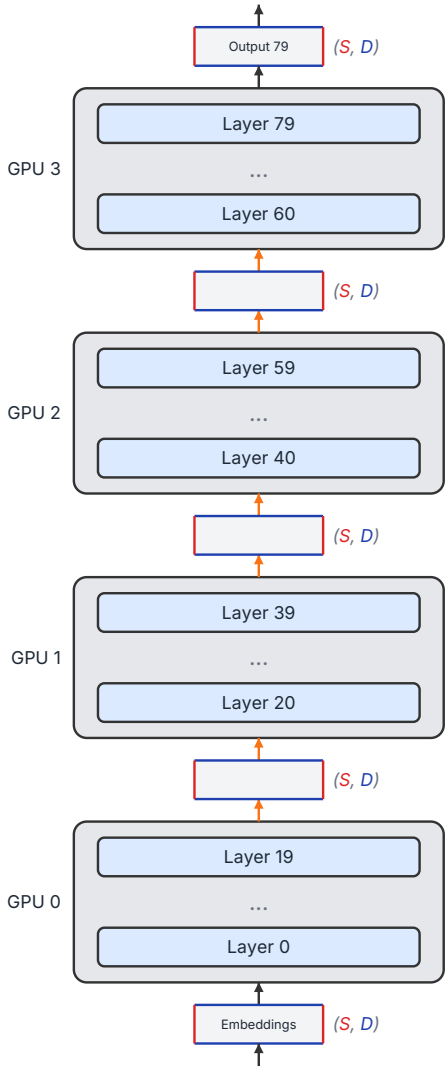


Figure 7.6.: Pipeline parallelism of an 80-layer model across 4 GPUs. There is minimal communication between GPUs.

However, PP does add latency to each decode step, since the activation must travel through all stages sequentially. With **P** stages, the minimum TPOT for a single forward pass is **P** times the per-stage compute time, plus the communication time between stages. For latency-sensitive applications, this overhead may be noticeable.

7.5. Expert Parallelism (EP)

Mixture-of-Experts (MoE) models, like Mixtral, DeepSeek-V2, and many recent large models, replace the dense MLP block with a set of expert MLPs, where a router selects a small number of experts for each token. This significantly reduces per-token compute, but it still requires the full set of expert weights to be in memory.

Expert parallelism assigns different experts to different GPUs. With 64 experts and 8 GPUs, each GPU holds 8 experts. When a token is routed to an expert on another GPU, the token's activation must be sent to that GPU and the result sent back.

All-to-all communication

Figure 7.7 shows the data flow for expert parallelism. The communication pattern for EP is **all-to-all**: every GPU potentially needs to send tokens to every other GPU and receive results back. Each MoE layer requires two all-to-all operations — one to dispatch tokens to the correct experts, and one to collect results.

The communication volume depends on the routing decisions, but in expectation, with **T** total GPUs, each GPU sends $(T-1)/T$ of its tokens to other GPUs (since only $1/T$ of experts are local). For a model with many MoE layers, this all-to-all traffic adds up.

Load balancing

A practical challenge with EP is **load imbalance**. A balanced distribution allows multiple devices to parallelize the work, reducing TPOT. If the router sends most tokens to a few popular experts, the GPUs holding those experts become bottlenecks while others sit idle. Training-time auxiliary losses encourage balanced routing, but inference workloads may still exhibit skewed expert utilization. Some systems address this by placing popular experts on multiple GPUs (expert replication) or by capping the number of tokens each expert processes and dropping overflow tokens.

Advanced routing strategies for EP also try to cap the number of devices hosting experts that are frequently co-activated, reducing the all-to-all communication. This is accomplished during training using techniques such as additional routing terms in the loss function (A. Liu et al. 2024a) and learnable bias weights in the routing network (A. Liu et al. 2024b).

EP pairs naturally with TP and PP. A common configuration for large MoE models is TP within a node for the attention layers and EP across nodes for the expert layers, since the

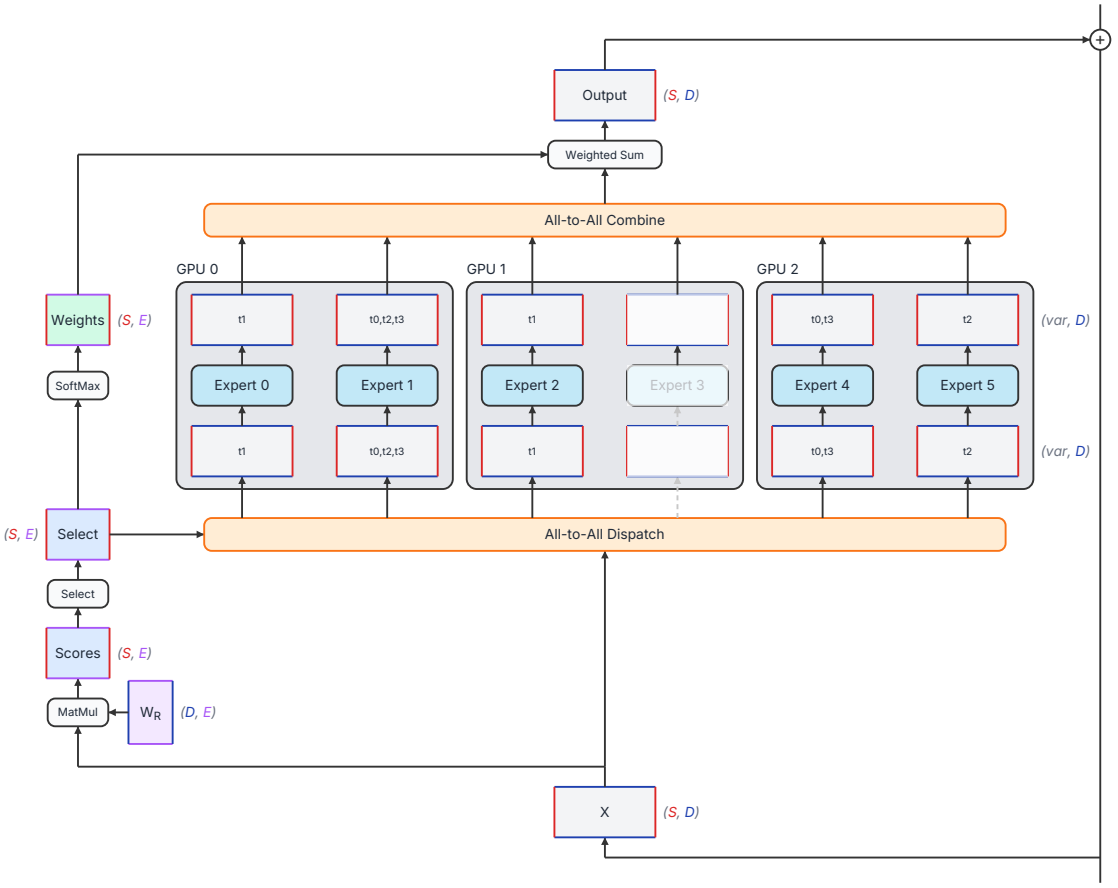


Figure 7.7.: Expert parallel MLP data flow. All-to-all communication is required between GPUs for the dispatch and the collection of results.

all-to-all communication pattern of EP is more tolerant of network latency than the all-reduce pattern of TP.

7.6. Context / Sequence Parallelism (CP/SP)

The parallelism strategies we've covered so far partition the model. **Context parallelism** (also called **sequence parallelism**) partitions the input sequence, splitting a long sequence of tokens across multiple GPUs so that each GPU processes a subsequence.

This is primarily useful during **prefill** of long contexts. If you're processing a 128K-token prompt, the attention computation is $O(S^2)$ and the KV cache for that single request may not fit on one GPU. Splitting the sequence across 4 GPUs gives each one a 32K-token chunk to process.

Ring attention

Ring self-attention (Li et al. 2021) arranges the GPUs in a logical ring. Each GPU starts with a local chunk of queries and an initial chunk of keys and values. Through a series of ring-pass steps, the K/V chunks rotate around the ring, and each GPU accumulates partial attention scores from all chunks. After **T** steps (where **T** is the number of GPUs), every GPU has computed attention over the full sequence. This pattern is depicted in Figure 7.8 for **T** = 4 GPUs.

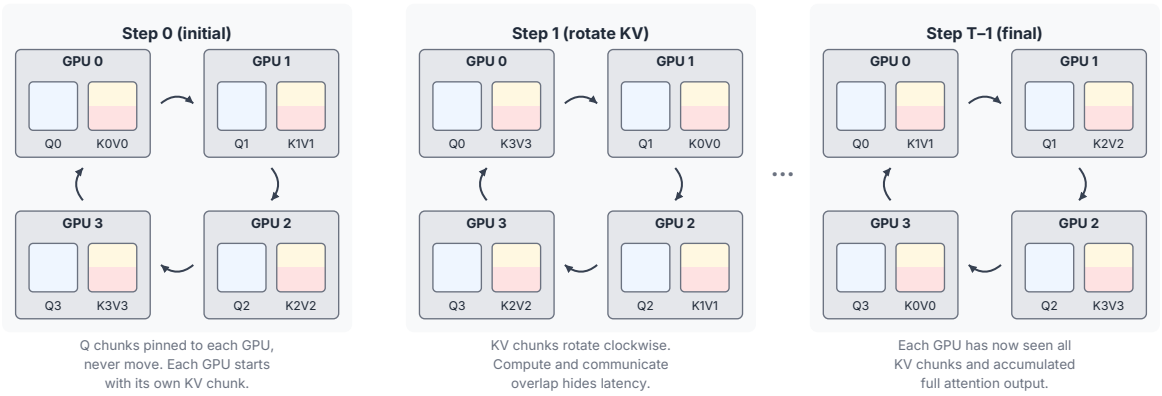


Figure 7.8.: Ring attention calculation

The ring communication pattern overlaps neatly with computation: while a GPU computes attention with the current K/V chunk, it simultaneously sends that chunk to its neighbor and receives the next chunk. With enough computation per chunk, the communication is fully hidden.

DistFlashAttn (D. Li et al. 2023) combines ring attention with FlashAttention's IO-aware tiling (see Section 6.1), getting both the distributed memory benefit and the HBM traffic reduction of FlashAttention.

NVIDIA sequence parallelism

NVIDIA's variant of **sequence parallelism** is designed to work alongside tensor parallelism. In a TP-sharded model, operations that are not parallelized by TP – like LayerNorm and dropout – are replicated across all GPUs, which wastes memory. NVIDIA SP partitions the sequence dimension for these operations, so each GPU handles a fraction of the sequence during LayerNorm and other non-TP layers, then all-gathers the full sequence before the TP-parallel matrix multiplications.

DeepSpeed Ulysses

DeepSpeed Ulysses (Jacobs et al. 2023) leverages the multi-head nature of attention. Instead of passing K/V chunks around a ring, it uses all-to-all communication to transpose the Q, K, and V tensors from a sequence-partitioned layout to a head-partitioned layout. Each GPU then computes full-sequence attention for its subset of heads, equivalent to tensor parallelism's head-based split, but derived from a sequence-partitioned starting point. After attention, another all-to-all transposes back.

The advantage of this approach is that each GPU computes standard FlashAttention on its local heads without the complexity of incremental partial-attention accumulation. The cost is two all-to-all operations per attention layer.

FlashDecoding

FlashDecoding is worth mentioning here even though it operates within a single GPU. It parallelizes the attention computation across the KV sequence dimension, splitting the cached keys and values across multiple thread blocks that compute partial attention in parallel, then reducing the results. This addresses the problem that during decode, the single query token produces very little parallelism across the head dimension, leaving most of the GPU idle. FlashDecoding gives the GPU more work to do in parallel by splitting the "long" dimension, which is the cached sequence.

When sequence parallelism helps

Context parallelism is most effective for **long-context prefill**, where the sequence length creates both a compute and memory bottleneck. For short-context decode, the sequence dimension is short and there's little benefit to splitting it further. The overhead of the extra communication typically outweighs any gains from utilizing multiple devices.

7.7. Communication Cost Analysis

With all the parallelism strategies laid out, let's compare their communication patterns and costs.

Table 7.2.: Communication patterns and costs for each parallelism strategy. **T** = number of devices, **D** = model dimension, **B** = batch size in tokens, **S** = sequence length.

Strategy	Collective op	Ops per layer	Communication volume per op	Best interconnect
TP	All-reduce	2	$2 \times \frac{T-1}{T} \times D \times B$ bytes	NVLink
PP	Point-to-point	1	$D \times B$ bytes (activation)	Network OK
EP	All-to-all	2	Routing-dependent; $\sim\frac{T-1}{T}$ of tokens	NVLink or network
CP/SP	Ring-pass or all-to-all	**T** or 2	$O((S/T) \times D)$ per step	NVLink

The key observation is that **TP is the most communication-hungry** strategy, requiring two all-reduce operations per layer. This is why TP is confined to within a single node over NVLink. PP has the lightest communication — just a single activation tensor between adjacent stages — making it the natural choice for spanning nodes.

Combined TP + PP

The most common multi-node configuration is **TP within a node, PP across nodes**. For example, serving a 70B model on 2 nodes of 8 GPUs each: use TP=8 within each node (splitting layers across all 8 GPUs via NVLink), and PP=2 across the two nodes (first half of layers on node 1, second half on node 2). The frequent all-reduces stay on the fast NVLink fabric, and only the occasional activation transfer crosses the network.

Combining EP with other strategies

Combining EP with other strategies works slightly differently because EP only uses a subset of a layer's experts for each token. DP, TP, and PP always require all devices to fully participate for each token. The averaging of work across devices with EP is more complex, and averages better with more tokens that smooth out the routing decisions. Whereas simple DP can be performed with replicas that don't interact, DP with EP works better if the replicas share a single load-balancing pool for the experts. It's also difficult to manage multiple communication patterns simultaneously. This is the reason why vLLM reuses the DP / TP communication mesh for EP, rather than introducing a separate communication mesh just for the experts.

Communication-bound regime

When does communication become the bottleneck? During decode with small batch sizes, the per-layer computation is tiny (a few matrix-vector multiplications), and each layer's all-reduce, even if it's only transferring tens of KB, has a latency floor that can dominate. The all-reduce latency depends on both bandwidth and the number of sequential steps in the algorithm.

A useful rule of thumb: if the communication time per layer exceeds the compute time per layer, you've over-parallelized. Adding more TP devices beyond this point will increase latency rather than decrease it, because you're spending more time synchronizing than computing. This is why TP=8 is a common sweet spot on 8-GPU nodes. It matches the NVLink connectivity, and going to TP=16 across nodes rarely makes sense. Note also that TP shards the KV cache efficiently up to the number of KV heads; beyond that, the KV cache must be replicated across devices, which is less beneficial.

7.8. Multi-Node Inference

When a model is too large for a single node, or when you need more throughput than one node can provide, you're in multi-node territory. This introduces the network as a hard constraint on performance.

Network latency as a floor

During decode, each token generation requires a full forward pass through the model. If that forward pass spans two nodes connected by InfiniBand, each pipeline stage boundary (or TP all-reduce, if you're running TP across nodes) adds network latency. InfiniBand NDR round-trip latency is roughly 2-5 s, compared to ~100 ns for NVLink. This latency is per layer for TP or per stage for PP, and it accumulates across the forward pass.

For a model using PP=2 across nodes, the activation transfer between stages adds a few microseconds per decode step. That's usually acceptable. But for TP across nodes, where you need two all-reduces per layer across potentially 80+ layers, the cumulative latency can add milliseconds to each token, directly increasing TPOT.

Practical deployment patterns

In practice, multi-node inference for large models follows a few common patterns:

- **DP across nodes**: when the model fits on a single node, the simplest scale-out is running independent replicas. No cross-node communication during inference at all.
- **TP within node, PP across nodes**: the standard approach, as discussed above. Minimizes cross-node communication.
- **TP within node, EP across nodes for MoE**: expert parallelism's all-to-all pattern is more latency-tolerant than TP's all-reduce, making it more viable across nodes.

> **Note**
>
> The distinction between what's practical within a node versus across nodes is driven almost entirely by the bandwidth gap in Table 7.1. NVLink provides 900 GB/s; the network provides 50 GB/s. That 18x gap is why the parallelism strategy changes at the node boundary.

Memory and KV cache considerations

One often-overlooked benefit of distributing a model across more GPUs is that it frees up memory for the **KV cache**. If a 70B FP16 model uses 140 GB across 8 GPUs (17.5 GB per GPU), that leaves about 62 GB per GPU on H100 for KV cache and activations. With quantized weights (e.g., INT4), the model only consumes ~35 GB total, leaving even more room. More KV cache memory means you can serve larger batches or longer sequences, directly improving throughput.

This is a genuine advantage of tensor parallelism beyond just enabling large models: by spreading weight memory across more devices, you increase the aggregate memory available for serving. Combined with KV cache sharding from GQA/MQA models (Section 4.4) and PagedAttention (Section 6.3), multi-GPU setups can sustain higher concurrency rates than the raw parameter count might suggest.

The next chapter (Chapter 8) will share how all of these techniques come together in production serving systems, including how frameworks like vLLM and TensorRT-LLM configure parallelism strategies automatically based on available hardware.

7.9. Further Reading

Overviews of parallelism strategies. Lilian Weng's "How to Train Really Large Models on Many GPUs?" (Weng 2021) is the most widely recommended introduction to all the parallelism strategies covered in this chapter. It covers data, tensor, pipeline, and expert parallelism with clear diagrams. The writing is from a training perspective, but the forward-pass mechanics are identical for inference. The Hugging Face documentation on parallelism methods (Hugging Face 2024a) is a good practical companion, with a decision table for choosing a strategy based on your hardware setup.

The hardware perspective. "How to Scale Your Model" (Austin et al. 2025), an online book from the JAX team at Google DeepMind, approaches parallelism from the hardware up. It starts with the TPU architecture, works through the roofline model, and then explains each parallelism strategy in terms of the communication costs they impose on different interconnects. Much of the focus is on training, but there is a chapter dedicated to inference, and all of the content is rich with details.

Scaling inference specifically. Pope et al. (2022) is one of the best analyses of how parallelism strategies interact with the unique characteristics of inference — the distinction

between prefill and decode, the impact of batch size on communication-to-compute ratio, and the tradeoffs between latency and throughput. For a more recent and practical treatment, Meta's engineering blog post on scaling LLM inference (Zhao et al. 2025) covers how they combine tensor, context, and expert parallelism in production, including their work on context parallelism for million-token sequences.

Mixture of Experts. The Hugging Face blog post "Mixture of Experts Explained" (Sanseviero et al. 2023) provides an accessible introduction to MoE architectures, including the routing mechanism, load balancing challenges, and how expert parallelism fits into the picture. For the systems-level challenges of serving MoE models at scale, the AlpaServe paper (Z. Li et al. 2023) explores how to automatically determine parallelism strategies across heterogeneous clusters, balancing latency constraints against hardware utilization.

8. Production LLM Serving Systems

Now that we understand LLM inference and many of the optimization techniques used, we can discuss some of the popular model serving frameworks and the implementation choices they have made for serving LLMs in the real world. Some of these systems are in production across many continents, scaling to thousands of concurrent users or more. This is not a usage tutorial for any particular framework. The goal is to understand which optimization combinations each system makes, why those choices matter, and what operational concerns arise once you move from benchmarking a single model on a single GPU to serving real traffic.

8.1. Serving Frameworks

The open-source LLM serving landscape has converged around a handful of frameworks, each with a different philosophy about what to optimize for. All of them implement continuous batching (Section 5.1), support tensor parallelism (Section 7.3), and integrate FlashAttention (Section 6.1). Where they differ is in their core innovation and the tradeoffs they make.

vLLM

vLLM is built around PagedAttention (Kwon et al. 2023a), the virtual-memory-inspired KV cache management system we covered in Section 6.3. By breaking the KV cache into fixed-size pages that don't need to be physically contiguous, vLLM nearly eliminates the memory fragmentation that plagued earlier systems. This is what allows it to pack more concurrent requests into the same GPU memory budget.

Beyond PagedAttention, vLLM implements continuous batching, chunked prefill (Section 5.2), prefix caching (Section 6.4), speculative decoding (Section 6.5), and multi-LoRA serving. It supports NVIDIA, AMD, and other hardware backends, making it one of the most broadly portable options. vLLM has become the default choice for many deployments, largely because of its active open-source community and broad model support.

