CHAPTER 9

Inference Optimization

New models come and go, but one thing will always remain relevant: making them better, cheaper, and faster. Up until now, the book has discussed various techniques for making models better. This chapter focuses on making them faster and cheaper.

No matter how good your model is, if it's too slow, your users might lose patience, or worse, its predictions might become useless—imagine a next-day stock price prediction model that takes two days to compute each outcome. If your model is too expensive, its return on investment won't be worth it.

Inference optimization can be done at the model, hardware, and service levels. At the model level, you can reduce a trained model's size or develop more efficient architectures, such as one without the computation bottlenecks in the attention mechanism often used in transformer models. At the hardware level, you can design more powerful hardware.

The inference service runs the model on the given hardware to accommodate user requests. It can incorporate techniques that optimize models for specific hardware. It also needs to consider usage and traffic patterns to efficiently allocate resources to reduce latency and cost.

Because of this, inference optimization is an interdisciplinary field that often sees collaboration among model researchers, application developers, system engineers, compiler designers, hardware architects, and even data center operators.

This chapter discusses bottlenecks for AI inference and techniques to overcome them. It'll focus mostly on optimization at the model and service levels, with an overview of AI accelerators.

This chapter also covers performance metrics and trade-offs. Sometimes, a technique that speeds up a model can also reduce its cost. For example, reducing a model's precision makes it smaller and faster. But often, optimization requires trade-offs. For example, the best hardware might make your model run faster but at a higher cost.

Given the growing availability of open source models, more teams are building their own inference services. However, even if you don't implement these inference optimization techniques, understanding these techniques will help you evaluate inference services and frameworks. If your application's latency and cost are hurting you, read on. This chapter might help you diagnose the causes and potential solutions.

Understanding Inference Optimization

There are two distinct phases in an AI model's lifecycle: training and inference. Training refers to the process of building a model. Inference refers to the process of using a model to compute an output for a given input.[1] Unless you train or finetune a model, you'll mostly need to care about inference.[2]

This section starts with an overview of inference that introduces a shared vocabulary to discuss the rest of the chapter. If you're already familiar with these concepts, feel free to skip to the section of interest.

Inference Overview

In production, the component that runs model inference is called an inference server. It hosts the available models and has access to the necessary hardware. Based on requests from applications (e.g., user prompts), it allocates resources to execute the appropriate models and returns the responses to users. An inference server is part of a broader inference service, which is also responsible for receiving, routing, and possibly preprocessing requests before they reach the inference server. A visualization of a simple inference service is shown in Figure 9-1.

[1] As discussed in Chapter 7, inference involves the forward pass while training involves both the forward and backward passes.

[2] A friend, Mark Saroufim, pointed me to an interesting relationship between a model's training cost and inference cost. Imagine you're a model provider. Let T be the total training cost, p be the cost you're charging per inference, and N be the number of inference calls you can sell. Developing a model only makes sense if the money you can recover from inference for a model is more than its training cost, i.e., $T <= p \times N$. The more a model is used in production, the more model providers can reduce inference cost. However, this doesn't apply for third-party API providers who sell inference calls on top of open source models.

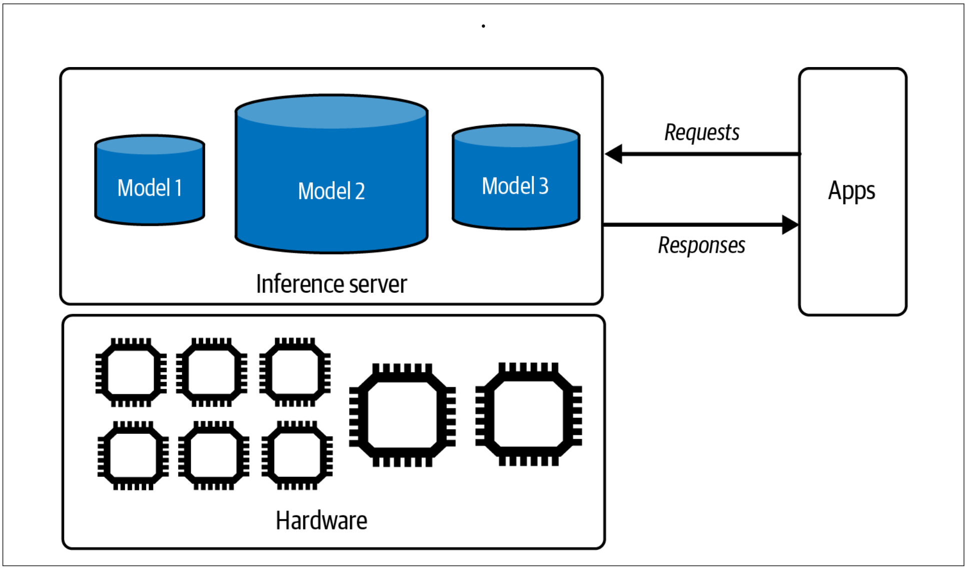


Figure 9-1. A simple inference service.

Model APIs like those provided by OpenAI and Google are inference services. If you use one of these services, you won't be implementing most of the techniques discussed in this chapter. However, if you host a model yourself, you'll be responsible for building, optimizing, and maintaining its inference service.

Computational bottlenecks

Optimization is about identifying bottlenecks and addressing them. For example, to optimize traffic, city planners might identify congestion points and take measures to alleviate congestion. Similarly, an inference server should be designed to address the computational bottlenecks of the inference workloads it serves. There are two main computational bottlenecks, *compute-bound* and *memory bandwidth-bound*:

Compute-bound
: This refers to tasks whose time-to-complete is determined by the computation needed for the tasks. For example, password decryption is typically compute-bound due to the intensive mathematical calculations required to break encryption algorithms.

Memory bandwidth-bound
: These tasks are constrained by the data transfer rate within the system, such as the speed of data movement between memory and processors. For example, if you store your data in the CPU memory and train a model on GPUs, you have to move data from the CPU to the GPU, which can take a long time. This can be

shortened as bandwidth-bound. In literature, memory bandwidth-bound is often referred to as memory-bound.

Terminology Ambiguity: Memory-Bound Versus Bandwidth-Bound

Memory-bound is also used by some people to refer to tasks whose time-to-complete is constrained by memory capacity instead of memory bandwidth. This occurs when your hardware doesn't have sufficient memory to handle the task, for example, if your machine doesn't have enough memory to store the entire internet. This memory is often manifested in the error recognizable by engineers everywhere: OOM, out-of-memory.[3]

However, this situation can often be mitigated by splitting your task into smaller pieces. For example, if you're constrained by GPU memory and cannot fit an entire model into the GPU, you can split the model across GPU memory and CPU memory. This splitting will slow down your computation because of the time it takes to transfer data between the CPU and GPU. However, if data transfer is fast enough, this becomes less of an issue. Therefore, the memory capacity limitation is actually more about memory bandwidth.

The concepts of compute-bound or memory bandwidth-bound were introduced in the paper "Roofline" (Williams et al., 2009).[4] Mathematically, an operation can be classified as compute-bound or memory bandwidth-bound based on its *arithmetic intensity*, which is the number of arithmetic operations per byte of memory access. Profiling tools like NVIDIA Nsight will show you a roofline chart to tell you whether your workload is compute-bound or memory bandwidth-bound, as shown in Figure 9-2. This chart is a *roofline* chart because it resembles a roof. Roofline charts are common in hardware performance analyses.

Different optimization techniques aim to mitigate different bottlenecks. For example, a compute-bound workload might be sped up by spreading it out to more chips or by leveraging chips with more computational power (e.g., a higher FLOP/s number). A memory bandwidth-bound workload might be sped up by leveraging chips with higher bandwidth.

[3] Anecdotally, I find that people coming from a system background (e.g., optimization engineers and GPU engineers) use *memory-bound* to refer to *bandwidth-bound*, and people coming from an AI background (e.g., ML and AI engineers) use to memory-bound to refer to memory capacity-bound.

[4] The Roofline paper uses the term memory-bound to refer to memory-bandwidth bound.

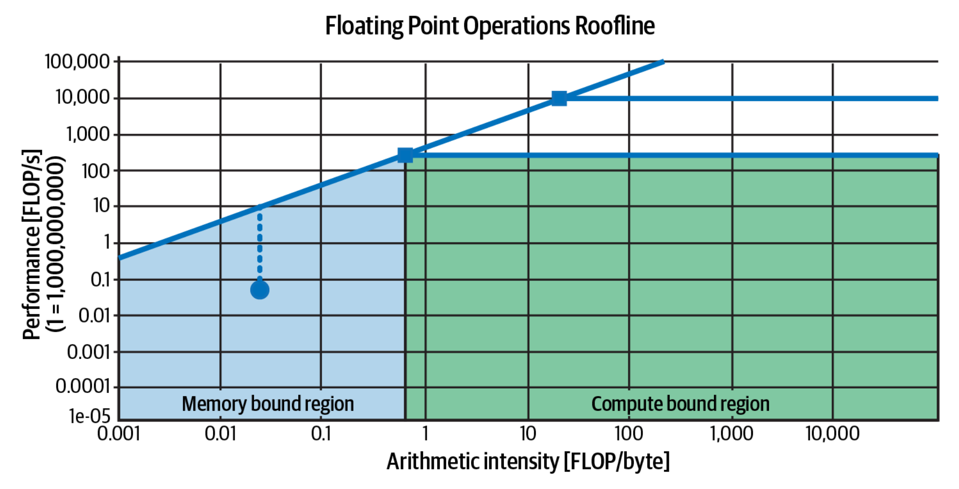


Figure 9-2. The roofline chart can help you visualize whether an operation is compute-bound or memory bandwidth-bound. This graph is on a log scale.