SGLang

SGLang (Zheng et al. 2023) innovates in two directions. First, its **RadixAttention** system extends prefix caching beyond simple exact-match lookups. It organizes the KV cache as a radix tree keyed by token sequences, enabling automatic, fine-grained prefix sharing across

requests (Section 6.4). When many requests share system prompts, few-shot examples, or RAG context, this can dramatically reduce redundant prefill computation.

Second, SGLang provides a programming model for **multi-call LLM programs** — workflows where a single user interaction involves multiple LLM calls with branching logic, structured output constraints, or iterative refinement. By co-designing the serving engine with this programming model, SGLang can optimize across calls in ways that a stateless API cannot.

TensorRT-LLM

TensorRT-LLM focuses on squeezing maximum throughput from NVIDIA hardware through aggressive kernel fusion and compilation (Section 6.2). Models are compiled into optimized execution plans that fuse operations across layers, eliminate unnecessary memory traffic, and exploit NVIDIA-specific hardware features like FP8 on Hopper GPUs.

The tradeoff is narrower hardware support — TensorRT-LLM runs only on NVIDIA GPUs — and a heavier compilation step before serving begins. But on NVIDIA hardware, it often achieves the highest raw throughput, particularly for large-batch, throughput-oriented workloads. It implements the same core techniques (continuous batching, paged KV cache, tensor parallelism) but through a compiled execution path rather than an interpreted one.

Framework comparison

The following table summarizes the key differences across these three leading frameworks. All three are actively developed, so specific feature gaps tend to close over time.

Table 8.1.: Framework comparison across major open-source LLM serving systems.

Feature	vLLM	TensorRT-LLM	SGLang
Core innovation	PagedAttention	Kernel fusion / compilation	RadixAttention + structured gen
KV cache management	Paged (virtual memory)	Paged	Radix tree
Continuous batching	Yes	Yes	Yes
Speculative decoding	Yes	Yes	Yes
Multi-LoRA	Yes	Yes	Yes
Hardware support	NVIDIA, AMD, others	NVIDIA only	NVIDIA, AMD
Best suited for	General-purpose serving	Max throughput on NVIDIA	Prefix-heavy / multi-call

Other frameworks

Several other serving frameworks are worth knowing about, even though vLLM, SGLang, and TensorRT-LLM have emerged as the dominant choices.

Text Generation Inference (TGI) is Hugging Face's serving engine. It saw wide adoption in 2023–2024 thanks to tight integration with the Hugging Face model ecosystem, and it implements continuous batching, paged KV cache, and speculative decoding. However, TGI has entered maintenance mode, and Hugging Face now offers vLLM and TensorRT-LLM as alternative backends within TGI — a signal that even its maintainers view the original engine as no longer competitive on raw performance.

LMDeploy, developed by the InternLM team at the Shanghai AI Lab, focuses on high throughput with quantized models. Its TurboMind backend delivers strong tokens-per-second numbers, particularly for 4-bit quantized models from the Llama family. It is less well-known outside of China but appears regularly in inference benchmarks.

DeepSpeed-Inference (Aminabadi et al. 2022) is the inference-focused sibling of the widely used DeepSpeed training library. Its ZeRO-Inference system distributes model weights across GPUs and even to NVMe storage, enabling inference on models that don't fit in aggregate GPU memory. DeepSpeed-Inference also implements custom CUDA kernels for fused operations, but its primary strength is multi-GPU and multi-node orchestration rather than single-device kernel optimization. As GPU memory capacities have grown and tensor parallelism support has improved across all frameworks, the niche for NVMe offloading has narrowed, and DeepSpeed-Inference sees less adoption as a standalone serving system than the three leading frameworks.

> **Note**
>
> This chapter focuses on GPU-based serving at scale. Closed-source serving systems from cloud providers are out of scope, and so is local and edge deployment, which operates under fundamentally different constraints — limited memory bandwidth, no HBM, CPU-only or unified-memory architectures. While we won't discuss them here, the top inference engines for local deployment are:
>
> - **llama.cpp** — the C/C++ inference engine that pioneered running LLMs on consumer hardware via aggressive quantization. Many of the tools below use llama.cpp under the hood.
> - **Ollama** — wraps llama.cpp in a Docker-like interface for pulling and running models with a single command. The fastest path from zero to a locally running LLM.
> - **Apple MLX** — Apple's framework optimized for the unified-memory architecture of Apple Silicon, achieving strong throughput on MacBook and Mac hardware.
> - **LM Studio** — a desktop application built on llama.cpp that provides a GUI for downloading, configuring, and chatting with local models.

8.2. Multi-LoRA Serving

In many production deployments, you don't just serve one model — you serve many fine-tuned variants of the same base model. **LoRA** (Hu et al. 2021) is a popular fine-tuning technique that adds small, trainable matrices to a frozen base model. Loading a separate full model for each customer is wildly impractical because of the loading time impact on TTFT and the memory overhead impact on concurrency. **Multi-LoRA serving** solves this by keeping a single copy of the base model weights in GPU memory and hot-swapping lightweight LoRA adapters at request time.

A LoRA adapter modifies the base model by adding low-rank update matrices to specific layers — typically the attention projections. These adapters are small, often just tens of megabytes compared to the base model's tens of gigabytes. This size difference is what makes the whole approach work: you can keep the base model loaded and swap adapters in and out without the memory cost of maintaining multiple full model copies.

The serving framework needs to handle three things for multi-LoRA to work well:

1. **Adapter routing**: Each incoming request specifies which adapter to use. The system must route the request to the correct adapter and apply it during the forward pass.
2. **Adapter memory management**: Popular adapters can be kept resident in GPU memory. Less frequently used adapters are loaded on demand from CPU memory or storage. The system needs an eviction policy — typically LRU — to manage which adapters are resident.
3. **Batching across adapters**: Requests using different adapters can still be batched together for the base model computation. The adapter-specific computation (the low-rank matrix multiplies) is applied per-request. This means you get some of the MFU benefits of shared base weights while still serving personalized models.

Both vLLM and SGLang support multi-LoRA serving, with adapter management integrated into their scheduling and memory systems.

8.3. Scheduling and Orchestration

The scheduling strategies from Chapter 5 operate at the level of a single serving instance — one model on one set of GPUs. In production, you often need to coordinate across a fleet of instances, potentially running different models on different hardware.

Multi-model routing

Many applications route requests to models of different sizes based on the task complexity. Simple classification tasks go to a smaller, cheaper model. Complex reasoning tasks go to a larger, more capable model. This **quality-tier routing** can significantly reduce cost while maintaining quality where it matters, but it requires a routing layer that can classify incoming requests and direct them appropriately.

Load balancing across replicas

When you scale out with data parallelism — running multiple identical copies of the same model — you need a load balancer that understands LLM workload characteristics. Naive round-robin doesn't work well because requests vary enormously in cost. A request with a 10,000-token prompt and 2,000-token output takes orders of magnitude more resources than a 100-token prompt with a 50-token output. Load balancers that account for estimated request cost (based on input length and expected output length) distribute work more evenly. They can be tuned not only to improve median metrics but also to reduce tail latencies, thereby maximizing goodput.

Disaggregated prefill at fleet level

In Section 5.2, we discussed disaggregating prefill and decode to separate GPU pools (Zhong et al. 2024). At fleet scale, this becomes an orchestration problem: how many prefill instances versus decode instances should you run, and how do you transfer KV caches between them? The optimal ratio depends on your workload's input/output length distribution (Section 3.3). Prefill-heavy workloads (long prompts, short outputs) need more prefill capacity. Decode-heavy workloads need more decode capacity. Production systems typically monitor queue depths on both pools and auto-scale the ratio to optimize the TTFT and TPOT tradeoff.

Heterogeneous hardware

Large deployments often mix GPU generations — perhaps H100s for the most latency-sensitive traffic and slower A100s or older GPUs for batch workloads. The orchestration layer must route requests based on both the model's requirements and each hardware pool's capabilities and current load.

8.4. Memory Management and Preemption

We covered the metrics around preemption in Section 3.2, and the paged KV cache in Section 6.3. In production, memory management becomes one of the most critical operational concerns, because running out of KV cache space under load can cascade into system-wide degradation.

Multi-tenant memory pressure

When many concurrent requests share the same GPU, the KV cache dominates memory usage. Each active request's cache grows with every decode step, and different requests may have very different sequence lengths. If we have new requests arriving, we'd like to admit them to increase concurrency, MFU, and throughput. The serving framework must track per-request

memory consumption and make admission decisions: can we accept this new request, or will its KV cache push us over our memory budget?

PagedAttention helps by eliminating fragmentation — memory is allocated in pages and freed when requests complete. But even with perfect paging, the total KV cache demand can exceed available memory when the system is under heavy load or processing long-context requests.

Preemption policies

When memory runs out, something has to give. The system **preempts** one or more active requests, freeing their KV cache to make room for others. The preempted request either has its KV cache swapped to CPU memory (to be restored later) or is simply evicted and must recompute its KV cache from scratch when it resumes.

The choice between swap and recompute involves a tradeoff between two negative impacts on TTFT and GPU resources:

- **Swap** preserves the computation already done but requires CPU memory and PCIe bandwidth to move the KV cache. For long sequences with large KV caches, the transfer time can be significant.
- **Recompute** avoids the transfer cost but wastes the prefill work. For short sequences, recompute is often cheaper than swap.

Production systems typically use **priority-based preemption** — lower-priority requests are evicted first, and SLA-aware policies protect requests that are close to completing or that belong to higher-priority tiers.

Graceful request degradation

Under sustained overload, the goal shifts from maximizing throughput to avoiding cascading failures and preserving as much goodput as possible. **Request shedding** — rejecting new requests at the admission layer when the system is at capacity — is blunt but effective. More sophisticated approaches include dynamically reducing maximum sequence lengths, lowering batch sizes, or routing overflow traffic to a degraded service tier that uses a smaller, faster model.

8.5. Monitoring and Benchmarking

You can't optimize what you don't measure. Production serving requires continuous monitoring of the metrics we introduced in Section 3.2, with particular attention to tail latencies and resource utilization.

Key production metrics

The metrics that matter most in production go beyond averages:

Table 8.2.: Key production metrics and their diagnostic value.

Metric	What it tells you	Warning threshold
P99 TTFT	Worst-case user wait for first token	Exceeds SLA
P99 TPOT	Worst-case streaming speed	Users perceive lag
GPU utilization (MFU/MBU)	Hardware efficiency	Below 50% suggests misconfiguration
Queue depth	Request backlog	Sustained growth means under-provisioned
Preemption rate	Memory pressure frequency	Any sustained preemption suggests need for more capacity
KV cache hit rate	Prefix caching effectiveness	Low rate with shared prefixes means caching misconfigured
Goodput	Useful throughput within SLA	Gap between throughput and goodput means SLA violations

Percentile latencies (P50, P90, P99) are essential because averages hide tail behavior. A system can have a perfectly acceptable P50 TTFT of 200ms while its P99 is 5 seconds — meaning one in a hundred users waits 25x longer than the median user. Production SLAs are typically defined at the P99 level.

Instrumentation

Most production deployments use **Prometheus** (Prometheus Authors 2024) for metrics collection and **Grafana** (Grafana Labs 2024) for dashboarding. vLLM, SGLang, and TensorRT-LLM all expose Prometheus-compatible metrics endpoints. For distributed tracing across the full request lifecycle — from the load balancer through the serving engine to response delivery — **OpenTelemetry** (OpenTelemetry Authors 2024) provides a standard instrumentation layer.

Benchmarking

Before deploying a new configuration, you need reproducible benchmarks that reflect your actual workload. Tools like **LLMPerf** generate synthetic traffic with configurable input/output length distributions and concurrency levels (Kadous et al. 2023). The critical mistake in benchmarking is using a uniform workload (e.g., all requests with 512 input tokens and 128

output tokens) when your production traffic has high variance. Benchmark with a distribution that matches production, or you'll be surprised by how differently your system behaves under real load.

8.6. Production Failure Modes

Understanding how systems fail is just as important as understanding how they work. Several failure modes are specific to LLM serving, and recognizing them early can prevent cascading outages.

OOM cascades

This is the most dangerous failure pattern. A single unusually long request consumes a large portion of the KV cache, triggering preemption of other requests. Those preempted requests eventually resume and recompute their KV caches, creating a burst of prefill work that competes with ongoing decode steps. This increases latency for all active requests, causing SLA violations, which may trigger retries from the client, which adds more load. The feedback loop can take down an entire serving instance.

Mitigation: Set hard limits on maximum sequence length per request. Use admission control to reject requests that would push memory usage above a safe threshold. Monitor preemption rate as an early warning signal.

Head-of-line blocking

A request with a very long prompt ties up GPU compute during its prefill phase. If the system uses naive scheduling, shorter requests queue behind it and experience inflated TTFT. This is particularly damaging in mixed-workload deployments where most requests are short but occasional long-context requests arrive.

Mitigation: Chunked prefill (Section 5.2) breaks long prefills into smaller pieces that are interleaved with decode steps, preventing any single prefill from monopolizing the GPU. Priority scheduling (Section 5.4) can also help by ensuring short requests aren't starved.

Thundering herd

A burst of simultaneous requests — perhaps triggered by a traffic spike, a retry storm, or a batch job — overwhelms the KV cache allocator and scheduler. The system attempts to admit all requests at once, runs out of memory, and begins preempting aggressively, which compounds the problem.

Mitigation: Admission control with request queuing and rate limiting. Gradually ramp admitted requests rather than accepting the full burst. Backpressure signals to upstream load balancers allow the fleet to absorb traffic spikes across replicas.

Graceful degradation strategies

We have touched on some strategies for mitigating overload, but it's worth reiterating. When a system is under sustained pressure, there are several levers that can be pulled before things break:

1. **Request shedding**: reject low-priority requests at the edge, returning a "service unavailable" rather than degrading quality for everyone
2. **Dynamic batch policy adjustment**: temporarily reduce maximum batch sizes to lower memory pressure, trading throughput for stability
3. **Quality-tier routing**: redirect overflow traffic to smaller, cheaper models that can handle the load, accepting lower quality to maintain availability
4. **Speculative decoding toggle**: disable speculative decoding under memory pressure, since draft tokens consume KV cache space

The common thread across all of these strategies is that controlled, deliberate degradation is vastly preferable to uncontrolled cascading failure. A system that gracefully sheds 10% of requests under a load spike is far better than one that falls over entirely and drops 100%.

8.7. vLLM

vLLM has become the most widely deployed open-source LLM serving framework, and it's worth understanding why. The short answer is breadth: vLLM implements nearly every optimization technique we've covered in this book, from PagedAttention and chunked prefill through speculative decoding and expert parallelism. But breadth alone doesn't explain adoption — the design decisions behind how these techniques fit together matter just as much. This section walks through those decisions, focusing on what's distinctive about vLLM's implementation rather than re-explaining the underlying techniques.

The V1 architecture

vLLM's original architecture (now called V0) grew organically as new features were added. By the time it supported prefix caching, speculative decoding, chunked prefill, and multi-modal inputs, the scheduler and worker code had accumulated enough special cases that adding new features meant touching many interacting code paths. The **V1 redesign** (vLLM Team 2025e) addressed this by rethinking the core abstractions.

The most visible change is the **persistent batch**. In V0, each scheduling step constructed a new set of input tensors from scratch — assembling token IDs, position indices, and attention metadata for every request in the batch. V1 caches these tensors and applies only incremental diffs each step. If a batch of 200 requests produces one token each, V1 updates 200 entries rather than rebuilding the entire input. This sounds like a minor optimization, but the CPU overhead of tensor construction was a real bottleneck at high batch sizes, and eliminating it delivers up to 1.7x higher throughput compared to V0 (vLLM Team 2025e).

The second major change is **symmetric tensor parallelism**. In V0, the scheduler ran on rank 0 and broadcast full request state to all other TP ranks every step. V1 caches request state on every worker and transmits only diffs — new requests entering the batch and completed requests leaving it. This removes the asymmetry where rank 0 was doing substantially more work than the other ranks, and it reduces the communication volume between the scheduler and the workers.

V1 also embraces `torch.compile` as a first-class optimization path. Rather than relying entirely on hand-written custom CUDA kernels, V1 passes models through PyTorch's Inductor compiler and then applies vLLM-specific fusion passes on top (vLLM Team 2024c). The compiler handles the boilerplate optimizations — operator fusion, memory planning, kernel selection — while custom passes add inference-specific fusions like RMSNorm + quantization and SiLU + quantized linear. This is a deliberate bet on maintainability: as new GPU architectures arrive, `torch.compile` can retarget them without rewriting kernel code.

Finally, V1 uses **piecewise CUDA graphs** aligned with its compilation strategy. The computation graph is split at attention operations — which need dynamic shapes because the batch composition changes every step — and CUDA graphs capture everything between them. This gives most of the kernel launch overhead reduction of full CUDA graphs while preserving the flexibility that attention requires (vLLM Team 2025e).

Scheduling and memory management

vLLM's scheduler operates at the iteration level (Section 5.1): after every forward pass, it decides which requests continue, which new requests enter, and which running requests should be preempted. This is the same continuous batching approach used by other frameworks, but vLLM's implementation has several distinctive features.

Chunked prefill is the default scheduling mode. Long prompts are split into chunks by capping the number of new tokens processed per step. The scheduler fills each step by first allocating budget to decode requests — since each one needs only a single token — and then filling the remaining budget with pending prefill chunks. This prioritization is what keeps inter-token latency stable for requests that are already generating: decode steps are never blocked behind a large prefill.

Priority-based scheduling lets you assign integer priorities to requests, and the scheduler dynamically reorders both the waiting and running queues. If a running request has lower priority than a waiting request, the scheduler can forcefully preempt it — swapping or freeing its KV cache blocks — to make room for the higher-priority request immediately. This goes beyond the simple memory-pressure preemption described in Section 8.4: it's policy-driven preemption in service of request-level SLAs.

When preemption is necessary, vLLM uses a hybrid strategy. It attempts to swap KV cache blocks to CPU memory first, preserving the computed state. If CPU memory is insufficient, it falls back to recomputation — freeing the blocks and requeuing the request to recompute its KV cache when it's rescheduled. The choice between swap and recompute is made automatically based on available resources, not configured manually.

SLA-aware scheduling is at the RFC stage as of early 2026, but worth noting because it reflects where the project is heading. The proposal introduces SLA tiers — interactive, batch, and background — as the primary sort key for scheduling, with integer priority as a tie-breaker within a tier. SLA tier would also influence preemption order under memory pressure: background requests are evicted first, then batch, then interactive. This is a natural extension of the priority-based scheduling already in place, moving from raw integer priorities toward semantically meaningful service classes.

OOM handling in vLLM follows a graceful degradation strategy rather than crashing. When the KV cache manager cannot allocate slots for a new step, the scheduler stops admitting new requests and may preempt the lowest-priority running requests to free blocks. If the waiting queue is full or no blocks can be freed, the engine rejects incoming requests outright rather than risking a system-wide failure. KV cache blocks can also be swapped to CPU memory during preemption, effectively extending available cache capacity at the cost of PCIe transfer latency (vLLM Team 2025b).

Prefix caching is enabled by default in V1 with near-zero overhead (vLLM Team 2025e). vLLM's approach hashes each KV block using the tokens it contains plus all preceding tokens, then stores these hashes in a global table. When a new request shares a prefix with a cached request, the matching blocks are reused without recomputation. The eviction policy is LRU among blocks with a reference count of zero, with ties broken by evicting the block at the end of the longest cached prefix — an approach that produces equivalent behavior to SGLang's RadixAttention for standard full-attention models. The V1 implementation achieves less than 1% throughput overhead even at a 0% cache hit rate, thanks to constant-time eviction data structures (vLLM Team 2025e).