Different model architectures and workloads result in different computational bottlenecks. For example, inference for image generators like Stable Diffusion is typically compute-bound, whereas inference for autoregression language models is typically memory bandwidth-bound.

As an illustration, let's look into language model inference. Recall from Chapter 2 that inference for a transformer-based language model consists of two steps, prefilling and decoding:

Prefill

The model processes the input tokens in parallel.[5] How many tokens can be processed at once is limited by the number of operations your hardware can execute in a given time. Therefore, prefilling is *compute-bound*.

Decode

The model generates one output token at a time. At a high level, this step typically involves loading large matrices (e.g., model weights) into GPUs, which is limited by how quickly your hardware can load data into memory. Decoding is, therefore, *memory bandwidth-bound*.

Figure 9-3 visualizes prefilling and decoding.

[5] Prefilling effectively populates the initial KV cache for the transformer model.

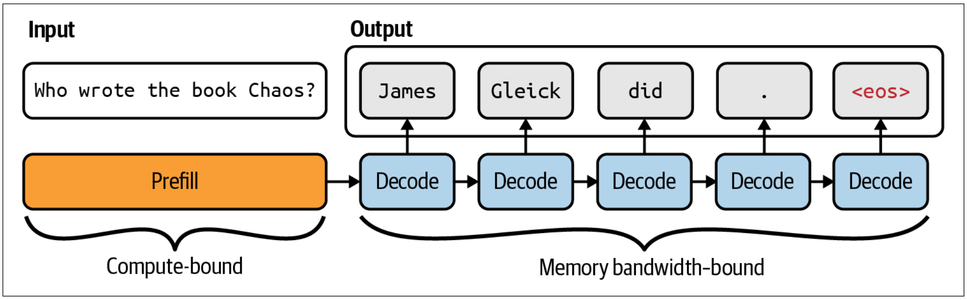


Figure 9-3. Autoregressive language models follow two steps for inference: prefill and decode. `<eos>` *denotes the end of the sequence token.*

Because prefill and decode have different computational profiles, they are often decoupled in production with separate machines. This technique will be discussed "Inference Service Optimization" on page 440.

The factors that affect the amount of prefilling and decoding computation in an LLM inference server, and therefore its bottlenecks, include context length, output length, and request batching strategies. Long context typically results in a memory bandwidth-bound workload, but clever optimization techniques, such as those discussed later in this chapter, can remove this bottleneck.

As of this writing, due to the prevalence of the transformer architecture and the limitations of the existing accelerator technologies, many AI and data workloads are memory bandwidth-bound. However, future software and hardware advancements will be able to make AI and data workloads compute-bound.

Online and batch inference APIs

Many providers offer two types of inference APIs, online and batch:

- Online APIs optimize for latency. Requests are processed as soon as they arrive.
- Batch APIs optimize for cost. If your application doesn't have strict latency requirements, you can send them to batch APIs for more efficient processing. Higher latency allows a broader range of optimization techniques, including batching requests together and using cheaper hardware. For example, as of this writing, both Google Gemini and OpenAI offer batch APIs at a 50% cost

reduction and significantly higher turnaround time, i.e., in the order of hours instead of seconds or minutes.[6]

Online APIs might still batch requests together as long as it doesn't significantly impact latency, as discussed in "Batching" on page 440. The only real difference is that an online API focuses on lower latency, whereas a batch API focuses on higher throughput.

Customer-facing use cases, such as chatbots and code generation, typically require lower latency, and, therefore, tend to use online APIs. Use cases with less stringent latency requirements, which are ideal for batch APIs, include the following:

- Synthetic data generation
- Periodic reporting, such as summarizing Slack messages, sentiment analysis of brand mentions on social media, and analyzing customer support tickets
- Onboarding new customers who require processing of all their uploaded documents
- Migrating to a new model that requires reprocessing of all the data
- Generating personalized recommendations or newsletters for a large customer base
- Knowledge base updates by reindexing an organization's data

APIs usually return complete responses by default. However, with autoregressive decoding, it can take a long time for a model to complete a response, and users are impatient. Many online APIs offer *streaming mode*, which returns each token as it's generated. This reduces the time the users have to wait until the first token. The downside of this approach is that you can't score a response before showing it to users, increasing the risk of users seeing bad responses. However, you can still retrospectively update or remove a response as soon as the risk is detected.

[6] If you run an inference service, separating your inference APIs into online and batch can help you prioritize latency for requests where latency matters the most. Let's say that your inference server can serve only a maximum of X requests/second without latency degradation, you have to serve Y requests/second, and Y is larger than X. In an ideal world, users with less-urgent requests can send their requests to the batch API, so that your service can focus on processing the online API requests first.

A batch API for foundation models differs from batch inference for traditional ML. In traditional ML:

- Online inference means that predictions are computed *after* requests have arrived.
- Batch inference means that predictions are precomputed *before* requests have arrived.

Precomption is possible for use cases with finite and predictable inputs like recommendation systems, where recommendations can be generated for all users in advance. These precomputed predictions are fetched when requests arrive, e.g., when a user visits the website. However, with foundation model use cases where the inputs are open-ended, it's hard to predict all user prompts.[7]

Inference Performance Metrics

Before jumping into optimization, it's important to understand what metrics to optimize for. From the user perspective, the central axis is latency (response quality is a property of the model itself, not of the inference service). However, application developers must also consider throughput and utilization as they determine the cost of their applications.

Latency, TTFT, and TPOT

Latency measures the time from when users send a query until they receive the complete response. For autoregressive generation, especially in the streaming mode, the overall latency can be broken into several metrics:

Time to first token

TTFT measures how quickly the first token is generated after users send a query. It corresponds to the duration of the prefill step and depends on the input's length. Users might have different expectations for TTFT for different applications. For example, for conversational chatbots, the TTFT should be instantaneous.[8] However, users might be willing to wait longer to summarize long documents.

[7] As discussed in "Prompt caching" on page 443, it's common to know in advance the system prompt of an application. It's just the exact user queries that are hard to predict.

[8] In the early days of chatbots, some people complained about chatbots responding too fast, which seemed unnatural. See "Lufthansa Delays Chatbot's Responses to Make It More 'Human'" (Ry Crozier, iTnews, May 2017). However, as people become more familiar with chatbots, this is no longer the case.

Time per output token

TPOT measures how quickly each output token is generated after the first token. If each token takes 100 ms, a response of 1,000 tokens will take 100 s.

In the streaming mode, where users read each token as it's generated, TPOT should be faster than human reading speed but doesn't have to be much faster. A very fast reader can read 120 ms/token, so a TPOT of around 120 ms, or 6–8 tokens/second, is sufficient for most use cases.

Time between tokens and inter-token latency

Variations of this metric include *time between tokens (TBT)* and i*nter-token latency (ITL)*.[9] Both measure the time between output tokens.

The total latency will equal `TTFT + TPOT × (number of output tokens)`.

Two applications with the same total latency can offer different user experiences with different TTFT and TPOT. Would your users prefer instant first tokens with a longer wait between tokens, or would they rather wait slightly longer for the first tokens but enjoy faster token generation afterward? User studies will be necessary to determine the optimal user experience. Reducing TTFT at the cost of higher TPOT is possible by shifting more compute instances from decoding to prefilling and vice versa.[10]

It's important to note that the TTFT and TPOT values observed by users might differ from those observed by models, especially in scenarios involving CoT (chain-of-thought) or agentic queries where models generate intermediate steps not shown to users. Some teams use the metric *time to publish* to make it explicit that it measures time to the first token users see.

Consider the scenario where, after a user sends a query, the model performs the following steps:

1. Generate a plan, which consists of a sequence of actions. This plan isn't shown to the user.
2. Take actions and log their outputs. These outputs aren't shown to the user.
3. Based on these outputs, generate a final response to show the user.

[9] Time between tokens (TBT) is used by LinkedIn and inter-token latency (ITL) is used by NVIDIA.

[10] An experiment by Anyscale shows that 100 input tokens have approximately the same impact on the overall latency as a single output token.

From the model's perspective, the first token is generated in step 1. This is when the model internally begins its token generation process. The user, however, only sees the first token of the final output generated in step 3. Thus, from their perspective, TTFT is much longer.

Because latency is a distribution, the average can be misleading. Imagine you have 10 requests whose TTFT values are 100 ms, 102 ms, 100 ms, 100 ms, 99 ms, 104 ms, 110 ms, 90 ms, 3,000 ms, 95 ms. The average TTFT value is 390 ms, which makes your inference service seem slower than it is. There might have been a network error that slowed down one request or a particularly long prompt that took a much longer time to prefill. Either way, you should investigate. With a large volume of requests, outliers that skew the average latency are almost inevitable.

It's more helpful to look at latency in percentiles, as they tell you something about a certain percentage of your requests. The most common percentile is the 50th percentile, abbreviated as p50 (median). If the median is 100 ms, half of the requests take longer than 100 ms to generate the first token, and half take less than 100 ms. Percentiles also help you discover outliers, which might be symptoms of something wrong. Typically, the percentiles you'll want to look at are p90, p95, and p99. It's also helpful to plot TTFT values against inputs' lengths.

Throughput and goodput

Throughput measures the number of output tokens per second an inference service can generate across all users and requests.

Some teams count both input and output tokens in throughput calculation. However, since processing input tokens (prefilling) and generating output tokens (decoding) have different computational bottlenecks and are often decoupled in modern inference servers, input and output throughput should be counted separately. When throughput is used without any modifier, it usually refers to output tokens.

Throughput is typically measured as tokens/s (TPS). If you serve multiple users, tokens/s/user is also used to evaluate how the system scales with more users.

Throughput can also be measured as the number of *completed* requests during a given time. Many applications use requests per second (RPS). However, for applications built on top of foundation models, a request might take seconds to complete, so many people use completed requests per minute (RPM) instead. Tracking this metric is useful for understanding how an inference service handles concurrent requests. Some providers might throttle your service if you send too many concurrent requests at the same time.

Throughput is directly linked to compute cost. A higher throughput typically means lower cost. If your system costs $2/h in compute and its throughput is 100 tokens/s, it costs around $5.556 per 1M output tokens. If each request generates 200 output tokens on average, the cost for decoding 1K requests would be $1.11.

The prefill cost can be similarly calculated. If your hardware costs $2 per hour and it can prefill 100 requests per minute, the cost for prefilling 1K requests would be $0.33.

The total cost per request is the sum of the prefilling and decoding costs. In this example, the total cost for 1K requests would be $1.11 + $0.33 = $1.44.

What's considered good throughput depends on the model, the hardware, and the workload. Smaller models and higher-end chips typically result in higher throughput. Workloads with consistent input and output lengths are easier to optimize than workloads with variable lengths.

Even for similarly sized models, hardware, and workloads, direct throughput comparisons might be only approximate because token count depends on what constitutes a token, and different models have different tokenizers. It's better to compare the efficiency of inference servers using metrics such as cost per request.

Just like most other software applications, AI applications have the latency/throughput trade-off. Techniques like batching can improve throughput but reduce latency. According to the LinkedIn AI team in their reflection after a year of deploying generative AI products (LinkedIn, 2024), it's not uncommon to double or triple the throughput if you're willing to sacrifice TTFT and TPOT.

Due to this trade-off, focusing on an inference service based solely on its throughput and cost can lead to a bad user experience. Instead, some teams focus on *goodput*, a metric adapted from networking for LLM applications. Goodput measures the number of requests per second that satisfies the SLO, software-level objective.

Imagine that your application has the following objectives: TTFT of at most 200 ms and TPOT of at most 100 ms. Let's say that your inference service can complete 100 requests per minute. However, out of these 100 requests, only 30 satisfy the SLO. Then, the goodput of this service is 30 requests per minute. A visualization of this is shown in Figure 9-4.

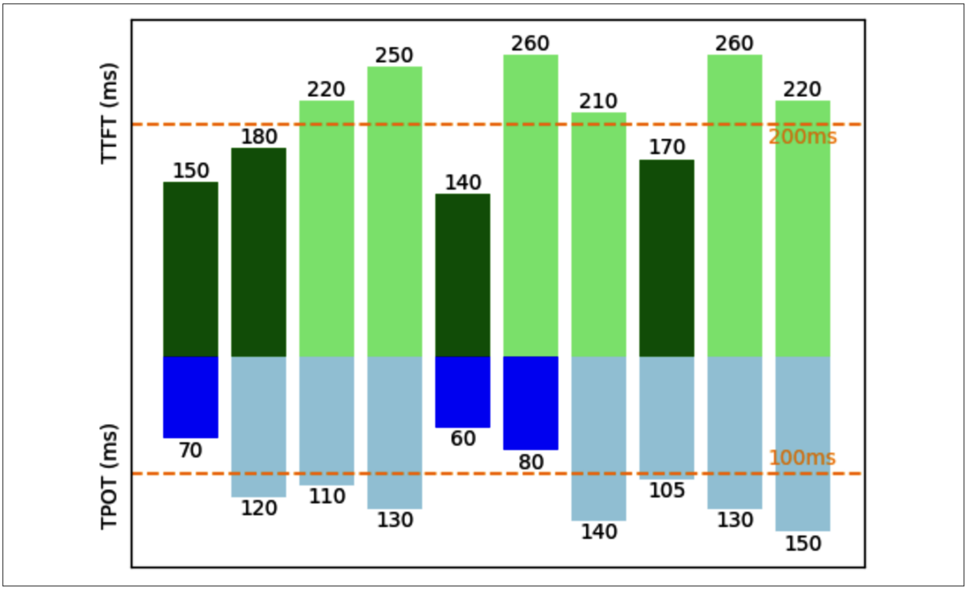


Figure 9-4. If an inference service can complete 10 RPS but only 3 satisfy the SLO, then its goodput is 3 RPS.

Utilization, MFU, and MBU

Utilization metrics measure how efficiently a resource is being used. It typically quantifies the proportion of the resource actively being used compared to its total available capacity.

A common but often misunderstood metric is *GPU utilization*, and NVIDIA is partially to blame for this misunderstanding. The official NVIDIA tool for monitoring GPU usage is `nvidia-smi`—SMI stands for System Management Interface. One metric this tool shows is GPU utilization, which represents the percentage of time during which the GPU is actively processing tasks. For example, if you run inference on a GPU cluster for 10 hours, and the GPUs are actively processing tasks for 5 of those hours, your GPU utilization would be 50%.

However, actively processing tasks doesn't mean doing so efficiently. For simplicity, consider a tiny GPU capable of doing 100 operations per second. In `nvidia-smi`'s definition of utilization, this GPU can report 100% utilization even if it's only doing one operation per second.

If you pay for a machine that can do 100 operations and use it for only 1 operation, you're wasting money. `nvidia-smi`'s GPU optimization metric is, therefore, not very useful. A utilization metric you might care about, out of all the operations a machine is capable of computing, is how many it's doing in a given time. This metric is called

MFU (Model FLOP/s Utilization), which distinguishes it from the NVIDIA GPU utilization metric.

MFU is the ratio of the observed throughput (tokens/s) relative to the theoretical maximum throughput of a system operating at peak FLOP/s. If at the peak FLOP/s advertised by the chip maker, the chip can generate 100 tokens/s, but when used for your inference service, it can generate only 20 tokens/s, your MFU is 20%.[11]

Similarly, because memory bandwidth is expensive, you might also want to know how efficiently your hardware's bandwidth is utilized. *MBU (Model Bandwidth Utilization)* measures the percentage of achievable memory bandwidth used. If the chip's peak bandwidth is 1 TB/s and your inference uses only 500 GB/s, your MBU is 50%.

Computing the memory bandwidth being used for LLM inference is straightforward:

```
parameter count × bytes/param × tokens/s
```

MBU is computed as follows:

```
(parameter count × bytes/param × tokens/s) / (theoretical bandwidth)
```

For example, if you use a 7B-parameter model in FP16 (two bytes per parameter) and achieve 100 tokens/s, the bandwidth used is:

```
7B × 2 × 100 = 700 GB/s
```

This underscores the importance of quantization (discussed in Chapter 7). Fewer bytes per parameter mean your model consumes less valuable bandwidth.

If this is done on an A100-80GB GPU with a theoretical 2 TB/s of memory bandwidth, the MBU is:

```
(700 GB/s) / (2 TB/s) = 70%
```

The relationships between throughput (tokens/s) and MBU and between throughput and MFU are linear, so some people might use throughput to refer to MBU and MFU.

What's considered a good MFU and MBU depends on the model, hardware, and workload. Compute-bound workloads typically have higher MFU and lower MBU, while bandwidth-bound workloads often show lower MFU and higher MBU.

Because training can benefit from more efficient optimization (e.g., better batching), thanks to having more predictable workloads, MFU for training is typically higher than MFU for inference. For inference, since prefill is compute-bound and decode is memory bandwidth-bound, MFU during prefilling is typically higher than MFU during decoding. For model training, as of this writing, an MFU above 50% is generally

[11] People have cared about FLOP/s utilization for a long time, but the term MFU was introduced in the PaLM paper (Chowdhery et al., 2022).

considered good, but it can be hard to achieve on specific hardware.[12] Table 9-1 shows MFU for several models and accelerators.