KV cache quantization provides an additional memory lever on top of paged allocation and prefix caching (Section 6.3). vLLM supports FP8 KV cache quantization with two granularity strategies: per-tensor quantization (a single scale factor for the entire Q, K, or V tensor) and per-attention-head quantization (a separate scale factor for each attention head) (vLLM Team 2024d). Three calibration methods are available: no calibration (scale factors default to 1.0), random token calibration performed on-the-fly during warmup, and dataset-based calibration through the companion `llm-compressor` library, which also supports the finer-grained per-head quantization (vLLM Team 2024d). FP8 KV cache roughly halves the memory footprint compared to FP16, enabling either more concurrent requests or longer context windows within the same GPU memory budget.

Attention backends and kernel optimization

vLLM doesn't commit to a single attention kernel. Instead, it auto-selects from a menu of backends based on the GPU architecture, and you can override the choice if needed.

The selection logic follows GPU compute capability (vLLM Team 2024a):

- **FlashAttention-2** is the default on Ada Lovelace (SM89) and earlier CUDA GPUs.

- **FlashAttention-3** is the default on Hopper (SM90) GPUs. It exploits Hopper-specific features — warp specialization, FP8 tensor cores, asynchronous memory operations — and is essential for V1's mixed prefill/decode batching where batch composition is highly dynamic.
- **FlashAttention-4** is the default on Blackwell (SM100+) GPUs.
- **FlashInfer** provides a unified API that abstracts over multiple kernel implementations and JIT-compiles CUDA kernels for the target architecture. It serves as the primary backend on NVIDIA HGX B200 systems (vLLM Team 2024a).
- **Triton attention** is implemented entirely in Triton (Tillet et al. 2019) and carries no external kernel dependencies (vLLM Team 2026). The same source code runs on NVIDIA, AMD, and Intel GPUs, making it the cross-platform fallback when FlashAttention or FlashInfer are unavailable.

Beyond attention kernel selection, vLLM applies **kernel fusion** through custom `torch.compile` Inductor passes that rewrite the computation graph rather than modifying model code (vLLM Team 2024c). The specific fusions include:

- **Attention output quantization**: eliminates a full-precision memory round-trip by quantizing the attention output in-place
- **Activation fusion**: fuses SiLU with a quantized linear layer, yielding up to 8% throughput improvement for quantized MLP blocks
- **QK norm + RoPE fusion**: combines split QKV, reshape, Q/K RMSNorm, and rotary embedding into a single kernel
- **RMSNorm + quantization**: eliminates an intermediate read/write of the full-precision activation tensor

These fusions are applied automatically when `torch.compile` is enabled — you don't need to select or configure them individually.

Parallelism

vLLM supports all four major parallelism strategies — data parallelism (DP), tensor parallelism (TP), pipeline parallelism (PP), and expert parallelism (EP) — but the way they compose is worth understanding, especially for Mixture-of-Experts models.

The standard pattern for dense models is **TP within a node and PP across nodes** (Section 7.3, Section 7.4). TP shards individual layers across GPUs on the same node and synchronizes via AllReduce, while PP distributes layers sequentially across nodes. vLLM optimizes pipeline bubbles by processing multiple requests concurrently through the pipeline.

For MoE models, the picture gets more interesting. **Expert parallelism is not a standalone strategy** — it's a modifier flag (`--enable-expert-parallel`) that changes how experts are distributed within an existing TP or DP configuration (AMD ROCm Team 2025).

Without the EP flag, all experts are present on every GPU with their weights sharded across devices, and synchronization uses AllReduce. With EP enabled alongside TP, experts are distributed across the TP GPUs — different experts on different devices — using AllToAll

communication to route tokens to the correct expert. With EP enabled alongside DP, experts are distributed across DP replicas, with AllToAll routing tokens across replicas.

This distinction matters because the two combinations have very different performance profiles. TP+EP excels at low concurrency — roughly 52% higher throughput than DP+EP at 64 concurrent requests for DeepSeek-R1. But DP+EP dominates at high concurrency, with 47% higher throughput at 1024 concurrent requests. The crossover occurs around 256–512 concurrent requests (AMD ROCm Team 2025). The reason is that DP+EP partitions the KV cache across replicas, so each replica handles a smaller share of the total batch. At high concurrency, this partitioning matters more than the communication overhead.

There's a subtlety for models with **Multi-Head Latent Attention** (MLA), like DeepSeek. MLA models have a single KV head, which means the KV cache can't be sharded along the head dimension. Under TP, the full KV cache must be duplicated on every TP rank, wasting memory. DP+EP avoids this duplication, making it the preferred configuration for MLA models at scale.

One more wrinkle: for **ultra-sparse models** with less than 1% activation density — models like Llama-4-Maverick where very few experts fire per token — enabling the EP flag actually hurts throughput by 7–12% (AMD ROCm Team 2025). AllReduce is more efficient than AllToAll when the number of active experts is tiny, so the communication pattern that EP introduces costs more than it saves.

Disaggregated prefill is available as an experimental feature (vLLM Team 2025a). The idea (Section 5.2) is to run two separate vLLM instances — one dedicated to prefill (compute-bound) and one to decode (memory-bandwidth-bound) — with a connector that transfers prefill KV caches from the prefill instance to the decode instance. The motivation is that co-locating prefill and decode on the same GPUs forces suboptimal resource allocation, since the two phases have fundamentally different performance profiles. The vLLM Router (Section 8.7) supports this configuration natively, routing new requests to prefill workers and completed prefills to decode workers. This is still experimental, but it mirrors the more mature disaggregated serving support in SGLang (Section 8.8).

vLLM integrates specialized communication kernels for large-scale expert parallelism: **DeepEP kernels** (high-throughput for prefill, low-latency for decode) and **PPLX kernels** (CUDA graph compatible and flexible for chunked prefill) (vLLM Team 2025c). For load balancing across experts, vLLM supports **EPLB** (Expert Parallel Load Balancer), which implements hierarchical and global load balancing strategies from DeepSeek (vLLM Team 2025c).

Speculative decoding

vLLM supports a broad menu of speculative decoding methods (Section 6.5), and the practical question is which one to use for a given workload.

Draft model decoding is the classic approach: a smaller model proposes candidate tokens and the full target model verifies them in a single forward pass. The draft and target models must share the same tokenizer and vocabulary. vLLM reports up to 1.5x speedup with this method (JarvisLabs 2025).

N-gram matching (prompt lookup decoding) is the simplest option. It searches the prompt and previously generated tokens for n-gram matches and proposes the tokens that followed the match. There's no separate model, so the VRAM overhead is zero. On summarization tasks where output heavily overlaps with the input, n-gram matching achieves up to 2.8x speedup — but for general-purpose chat, gains are modest at 1.10–1.17x (JarvisLabs 2025).

Suffix decoding builds dual suffix tree data structures from the current request and previous request outputs. It's CPU-based, model-free, and ideal for repetitive workloads like code generation or agentic loops where outputs follow predictable patterns (JarvisLabs 2025).

MLP speculators attach lightweight multi-headed MLPs to the target model, with separate prediction heads for 1, 2, and 3 steps ahead. They add roughly one-tenth the parameters of a full draft model (JarvisLabs 2025).

EAGLE is where the state of the art currently sits for general-purpose inference. EAGLE-3, the latest version, uses multi-layer fusion that extracts features from low, mid, and high layers of the target model, plus a training-time technique for resolving distribution mismatch between draft and target. It uses tree attention verification to check multiple candidate sequences simultaneously. On 70B models, EAGLE-3 achieves 1.57–1.60x speedup and is consistently the best method for large models where memory bandwidth is the bottleneck (JarvisLabs 2025).

The practical guidance breaks down by model size and workload:

- **Smaller models (8B)**: suffix decoding excels for code generation; EAGLE for unpredictable chat
- **Larger models (70B+)**: EAGLE-3 consistently wins due to the memory bandwidth bottleneck
- **High prompt/output overlap**: n-gram matching is hard to beat and costs nothing

vLLM supports CUDA graphs for EAGLE-1 and EAGLE-3, and exposes speculative decoding metrics — draft acceptance rate, per-position acceptance rates, and mean acceptance length — for monitoring effectiveness in production (vLLM Team 2025c).

Operational features

Several features round out vLLM's production story beyond raw inference performance.

Multi-LoRA serving in vLLM processes requests using different LoRA adapters concurrently in the same batch — base model, LoRA-A, and LoRA-B requests all run in parallel. Adapters can be added and removed at runtime via API endpoints without restarting the server. The key tuning parameters are `max_loras` (how many adapters can be active in a single batch), `max_cpu_loras` (the LRU cache size for adapters in CPU memory), and `max_lora_rank` (set as low as your adapters allow to save memory) (vLLM Team 2024e).

The vLLM Router is a FastAPI-based HTTP proxy for distributing requests across a fleet of vLLM engines (vLLM Team 2025d). It implements four load balancing algorithms:

- **Consistent hashing**: routes requests with the same key (session or user ID) to the same worker, maximizing KV cache reuse across requests from the same conversation. This is the recommended policy for most deployments.
- **Power of Two (PoT)**: randomly samples two workers and routes to the less loaded one. Low overhead with good load distribution.
- **Round robin** and **random**: standard stateless policies.

The router also supports **prefill/decode disaggregated routing**, sending new requests to prefill workers and completed prefills to decode workers. It integrates with Kubernetes for automatic pod discovery and exposes its own Prometheus metrics endpoint for request volume, latency, and per-worker health (vLLM Team 2025d).

Monitoring uses the standard Prometheus + Grafana stack (Section 8.5). vLLM exposes two categories of metrics through its `/metrics` endpoint: server-level gauges and counters (engine state, KV cache utilization, token throughput) and request-level histograms (TTFT, TPOT, ITL, prompt and generation lengths). For distributed tracing, vLLM supports OpenTelemetry, sending traces via OTLP to backends like Jaeger for end-to-end visibility across the serving pipeline (vLLM Team 2024e).

Hardware support is unusually broad. Beyond NVIDIA and AMD GPUs, vLLM runs on Intel GPUs and CPUs, Arm CPUs, Google TPUs, and through plugins on Intel Gaudi, IBM Spyre, and Huawei Ascend accelerators (Red Hat 2025). The Triton attention backend and `torch.compile` integration are what make this portability practical — the same model code and attention implementation can target different hardware without per-platform kernel rewrites. CPU inference supports DP, TP, and PP across multiple CPU sockets, and vLLM even supports **heterogeneous speculative decoding** where the draft model runs on CPU while the target model runs on GPU (Red Hat 2025).

Weight quantization is well-supported through multiple methods. FP8 W8A8 (8-bit weights and activations) runs on Hopper and Ada Lovelace GPUs, with a weight-only W8A16 variant available on Turing and Ampere via Marlin kernels — providing up to 2x memory reduction and up to 1.6x throughput improvement (vLLM Team 2024e). INT8 W8A8 is supported on compute capability 7.5 and above (Turing through Hopper), though not on Blackwell. For 4-bit weight-only quantization, vLLM supports both AWQ (using the official AWQ kernel) and GPTQ (using the ExLlamaV2 kernel by default). The `llm-compressor` companion library handles the calibration and quantization workflow for all of these methods (vLLM Team 2024d).

Guided decoding constrains model output to conform to a grammar-based finite-state machine, supporting both regular and context-free grammars through backends like xgrammar (vLLM Team 2025b). This enables structured output generation — producing valid JSON matching a schema, for instance — without post-hoc validation and retry.

Multimodal and VLM support received significant attention in V1. Input processing for vision-language models is offloaded to separate processes to avoid blocking the main inference loop. V1 implements encoder caching to avoid redundant vision encoder forward passes, chunked-prefill scheduling adapted for VLM inputs, and image-hash-based prefix caching that extends the prefix caching system to multimodal inputs (vLLM Team 2025e).

Benchmarking tools ship with vLLM for evaluating serving performance across different scenarios (vLLM Team 2024b). Three built-in benchmark modes cover the main use cases: a latency benchmark (short inputs, fixed output length) for measuring per-request latency, a throughput benchmark (1000+ samples submitted at once) for measuring maximum offline batch throughput, and a serving benchmark with Poisson-distributed request arrivals for measuring online serving under realistic load. The serving benchmark reports TTFT, TPOT, and ITL at P50, P90, and P99 — the same percentile latencies that production SLAs are typically defined against (Section 8.5). vLLM also runs continuous benchmarks on every labeled commit and merged PR, publishing results to a public performance dashboard for tracking regressions over time (vLLM Team 2024b).

8.8. SGLang

SGLang's identity is built around two ideas: RadixAttention for fine-grained prefix caching, and a co-designed programming model for multi-call LLM workflows. But the framework has grown well beyond those roots. SGLang now implements a comprehensive set of serving optimizations — from a zero-overhead batch scheduler and hierarchical KV caching to chunked pipeline parallelism and an extensive menu of attention backends. This section walks through those implementations, focusing on what SGLang does differently and where its design choices lead to distinct tradeoffs.

RadixAttention and prefix caching

RadixAttention is SGLang's signature contribution to KV cache management (Zheng et al. 2023). Where vLLM hashes KV blocks and stores them in a flat lookup table, SGLang organizes the entire KV cache as a **radix tree** — a compressed trie where edges are labeled with token sequences of varying lengths and nodes point to the corresponding KV cache tensors.

When a new request arrives, the scheduler walks the tree to find the longest prefix match. If the request's prompt shares a prefix with any cached sequence, the matching KV cache is reused without recomputation. The tree structure makes this automatic and fine-grained: it handles multi-level sharing across few-shot prompts, branching reasoning trees, multi-turn chat histories, and self-consistency sampling without any explicit configuration.

The key architectural difference from vLLM's block-level hashing is granularity. vLLM matches at block boundaries using O(1) hash lookups — fast, but it can only reuse cache at the granularity of fixed-size blocks. SGLang's token-level matching in a radix tree enables finer-grained reuse at the cost of a tree traversal. For workloads with many shared prefixes — the exact scenario RadixAttention was designed for — the finer granularity pays off.

RadixAttention is enabled by default and uses **LRU eviction** that recursively frees leaf nodes when GPU memory fills up. The overhead when there is no cache hit is negligible — the tree traversal cost is dwarfed by the GPU computation (Zheng et al. 2023). It can be

disabled with `--disable-radix-cache` for workloads where prefix sharing is rare and the tree maintenance isn't worth it.

Scheduling and memory management

SGLang's scheduler operates at the iteration level (Section 5.1), forming a new batch from the waiting queue after every forward pass. The main event loop receives requests, processes input, calls `get_next_batch_to_run()`, executes a GPU forward pass, then calls `process_batch_result`. Unfinished requests loop back for further processing. This is the same continuous batching approach used by other frameworks, but SGLang layers several distinctive features on top.

Zero-overhead batch scheduling. Introduced in v0.4 (LMSYS 2024b), this optimization overlaps CPU batch preparation with GPU execution. While the GPU runs the current batch's forward pass, the CPU concurrently prepares the next batch — creating "future tokens" and carefully scheduling CUDA events and synchronization points so that there is no idle time on the GPU between consecutive decoding batches, reducing TPOT. The SGLang team verified this via Nsight profiling: the GPU stays fully occupied across batch boundaries. The result is a 1.1x throughput improvement over SGLang v0.3 and a 1.3x speedup over other frameworks at the time of release (LMSYS 2024b).

Cache-aware scheduling policies. The default scheduling policy is **Longest Prefix Match** (`lpm`), which prioritizes requests whose prompts share the longest prefix already in the radix cache. This is a natural pairing with RadixAttention — the scheduler actively steers toward cache hits rather than leaving them to chance. SGLang also supports `dfs-weight` (balances cache hits with tree traversal efficiency), `fcfs` (first come first serve), `lof` (longest output first), and `random`, configured via `--schedule-policy` (SGLang Team 2024).

Ragged batching. Rather than using a traditional prefill mode, SGLang uses what it calls **extend mode**: it incrementally updates existing KV cache using ragged tensors. This is SGLang's version of ragged batching, where sequences of different lengths in the same batch are packed together without padding. The extend mode determines whether a ragged forward pass is needed based on token count and wrapper count.

Chunked prefill is supported and enabled by default, configured via `--chunked-prefill-size` (default 8192, favoring throughput). If a prompt exceeds this value, it's split into smaller chunks processed sequentially. Setting it to `-1` disables chunking. SGLang also supports **mixed chunked prefill** (`--enable-mixed-chunk`, disabled by default), where prefill and decode operations are mixed within the same batch in a single forward pass. The current limitation is that mixed mode uses a single prefill attention kernel for the whole batch rather than separate optimal kernels for prefill versus decode requests (SGLang Team 2024).

Priority-based scheduling is opt-in via `--enable-priority-scheduling`. When enabled, requests can carry explicit priorities that override the scheduling policy ordering. The scheduler can preempt running requests to make room for higher-priority arrivals, evicting lowest-priority running requests first. The preemption threshold is controlled by `priority_scheduling_preemption_threshold` (default: 10). This feature is still maturing

— a known issue is that excessive preemption can occur where low-priority requests keep getting preempted even after the minimum token removal threshold is satisfied (SGLang Team 2024).

Preemption in SGLang uses **recomputation only**. There is no swap-to-CPU preemption strategy. When a request is preempted, its KV cache is discarded, and when the request is rescheduled, its KV cache must be recomputed from scratch. This is a deliberate simplification compared to vLLM's hybrid swap/recompute approach — it avoids the complexity and PCIe bandwidth cost of CPU swapping, but it means preemption is more expensive for long sequences that took significant prefill work to generate.

OOM handling is adaptive rather than static. The `new_token_ratio` parameter controls how aggressively the scheduler estimates future token consumption. When a batch executes successfully, the ratio decreases (more aggressive batching). When OOM occurs, the ratio increases (smaller batches). The `--max-running-requests` flag provides an additional hard cap on concurrent decoding sequences. The `--mem-fraction-static` parameter controls the ratio of GPU memory allocated to model weights plus the KV cache pool; the rule of thumb is to reserve 5–8 GB for activations, reducing the fraction to 0.8 or 0.7 if OOM occurs (SGLang Team 2024).

SGLang does not implement explicit request rejection under memory pressure at the engine level. The Model Gateway layer does support token-bucket rate limiting with FIFO queuing, returning 429 or 408 status codes, but this is admission control at the routing layer rather than backpressure from the GPU engine.

HiCache extends RadixAttention with a hierarchical KV caching system across three memory tiers: GPU memory (L1), host/CPU memory (L2), and distributed storage (L3) (LMSYS 2025c). A `HiRadixTree` acts as a page table for KV caches across these tiers. GPU-assisted I/O kernels provide up to 3x higher throughput for CPU-GPU transfers compared to naive copy. HiCache supports multiple storage backends including 3FS, Mooncake, NIXL, AIBrix KVCache, and local file, with configurable write policies: write-through, write-through-selective (which uses hit-count tracking to identify hot spots), and write-back. The performance impact is substantial: up to 6x throughput improvement and up to 80% reduction in TTFT for cache-heavy workloads (LMSYS 2025c).

Attention backends

SGLang doesn't commit to a single attention kernel either, but its backend selection matrix is unusually broad — spanning not just GPU generations but entire hardware families.

The auto-selection logic follows GPU compute capability (SGLang Team 2024):

- **Triton** is the default on Turing (SM75) GPUs.
- **FlashInfer** is the default on Ampere (SM80, SM86) and Ada Lovelace (SM89) GPUs. FlashInfer provides a unified API that abstracts over multiple kernel implementations and serves as the primary backend for older NVIDIA architectures. NVIDIA releases its most performant inference kernels — including TensorRT-LLM kernels — through FlashInfer for integration into SGLang.

- **FlashAttention-3** is the default on Hopper (SM90) for standard multi-head attention models. SGLang skipped FA1 and FA2 as explicit backends, jumping directly to FA3 with FlashInfer covering earlier-generation workloads. FA3 became the default attention backend as of v0.4.6 for mainstream MHA models on Hopper, chosen for its native paged KV cache support via the `flash_attn_with_kvcache` API. It consistently delivers the highest throughput across tested scenarios, outperforming FlashInfer and Triton especially as input/output size increases (SGLang Team 2024).
- **FlashInfer** remains the default on Hopper for non-MHA models (MLA, GQA variants).
- **FlashAttention-4** is supported for Blackwell (SM100+) GPUs with `--page-size 128` for MHA and `--page_size 1` for MLA. FA4 supports FP4 KV cache, but currently has a limitation: decode speed degrades as sequence length grows due to lack of SplitKV support on Hopper, making it primarily useful for prefill in some configurations.