Table 9-1. MFU examples from "PaLM: Scaling Language Modeling with Pathways" (Chowdhery et al., 2022).

Model	Number of parameters (in billions)	Accelerator chips	Model FLOP/s utilization
GPT-3	175B	V100	21.3%
Gopher	280B	4096 TPU v3	32.5%
Megatron-Turing NLG	530B	2240 A100	30.2%
PaLM	540B	6144 TPU v4	46.2%

Figure 9-5 shows the MBU for the inference process using Llama 2-70B in FP16 on different hardware. The decline is likely due to the higher computational load per second with more users, shifting the workload from being bandwidth-bound to compute-bound.

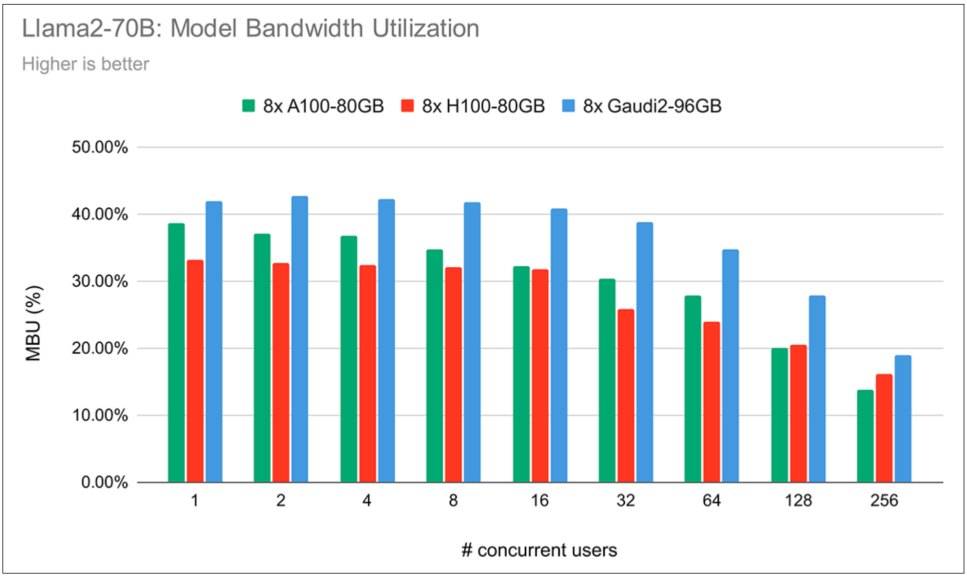


Figure 9-5. Bandwidth utilization for Llama 2-70B in FP16 across three different chips shows a decrease in MBU as the number of concurrent users increases. Image from "LLM Training and Inference with Intel Gaudi 2 AI Accelerators" (Databricks, 2024).

[12] Chip makers might also be doing what I call *peak FLOP/s hacking*. This might run experiments in certain conditions, such as using sparse matrices with specific shapes, to increase their peak FLOP/s. Higher peak FLOP/s numbers make their chips more attractive, but it can be harder for users to achieve high MFU.

Utilization metrics are helpful to track your system's efficiency. Higher utilization rates for similar workloads on the same hardware generally mean that your services are becoming more efficient. However, *the goal isn't to get the chips with the highest utilization*. What you really care about is how to get your jobs done faster and cheaper. A higher utilization rate means nothing if the cost and latency both increase.

AI Accelerators

How fast and cheap software can run depends on the hardware it runs on. While there are optimization techniques that work across hardware, understanding hardware allows for deeper optimization. This section looks at hardware from an inference perspective, but it can be applied to training as well.

The development of AI models and hardware has always been intertwined. The lack of sufficiently powerful computers was one of the contributing factors to the first AI winter in the 1970s.[13]

The revival of interest in deep learning in 2012 was also closely tied to compute. One commonly acknowledged reason for the popularity of AlexNet (Krizhevsky et al., 2012) is that it was the first paper to successfully use GPUs, graphics processing units, to train neural networks.[14] Before GPUs, if you wanted to train a model at AlexNet's scale, you'd have to use thousands of CPUs, like the one Google released just a few months before AlexNet. Compared to thousands of CPUs, a couple of GPUs were a lot more accessible to PhD students and researchers, setting off the deep learning research boom.

[13] In the 1960s, computers could run only one-layer neural networks, which had very limited capabilities. In their famous 1969 book *Perceptrons: An Introduction to Computational Geometry* (MIT Press), two AI pioneers, Marvin Minsky and Seymour Papert, argued that neural networks with hidden layers would still be able to do little. Their exact quote was: "Virtually nothing is known about the computational capabilities of this latter kind of machine. We believe that it can do little more than can a low order perceptron." There wasn't sufficient compute power to dispute their argument, which was then cited by many people as a key reason for the drying up of AI funding in the 1970s.

[14] There have been discussions on whether to rename the GPU since it's used for a lot more than graphics (Jon Peddie, "Chasing Pixels," July 2018). Jensen Huang, NVIDIA's CEO, said in an interview (*Stratechery*, March 2022) that once the GPU took off and they added more capabilities to it, they considered renaming it to something more general like GPGPU (general-purpose GPU) or XGU. They decided against renaming because they assumed that people who buy GPUs will be smart enough to know what a GPU is good for beyond its name.

What's an accelerator?

An accelerator is a chip designed to accelerate a specific type of computational workload. An AI accelerator is designed for AI workloads. The dominant type of AI accelerator is GPUs, and the biggest economic driver during the AI boom in the early 2020s is undoubtedly NVIDIA.

The main difference between CPUs and GPUs is that CPUs are designed for general-purpose usage, whereas GPUs are designed for parallel processing:

- CPUs have a few powerful cores, typically up to 64 cores for high-end consumer machines. While many CPU cores can handle multi-threaded workloads effectively, they excel at tasks requiring high single-thread performance, such as running an operating system, managing I/O (input/output) operations, or handling complex, sequential processes.
- GPUs have thousands of smaller, less powerful cores optimized for tasks that can be broken down into many smaller, independent calculations, such as graphics rendering and machine learning. The operation that constitutes most ML workloads is matrix multiplication, which is highly parallelizable.[15]

While the pursuit of efficient parallel processing increases computational capabilities, it imposes challenges on memory design and power consumption.

The success of NVIDIA GPUs has inspired many accelerators designed to speed up AI workloads, including Advanced Micro Devices (AMD)'s newer generations of GPUs, Google's TPU (Tensor Processing Unit), Intel's Habana Gaudi, Graphcore's Intelligent Processing Unit (IPU), Groq's Language Processing Unit (LPU), Cerebras' Wafer-Scale Quant Processing Unit (QPU), and many more being introduced.

While many chips can handle both training and inference, one big theme emerging is specialized chips for inference. A survey by Desislavov et al. (2023) shares that inference can exceed the cost of training in commonly used systems, and that inference accounts for up to 90% of the machine learning costs for deployed AI systems.

[15] Matrix multiplication, affectionately known as matmul, is estimated to account for more than 90% of all floating point operations in a neural network, according to "Data Movement Is All You Need: A Case Study on Optimizing Transformers" (Ivanov et al., *arXiv*, v3, November 2021) and "Scalable MatMul-free Language Modeling" (Zhu et al., *arXiv*, June 2024).

As discussed in Chapter 7, training demands much more memory due to backpropagation and is generally more difficult to perform in lower precision. Furthermore, training usually emphasizes throughput, whereas inference aims to minimize latency.

Consequently, chips designed for inference are often optimized for lower precision and faster memory access, rather than large memory capacity. Examples of such chips include the Apple Neural Engine, AWS Inferentia, and MTIA (Meta Training and Inference Accelerator). Chips designed for edge computing, like Google's Edge TPU and the NVIDIA Jetson Xavier, are also typically geared toward inference.

There are also chips specialized for different model architectures, such as chips specialized for the transformer.[16] Many chips are designed for data centers, with more and more being designed for consumer devices (such as phones and laptops).

Different hardware architectures have different memory layouts and specialized compute units that evolve over time. These units are optimized for specific data types, such as scalars, vectors, or tensors, as shown in Figure 9-6.

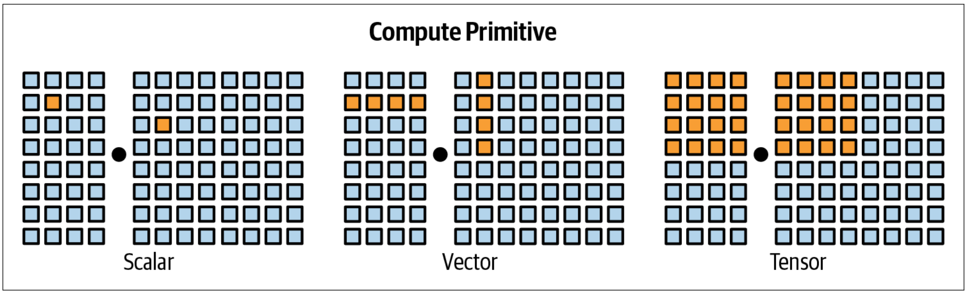


Figure 9-6. Different compute primitives. Image inspired by Chen et al. (2018).

A chip might have a mixture of different compute units optimized for various data types. For example, GPUs traditionally supported vector operations, but many modern GPUs now include tensor cores optimized for matrix and tensor computations. TPUs, on the other hand, are designed with tensor operations as their primary compute primitive. To efficiently operate a model on a hardware architecture, its memory layout and compute primitives need to be taken into account.

A chip's specifications contain many details that can be useful when evaluating this chip for each specific use case. However, the main characteristics that matter across use cases are computational capabilities, memory size and bandwidth, and power consumption. I'll use GPUs as examples to illustrate these characteristics.

[16] While a chip can be developed to run one model architecture, a model architecture can be developed to make the most out of a chip, too. For example, the transformer was originally designed by Google to run fast on TPUs and only later optimized on GPUs.

Computational capabilities

Computational capabilities are typically measured by the number of operations a chip can perform in a given time. The most common metric is *FLOP/s*, often written as FLOPS, which measures the *peak* number of floating-point operations per second. In reality, however, it's very unlikely that an application can achieve this peak FLOP/s. The ratio between the actual FLOP/s and the theoretical FLOP/s is one *utilization* metric.

The number of operations a chip can perform in a second depends on the numerical precision—the higher the precision, the fewer operations the chip can execute. Think about how adding two 32-bit numbers generally requires twice the computation of adding two 16-bit numbers. The number of 32-bit operations a chip can perform in a given time is not exactly half that of 16-bit operations because of different chips' optimization. For an overview of numerical precision, revisit "Numerical Representations" on page 325.

Table 9-2 shows the FLOP/s specs for different precision formats for NVIDIA H100 SXM chips.

Table 9-2. FLOP/s specs for NVIDIA H100 SXM chips.

Numerical precision	teraFLOP/s (trillion FLOP/s) with sparsity
TF32 Tensor Core[a]	989
BFLOAT16 Tensor Core	1,979
FP16 Tensor Core	1,979
FP8 Tensor Core	3,958

[a] Recall from Chapter 7 that TF32 is a 19-bit, not 32-bit, format.

Memory size and bandwidth

Because a GPU has many cores working in parallel, data often needs to be moved from the memory to these cores, and, therefore, data transfer speed is important. Data transfer is crucial when working with AI models that involve large weight matrices and training data. These large amounts of data need to be moved quickly to keep the cores efficiently occupied. Therefore, GPU memory needs to have higher bandwidth and lower latency than CPU memory, and thus, GPU memory requires more advanced memory technologies. This is one of the factors that makes GPU memory more expensive than CPU memory.

To be more specific, CPUs typically use DDR SDRAM (Double Data Rate Synchronous Dynamic Random-Access Memory), which has a 2D structure. GPUs, particularly high-end ones, often use HBM (high-bandwidth memory), which has a 3D stacked structure.[17]

An accelerator's memory is measured by its *size and bandwidth*. These numbers need to be evaluated within the system an accelerator is part of. An accelerator, such as a GPU, typically interacts with three levels of memory, as visualized in Figure 9-7:

CPU memory (DRAM)

Accelerators are usually deployed alongside CPUs, giving them access to the CPU memory (also known as system memory, host memory, or just CPU DRAM).

CPU memory usually has the lowest bandwidth among these memory types, with data transfer speeds ranging from 25 GB/s to 50 GB/s. CPU memory size varies. Average laptops might have around 16–64 GB, whereas high-end workstations can have one TB or more.

GPU high-bandwidth memory (HBM)

This is the memory dedicated to the GPU, located close to the GPU for faster access than CPU memory.

HBM provides significantly higher bandwidth, with data transfer speeds typically ranging from 256 GB/s to over 1.5 TB/s. This speed is essential for efficiently handling large data transfers and high-throughput tasks. A consumer GPU has around 24–80 GB of HBM.

GPU on-chip SRAM

Integrated directly into the chip, this memory is used to store frequently accessed data and instructions for nearly instant access. It includes L1 and L2 caches made of SRAM, and, in some architectures, L3 caches as well. These caches are part of the broader on-chip memory, which also includes other components like register files and shared memory.

RAM has extremely high data transfer speeds, often exceeding 10 TB/s. The size of GPU SRAM is small, typically 40 MB or under.

[17] Lower-end to mid-range GPUs might use GDDR (Graphics Double Data Rate) memory.

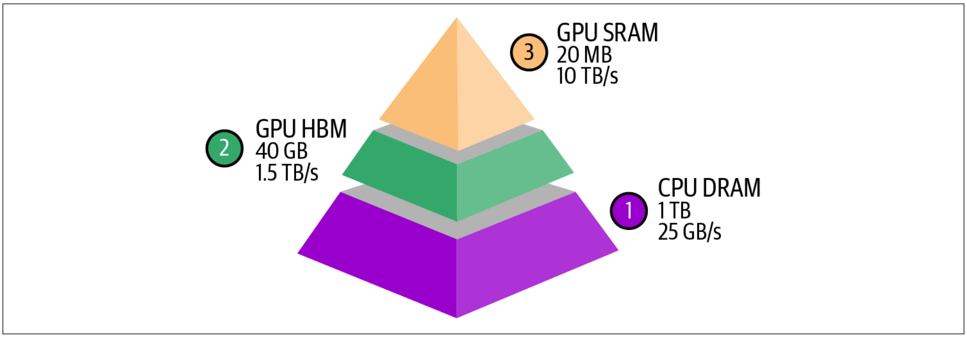


Figure 9-7. The memory hierarchy of an AI accelerator. The numbers are for reference only. The actual numbers vary for each chip.

A lot of GPU optimization is about how to make the most out of this memory hierarchy. However, as of this writing, popular frameworks such as PyTorch and TensorFlow don't yet allow fine-grained control of memory access. This has led many AI researchers and engineers to become interested in GPU programming languages such as CUDA (originally Compute Unified Device Architecture), OpenAI's Triton, and ROCm (Radeon Open Compute). The latter is AMD's open source alternative to NVIDIA's proprietary CUDA.

Power consumption

Chips rely on transistors to perform computation. Each computation is done by transistors switching on and off, which requires energy. A GPU can have billions of transistors—an NVIDIA A100 has 54 billion transistors, while an NVIDIA H100 has 80 billion. When an accelerator is used efficiently, billions of transistors rapidly switch states, consuming a substantial amount of energy and generating a nontrivial amount of heat. This heat requires cooling systems, which also consume electricity, adding to data centers' overall energy consumption.

Chip energy consumption threatens to have a staggering impact on the environment, increasing the pressure on companies to invest in technologies for green data centers. An NVIDIA H100 running at its peak for a year consumes approximately 7,000 kWh. For comparison, the average US household's annual electricity consumption is 10,000 kWh. That's why electricity is a bottleneck to scaling up compute.[18]

[18] A main challenge in building data centers with tens of thousands of GPUs is finding a location that can guarantee the necessary electricity. Building large-scale data centers requires navigating electricity supply, speed, and geopolitical constraints. For example, remote regions might provide cheaper electricity but can increase network latency, making the data centers less appealing for use cases with stringent latency requirements like inference.

Accelerators typically specify their power consumption under *maximum power draw* or a proxy metric *TDP (thermal design power):*

- Maximum power draw indicates the peak power that the chip could draw under full load.
- *TDP* represents the maximum heat a cooling system needs to dissipate when the chip operates under typical workloads. While it's not an exact measure of power consumption, it's an indication of the expected power draw. For CPUs and GPUs, the maximum power draw can be roughly 1.1 to 1.5 times the TDP, though the exact relationship varies depending on the specific architecture and workload.

If you opt for cloud providers, you won't need to worry about cooling or electricity. However, these numbers can still be of interest to understand the impact of accelerators on the environment and the overall electricity demand.

Selecting Accelerators

What accelerators to use depends on your workload. If your workloads are compute-bound, you might want to look for chips with more FLOP/s. If your workloads are memory-bound, shelling out money for chips with higher bandwidth and more memory will make your life easier.

When evaluating which chips to buy, there are three main questions:

- Can the hardware run your workloads?
- How long does it take to do so?
- How much does it cost?

FLOP/s, memory size, and memory bandwidth are the three big numbers that help you answer the first two questions. The last question is straightforward. Cloud providers' pricing is typically usage-based and fairly similar across providers. If you buy your hardware, the cost can be calculated based on the initial price and ongoing power consumption.