For **Multi-Head Latent Attention** models like DeepSeek, SGLang provides a dedicated backend matrix: FlashInfer MLA, FlashMLA (from DeepSeek), Cutlass MLA, TRTLLM MLA, FA3, and FA4. FlashInfer MLA operates with page_size=1 and supports FP4 KV cache and prefix caching. On Blackwell, TRTLLM prefill/decode DSA kernels are the default (SGLang Team 2024).

SGLang also supports hybrid prefill/decode backend configuration via `--prefill-attention-backend` and `--decode-attention-backend`, allowing different kernels for each phase — a recognition that the optimal kernel for compute-bound prefill is often different from the optimal kernel for memory-bound decode.

Beyond the attention backends, hardware support extends to AMD (ROCm via AITER and Wave backends), Huawei Ascend (NPU), and Intel (XPU), though the NVIDIA backends are the most mature (SGLang Team 2024).

Kernel optimization

SGLang's approach to kernel optimization centers on **sgl-kernel**, a standalone high-performance CUDA kernel library that provides optimized primitives for quantization, MoE, attention, and GEMM operations. sgl-kernel supports multiple architectures (SM80, SM89, SM90, SM90a, SM100a+) and integrates third-party libraries including CUTLASS, FlashInfer, and DeepGEMM.

Piecewise CUDA graphs are enabled by default. SGLang splits model computation into separate pieces captured as individual CUDA graphs for predefined token lengths. Inputs are padded to the nearest captured size at runtime, eliminating kernel launch overhead. Memory is optimized via a global shared memory pool across runners, with capture in reverse order (largest-first). The approach auto-disables for speculative decoding, pipeline parallelism, LoRA, VLM models, non-CUDA hardware, and deterministic inference — cases where the static capture assumption breaks down (SGLang Team 2024).

`torch.compile` integration is experimental. SGLang implements a custom `SGLangBackend` that traces model forward passes as FX graphs, splits at registered split points (such as MoE dispatch operations), compiles each piece separately for dynamic shapes, and dispatches

at runtime through eager split ops and per-piece replay. Custom CUDA kernels must be registered via `register_custom_op` for compatibility. Currently, `torch.compile` is only supported in combination with CUDA graphs, not standalone (SGLang Team 2024).

KV cache engineering

Beyond RadixAttention's tree-based management, SGLang supports several KV cache optimization techniques.

Paged KV cache is supported across backends, though with different page size constraints. FA3 provides native paged KV cache support via the `flash_attn_with_kvcache` API, which accepts the entire page table directly. Different backends impose different page size requirements: FA4 requires page_size=128 for MHA, FlashInfer MLA uses page_size=1, and TRTLLM uses fixed sizes of 16, 32, or 64 (SGLang Team 2024).

KV cache quantization supports FP8 E5M2 (larger dynamic range), FP8 E4M3 (higher precision, recommended), and experimental FP4 E2M1 (MXFP4 with 16-element blocks), enabled via `--kv-cache-dtype`. FP8 provides roughly 2x memory savings over BF16; FP4 provides roughly 3.56x savings accounting for scaling overhead. FP8 requires per-tensor scaling factors from checkpoints or a JSON file (defaulting to 1.0 with an accuracy warning), while FP4 handles scaling automatically. INT8 KV cache is not supported — only FP8 and FP4. A critical caveat: not all backends support fused dequantization with the attention kernel, and using quantized KV with an unsupported backend can be "extremely slow" (SGLang Team 2024).

MLA support is first-class. MLA stores latent representations rather than full K/V heads, significantly reducing KV cache memory. SGLang provides dedicated MLA backends and supports FP4 KV cache quantization for MLA models. Under data parallelism, MLA's latent state caches are stored on different GPUs, with requests split across them (SGLang Team 2024).

Selective KV cache and token eviction are supported through multiple mechanisms. DoubleSparsity provides approximated attention with token selection for KV cache compression. Token eviction methods (SnapKV, PyramidKV) permanently drop non-important tokens for long-context and streaming scenarios. FlashInfer supports vector-sparsity (page_size=1), enabling efficient KV cache token pruning at token granularity. **Native Sparse Attention** (NSA) is also supported, with a fuse store indexer for K cache and a configurable KV length threshold for sparse MLA attention at prefill — improving throughput for DeepSeek V3.2 and GLM-5 (SGLang Team 2024).

Speculative decoding

SGLang supports several speculative decoding methods (Section 6.5), with a particular emphasis on the EAGLE family.

EAGLE-2 and EAGLE-3 are the primary methods, using tree-structured speculative decoding with branching controlled by `--speculative-eagle-topk`. EAGLE-2 uses a draft

model to evaluate branch probability and dynamically stops expansion of unlikely branches; after expansion, reranking selects the top `--speculative-num-draft-tokens` final nodes for verification. EAGLE-3 removes the feature prediction objective, incorporates low and mid-layer features, and is trained on-policy. The performance difference is significant: on LLaMA-3.1-8B with an H100, EAGLE-2 achieves 244 tokens/s (54% improvement over baseline), while EAGLE-3 reaches 373 tokens/s (136% improvement over baseline of 158 tokens/s). SGLang also supports EAGLE-2 combined with **FR-Spec**, which uses a truncated high-frequency vocabulary to reduce lm_head overhead, configured via `--speculative-token-map` (SGLang Team 2024).

SGLang provides **SpecForge**, its own training framework for EAGLE draft models — making it one of the few serving frameworks that includes tooling for the draft model training side, not just inference.

Standalone draft model decoding is supported (`--speculative-algorithm STANDALONE`), using a separate smaller model for token-level drafting. The limitation is that it cannot be combined with `--enable-dp-attention` (SGLang Team 2024).

N-gram speculative decoding (`--speculative-algorithm NGRAM`) drafts tokens from the previous context cache with no separate model required. It's configurable via `--speculative-ngram-max-bfs-breadth` (1–10, default 10), `--speculative-ngram-match-type` ("BFS" or "PROB"), `--speculative-ngram-max-trie-depth` (max 18), and `--speculative-ngram-capacity` (up to 10M entries). Like standalone decoding, it's incompatible with data-parallel attention, overlap scheduling, and mixed chunked prefill (SGLang Team 2024).

Multi-token prediction is supported for models with built-in MTP heads, specifically DeepSeek V3/V3.1/V3.2. MTP operates in two stages: lightweight heads generate n candidates in a single pass, then the full model validates all drafts in parallel, accepting the longest matching prefix. At moderate scale (16 H200 GPUs), MTP delivers +59.8% throughput with 3-token prediction and +60.8% with 4-token prediction. At larger scale (128 H200 GPUs), the gains are more modest at +14.2% output throughput. Average acceptance length is 2.4 tokens, with a recommended starting draft_token_num of 2 (LMSYS 2025a).

Lookahead decoding (Jacobi iteration) has partial support, contributed via a community PR. An experimental **Speculative Decoding V2** with an overlap scheduler is available via `SGLANG_ENABLE_SPEC_V2=True`, though it requires `--speculative-eagle-topk 1` (SGLang Team 2024).

Parallelism

SGLang supports data parallelism, tensor parallelism, pipeline parallelism, expert parallelism, and context parallelism, with total world size computed as TP × PP × EP × DP.

Tensor parallelism (`--tp <size>`) shards model weights across GPUs within a node, synchronized via AllReduce — the standard approach (Section 7.3).

Pipeline parallelism is where SGLang has made a distinctive contribution. **Chunked Pipeline Parallelism**, introduced in January 2026 (LMSYS 2026), splits long sequences into

chunks and processes different chunks in parallel across pipeline stages. It uses asynchronous P2P communication with non-blocking sends and receives, and a micro-batching event loop that overlaps GPU computation with CPU metadata processing and PP communication. Dynamic chunking adapts chunk sizes at runtime rather than using a fixed split.

The performance results are notable: PP4 TP8 multi-node yields a 3.31x prefill throughput improvement for DeepSeek-V3.1 on an H20 cluster compared to TP8 with 12K chunked prefill. Scaling efficiency stays above 80% at PP4. For ultra-long prompts (Qwen3-235B-A22B-FP8 on H20 with PP8), chunked pipeline parallelism delivers up to 81% reduction in TTFT (LMSYS 2026).

Expert parallelism (`--ep <size>` plus `--enable-ep-moe`) distributes expert weights across multiple devices in MoE models using optimized all-to-all communication and grouped GEMMs. The all-to-all backend is configurable via `--moe-a2a-backend` with options including DeepEP, Mooncake (DeepEP extension with RDMA), NIXL (NVIDIA NIXL with RDMA and NVLink), MORI (AMD ROCm native), FlashInfer, and Ascend. A key constraint: DeepEP, Mooncake, NIXL-EP, Ascend, and MORI only support ep_size = tp_size. For ep_size < tp_size, only the default AllReduce/AllGather backend works (SGLang Team 2024).

The MoE computation backend is separately configurable via `--moe-runner-backend`, with options including Triton, DeepGEMM, CUTLASS, and FlashInfer variants for Blackwell (TRT-LLM kernels), FP4/FP8 (CUTLASS), MXFP4, and cuDSL. DeepEP dispatch mode (`--deepep-mode`) can be set to `auto` (runtime switching), `normal` (optimized for prefill), or `low_latency` (optimized for decode, CUDA graph compatible) (SGLang Team 2024).

For expert load balancing, SGLang supports **EPLB** (Expert Parallelism Load Balancer from DeepSeek) via `--enable-eplb`, plus `--enable-two-batch-overlap` (up to 2x throughput) and `--enable-single-batch-overlap` for overlapping communication with computation (SGLang Team 2024).

Data parallelism (`--dp <size>`) replicates the model across multiple scheduler instances. SGLang also supports `--enable-dp-attention` for data-parallel attention, which is distinct from standard DP: latent state caches for different requests are stored on different GPUs, which is particularly useful for MLA models where the KV cache can't be sharded along the head dimension (SGLang Team 2024).

Context parallelism is supported for the prefill phase, distributing long sequences across GPUs for MHA models. This is distinct from pipeline parallelism — it splits a single sequence's tokens across GPUs rather than distributing layers (LMSYS 2026).

Combined parallelism patterns are well-supported. PP4 TP8 has been demonstrated for DeepSeek-V3.1 on multi-node H20 clusters. TP within a node combined with EP across nodes has been demonstrated at scale with 96 H100 GPUs for DeepSeek deployment. Different parallelism configurations can be used per phase in disaggregated serving: for example, PP8 TP8 for prefill and PP1 DP16 EP16 for decode (LMSYS 2025b).

Multi-node deployment is configured via `--nnodes`, `--node-rank`, and `--dist-init-addr`, with support for Kubernetes (via LWS_LEADER_ADDRESS and LWS_GROUP_SIZE environment variables) and SLURM clusters (SGLang Team 2024).

Disaggregated prefill

SGLang supports disaggregated prefill (Section 5.2) where prefill and decode run on separate node pools connected via high-performance communication backends. KV cache transfer between pools uses RDMA-based transports including the Mooncake Transfer Engine and NIXL. Send and receive operations are non-blocking, running in background threads, and use scatter-gather elements (SGE) in RDMA to transfer non-contiguous memory chunks efficiently.

At scale, this has been demonstrated on 12 nodes of 8 H100 GPUs, achieving 52.3k input tokens/s and 22.3k output tokens/s per node for 2000-token inputs. The Model Gateway supports PD disaggregation routing with separate prefill and decode worker pools (LMSYS 2025b).

Quantization

SGLang supports a broad range of weight quantization methods: `fp8`, `mxfp4`, `blockwise_int8`, `awq`, `gptq`, `compressed-tensors`, `quark`, `auto-round`, `awq_marlin`, `gptq_marlin`, `gguf`, `modelopt/modelopt_fp8` (Hopper SM90+), `modelopt_fp4` (Blackwell SM100+), `bitsandbytes`, and `torchao`. Both offline quantization (pre-quantized weights) and online quantization (dynamic scaling at runtime) are supported, though offline is recommended for better performance (SGLang Team 2024).

For **weight-activation quantization**, SGLang supports W8A8 with `--quantization w8a8_int8` or `w8a8_fp8`, using optimized CUTLASS int8 or fp8 kernels from sgl-kernel for per-channel W8A8 with per-token dynamic activation quantization (SGLang Team 2024).

GPTQ and **AWQ** are supported both in their standard forms and with Marlin kernel optimization (`gptq_marlin`, `awq_marlin`) (SGLang Team 2024).

The GEMM backend is configurable for both FP8 (`--fp8-gemm-backend` with options including DeepGEMM, FlashInfer TRT-LLM, FlashInfer CUTLASS, CUTLASS, Triton, and AITER) and FP4 (`--fp4-gemm-backend` with CUTLASS, FlashInfer CUTLASS, FlashInfer cuDNN, and FlashInfer TRT-LLM). AMD GPUs have their own supported quantization methods including `fp8`, `mxfp4`, `w8a8_int8`, `w8a8_fp8`, and `petit_nvfp4` (NVFP4-on-ROCm) (SGLang Team 2024).

Torchao online quantization options include `int8dq`, `int8wo`, `fp8wo`, `fp8dq` (per_tensor/per_row), and `int4wo` (group sizes 32/64/128/256). Note that `int8dq` has CUDA graph capture bugs requiring `--disable-cuda-graph` (SGLang Team 2024).

There are several practical limitations to be aware of. Mixed-bit quantization is incompatible with layer fusion (QKV fusion issues). Quantized MoE models may hit kernel limitations with mlp.gate layers. Quantized VLM support is limited, with some models showing near-zero accuracy. ModelOpt online quantization causes high startup overhead and increased VRAM usage. Pre-quantized models (such as DeepSeek V3/R1 with native FP8) should not have additional `--quantization` flags applied (SGLang Team 2024).

Operational features

Structured output and guided decoding. SGLang supports JSON schema (via Pydantic BaseModel or raw schema), regular expressions, and EBNF grammar (GGML BNF format for XGrammar). The grammar backend is configurable via `--grammar-backend`: XGrammar (default, best performance, supports JSON/regex/EBNF), Outlines (JSON/regex only), or Llguidance (JSON/regex/EBNF). The key optimization is that per-step mask generation is overlapped with LLM inference, hiding grammar processing latency and achieving constrained decoding with near-zero overhead at production scale. SGLang also supports structural tags for multiple constrained regions within a single response — useful for tool calling where some parts of the response are free-form and others must conform to a schema (SGLang Team 2024).

Multi-LoRA serving integrates S-LoRA and Punica for efficient multi-adapter batching. Key configuration parameters include `--max-loras-per-batch` (default 8), `--max-loaded-loras` (CPU memory limit), and `--lora-backend` (`triton` or `csgmv`; csgmv is the default, providing 20–80% latency improvement at high concurrency). SGLang supports GPU pinning for frequently-used adapters, LRU/FIFO eviction policies, and async overlap loading (`--enable-lora-overlap-loading`, yielding roughly 35% TTFT reduction). Adapters can be loaded and unloaded dynamically via REST endpoints. The API supports both a native `lora_path` parameter and OpenAI-compatible `model:adapter-name` syntax. Piecewise CUDA graphs auto-disable when LoRA is active (SGLang Team 2024).

The Model Gateway is a Rust-based router that sits in front of one or more SGLang engine instances. It implements five load balancing policies: `random`, `round_robin`, `power_of_two` (sample 2 workers, pick the lighter one), `cache_aware` (default, balancing cache locality with load distribution), and `bucket` (dynamic load buckets). The cache-aware policy is particularly interesting: it routes requests to the instance most likely to have relevant prefix cache entries in its radix tree, achieving up to 1.9x throughput increase with a 3.8x higher cache hit rate compared to cache-oblivious routing (LMSYS 2024b). Cache-aware tuning parameters include `--cache-threshold`, `--balance-abs-threshold`, `--balance-rel-threshold`, and `--eviction-interval-secs` (SGLang Team 2024).

For reliability, the Model Gateway provides circuit breakers (three states: closed, open, half-open), exponential backoff retries with jitter (retryable status codes: 408, 429, 500, 502, 503, 504), API key authentication, and TLS/mTLS (SGLang Team 2024).

Monitoring. The Model Gateway exposes over 40 Prometheus metrics across HTTP (requests_total, request_duration_seconds, rate_limit_total), router (ttft_seconds, tpot_seconds, generation_duration_seconds), worker (pool_size, connections_active, health_checks_total), and circuit breaker categories, with duration histogram buckets from 1ms to 240s. OpenTelemetry tracing exports to OTLP/gRPC with W3C Trace Context propagation for distributed tracing across the full request lifecycle (SGLang Team 2024).

Multimodal support covers 30+ model architectures including Llama 3.2 Vision, LLaVA 1.5/NeXT, Qwen3-VL, NVILA, and DeepSeek-VL2. FA4 support has been added for multimodal encoders. VLM models auto-disable piecewise CUDA graphs due to the dynamic

nature of vision token lengths. SGLang Diffusion extends the framework to video and image generation workloads (SGLang Team 2024).

Deployment modes include co-launch (single process), separate HTTP workers, gRPC workers, PD disaggregation, OpenAI-compatible proxy, and a multi-model gateway (IGW) with per-model policies. WASM middleware supports custom request/response processing, and MCP integration enables tool execution. Reasoning parser integration supports DeepSeek-R1, Qwen-3, and GLM-4.5 thinking formats (SGLang Team 2024).

8.9. TensorRT-LLM

TensorRT-LLM takes a fundamentally different design path from vLLM and SGLang. Where those frameworks operate primarily at the Python level — scheduling, memory management, and model execution all live in Python with selective CUDA kernel calls — TensorRT-LLM compiles the entire model into an optimized execution plan before serving begins. The result is a system that trades flexibility and hardware breadth for raw throughput on NVIDIA hardware.

Architecture and compilation

TensorRT-LLM has gone through a significant architectural shift. The **legacy TensorRT backend** compiled the model graph through NVIDIA's TensorRT engine, which performed kernel selection, operator fusion, and memory optimization at build time. This produced highly optimized execution plans but required a heavyweight compilation step — rebuilding the engine for different batch sizes, sequence lengths, or quantization configurations.

The **PyTorch workflow**, which became the default in v1.0, takes a different approach (NVIDIA 2024g). Instead of compiling through TensorRT's graph optimizer, it runs the model through PyTorch and applies optimizations via `torch.compile` and **piecewise CUDA graphs** — the same strategy vLLM V1 adopted (Section 8.7). The PyTorch workflow offers more flexibility: you can modify model code without recompiling, swap attention backends, and iterate faster during development. The tradeoff is that the legacy backend can still squeeze out a few extra percent of throughput for well-characterized workloads where the compilation cost is amortized across many requests.

An **AutoTuner** benchmarks and selects optimal kernel implementations for each specific configuration, trading compile-time for runtime efficiency (NVIDIA 2024g, 2024l). This is the compilation philosophy in a nutshell: spend time upfront to make every kernel call as fast as possible.

Scheduling and memory management

TensorRT-LLM's scheduler uses the same continuous batching approach as the other frameworks (Section 5.1), but NVIDIA's terminology calls it **in-flight batching** — the batch

composition changes at every iteration, with finished sequences immediately evicted and new requests inserted (NVIDIA 2024g, 2024i). The constraint is that context-phase sequences must precede generation-phase sequences in the packed input tensor, which affects how the scheduler orders work within each step.

Chunked prefill splits context processing into chunks that are batched with generation tokens, just as in vLLM and SGLang (Section 5.2). It requires paged KV cache and fused multi-head attention (FMHA) to be enabled, and chunk sizes (except the final one) must be integer multiples of the KV cache block size (NVIDIA 2024i, 2024g).

Preemption is where TensorRT-LLM's design choices diverge most from the other frameworks. It offers two **separate** preemption strategies under the `MAX_UTILIZATION` allocation policy, rather than the hybrid swap-with-recompute-fallback that vLLM uses (NVIDIA 2024j; SqueezeBits 2025):

- **Swap**: the preempted request's KV cache is copied to host (CPU) memory and reloaded when the request resumes. This preserves computed state but introduces "significant overhead" from PCIe transfers, particularly for long sequences.
- **Drop** (recomputation): the KV cache is discarded entirely. On resumption, the original context plus any tokens generated so far are concatenated and run through a single prefill pass to reconstruct the cache. This is preferred over swapping because it only requires one forward pass rather than a round-trip through host memory.