Inference Optimization

Inference optimization can be done at the model, hardware, or service level. To illustrate their differences, consider archery. Model-level optimization is like crafting better arrows. Hardware-level optimization is like training a stronger and better archer. Service-level optimization is like refining the entire shooting process, including the bow and aiming conditions.

Ideally, optimizing a model for speed and cost shouldn't change the model's quality. However, many techniques might cause model degradation. Figure 9-8 shows the same Llama models' performance on different benchmarks, served by different inference service providers.

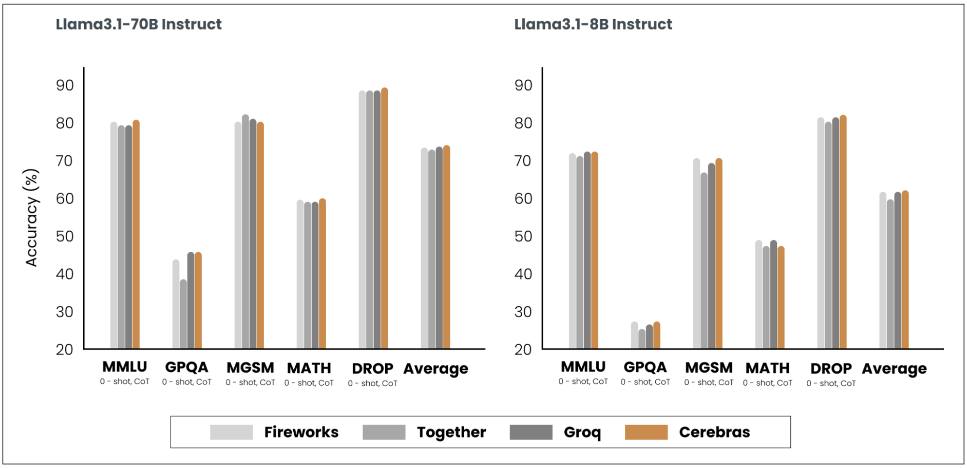


Figure 9-8. An inference service provider might use optimization techniques that can alter a model's behavior, causing different providers to have slight model quality variations. The experiment was conducted by Cerebras (2024).

Since hardware design is outside the scope of this book, I'll discuss techniques at the model and service levels. While the techniques are discussed separately, keep in mind that, in production, optimization typically involves techniques at more than one level.

Model Optimization

Model-level optimization aims to make the model more efficient, often by modifying the model itself, which can alter its behavior. As of this writing, many foundation models follow the transformer architecture and include an autoregressive language model component. These models have three characteristics that make inference resource-intensive: model size, autoregressive decoding, and the attention mechanism. Let's discuss approaches to address these challenges.

Model compression

Model compression involves techniques that reduce a model's size. Making a model smaller can also make it faster. This book has already discussed two model compression techniques: quantization and distillation. Quantization, reducing the precision of a model to reduce its memory footprint and increase its throughput, is discussed in Chapter 7. Model distillation, training a small model to mimic the behavior of the large model, is discussed in Chapter 8.

Model distillation suggests that it's possible to capture a large model's behaviors using fewer parameters. Could it be that within the large model, there exists a subset of parameters capable of capturing the entire model's behavior? This is the core concept behind pruning.

Pruning, in the context of neural networks, has two meanings. One is to remove entire nodes of a neural network, which means changing its architecture and reducing its number of parameters. Another is to find parameters least useful to predictions and set them to zero. In this case, pruning doesn't reduce the total number of parameters, only the number of non-zero parameters. This makes the model more sparse, which both reduces the model's storage space and speeds up computation.

Pruned models can be used as-is or be further finetuned to adjust the remaining parameters and restore any performance degradation caused by the pruning process. Pruning can help discover promising model architectures (Liu et al., 2018). These pruned architectures, smaller than the pre-pruned architectures, can also be trained from scratch (Zhu et al., 2017).

In the literature, there have been many encouraging pruning results. For example, Frankle and Carbin (2019) showed that pruning techniques can reduce the non-zero parameter counts of certain trained networks by over 90%, decreasing memory footprints and improving speed without compromising accuracy. However, in practice, as of this writing, pruning is less common. It's harder to do, as it requires an understanding of the original model's architecture, and the performance boost it can bring is often much less than that of other approaches. Pruning also results in sparse models, and not all hardware architectures are designed to take advantage of the resulting sparsity.

Weight-only quantization is by far the most popular approach since it's easy to use, works out of the box for many models, and is extremely effective. Reducing a model's precision from 32 bits to 16 bits reduces its memory footprint by half. However, we're close to the limit of quantization—we can't go lower than 1 bit per value. Distillation is also common because it can result in a smaller model whose behavior is comparative to that of a much larger one for your needs.

Overcoming the autoregressive decoding bottleneck

As discussed in Chapter 2, autoregressive language models generate one token after another. If it takes 100 ms to generate one token, a response of 100 tokens will take 10 s.[19] This process is not just slow, it's also expensive. Across model API providers, an output token costs approximately two to four times an input token. In an experiment, Anyscale found that a single output token can have the same impact on latency as 100 input tokens (Kadous et al., 2023). Improving the autoregressive generation process by a small percentage can significantly improve user experience.

As the space is rapidly evolving, new techniques are being developed to overcome this seemingly impossible bottleneck. Perhaps one day, there will be architectures that don't have this bottleneck. The techniques covered here are to illustrate what the solution might look like, but the techniques are still evolving.

Speculative decoding. Speculative decoding (also called speculative sampling) uses a faster but less powerful model to generate a sequence of tokens, which are then verified by the target model. The target model is the model you want to use. The faster model is called the draft or proposal model because it proposes the draft output.

Imagine the input tokens are $x_1, x_2, \ldots, x_t$:

1. The draft model generates a sequence of K tokens: $x_{t+1}, x_{t+2}, \ldots, x_{t+K}$.
2. The target model verifies these K generated tokens in parallel.
3. The target model *accepts* the longest subsequence of draft tokens, from left to right, which the target model agrees to use.
4. Let's say the target model accepts j draft tokens, $x_{t+1}, x_{t+2}, \ldots, x_{t+j}$. The target model then generates one extra token, x_{t+j+1}.

The process returns to step 1, with the draft model generating K tokens conditioned on $x_1, x_2, \ldots, x_t, x_{t+1}, x_{t+2}, \ldots, x_{t+j}$. The process is visualized in Figure 9-9.

If no draft token is accepted, this loop produces only one token generated by the target model. If all draft tokens are accepted, this loop produces $K + 1$ tokens, with K generated by the draft model and one by the target model.

[19] Each token generation step necessitates the transfer of the entire model's parameters from the accelerator's high-bandwidth memory to its compute units. This makes this operation bandwidth-heavy. Because the model can produce only one token at a time, the process consumes only a small number of FLOP/s, resulting in computational inefficiency.

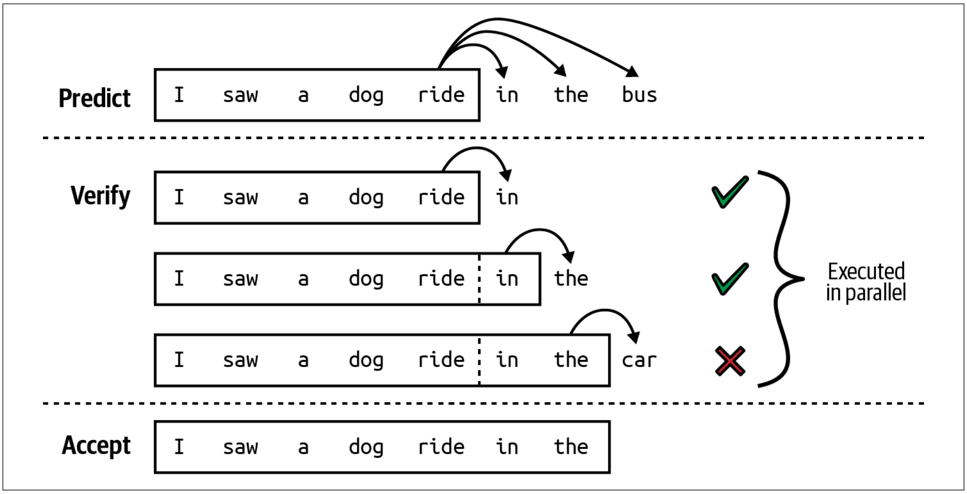


Figure 9-9. A draft model generates a sequence of K tokens, and the main model accepts the longest subsequence that it agrees with. The image is from "Blockwise Parallel Decoding for Deep Autoregressive Models" (Stern et al., 2018).

If all draft sequences are rejected, the target model must generate the entire response in addition to verifying it, potentially leading to increased latency. However, this can be avoided because of these three insights:

1. The time it takes for the target model to verify a sequence of tokens is less than the time it takes to generate it, because verification is parallelizable, while generation is sequential. Speculative decoding effectively turns the computation profile of decoding into that of prefilling.
2. In an output token sequence, some tokens are easier to predict than others. It's possible to find a weaker draft model capable of getting these easier-to-predict tokens right, leading to a high acceptance rate of the draft tokens.
3. Decoding is memory bandwidth-bound, which means that during the coding process, there are typically idle FLOPs that can be used for free verification.[20]

Acceptance rates are domain-dependent. For texts that follow specific structures like code, the acceptance rate is typically higher. Larger values of *K* mean fewer verifying calls for the target model but a low acceptance rate of the draft tokens. The draft model can be of any architecture, though ideally it should share the same vocabulary and tokenizer as the target model. You can train a custom draft model or use an existing weaker model.

[20] This also means that if your MFU is already maxed out, speculative decoding makes less sense.

For example, to speed up the decoding process of Chinchilla-70B, DeepMind trained a 4B-parameter draft model of the same architecture (Chen et al., 2023). The draft model can generate a token eight times faster than the target model (1.8 ms/token compared to 14.1 ms/token). This reduces the overall response latency by more than half without compromising response quality. A similar speed-up was achieved for T5-XXL (Laviathan et al., 2022).

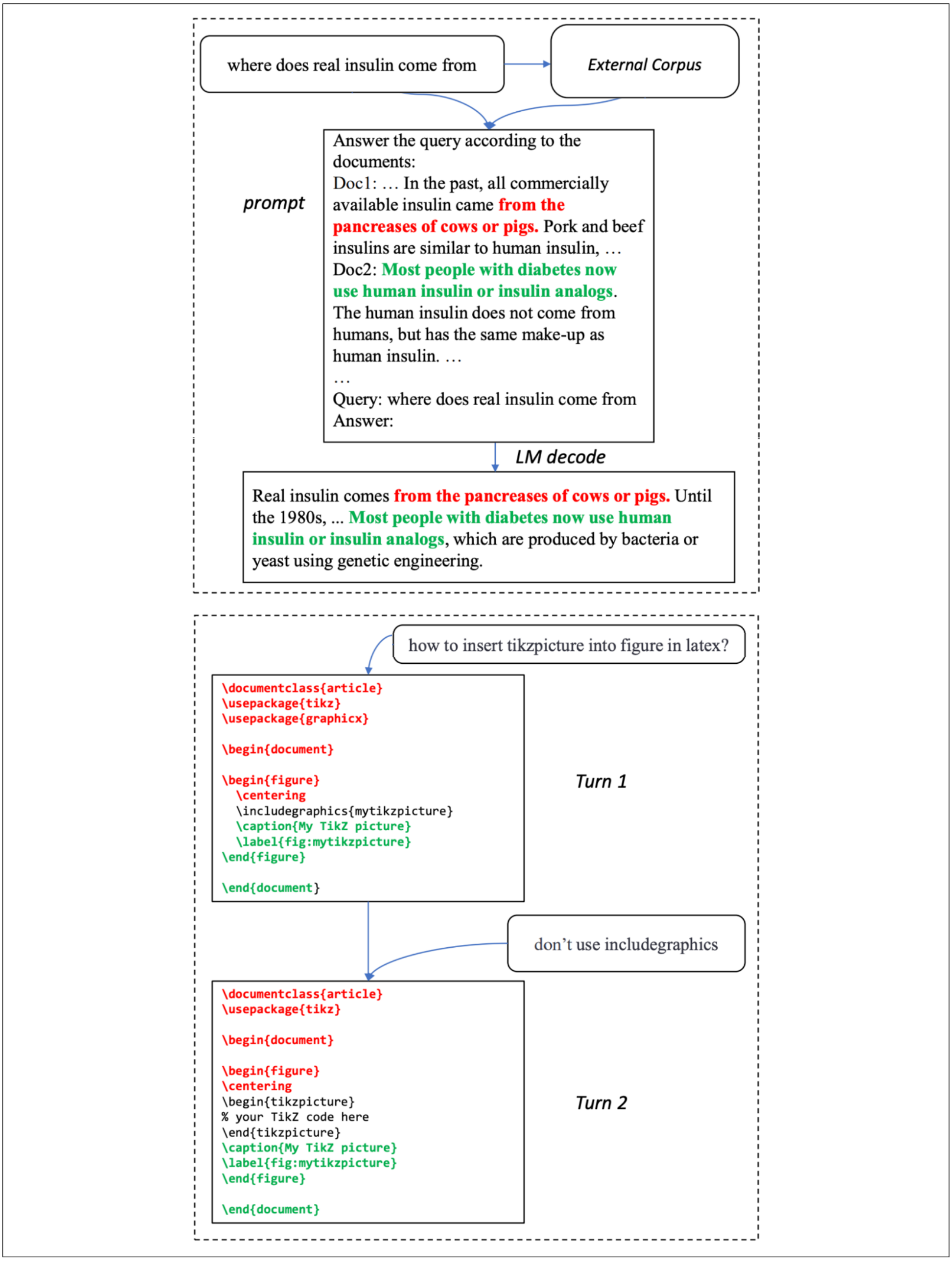
This approach has gained traction because it's relatively easy to implement and doesn't change a model's quality. For example, it's possible to do so in 50 lines of code in PyTorch. It's been incorporated into popular inference frameworks such as vLLM, TensorRT-LLM, and llama.cpp.

Inference with reference. Often, a response needs to reference tokens from the input. For example, if you ask your model a question about an attached document, the model might repeat a chunk of text verbatim from the document. Another example is if you ask the model to fix bugs in a piece of code, the model might reuse the majority of the original code with minor changes. Instead of making the model generate these repeated tokens, what if we copy these tokens from the input to speed up the generation? This is the core idea behind inference with reference.

Inference with reference is similar to speculative decoding, but instead of using a model to generate draft tokens, it selects draft tokens from the input. The key challenge is to develop an algorithm to identify the most relevant text span from the context at each decoding step. The simplest option is to find a text span that matches the current tokens.

Unlike speculative decoding, inference with reference doesn't require an extra model. However, it's useful only in generation scenarios where there's a significant overlap between contexts and outputs, such as in retrieval systems, coding, or multi-turn conversations. In "Inference with Reference: Lossless Acceleration of Large Language Models" (Yang et al., 2023), this technique helps achieve two times generation speedup in such use cases.

Examples of how inference with reference works are shown in Figure 9-10.



Answer the query according to the documents:
Doc1: … In the past, all commercially available insulin came **from the pancreases of cows or pigs.** Pork and beef insulins are similar to human insulin, …
Doc2: **Most people with diabetes now use human insulin or insulin analogs**. The human insulin does not come from humans, but has the same make-up as human insulin. …
…
Query: where does real insulin come from
Answer:

Real insulin comes **from the pancreases of cows or pigs.** Until the 1980s, ... **Most people with diabetes now use human insulin or insulin analogs**, which are produced by bacteria or yeast using genetic engineering.

```
\documentclass{article}
\usepackage{tikz}
\usepackage{graphicx}

\begin{document}

\begin{figure}
  \centering
  \includegraphics{mytikzpicture}
  \caption{My TikZ picture}
  \label{fig:mytikzpicture}
\end{figure}

\end{document}
```

```
\documentclass{article}
\usepackage{tikz}

\begin{document}

\begin{figure}
\centering
\begin{tikzpicture}
% your TikZ code here
\end{tikzpicture}
\caption{My TikZ picture}
\label{fig:mytikzpicture}
\end{figure}

\end{document}
```

Figure 9-10. Two examples of inference with reference. The text spans that are successfully copied from the input are in red and green. Image from Yang et al. (2023). The image is licensed under CC BY 4.0.

Parallel decoding. Instead of making autoregressive generation faster with draft tokens, some techniques aim to break the sequential dependency. Given an existing sequence of tokens $x_1, x_2, \ldots, x_t$, these techniques attempt to generate $x_{t+1}, x_{t+2}, \ldots, x_{t+k}$ simultaneously. This means that the model generates x_{t+2} before it knows that the token before it is x_{t+1}.

This can work because the knowledge of the existing sequence often is sufficient to predict the next few tokens. For example, given "the cat sits", without knowing that the next token is "on", "under", or "behind", you might still predict that the word after it is "the".

The parallel tokens can be generated by the same decoder, as in Lookahead decoding (Fu et al., 2024), or by different decoding heads, as in Medusa (Cai et al., 2024). In Medusa, the original model is extended with multiple decoding heads, and each head is a small neural network layer that is then trained to predict a future token at a specific position. If the original model is trained to predict the next token x_{t+1}, the k^{th} head will predict the token x_{t+k+1}. These heads are trained together with the original model, but the original model is frozen. NVIDIA claimed Medusa helped boost Llama 3.1 token generation by up to 1.9× on their HGX H200 GPUs (Eassa et al., 2024).

However, because these tokens aren't generated sequentially, they need to be verified to make sure that they fit together. An essential part of parallel decoding is verification and integration. Lookahead decoding uses the Jacobi method[21] to verify the generated tokens, which works as follows:

1. K future tokens are generated in parallel.
2. These *K* tokens are verified for coherence and consistency with the context.
3. If one or more tokens fail verification, instead of aggregating all *K* future tokens, the model regenerates or adjusts only these failed tokens.