The default policy is `GUARANTEED_NO_EVICT`, which takes a more conservative approach than either preemption strategy: it preallocates memory for the maximum output length of each request at admission time. If there isn't enough KV cache memory to guarantee a request can run to completion, the request is simply not admitted. This eliminates preemption entirely at the cost of smaller batch sizes, since reserved-but-unused memory can't serve other requests. It's the safe choice for latency-sensitive deployments where preemption would cause unacceptable tail latency spikes.

Priority-based KV cache eviction adds a finer-grained control layer on top of these policies. You can assign priority levels (0–100) to specific token ranges within a request, plus duration values that control how long the priority applies (NVIDIA 2024d). The practical use case is system prompts: assign them maximum priority so they remain cached longer, improving reuse for requests that share the same system prompt.

OOM handling is less adaptive than what vLLM and SGLang provide. There is no automatic dynamic batch size reduction at runtime — `max_batch_size` is configured at build or launch time. NVIDIA's guidance is to profile throughput across batch sizes, set `max_batch_size` 20–30% above the throughput knee, and reduce it if KV cache utilization consistently exceeds 80% (NVIDIA 2024g). This is a manual tuning process rather than the self-adjusting ratios that SGLang uses (Section 8.8).

There is no built-in **SLA-aware scheduling** — no mechanism to automatically adjust scheduling based on per-request latency targets. The priority-based KV cache eviction and request admission policies provide building blocks, but tying them to SLA tiers is left to the orchestration layer.

Attention backends and kernel optimization

This is where TensorRT-LLM's compilation heritage shows most clearly. Rather than selecting from a menu of third-party attention backends the way vLLM and SGLang do, TensorRT-LLM primarily uses its own custom attention kernels, with FlashInfer available as a pluggable alternative (NVIDIA 2024g, 2024i).

The **context phase** (prefill) uses NVIDIA's fused multi-head attention (FMHA) kernels, which implement FlashAttention v1 and v2 algorithms for larger sequences. For short sequences, vanilla MHA/MQA runs instead. Two FMHA modes are available: standard FP32 accumulation and a variant that forces FP32 in the first batched matrix multiplication for improved numerical accuracy. **FP8 context FMHA** accelerates attention on Ada Lovelace and Hopper GPUs and works simultaneously with paged KV cache (NVIDIA 2024i).

The **generation phase** uses **XQA** (eXtended Query Attention), a specialized kernel for MQA and GQA models that is distinct from TensorRT-LLM's other attention paths (NVIDIA 2024i). XQA supports FP16 and BF16 compute with FP16, BF16, FP8, and INT8 KV cache, and paged KV cache with block sizes of 8, 16, 32, 64, or 128 tokens. It's enabled by default and can be disabled with `--disable_xqa`.

Multi-block mode is TensorRT-LLM's version of FlashDecoding (Section 6.1) — it distributes attention computation across multiple CUDA thread blocks during the generation phase to improve GPU occupancy when a single query attends to a long KV cache. It's enabled by default since v0.13 (NVIDIA 2024i).

The kernel fusion story goes well beyond attention. The **masked multi-head attention kernel** fuses QKV bias addition, RoPE application, and dequantization/quantization into a single kernel launch. A fused **GEMM-SwiGLU kernel** is available on Hopper (SM90). Fused **MoE finalize and AllReduce** operations reduce overhead for Mixture-of-Experts models, and one-sided AlltoAll over NVLink further optimizes MoE communication (NVIDIA 2024i, 2024l).

Piecewise CUDA graphs capture operation sequences into optimized graphs for reduced CPU launch overhead, combined with `torch.compile` in the PyTorch workflow (NVIDIA 2024g). This is the same approach vLLM V1 takes (Section 8.7), but TensorRT-LLM was an earlier adopter.

One notable absence: **FlexAttention** is not supported. FlexAttention is a PyTorch-native API (`torch.nn.attention.flex_attention`) for user-defined attention patterns via compilation. TensorRT-LLM uses its own custom attention kernels and plugin system rather than PyTorch's attention APIs, so FlexAttention's composability advantage doesn't apply here.

KV cache engineering

TensorRT-LLM implements paged KV cache with configurable block sizes of 8, 16, 32, 64, or 128 tokens (default 128, must be a power of 2) (NVIDIA 2024i). The contrast with the legacy **contiguous KV cache** is stark: the old approach allocated a monolithic tensor of shape `[max_batch_size * max_beam_width, 2, num_heads, max_seqlen,`

`hidden_dim_per_head]`, wasting memory whenever sequences were shorter than the maximum — which was almost always.

KV cache quantization supports INT8 and FP8 modes with per-tensor scaling factors. The XQA kernel handles FP16, BF16, FP8, and INT8 KV cache combinations, so quantization integrates cleanly with the generation-phase attention path (NVIDIA 2024i). There is no support for **TurboQuant**, the rotation-plus-vector-quantization technique for KV cache compression that has community integrations in vLLM and SGLang.

Prefix caching (KV cache reuse) uses **block hashing** — only full blocks can be shared across requests, similar to vLLM's approach and unlike SGLang's token-level radix tree (Section 8.8). It's enabled by default when built with paged context FMHA, with the default block size of 128 tokens (NVIDIA 2024d). There's an important practical limitation: reuse requires the first request to complete its block before subsequent requests can access it. At high batch sizes, requests may launch before prior ones finish filling their blocks, preventing reuse.

A **KV Cache Event API** provides real-time tracking of cache state changes — block creation, storage, removal, and updates — enabling downstream routing decisions (NVIDIA 2024d). This is particularly useful for multi-instance deployments where a load balancer needs to know which instances already have specific token sequences cached.

Host memory offloading extends KV cache reuse by preserving evicted blocks in pinned host memory with LRU eviction. A `secondary_offload_min_priority` threshold prevents low-priority blocks from being offloaded at all, evicting them directly to reduce GPU-CPU traffic (NVIDIA 2024j, 2024d).

MLA (Multi-Head Latent Attention) support is first-class. KV cache reuse for MLA was added in v1.1, chunked prefill for MLA in v1.0, FP8 MLA on Hopper and Blackwell in v0.19.0, and FlashMLA for SM90 in v0.19.0 (NVIDIA 2024l).

Sliding window attention is handled through a **cyclic KV cache** that treats storage as a circular buffer retaining the last N tokens. Per-layer window sizes are supported, and a **StreamingLLM** mode maintains a set of "sink tokens" permanently while cycling other positions, adjusting position embeddings to use in-cache positions (NVIDIA 2024i). The limitation is that it's incompatible with beam search.

Keyformer-style token-level eviction based on attention scores is not supported. TensorRT-LLM's selective eviction operates at the block level via prioritized LRU, not at the individual token level.

Speculative decoding

TensorRT-LLM supports a broad set of speculative decoding methods (Section 6.5), with particularly strong performance numbers on large models (NVIDIA 2024m, 2024d).

Draft model decoding uses two independently trained models that share the same vocabulary. The draft model generates up to K candidate tokens, and the target model validates them in a single forward pass. The performance on H200 GPUs tells the story: Llama 405B with a 3B draft model achieves 120.75 tokens/sec — a 3.61x speedup over the 33.46 tok/sec

baseline. Llama 70B with a 1B draft achieves 146.05 tokens/sec (2.86x speedup). These numbers are notably higher than what vLLM reports for draft model decoding (Section 8.7), reflecting TensorRT-LLM's kernel-level optimizations. Draft model decoding works with FP8 quantization and is compatible with Triton Inference Server (NVIDIA 2024d).

EAGLE is well-supported across versions. EAGLE-1 uses predefined decoding trees, while EAGLE-2 assembles trees dynamically via beam search. EAGLE-3 was added in v0.19.0 with disaggregated serving support following in v0.21.0 (NVIDIA 2024l, 2024m).

Medusa adds multiple lightweight LM heads predicting future tokens, with configurable tree structures at runtime via `medusa_choices`. Inflight batching support has been available since v0.9.0. The current limitations are notable: it only supports Vicuna (fine-tuned LLaMA), requires `medusa_temperature=0`, and is incompatible with beam search (NVIDIA 2024m).

ReDrafter takes a different approach — recurrent prediction where each draft token depends on the previous one. The engine performs logits prediction, beam search, and acceptance internally, supporting both inflight fused batching and static batching (NVIDIA 2024m).

N-gram speculative decoding copies input prompt and previously generated output as draft tokens, requiring only the target model. It performs best on tasks with high n-gram overlap — summarization, question answering, code editing — where the output draws heavily from the input. NGram v2 was added in v1.0 (NVIDIA 2024m, 2024l).

Multi-token prediction is supported for models with built-in MTP heads, notably DeepSeek V3/R1. When combined with disaggregated serving, MTP adds another 1.6x–2.5x speedup on top of the disaggregation benefits (NVIDIA 2024l, 2024c).

Lookahead decoding (Jacobi iteration) runs two parallel computation branches — a lookahead branch and a verification branch — within the same model, requiring no additional training or fine-tuning. It has been experimental since v0.13.0, with inflight batching support since v0.16.0 (NVIDIA 2024m, 2024l).

An **overlap scheduler** between draft forwards was added in v0.21.0, implementing the pipelined draft-verify idea from speculative speculative decoding (Section 6.5) (NVIDIA 2024l).

Inference with reference — reusing a reference output to accelerate decoding — is not supported.

Parallelism

TensorRT-LLM supports data parallelism, tensor parallelism, pipeline parallelism, expert parallelism, and context parallelism, with the constraint that `tensor_parallel_size * pipeline_parallel_size` must equal the total GPU count (NVIDIA 2024k, 2024b).

Tensor parallelism splits individual weight matrices across devices with AllReduce synchronization via NCCL. It's the preferred strategy within a single node where fast NVLink connections keep the AllReduce cost low.

Pipeline parallelism divides the model into sets of contiguous layers, with each GPU housing one set. The recommended default for multi-node deployment is **TP within a node, PP across nodes**, with one exception: NVIDIA's NVL36/NVL72 Blackwell systems have multi-node NVLink, so TP can span their full GPU sets without the usual cross-node bandwidth penalty (NVIDIA 2024b).

Expert parallelism distributes complete expert weights across GPUs, so each GPU holds the full weights of its assigned experts. Configuration uses `--moe_tp_size` and `--moe_ep_size`, with the constraint that their product must equal `tp_size`. A hybrid approach combining EP with TP enables load balancing across experts (NVIDIA 2024h).

Wide Expert Parallel introduces expert slots that are decoupled from specific experts, enabling replication of high-demand experts for load balancing (NVIDIA 2024k). The **Expert Parallelism Load Balancer** (EPLB) handles both offline and online load balancing, with optimized communication kernels for GB200 multi-node NVLink. Large-scale EP was extended in v0.21.0, and **DeepEP** integration provides optimized MoE communication kernels (NVIDIA 2024l, 2024g). Token dropping for load balancing is not implemented — load balancing relies on expert replication and EPLB instead.

Attention Data Parallel (ADP) is a distinctive feature that doesn't appear in the other frameworks under this name. ADP replicates GEMM weights on every GPU but partitions the KV cache across devices, eliminating cross-GPU communication during the attention phase. This is conceptually similar to what vLLM calls DP+EP for MLA models (Section 8.7) and what SGLang calls `--enable-dp-attention` (Section 8.8), but TensorRT-LLM provides it as an explicit mode enabled via `enable_attention_dp: true` (NVIDIA 2024k).

Context parallelism distributes long sequences across GPUs during the prefill phase, including Ulysses-style context parallel support added in v0.16.0 (NVIDIA 2024l, 2024g).

Disaggregated prefill is more mature in TensorRT-LLM than in the other frameworks. It fully separates context (prefill) and generation (decode) phases onto different GPU pools, eliminating interference between phases and enabling independent optimization of TTFT and TPOT. Three deployment approaches are available (NVIDIA 2024c):

1. **trtllm-serve**: a REST-based orchestrator with round-robin or KV-cache-aware routing
2. **Dynamo**: datacenter-scale deployment with a smart router and Kubernetes support
3. **Triton Inference Server**: an ensemble model with a BLS (Business Logic Scripting) orchestrator

KV cache transfer between prefill and decode GPUs uses MPI, UCX, or NIXL backends over RDMA or NVLink. UCX and NIXL are recommended for deployments that need dynamic scaling. A particularly useful capability is support for **different parallelism strategies between phases** — for example, context with TP2 paired with generation using PP2, with orchestrated block mapping for layout transformation between the two (NVIDIA 2024c).

The performance gains are substantial: DeepSeek R1 shows 1.4x–2.5x speedup, and Qwen 3 shows 1.7x–6.11x speedup on GB200 GPUs. The current limitation is that the number of context and generation instances is fixed at deployment time; dynamic scaling is under development (NVIDIA 2024c).

Quantization

TensorRT-LLM's quantization support reflects its NVIDIA-first philosophy — it provides deep integration with NVIDIA's quantization formats and hardware features rather than broad third-party method support.

Weight quantization covers FP4/NVFP4 (Blackwell-only, with native support and optimized kernels), FP8 (Hopper and Blackwell, with automatic conversion via Transformer Engine), INT4, and INT8 (NVIDIA 2024g, 2024l). **Mixed precision quantization** is supported, with AutoQ for automated mixed-precision selection since v0.15.0. Block scaling (per-block scale factors) is available for improved accuracy at low precision (NVIDIA 2024g).

Weight-activation quantization includes W4A8 (INT4 weights, FP8 activations) with CUTLASS kernels on Ada Lovelace, and MXFP8xMXFP4 added in v1.0 (NVIDIA 2024l).

GPTQ (INT8, added in v0.15.0), **AWQ** (INT4), and **SmoothQuant** (TensorRT-native INT8, added in v0.15.0) are all supported (NVIDIA 2024l, 2024g).

Operational features

Guided decoding uses the XGrammar backend for grammar-based constrained generation, supporting JSON schema, regular expressions, EBNF grammar, and structural tags. Integration with the overlap scheduler arrived in v1.0 and with speculative decoding in v1.1 (NVIDIA 2024g, 2024l).

Multi-LoRA serving supports dynamic LoRA loading per request with Hugging Face and NeMo format compatibility. FP8 base model with FP16/BF16 LoRA adapters has been supported since v0.11.0, MoE model LoRA since v0.12.0, and PyTorch backend LoRA with adapter eviction since v1.0 (NVIDIA 2024l, 2024g).

Multimodal support covers a broad range of vision-language models including LLaVA-NeXT, Qwen2-VL, Llama 3.2 Vision, and Mistral Small 3.1 VLM, plus visual generation models like FLUX and Wan 2.1/2.2 for image and video (NVIDIA 2024g, 2024l).

Sparse attention support is available for structured sparsity patterns including Native Sparse Attention (NSA) and Skip Softmax Attention, which approximates attention for long-context inference acceleration (NVIDIA 2024g).

Monitoring is less standardized than vLLM's Prometheus endpoint or SGLang's 40+ metric categories. The KV Cache Event API provides real-time tracking of cache state changes for monitoring and routing decisions, and per-request stats are available in the PyTorch workflow since v0.20.0. The `trtllm-bench` benchmarking tool supports streaming with TTFT and ITL metrics since v0.10.0 (NVIDIA 2024l, 2024d).

Hardware support is NVIDIA-only. This is the fundamental tradeoff: TensorRT-LLM doesn't run on AMD, Intel, or any other hardware, but it can exploit NVIDIA-specific features — Hopper's FP8 tensor cores, Blackwell's FP4 support, NVLink topology-aware communication — more deeply than frameworks that target multiple backends.

Production LLM serving sits at the intersection of systems engineering and machine learning optimization. The techniques from the preceding chapters provide the building blocks, but assembling them into a reliable, efficient serving system requires understanding the operational realities — memory pressure, failure modes, monitoring, and orchestration — that don't show up in benchmark results. The frameworks surveyed here each represent a different set of answers to these challenges, and the right choice depends on your specific workload, hardware, and reliability requirements.

8.10. Further Reading

Multi-LoRA serving. The original LoRA paper (Hu et al. 2021) explains the low-rank adaptation technique and why the adapter weights are small enough to swap efficiently. For serving multiple adapters simultaneously, the S-LoRA paper (Sheng et al. 2023) introduces the idea of a unified base model with dynamically loaded adapters, including a custom CUDA kernel for batched LoRA computation across requests using different adapters. Punica (L. Chen et al. 2023) takes a similar approach with a focus on the GPU kernel design for multi-adapter batching. For a practical walkthrough of deploying one base model with many adapters, the Hugging Face blog post "TGI Multi-LoRA: Deploy Once, Serve 30 Models" (Thomas et al. 2024) shows the end-to-end process with real adapter switching.

Disaggregated serving. The DistServe paper (Zhong et al. 2024) introduced the case for separating prefill and decode onto different GPU pools, but the idea has evolved rapidly since then. "Disaggregated Inference: 18 Months Later" (Chen et al. 2025), a retrospective from the DistServe authors, documents how disaggregation went from a research prototype to an industry-standard pattern adopted by NVIDIA Dynamo, vLLM, SGLang, and others, and explores emerging directions like attention-FFN disaggregation.

Multi-model routing. Quality-tier routing — sending easy requests to a small model and hard ones to a large model — can dramatically reduce cost. The RouteLLM blog post (Ong et al. 2024) describes a framework of trained routers that achieve up to 85% cost reduction while maintaining 95% of GPT-4 quality, with open-source router weights and evaluation code.

Benchmarking. If you're evaluating frameworks for a specific deployment, LLMPerf (Anyscale 2024b) provides standardized benchmarking scripts that measure the metrics from Section 3.2 across different frameworks and hardware configurations. For understanding how to interpret benchmark results and avoid common pitfalls, the Anyscale metrics blog post (Kadous et al. 2023) is a practical companion.

System design. For the broader context of how LLM serving fits into production ML systems, Miao et al. (2023) surveys the full stack from model optimization through serving infrastructure. For fleet-level scheduling across multiple model replicas, Wu et al. (2023) covers how to route requests across heterogeneous hardware to maximize goodput under latency constraints.

Framework documentation. Each of the major frameworks has comprehensive documentation that goes beyond what we've covered here. Here again are links to their main documentation sites. The vLLM docs (vLLM Team 2024e) include guides on configuring PagedAttention parameters, enabling speculative decoding, and tuning scheduler policies. The TensorRT-LLM docs (NVIDIA 2024g) cover the compilation pipeline and how to exploit FP8 and custom kernels on NVIDIA hardware. SGLang's docs (SGLang Team 2024) are particularly useful for understanding the RadixAttention cache and the structured generation programming model.

A. Additional Material

This appendix contains additional material to supplement the main content of the book.

A.1. Attention calculation

In order to deeply understand the self-attention mechanism in LLMs, it is helpful to understand the concepts being applied, the size of the data, and how the data flows through the calculations. In this section, we will walk through the multi-head attention tensor data flow diagram in depth, illustrating exactly how the masked self-attention works. We will walk through sample code and the diagram, side-by-side, to provide a unified picture of both the concepts and the data flow.

Figure A.1 is a listing of the PyTorch code we will use for our walk through. It was based on the implementation from Andrej Karpathy's nanoGPT repository (Karpathy 2022). (A copy of the listing can be downloaded from the book repo source, and the full LLM implementation can be reviewed in the nanoGPT repo.) Note that the weight matrices are inside linear layers named `c_attn` and `c_proj`, which are not included in the listing for brevity. Since this code provides the full batched implementation of attention, we will include the batch dimension, **B**, in the tensor dimension labels, so that they match the code. Variable names and comments for other dimensions, such as **S** for the sequence length, have been changed in the code to match the naming conventions in this book (which are listed in the **Conventions** section in the front matter). Figure A.2 provides an overview of the full data flow.

Next, we will walk through the code line by line.

Step 1: calculating Q, K, and V

Line 7 is the first data operation in the forward pass. We see it highlighted, along with the relevant portion of the data flow, in Figure A.3. Our input, X, is a tensor of shape **(B, S, D)**, where **B** is the batch size, **S** the sequence length (number of tokens), and **D** the model dimension (embedding size) of the LLM. The input is multiplied by the three weight matrices for the queries, keys, and values. Mathematically, this is the embedding for Q, K, and V:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

Conceptually, the queries represent the kind of information being sought in prior tokens, the keys list the kinds of information available for each token, and the values hold the information to be copied when the queries and keys match.

```python
def forward(self, x):
    # Store batch size, sequence length, model dimension
    B, S, D = x.size()

    # Calculate query, key, values for all heads in batch
    # (B, S, D) x (D, 3D) -> (B, S, 3D) -> 3 separate (B, S, D)
    q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)

    # Split out and transpose head forward next to the batch dim
    # (B, S, D) -> (B, H, S, d_k) where d_k = D / H
    q = q.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    k = k.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    v = v.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)

    # Calculate scores for self-attention
    # (B, H, S, d_k) x (B, H, d_k, S) -> (B, H, S, S)
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))

    # Causal mask and SoftMax
    # (B, H, S, S)
    att = att.masked_fill(self.bias[:,:,:S,:S] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)

    # Combine values by attention weights
    # (B, H, S, S) x (B, H, S, d_k) -> (B, H, S, d_k)
    y = att @ v

    # Re-assemble weighted values from all heads side by side
    # (B, H, S, d_k) -> (B, S, D)
    y = y.transpose(1, 2).contiguous().view(B, S, D)

    # Output projection
    # (B, S, D) x (D, D) -> (B, S, D)
    y = self.resid_dropout(self.c_proj(y))
    return y
```

Figure A.1.: Sample PyTorch code for the multi-head attention calculation

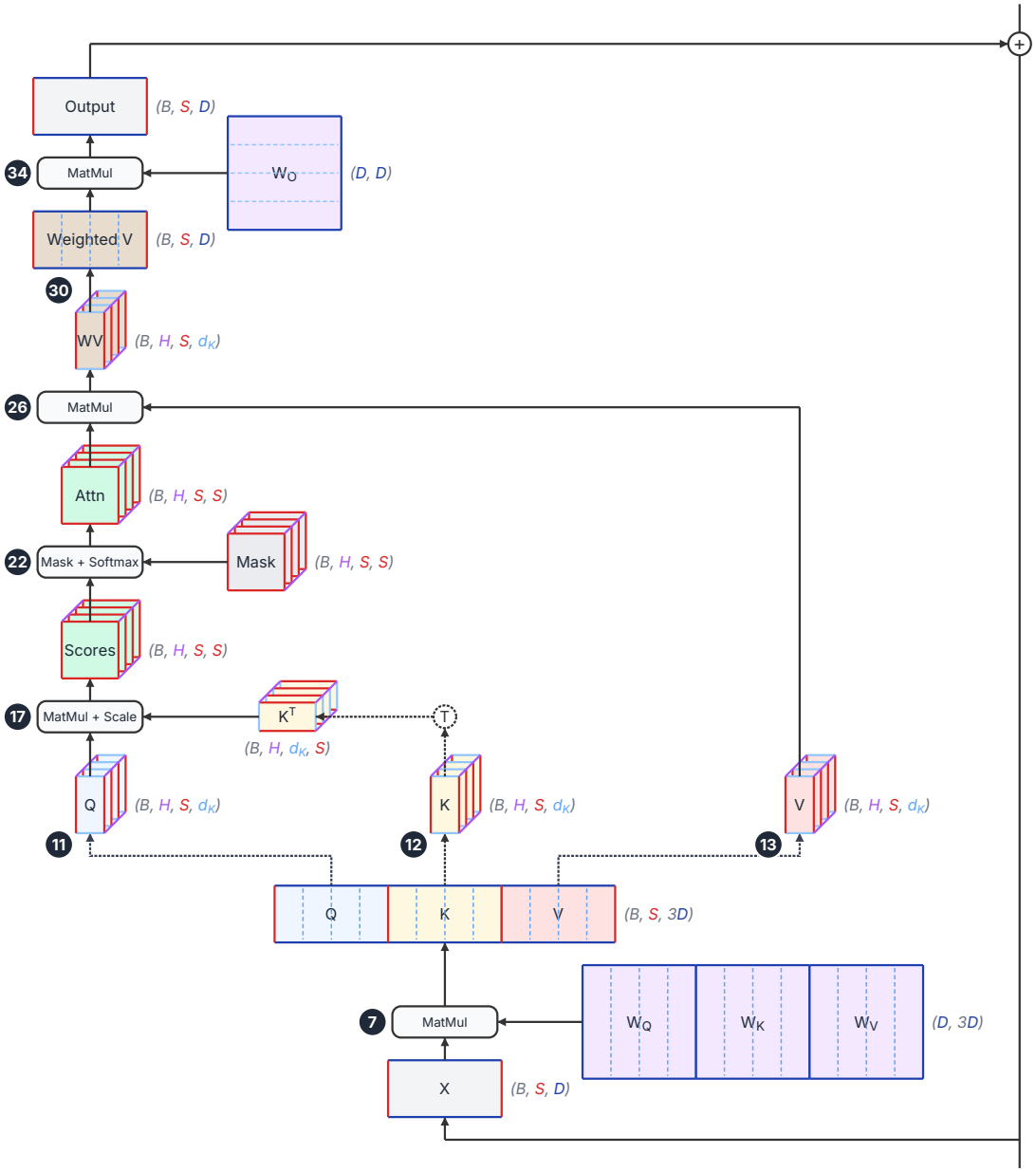
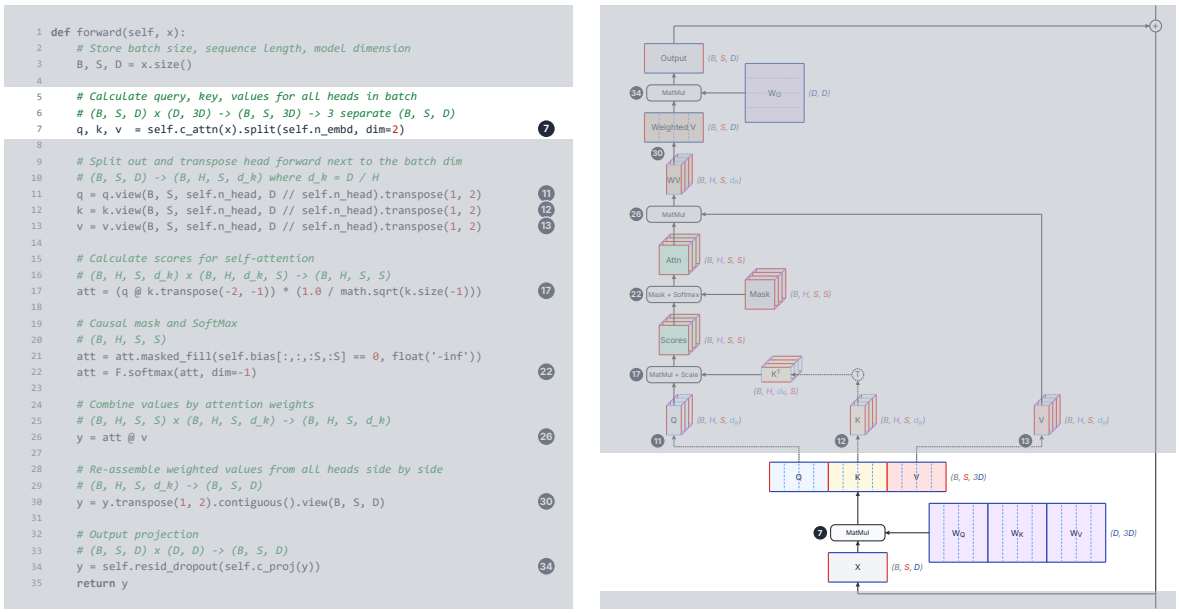


Figure A.2.: Attention flow corresponding to the sample PyTorch code



```
def forward(self, x):
    # Store batch size, sequence length, model dimension
    B, S, D = x.size()

    # Calculate query, key, values for all heads in batch
    # (B, S, D) x (D, 3D) -> (B, S, 3D) -> 3 separate (B, S, D)
    q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)

    # Split out and transpose head forward next to the batch dim
    # (B, S, D) -> (B, H, S, d_k) where d_k = D / H
    q = q.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    k = k.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    v = v.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)

    # Calculate scores for self-attention
    # (B, H, S, d_k) x (B, H, d_k, S) -> (B, H, S, S)
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))

    # Causal mask and SoftMax
    # (B, H, S, S)
    att = att.masked_fill(self.bias[:,:,:S,:S] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)

    # Combine values by attention weights
    # (B, H, S, S) x (B, H, S, d_k) -> (B, H, S, d_k)
    y = att @ v

    # Re-assemble weighted values from all heads side by side
    # (B, H, S, d_k) -> (B, S, D)
    y = y.transpose(1, 2).contiguous().view(B, S, D)

    # Output projection
    # (B, S, D) x (D, D) -> (B, S, D)
    y = self.resid_dropout(self.c_proj(y))
    return y
```

Figure A.3.: Step 1, line 7 of the code, calculating Q, K, and V

Line 7 sends the input batch to `c_attn`, a linear layer with a single matrix with all three sets of weights concatenated, for efficiency. W_Q, W_K, and W_V each have shape (**D**, **D**), so (**D**, **3D**) when concatenated. The weight matrix is broadcast along the batch dimension when the input X is multiplied with it. Once split, each of the outputs Q, K, and V each has shape (**B**, **S**, **D**).

Note that in this data flow diagram, **S** edges are drawn shorter than **D** edges. Llama 3 70B has model dimension **D** = 8192, so this will be accurate when **S** < 8192. If the sequence were longer and we had **S** > 8192, then the **S** edges would be longer than **D** edges, though this static diagram cannot show the change.

Step 2: splitting Q, K, and V into heads

The second step in the attention calculation, on lines 11-13, splits the Q, K, and V tensors into heads and reorders the dimensions. Having multiple heads allows multiple query-key patterns to be matched for each token. Mathematically, this splitting and reshaping involves reinterpreting the last dimension — breaking **D** into **H** groups of size d_K — and transposing the dimensions so that the head dimension comes before the sequence dimension. This does not require any computation or data movement — just changing the metadata describing how to index each tensor. The tensors have changed from shape (**B**, **S**, **D**) to (**B**, **H**, **S**, d_K). In Figure A.4, dotted line arrows show the view operations that do the reshape on each of Q, K, and V.

If you trace back the individual heads, you will see they represent ranges of columns in the Q, K, and V tensors. And if you trace those back further, you will see that independent ranges of columns of the W_Q, W_K, and W_V weights multiply with X to form those column ranges. The

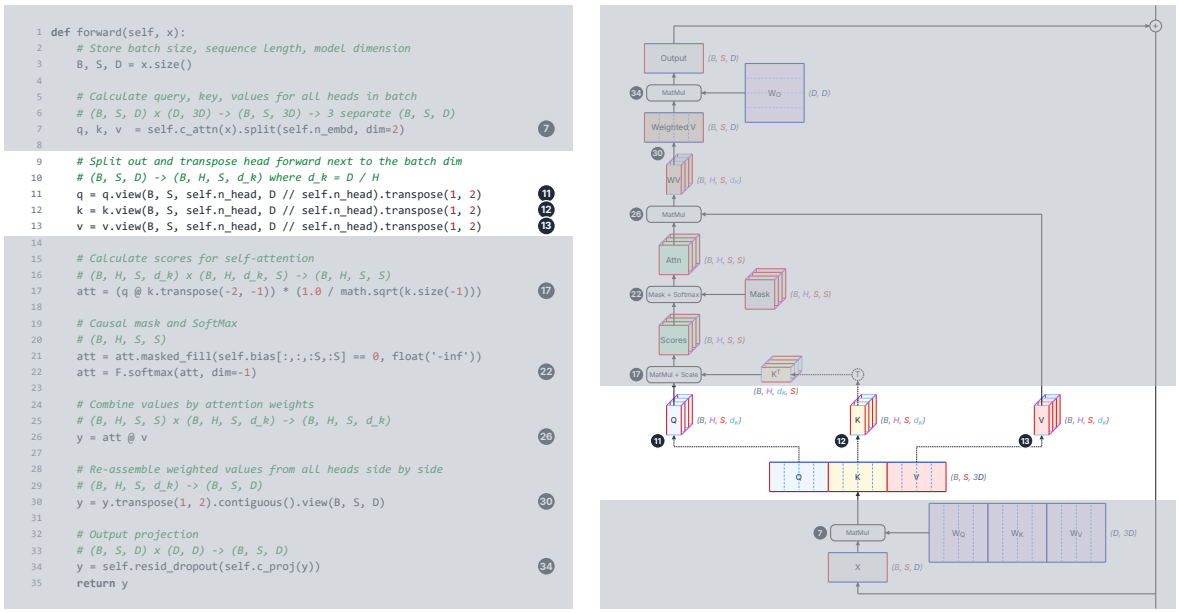


Figure A.4.: Step 2, lines 11-13 of the code, splitting Q, K, and V into heads

vertical dashed lines on the weight matrices and Q, K, and V tensors show the conceptual division of those tensors into heads, even though the explicit reshape doesn't happen until the next step.

Step 3: calculating and scaling similarity scores

Step 3 is calculating the similarity scores between queries and keys. We compute the raw attention scores by taking the dot product of each query with each key. In matrix notation, we have:

$$\text{Scores} = \frac{QK^T}{\sqrt{d_K}}$$

The division by $\sqrt{d_K}$ prevents the dot products from growing larger as the dimension d_K increases.

The multiplication is on line 17 and is highlighted in Figure A.5. The transpose is another view operation that changes the K tensor of shape **(B, H, S,** d_K**)** into the K^T tensor of shape **(B, H,** d_K**, S)**. The product QK^T multiplies tensors with shapes **(B, H, S,** d_K**)** and **(B, H,** d_K**, S)**, producing a scores tensor of shape **(B, H, S, S)**. For each batch sample and each head, the scores form a matrix with shape **(S, S)**, where entry (i, j) is the attention score between the query at token position i and the key at token position j.

Step 4: masking the scores and performing the SoftMax

Next, we apply a causal mask to the scores and perform a SoftMax to convert them to values that sum to 1. Figure A.6 highlights lines 21-22 and the associated portion of the data flow.

```python
def forward(self, x):
    # Store batch size, sequence length, model dimension
    B, S, D = x.size()

    # Calculate query, key, values for all heads in batch
    # (B, S, D) x (D, 3D) -> (B, S, 3D) -> 3 separate (B, S, D)
    q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)

    # Split out and transpose head forward next to the batch dim
    # (B, S, D) -> (B, H, S, d_k) where d_k = D / H
    q = q.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    k = k.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    v = v.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)

    # Calculate scores for self-attention
    # (B, H, S, d_k) x (B, H, d_k, S) -> (B, H, S, S)
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))

    # Causal mask and SoftMax
    # (B, H, S, S)
    att = att.masked_fill(self.bias[:,:,:S,:S] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)

    # Combine values by attention weights
    # (B, H, S, S) x (B, H, S, d_k) -> (B, H, S, d_k)
    y = att @ v

    # Re-assemble weighted values from all heads side by side
    # (B, H, S, d_k) -> (B, S, D)
    y = y.transpose(1, 2).contiguous().view(B, S, D)

    # Output projection
    # (B, S, D) x (D, D) -> (B, S, D)
    y = self.resid_dropout(self.c_proj(y))
    return y
```

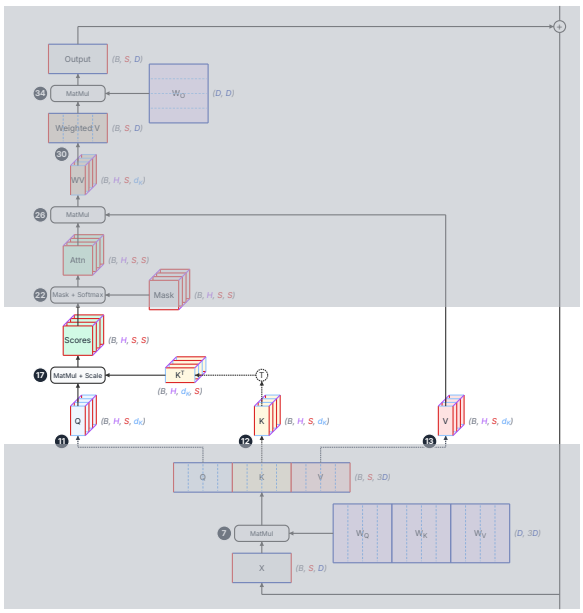


Figure A.5.: Step 3, line 17 of the code, calculating and scaling similarity scores

```python
def forward(self, x):
    # Store batch size, sequence length, model dimension
    B, S, D = x.size()

    # Calculate query, key, values for all heads in batch
    # (B, S, D) x (D, 3D) -> (B, S, 3D) -> 3 separate (B, S, D)
    q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)

    # Split out and transpose head forward next to the batch dim
    # (B, S, D) -> (B, H, S, d_k) where d_k = D / H
    q = q.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    k = k.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    v = v.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)

    # Calculate scores for self-attention
    # (B, H, S, d_k) x (B, H, d_k, S) -> (B, H, S, S)
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))

    # Causal mask and SoftMax
    # (B, H, S, S)
    att = att.masked_fill(self.bias[:,:,:S,:S] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)

    # Combine values by attention weights
    # (B, H, S, S) x (B, H, S, d_k) -> (B, H, S, d_k)
    y = att @ v

    # Re-assemble weighted values from all heads side by side
    # (B, H, S, d_k) -> (B, S, D)
    y = y.transpose(1, 2).contiguous().view(B, S, D)

    # Output projection
    # (B, S, D) x (D, D) -> (B, S, D)
    y = self.resid_dropout(self.c_proj(y))
    return y
```

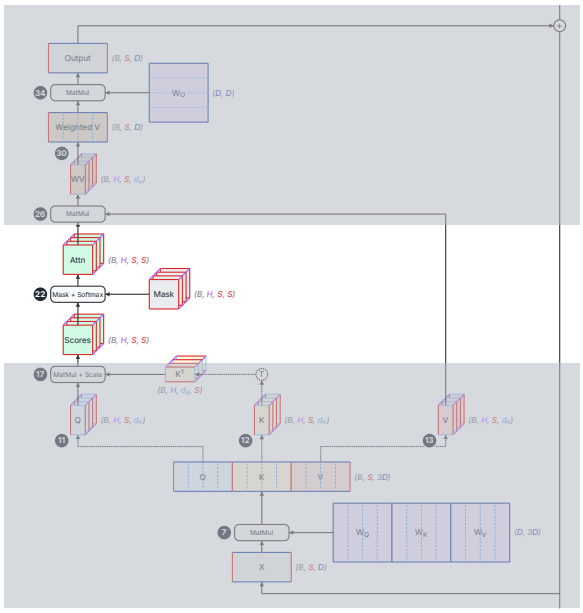


Figure A.6.: Step 4, lines 21-22 of the code, masking the scores and performing the SoftMax

Line 21 sets masked out entries to $-\infty$. These values become zeroes when the SoftMax is applied, on line 22.

All of these operations are working with **(B, H, S, S)** shaped tensors. When the sequence length is small, these tensors don't require much memory, but as **S** grows very large, these become enormous, since they increase quadratically with **S**. Section 6.1 discusses how efficient attention kernels avoid materializing the attention tensors in GPU memory.

Step 5: calculating weighted values

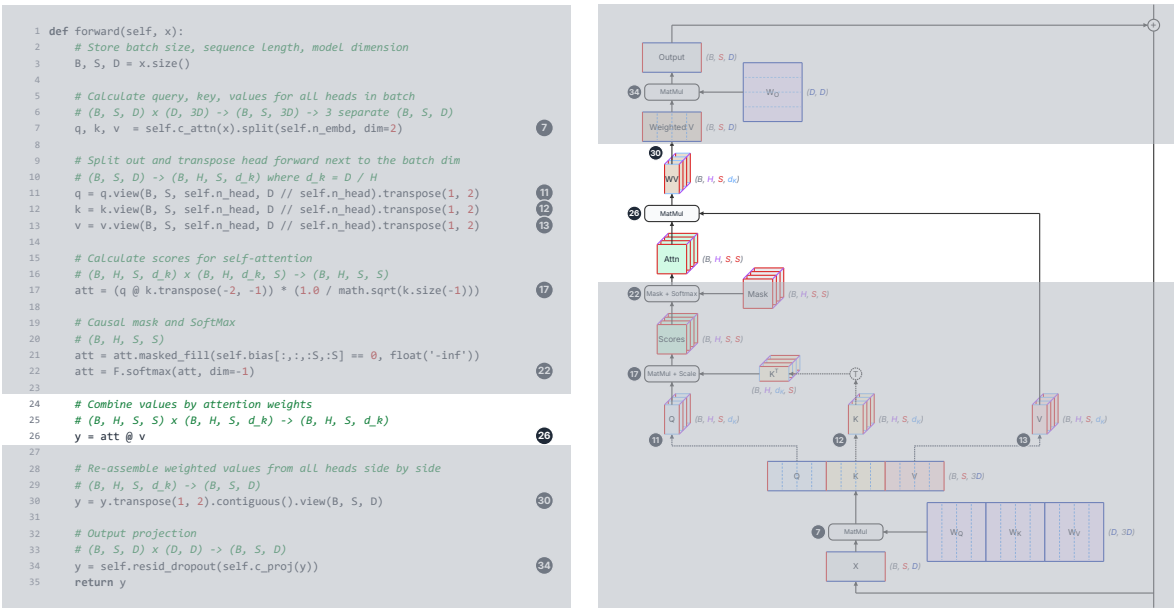


Figure A.7.: Step 5, line 26 of the code, calculating weighted values

The fifth step multiplies our attention pattern by the values, creating one weighted value vector for each batch sample, head, and query. Line 26 performs the matrix multiplication and is shown in Figure A.7. After the matrix multiplication, we have calculated:

$$\text{Weighted Values} = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_K}} + \text{mask}\right)V$$

The attention weights have shape **(B, H, S, S)** and V has shape **(B, H, S,** d_K**)**, so the result has shape **(B, H, S,** d_K**)**.

Step 6: reshaping the weighted values

Figure A.8 highlights step 6. The weighted values are reshaped from **(B, H, S,** d_K**)** to **(B, S, D)** by concatenating the **H** heads into the final dimension of length **D**. This happens on line 30 of the code. Note that this cannot be accomplished with a view, because the data

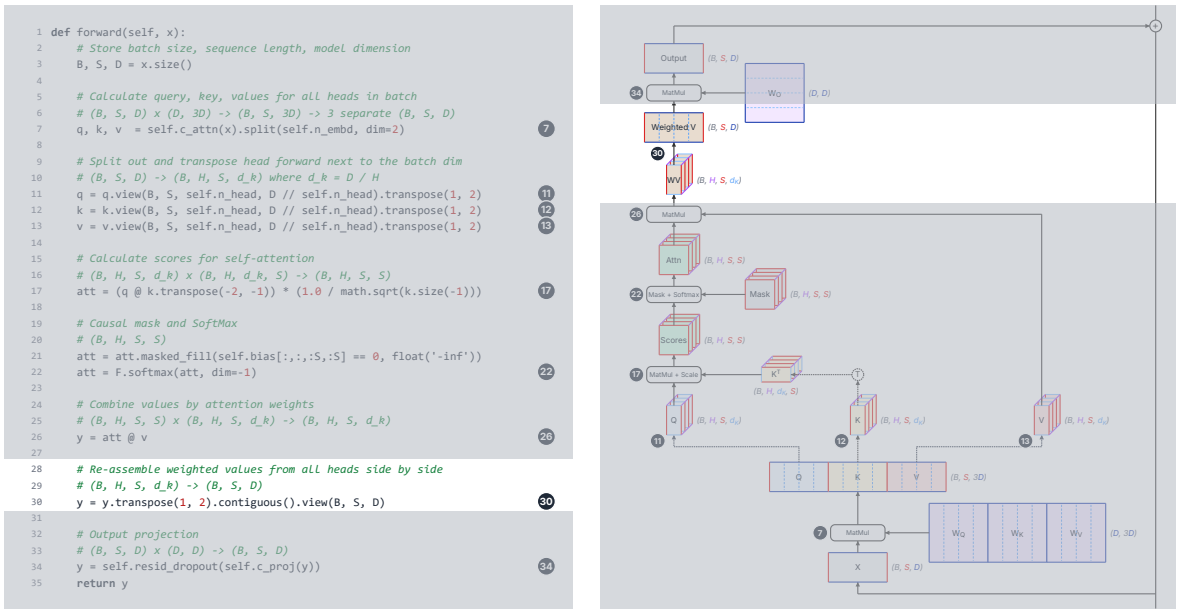


Figure A.8.: Step 6, line 30 of the code, reshaping the weighted values

for the heads of the same batch sample and token position are not contiguous. This reshape requires copying the data into a new, contiguous memory block.

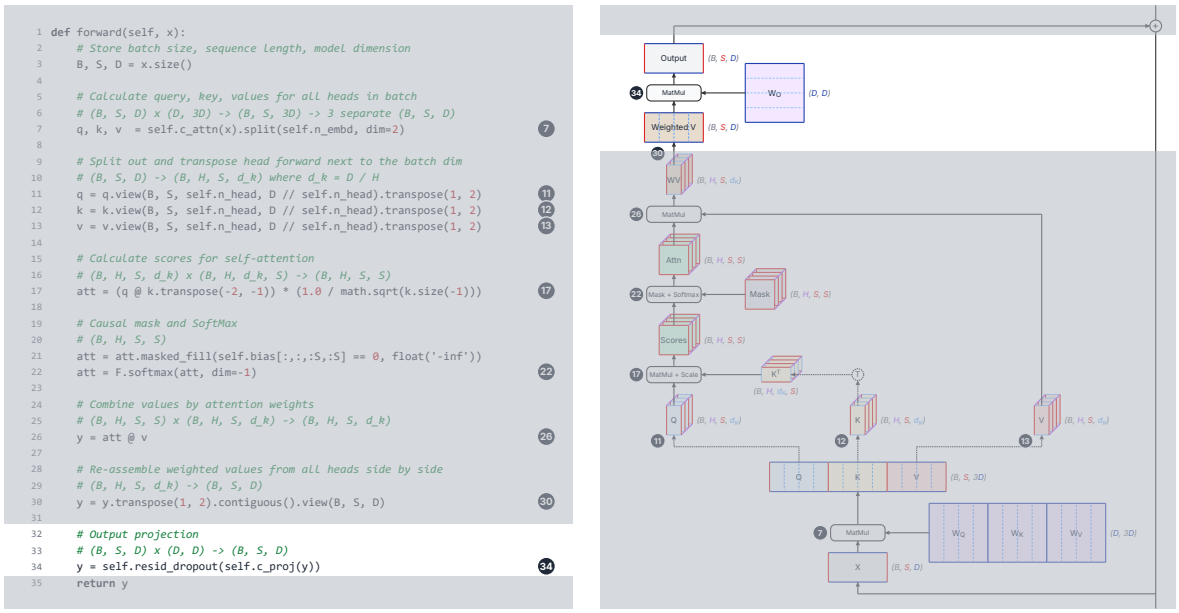
Step 7: calculating the final output

The final step creates the attention layer's output. Line 34 performs this using `c_proj`, a linear layer with weight matrix W_O (not shown). The weighted values tensor now has shape **(B, S, D)**. W_O has shape **(D, D)** and is broadcast along the batch dimension when the input to `c_proj` is multiplied with it. Note that because the weighted values had the output of attention heads concatenated into ranges of columns, implicitly ranges of rows of W_O act upon individual head values. This is indicated with the horizontal dashed lines on the W_O weight tensor.

The layer's final output has shape **(B, S, D)**. This is the same as the attention layer's input, so the layer does not modify the shape of the data it processes. This allows us to stack arbitrary numbers of layers without having to worry about dimensions.

Putting it all together

Now that we have traced each step in isolation, the data flow diagram in Figure A.2 can be read end-to-end as a single picture. The seven steps split naturally into three groups: the input and output projections (steps 1 and 7), the attention computation itself (steps 3 through 5), and the reshapes that bracket the attention block (steps 2 and 6). The reshapes do little or no real work — they cost only a metadata change or a single contiguous copy — while the projections and the attention block account for essentially all of the arithmetic.



```python
def forward(self, x):
    # Store batch size, sequence length, model dimension
    B, S, D = x.size()

    # Calculate query, key, values for all heads in batch
    # (B, S, D) x (D, 3D) -> (B, S, 3D) -> 3 separate (B, S, D)
    q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)

    # Split out and transpose head forward next to the batch dim
    # (B, S, D) -> (B, H, S, d_k) where d_k = D / H
    q = q.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    k = k.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)
    v = v.view(B, S, self.n_head, D // self.n_head).transpose(1, 2)

    # Calculate scores for self-attention
    # (B, H, S, d_k) x (B, H, d_k, S) -> (B, H, S, S)
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))

    # Causal mask and SoftMax
    # (B, H, S, S)
    att = att.masked_fill(self.bias[:,:,:S,:S] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)

    # Combine values by attention weights
    # (B, H, S, S) x (B, H, S, d_k) -> (B, H, S, d_k)
    y = att @ v

    # Re-assemble weighted values from all heads side by side
    # (B, H, S, d_k) -> (B, S, D)
    y = y.transpose(1, 2).contiguous().view(B, S, D)

    # Output projection
    # (B, S, D) x (D, D) -> (B, S, D)
    y = self.resid_dropout(self.c_proj(y))
    return y
```


Figure A.9.: Step 7, line 34 of the code, calculating the final output

A few observations are worth carrying forward as various chapters examine attention through the lens of inference optimization:

- The **(B, H, S, S)** attention matrix in the middle of the calculation grows quadratically with sequence length, and is the central memory pressure point that motivates fused attention kernels (Section 6.1).
- Each attention head operates on its own column range of W_Q, W_K, W_V, and on its own row range of W_O, with no cross-talk between heads until the output projection mixes them. This independence is what lets variants like multi-query and grouped-query attention (Section 4.4) share keys and values across heads without disturbing the rest of the pipeline.
- This column- and row-independence of slices of the weights also makes tensor parallelism (Section 7.3) easy to implement for attention layers by sharding attention heads across devices.
- During prefill, the queries, keys, and values all have the same sequence length S. During decode, there is only one query per batch sample, but the number of keys and values remains S because the key and value for the last token are appended to keys and values from prior tokens saved in the KV cache (Section 2.5); a version of the decode data flow is shown in Figure 2.12.
- The entire calculation broadcasts cleanly over the batch dimension. Since batch samples never interact with one another, it makes batched inference straightforward to parallelize on GPUs.

We hope this unified presentation of the concepts and the data flow clarifies precisely what is happening inside masked self-attention and gives a solid foundation for the optimization techniques in this book.

References

Adnan, Muhammad, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. "Keyformer: KV Cache Reduction Through Key Tokens Selection for Efficient Generative Inference." *MLSys*. https://proceedings.mlsys.org/paper_files/paper/2024/hash/48fecef47b19fe501d27d338b6d52582-Abstract-Conference.html.

Agrawal, Amey, Nitin Kedia, Ashish Panwar, et al. 2024. "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve." *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*. https://arxiv.org/abs/2403.02310.

Agrawal, Amey, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. "Sarathi: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills." *arXiv Preprint arXiv:2308.16369*. https://arxiv.org/abs/2308.16369.

Ainslie, Joshua, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints." *arXiv Preprint arXiv:2305.13245*. https://arxiv.org/abs/2305.13245.

Alammar, Jay. 2018. "The Illustrated Transformer." https://jalammar.github.io/illustrated-transformer/.

AMD ROCm Team. 2025. "The vLLM MoE Playbook: A Practical Guide to TP, DP, PP and Expert Parallelism." https://rocm.blogs.amd.com/software-tools-optimization/vllm-moe-guide/README.html.

Aminabadi, Reza Yazdani, Samyam Rajbhandari, Ammar Ahmad Awan, et al. 2022. "DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale." *arXiv Preprint arXiv:2207.00032*. https://arxiv.org/abs/2207.00032.

Anyscale. 2024a. "How Continuous Batching Enables 23x Throughput in LLM Inference While Reducing P50 Latency." https://www.anyscale.com/blog/continuous-batching-llm-inference.

Anyscale. 2024b. "LLMPerf: A Tool for Benchmarking LLM Inference." https://github.com/ray-project/llmperf.

Austin, Jacob, Sholto Douglas, Roy Frostig, et al. 2025. "How to Scale Your Model."

https://jax-ml.github.io/scaling-book/.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. "Layer Normalization." *arXiv Preprint arXiv:1607.06450*. https://arxiv.org/abs/1607.06450.

Bai, Yinmin, Peng Li, Hanyu Qin, et al. 2023. *Fast Distributed Inference Serving for Large Language Models*. https://arxiv.org/abs/2305.05920.

Banatt, Eryk. 2025. "Understanding Multi-Head Latent Attention." https://planetbanatt.net/articles/mla.html.

Beltagy, Iz, Matthew E. Peters, and Arman Cohan. 2020. "Longformer: The Long-Document Transformer." *arXiv Preprint arXiv:2004.05150*. https://arxiv.org/abs/2004.05150.

Bloem, Peter. 2019. "Transformers from Scratch." https://peterbloem.nl/blog/transformers.

Brandon, William, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. 2024. "Reducing Transformer Key-Value Cache Size with Cross-Layer Attention." *arXiv Preprint arXiv:2405.12981*. https://arxiv.org/abs/2405.12981.

Bycroft, Brendan. 2023. "LLM Visualization." https://bbycroft.net/llm.

Cai, Tianle, Yuhong Li, Zhengyang Geng, et al. 2024. "Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads." *arXiv Preprint arXiv:2401.10774*. https://arxiv.org/abs/2401.10774.

Cai, Zefan. 2024. "Awesome-LLM-KV-Cache." https://github.com/Zefan-Cai/Awesome-LLM-KV-Cache.

Casey, Matt. 2024. "LLM Distillation Demystified: A Complete Guide." https://snorkel.ai/blog/llm-distillation-demystified-a-complete-guide/.

Chen, Carol. 2022. "Transformer Inference Arithmetic." https://kipp.ly/transformer-inference-arithmetic/.

Chen, Charlie, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. "Accelerating Large Language Model Decoding with Speculative Sampling." *arXiv Preprint arXiv:2302.01318*. https://arxiv.org/abs/2302.01318.

Chen, Junda, Yonghao Zhuang, and Hao Zhang. 2025. "Disaggregated Inference: 18 Months Later." https://haoailab.com/blogs/distserve-retro/.

Chen, Lequn, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2023. "Punica: Multi-Tenant LoRA Serving." *arXiv Preprint arXiv:2310.18547*. https://arxiv.org/abs/2310.18547.

Chips and Cheese. 2023. "Nvidia's H100: Funny L2, and Tons of Bandwidth." https://chipsandcheese.com/2023/07/02/nvidias-h100-funny-l2-and-tons-of-bandwidth/.

Cho, Aeree, Grace C. Kim, Alexander Karpekov, et al. 2024. "Transformer Explainer: Interactive Learning of Text-Generative Models." https://poloclub.github.io/transformer-explainer/.

Chowdhery, Aakanksha, Sharan Narang, Jacob Devlin, et al. 2022. "PaLM: Scaling Language Modeling with Pathways." *arXiv Preprint arXiv:2204.02311*. https://arxiv.org/abs/2204.02311.

Dao, Tri. 2023. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning." *arXiv Preprint arXiv:2307.08691*. https://arxiv.org/abs/2307.08691.

Dao, Tri. 2024. "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-Precision." https://tridao.me/blog/2024/flash3/.

Dao, Tri, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness." *arXiv Preprint arXiv:2205.14135*. https://arxiv.org/abs/2205.14135.

Dao, Tri, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. *Flash-Decoding for Long-Context Inference*. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

Dao, Tri, Jay Shah, Ganesh Bikshandi, et al. 2025. *FlashAttention-4: Hardware-Friendly Attention on Hopper and Blackwell GPUs*. https://tridao.me/blog/2025/flash4/.

DeepSeek-AI. 2025. *FlashMLA: Efficient Multi-Head Latent Attention Decoding Kernels*. https://github.com/deepseek-ai/FlashMLA.

DeepSpeed Team. 2023. "DeepSpeed-FastGen: High-Throughput Text Generation for LLMs via MII and DeepSpeed-Inference." https://github.com/microsoft/DeepSpeed/blob/master/blogs/deepspeed-fastgen/README.md.

Dettmers, Tim et al. 2021. "Bitsandbytes: Accessible Large Language Models via k-Bit Quantization for PyTorch." https://github.com/bitsandbytes-foundation/bitsandbytes.

Dettmers, Tim. 2022. "LLM.int8() and Emergent Features." https://timdettmers.com/2022/08/17/llm-int8-and-emergent-features/.

Dong, Juechu, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2024. "Flex-Attention: A Programming Model for Generating Optimized Attention Kernels." *arXiv Preprint arXiv:2412.05496*. https://arxiv.org/abs/2412.05496.

Frantar, Elias, and Dan Alistarh. 2023. "SparseGPT: Massive Language Models Can Be

Accurately Pruned in One-Shot." *Proceedings of the 40th International Conference on Machine Learning*. https://arxiv.org/abs/2301.00774.

Frantar, Elias, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. "GPTQ: Accurate Post-Training Quantization for Generative Pre-Trained Transformers." *arXiv Preprint arXiv:2210.17323*. https://arxiv.org/abs/2210.17323.

Fu, Yichao, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. "Break the Sequential Dependency of LLM Inference Using Lookahead Decoding." *arXiv Preprint arXiv:2402.02057*. https://arxiv.org/abs/2402.02057.

Gloeckle, Fabian, Badr Youbi Idrissi, Baptiste Roziere, David Lopez-Paz, and Gabriel Synnaeve. 2024. "Better & Faster Large Language Models via Multi-Token Prediction." *arXiv Preprint arXiv:2404.19737*. https://arxiv.org/abs/2404.19737.

Gordić, Aleksa. 2023. "ELI5: FlashAttention." https://gordicaleksa.medium.com/eli5-flash-attention-5c44017022ad.

Grafana Labs. 2024. "Grafana Documentation." https://grafana.com/docs/grafana/latest/.

Grattafiori, Aaron, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. "The Llama 3 Herd of Models." *arXiv Preprint arXiv:2407.21783*. https://arxiv.org/abs/2407.21783.

Grootendorst, Maarten. 2024. "A Visual Guide to Quantization." https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization.

Gu, Albert, and Tri Dao. 2023. "Mamba: Linear-Time Sequence Modeling with Selective State Spaces." *arXiv Preprint arXiv:2312.00752*. https://arxiv.org/abs/2312.00752.

Gu, Albert, and Tri Dao. 2024. "GPUs Go Brrr." https://hazyresearch.stanford.edu/blog/2024-05-12-tk.

Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. 2015. "Distilling the Knowledge in a Neural Network." *arXiv Preprint arXiv:1503.02531*. https://arxiv.org/abs/1503.02531.

Holmes, Connor, Masahiro Tanaka, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, and Yuxiong He. 2024. "DeepSpeed-FastGen: High-Throughput Text Generation for LLMs via MII and DeepSpeed-Inference." *arXiv Preprint arXiv:2401.08671*. https://arxiv.org/abs/2401.08671.

Hong, Ke, Guohao Dai, Jiaming Xu, et al. 2023. "FlashDecoding++: Faster Large Language Model Inference on GPUs." *arXiv Preprint arXiv:2311.01282*. https://arxiv.org/abs/2311.01282.

Hu, Edward J., Yelong Shen, Phillip Wallis, et al. 2021. "LoRA: Low-Rank Adaptation of Large

Language Models." *arXiv Preprint arXiv:2106.09685*. https://arxiv.org/abs/2106.09685.

Hugging Face. 2024a. "Methods and Tools for Efficient Training on a Single GPU." https://huggingface.co/docs/transformers/en/perf_train_gpu_many.

Hugging Face. 2024b. "Quantization Concept Guide." https://huggingface.co/docs/transformers/en/quantization/concept_guide.

Jacobs, Sam Ade, Masahiro Tanaka, Chengming Zhang, et al. 2023. "DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models." *arXiv Preprint arXiv:2309.14509*. https://arxiv.org/abs/2309.14509.

JarvisLabs. 2025. "Speculative Decoding in vLLM: Complete Guide to Faster LLM Inference." https://docs.jarvislabs.ai/blog/speculative-decoding-vllm-faster-llm-inference.

Jiang, Albert Q., Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, et al. 2024. "Mixtral of Experts." *arXiv Preprint arXiv:2401.04088*. https://arxiv.org/abs/2401.04088.

Kadous, Waleed, Kyle Huang, Wendi Ding, Liguang Xie, Avnish Narayan, and Ricky Xu. 2023. "Reproducible Performance Metrics for LLM Inference." https://www.anyscale.com/blog/reproducible-performance-metrics-for-llm-inference.

Karpathy, Andrej. 2022. *nanoGPT*. https://github.com/karpathy/nanoGPT.

Karpathy, Andrej. 2023. "Let's Build GPT: From Scratch, in Code, Spelled Out." https://www.youtube.com/watch?v=kCc8FmEb1nY.

Kazemnejad, Amirhossein, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. 2023. "The Impact of Positional Encoding on Length Generalization in Transformers." *arXiv Preprint arXiv:2305.19466*. https://arxiv.org/abs/2305.19466.

Kimi Team, Yu Zhang, Zongyu Lin, et al. 2025. "Kimi Linear: An Expressive, Efficient Attention Architecture." *arXiv Preprint arXiv:2510.26692*. https://arxiv.org/abs/2510.26692.

Kumar, Tanishq, Tri Dao, and Avner May. 2026. "Speculative Speculative Decoding." *arXiv Preprint arXiv:2603.03251*. https://arxiv.org/abs/2603.03251.

Kwon, Woosuk, Zhuohan Li, Siyuan Zhuang, et al. 2023a. "Efficient Memory Management for Large Language Model Serving with PagedAttention." *arXiv Preprint arXiv:2309.06180*. https://arxiv.org/abs/2309.06180.

Kwon, Woosuk, Zhuohan Li, Siyuan Zhuang, et al. 2023b. "vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention." https://vllm.ai/blog/vllm.

Leviathan, Yaniv, Matan Kalman, and Yossi Matias. 2022. "Fast Inference from Transformers via Speculative Decoding." *arXiv Preprint arXiv:2211.17192*. https://arxiv.org/abs/2211.17192.

Leviathan, Yaniv, Matan Kalman, and Yossi Matias. 2024. "Looking Back at Speculative Decoding." https://research.google/blog/looking-back-at-speculative-decoding/.

Li, Dacheng, Rulin Shao, Anze Xie, et al. 2023. "DistFlashAttn: Distributed Memory-Efficient Attention for Long-Context LLMs Training." *arXiv Preprint arXiv:2310.03294*. https://arxiv.org/abs/2310.03294.

Li, Shenggui, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2021. "Sequence Parallelism: Long Sequence Training from System Perspective." *arXiv Preprint arXiv:2105.13120*. https://arxiv.org/abs/2105.13120.

Li, Yuhong, Yingbing Huang, Bowen Yang, et al. 2024. "SnapKV: LLM Knows What You Are Looking for Before Generation." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2404.14469.

Li, Zhuohan, Lianmin Zheng, Yinmin Zhong, et al. 2023. "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving." *arXiv Preprint arXiv:2302.11665*. https://arxiv.org/abs/2302.11665.

Lin, Ji, Jiaming Tang, Haotian Tang, et al. 2023. "AWQ: Activation-Aware Weight Quantization for LLM Compression and Acceleration." *arXiv Preprint arXiv:2306.00978*. https://arxiv.org/abs/2306.00978.

Liu, Aixin et al. 2024a. "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model." *arXiv Preprint arXiv:2405.04434*. https://arxiv.org/abs/2405.04434.

Liu, Aixin et al. 2024b. "DeepSeek-V3 Technical Report." *arXiv Preprint arXiv:2412.19437*. https://arxiv.org/abs/2412.19437.

Liu, Zirui, Jiayi Yuan, Hongye Jin, et al. 2024. "KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache." *Proceedings of the 41st International Conference on Machine Learning*. https://arxiv.org/abs/2402.02750.

LMSYS. 2024a. "Fast and Expressive LLM Inference with RadixAttention and SGLang." https://lmsys.org/blog/2024-01-17-sglang/.

LMSYS. 2024b. "SGLang V0.4: Zero-Overhead Batch Scheduler, Cache-Aware Load Balancer, and Faster Structured Outputs." https://www.lmsys.org/blog/2024-12-04-sglang-v0-4/.

LMSYS. 2025a. "Accelerating SGLang with Multiple Token Prediction." https://lmsys.org/

blog/2025-07-17-mtp/.

LMSYS. 2025b. "Deploying DeepSeek with PD Disaggregation and Large-Scale Expert Parallelism." https://www.lmsys.org/blog/2025-05-05-large-scale-ep/.

LMSYS. 2025c. "HiCache: Hierarchical KV Caching for SGLang." https://lmsys.org/blog/2025-09-10-sglang-hicache/.

LMSYS. 2026. "Chunked Pipeline Parallelism in SGLang." https://www.lmsys.org/blog/2026-01-15-chunked-pipeline/.

Miao, Xupeng, Gabriele Oliaro, Zhihao Zhang, et al. 2023. "Towards Efficient Generative Large Language Model Serving: A Survey from Algorithms to Systems." *arXiv Preprint arXiv:2312.15234*. https://arxiv.org/abs/2312.15234.

Microsoft Research. 2026. "Memento: Teaching LLMs to Manage Their Own Context." https://www.microsoft.com/en-us/research/articles/memento-teaching-llms-to-manage-their-own-context/.

Modal. 2024. "GPU Glossary." https://modal.com/gpu-glossary.

NVIDIA. 2021. "Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT." https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/.

NVIDIA. 2022. "NVIDIA Hopper Architecture in-Depth." https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

NVIDIA. 2023. "NVIDIA H100 Tensor Core GPU." https://www.nvidia.com/en-us/data-center/h100/.

NVIDIA. 2024a. "CUDA Programming Guide." https://docs.nvidia.com/cuda/cuda-programming-guide/.

NVIDIA. 2024b. "Deciding Model Sharding Strategy." https://nvidia.github.io/TensorRT-LLM/performance/performance-tuning-guide/deciding-model-sharding-strategy.html.

NVIDIA. 2024c. "Disaggregated Serving in TensorRT-LLM." https://nvidia.github.io/TensorRT-LLM/blogs/tech_blog/blog5_Disaggregated_Serving_in_TensorRT-LLM.html.

NVIDIA. 2024d. "KV Cache Reuse Optimizations in TensorRT-LLM." https://developer.nvidia.com/blog/introducing-new-kv-cache-reuse-optimizations-in-nvidia-tensorrt-llm/.

NVIDIA. 2024e. "Optimizing llama.cpp AI Inference with CUDA Graphs." https://developer.

nvidia.com/blog/optimizing-llama-cpp-ai-inference-with-cuda-graphs/.

NVIDIA. 2024f. "Streamlining AI Inference Performance and Deployment with NVIDIA TensorRT-LLM Chunked Prefill." https://developer.nvidia.com/blog/streamlining-ai-inference-performance-and-deployment-with-nvidia-tensorrt-llm-chunked-prefill/.

NVIDIA. 2024g. "TensorRT-LLM Documentation." https://nvidia.github.io/TensorRT-LLM/.

NVIDIA. 2024h. "TensorRT-LLM Expert Parallelism Documentation." https://nvidia.github.io/TensorRT-LLM/advanced/expert-parallelism.html.

NVIDIA. 2024i. "TensorRT-LLM GPT Attention Documentation." https://nvidia.github.io/TensorRT-LLM/advanced/gpt-attention.html.

NVIDIA. 2024j. "TensorRT-LLM KV Cache System Documentation." https://nvidia.github.io/TensorRT-LLM/latest/features/kvcache.html.

NVIDIA. 2024k. "TensorRT-LLM Parallelism Strategies Documentation." https://nvidia.github.io/TensorRT-LLM/features/parallelism.html.

NVIDIA. 2024l. "TensorRT-LLM Release Notes." https://nvidia.github.io/TensorRT-LLM/release-notes.html.

NVIDIA. 2024m. "TensorRT-LLM Speculative Decoding Documentation." https://nvidia.github.io/TensorRT-LLM/advanced/speculative-decoding.html.

Ong, Isaac, Amjad Almahairi, Vincent Wu, et al. 2024. "RouteLLM: An Open-Source Framework for Cost-Effective LLM Routing." https://www.lmsys.org/blog/2024-07-01-routellm/.

OpenTelemetry Authors. 2024. "OpenTelemetry Documentation." https://opentelemetry.io/docs/.

Patel, Pratyush, Esha Choukse, Chaojie Zhang, et al. 2024. "Splitwise: Efficient Generative LLM Inference Using Phase Splitting." *arXiv Preprint arXiv:2311.18677*. https://arxiv.org/abs/2311.18677.

Pope, Reiner, Sholto Douglas, Aakanksha Chowdhery, et al. 2022. "Efficiently Scaling Transformer Inference." *arXiv Preprint arXiv:2211.05102*. https://arxiv.org/abs/2211.05102.

Press, Ofir, Noah A. Smith, and Mike Lewis. 2021. "Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation." *arXiv Preprint arXiv:2108.12409*. https://arxiv.org/abs/2108.12409.

Prometheus Authors. 2024. "Prometheus Documentation." https://prometheus.io/docs/.

PyTorch Team. 2024. "A Hitchhiker's Guide to Speculative Decoding." https://pytorch.org/blog/hitchhikers-guide-speculative-decoding/.

Qin, Ruoyu, Zheming Li, Weiran He, et al. 2024. "Mooncake: A KVCache-Centric Disaggregated Architecture for LLM Serving." *arXiv Preprint arXiv:2407.00079*. https://arxiv.org/abs/2407.00079.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models Are Unsupervised Multitask Learners." *OpenAI Blog*. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Raschka, Sebastian. 2025. "LLM Architecture Gallery." https://sebastianraschka.com/llm-architecture-gallery/.

Raschka, Sebastian. 2026. "A Visual Guide to Attention Variants in Modern LLMs." https://magazine.sebastianraschka.com/p/visual-attention-variants.

Red Hat. 2025. "Why vLLM Is the Best Choice for AI Inference Today." https://developers.redhat.com/articles/2025/10/30/why-vllm-best-choice-ai-inference-today.

Rush, Alexander. 2018. "The Annotated Transformer." https://nlp.seas.harvard.edu/annotated-transformer/.

Sanderson, Grant. 2024. "Attention in Transformers, Visually Explained." https://www.3blue1brown.com/lessons/attention.

Sanseviero, Omar, Lewis Tunstall, Philipp Schmid, Sourab Mangrulkar, Younes Belkada, and Pedro Cuenca. 2023. "Mixture of Experts Explained." https://huggingface.co/blog/moe.

SGLang Team. 2024. "SGLang Documentation." https://docs.sglang.ai/.

Shah, Jay, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-Precision." *arXiv Preprint arXiv:2407.08608*. https://arxiv.org/abs/2407.08608.

Shaw, Robert, and Michael Goin. 2023. "SparseGPT: Remove 100 Billion Parameters for Free." https://developers.redhat.com/articles/2023/03/21/sparsegpt-remove-100-billion-parameters-free.

Shazeer, Noam. 2019. "Fast Transformer Decoding: One Write-Head Is All You Need." *arXiv Preprint arXiv:1911.02150*. https://arxiv.org/abs/1911.02150.

Shazeer, Noam. 2020. "GLU Variants Improve Transformer." *arXiv Preprint arXiv:2002.05202*. https://arxiv.org/abs/2002.05202.

Sheng, Ying, Shiyi Cao, Dacheng Li, et al. 2023. "S-LoRA: Serving Thousands of Concurrent LoRA Adapters." *arXiv Preprint arXiv:2311.03285*. https://arxiv.org/abs/2311.03285.

Spheron. 2025. "vLLM Vs TensorRT-LLM Vs SGLang: Benchmarks on H100." https://www.spheron.network/blog/vllm-vs-tensorrt-llm-vs-sglang-benchmarks/.

SqueezeBits. 2025. "vLLM Vs TensorRT-LLM #4: Which Scheduler Wins?" https://blog.squeezebits.com/vllm-vs-tensorrtllm-4-which-scheduler-wins--33083.

Stern, Mitchell, Noam Shazeer, and Jakob Uszkoreit. 2018. "Blockwise Parallel Decoding for Deep Autoregressive Models." *arXiv Preprint arXiv:1811.03115*. https://arxiv.org/abs/1811.03115.

Su, Jianlin, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2021. "RoFormer: Enhanced Transformer with Rotary Position Embedding." *arXiv Preprint arXiv:2104.09864*. https://arxiv.org/abs/2104.09864.

Sun, Biao, Ziming Huang, Hanyu Chen, et al. 2024. "Llumnix: Dynamic Scheduling for Large Language Model Serving." *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*. https://arxiv.org/abs/2406.03243.

Sun, Mingjie, Zhuang Liu, Anna Bair, and J. Zico Kolter. 2024. "A Simple and Effective Pruning Approach for Large Language Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2306.11695.

Tao, Chaofan, Qian Jia, Longxu Dou, et al. 2024. "Scaling Laws with Vocabulary: Larger Models Deserve Larger Vocabularies." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2407.13623.

Thomas, Derek, Diego Maniloff, and David Holtz. 2024. "TGI Multi-LoRA: Deploy Once, Serve 30 Models." https://huggingface.co/blog/multi-lora-serving.

Tillet, Philippe, H. T. Kung, and David Cox. 2019. "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations." *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*. https://doi.org/10.1145/3315508.3329973.

Touvron, Hugo, Thibaut Lavril, Gautier Izacard, et al. 2023. "LLaMA: Open and Efficient Foundation Language Models." *arXiv Preprint arXiv:2302.13971*. https://arxiv.org/abs/2302.13971.

Touvron, Hugo, Louis Martin, Kevin Stone, et al. 2023. "Llama 2: Open Foundation and Fine-

Tuned Chat Models." *arXiv Preprint arXiv:2307.09288*. https://arxiv.org/abs/2307.09288.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. 2017. "Attention Is All You Need." *arXiv Preprint arXiv:1706.03762*. https://arxiv.org/abs/1706.03762.

Verma, Shashank, and Neal Vaidya. 2023. "Mastering LLM Techniques: Inference Optimization." https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/.

vLLM Team. 2024a. "Attention Backend Feature Support." https://docs.vllm.ai/en/latest/design/attention_backends/.

vLLM Team. 2024b. "Benchmark CLI." https://docs.vllm.ai/en/latest/benchmarking/cli/.

vLLM Team. 2024c. "Fusion Torch.compile Passes." https://docs.vllm.ai/en/latest/design/fusions/.

vLLM Team. 2024d. "Quantized KV Cache." https://docs.vllm.ai/en/latest/features/quantization/quantized_kvcache/.

vLLM Team. 2024e. "vLLM Documentation." https://docs.vllm.ai/.

vLLM Team. 2025a. "Disaggregated Prefilling (Experimental)." https://docs.vllm.ai/en/latest/features/disagg_prefill/.

vLLM Team. 2025b. "Inside vLLM: Anatomy of a High-Throughput LLM Inference System." https://vllm.ai/blog/anatomy-of-vllm.

vLLM Team. 2025c. "vLLM Large Scale Serving: DeepSeek at 2.2k Tok/s/H200 with Wide-EP." https://vllm.ai/blog/large-scale-serving.

vLLM Team. 2025d. "vLLM Router: A High-Performance and Prefill/Decode Aware Load Balancer for Large-Scale Serving." https://vllm.ai/blog/vllm-router-release.

vLLM Team. 2025e. "vLLM V1: A Major Upgrade to vLLM's Core Architecture." https://vllm.ai/blog/v1-alpha-release.

vLLM Team. 2026. "vLLM Triton Attention Backend Deep Dive." https://vllm.ai/blog/vllm-triton-backend-deep-dive.

Wan, Zhongwei, Xin Wang, Che Liu, et al. 2024. "Efficient Large Language Models: A Survey." *Transactions on Machine Learning Research*. https://arxiv.org/abs/2312.03863.

Weng, Lilian. 2021. "How to Train Really Large Models on Many GPUs?" https://lilianweng.github.io/posts/2021-09-25-train-large/.

Weng, Lilian. 2023a. "Large Transformer Model Inference Optimization." https://lilianweng.github.io/posts/2023-01-10-inference-optimization/.

Weng, Lilian. 2023b. "The Transformer Family Version 2.0." https://lilianweng.github.io/posts/2023-01-27-the-transformer-family-v2/.

Williams, Samuel, Andrew Waterman, and David Patterson. 2009. "Roofline: An Insightful Visual Performance Model for Multicore Architectures." *Communications of the ACM* 52 (4): 65–76. https://dl.acm.org/doi/pdf/10.1145/1498765.1498785.

Wu, Bingyang, Yinmin Zhong, Zili Zhang, et al. 2023. "Fast Distributed Inference Serving for Large Language Models." *arXiv Preprint arXiv:2305.05920*. https://arxiv.org/abs/2305.05920.

Xia, Heming et al. 2024. "Unlocking Efficiency in Large Language Model Inference: A Comprehensive Survey of Speculative Decoding." *Findings of the Association for Computational Linguistics: ACL*. https://arxiv.org/abs/2401.07851.

Xiao, Guangxuan, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models." *arXiv Preprint arXiv:2211.10438*. https://arxiv.org/abs/2211.10438.

Xu, Xiaohan, Ming Li, Chongyang Tao, et al. 2024. "A Survey on Knowledge Distillation of Large Language Models." *arXiv Preprint arXiv:2402.13116*. https://arxiv.org/abs/2402.13116.

Yang, An, Anfeng Li, Baosong Yang, et al. 2025. "Qwen3 Technical Report." *arXiv Preprint arXiv:2505.09388*. https://arxiv.org/abs/2505.09388.

Yang, Nan, Tao Ge, Liang Wang, et al. 2023. "Inference with Reference: Lossless Acceleration of Large Language Models." *arXiv Preprint arXiv:2304.04487*. https://arxiv.org/abs/2304.04487.

Ye, Zihao, Lianmin Zheng, Lequn Chen, et al. 2025. "FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving." *arXiv Preprint arXiv:2501.01005*. https://arxiv.org/abs/2501.01005.

Yu, Gyeong-In, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. "Orca: A Distributed Serving System for Transformer-Based Generative Models." *OSDI*. https://www.usenix.org/conference/osdi22/presentation/yu.

Yuan, Jingyang et al. 2025. "Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention." *arXiv Preprint arXiv:2502.11089*. https://arxiv.org/abs/2502.11089.

Yuan, Zhihang, Yuzhang Shang, Yang Zhou, et al. 2024. "LLM Inference Unveiled: Survey and Roofline Model Insights." *arXiv Preprint arXiv:2402.16363*. https://arxiv.org/abs/2402.16363.

Zandieh, Amir, Majid Daliri, Majid Hadian, and Vahab Mirrokni. 2025. "TurboQuant: Online Vector Quantization with Near-Optimal Distortion Rate." *ICLR*. https://arxiv.org/abs/2504.19874.

Zhang, Biao, and Rico Sennrich. 2019. "Root Mean Square Layer Normalization." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/1910.07467.

Zhang, Zhenyu, Ying Sheng, Tianyi Zhou, et al. 2023. "H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2306.14048.

Zhao, Cen, Xiaodong Wang, and Jianyu Huang. 2025. "Scaling LLM Inference: Innovations in Tensor Parallelism, Context Parallelism, and Expert Parallelism." https://engineering.fb.com/2025/10/17/ai-research/scaling-llm-inference-innovations-tensor-parallelism-context-parallelism-expert-parallelism/.

Zheng, Lianmin, Liangsheng Yin, Zhiqiang Xie, et al. 2023. "SGLang: Efficient Execution of Structured Language Model Programs." *arXiv Preprint arXiv:2312.07104*. https://arxiv.org/abs/2312.07104.

Zhong, Yinmin, Shengyu Liu, Junda Chen, et al. 2024. "DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving." *arXiv Preprint arXiv:2401.09670*. https://arxiv.org/abs/2401.09670.