The model keeps refining the generated tokens until they all pass verification and are integrated into the final output. This family of parallel decoding algorithms is also called Jacobi decoding.

On the other hand, Medusa uses a tree-based attention mechanism to verify and integrate tokens. Each Medusa head produces several options for each position. These options are then organized into a tree-like structure to select the most promising combination. The process is visualized in Figure 9-11.

[21] The Jacobi method is an iterative algorithm where multiple parts of a solution can be updated simultaneously and independently.

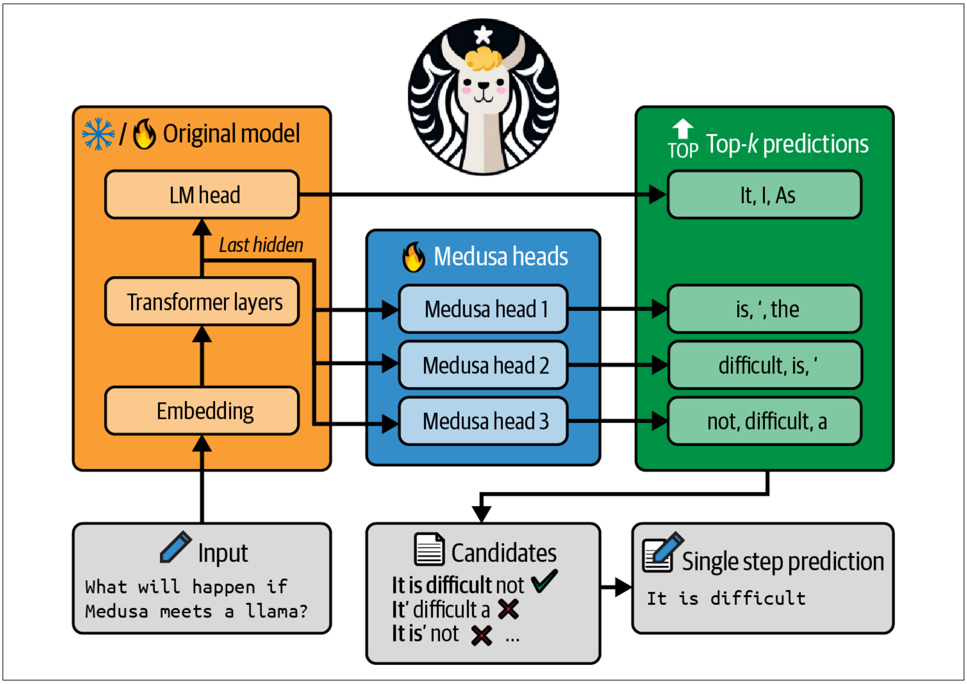


Figure 9-11. In Medusa (Cai et al., 2024), each head predicts several options for a token position. The most promising sequence from these options is selected. Image adapted from the paper, which is licensed under CC BY 4.0.

While the perspective of being able to circumvent sequential dependency is appealing, parallel decoding is not intuitive, and some techniques, like Medusa, can be challenging to implement.

Attention mechanism optimization

Recall from Chapter 2 that generating the next token requires the key and value vectors for all previous tokens. This means that the following applies:

- Generating token x_t requires the key and value vectors for tokens $x_1, x_2, \ldots, x_{t-1}$.
- Generating token x_{t+1} requires the key and value vectors for tokens $x_1, x_2, \ldots, x_{t-1}, x_t$.

When generating token x_{t+1}, instead of computing the key and value vectors for tokens $x_1, x_2, \ldots, x_{t-1}$ again, you reuse these vectors from the previous step. This means that you'll need to compute the key and value vectors for only the most recent token, x_t. The cache that stores key and value vectors for reuse is called the KV cache. The newly computed key and value vectors are then added to the KV cache, which is visualized in Figure 9-12.

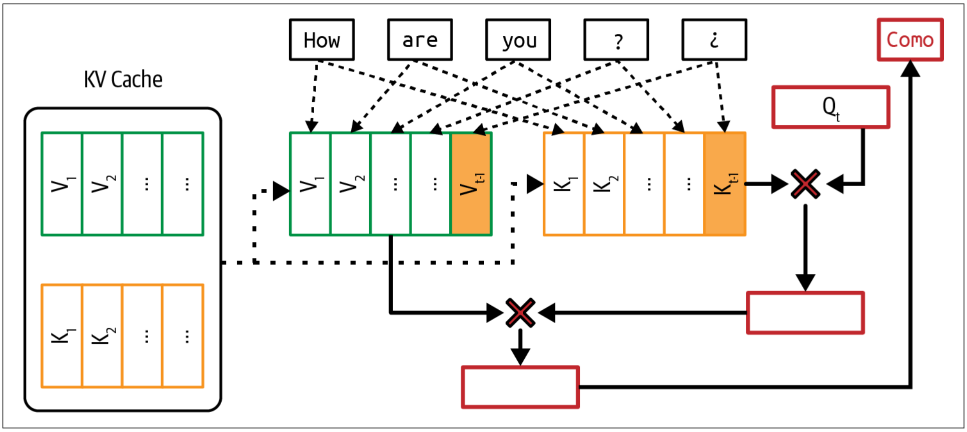


Figure 9-12. To avoid recomputing the key and value vectors at each decoding step, use a KV cache to store these vectors to reuse.

A KV cache is used only during inference, not training. During training, because all tokens in a sequence are known in advance, next token generation can be computed all at once instead of sequentially, as during inference. Therefore, there's no need for a KV cache.

Because generating a token requires computing the attention scores with all previous tokens, the number of attention computations grows exponentially with sequence length.[22] The KV cache size, on the other hand, grows linearly with sequence length.

The KV cache size also grows with larger batch sizes. A Google paper calculated that for a 500B+ model with multi-head attention, batch size 512, and context length 2048, the KV cache totals 3TB (Pope et al., 2022). This is three times the size of that model's weights.

The KV cache size is ultimately limited by the available hardware storage, creating a bottleneck for running applications with long context. A large cache size also takes time to load into memory, which can be an issue for applications with strict latency.

The computation and memory requirements of the attention mechanism are one of the reasons why it's so hard to have longer context.

Many techniques have been developed to make the attention mechanism more efficient. In general, they fall into three buckets: redesigning the attention mechanism, optimizing the KV cache, and writing kernels for attention computation.

[22] The number of attention computations for an autoregressive model is $O(n^2)$.

Calculating the KV Cache Size

The memory needed for the KV cache, without any optimization, is calculated as follows:

$$2 \times B \times S \times L \times H \times M$$

- B: batch size
- S: sequence length
- L: number of transformer layers
- H: model dimension
- M: memory needed for the cache's numerical representation (e.g., FP16 or FP32).

This value can become substantial as the context length increases. For example, LLama 2 13B has 40 layers and a model dimension of 5,120. With a batch size of 32, sequence length of 2,048, and 2 bytes per value, the memory needed for its KV cache, without any optimization, is 2 × 32 × 2,048 × 40 × 5,120 × 2 = 54 GB.

Redesigning the attention mechanism. These techniques involve altering how the attention mechanism works. Even though these techniques help optimize inference, because they change a model's architecture directly, they can be applied only during training or finetuning.

For example, when generating a new token, instead of attending to all previous tokens, *local windowed attention* attends only to a fixed size window of nearby tokens (Beltagy et al., 2020). This reduces the effective sequence length to a fixed size window, reducing both the KV cache and the attention computation. If the average sequence length is 10,000 tokens, attending to a window size of 1,000 tokens reduces the KV cache size by 10 times.

Local windowed attention can be interleaved with global attention, with local attention capturing nearby context; the global attention captures task-specific information across the document.

Both *cross-layer attention* (Brandon et al., 2024) and *multi-query attention* (Shazeer, 2019) reduce the memory footprint of the KV cache by reducing the number of key-value pairs. Cross-layer attention shares key and value vectors across adjacent layers. Having three layers sharing the same key-value vectors means reducing the KV cache three times. On the other hand, multi-query attention shares key-value vectors across query heads.

Grouped-query attention (Ainslie et al., 2023) is a generalization of multi-query attention. Instead of using only one set of key-value pairs for all query heads, its grouped-query attention puts query heads into smaller groups and shares key-value pairs only among query heads in the same group. This allows for a more flexible balance between the number of query heads and the number of key-value pairs.

Character.AI, an AI chatbot application, shares that their average conversation has a dialogue history of 180 messages (2024). Given the typically long sequences, the primary bottleneck for inference throughput is the KV cache size. Three attention mechanism designs—multi-query attention, interleaving local attention and global attention, and cross-layer attention—help them *reduce KV cache by over 20 times*. More importantly, this significant KV cache reduction means that memory is no longer a bottleneck for them for serving large batch sizes.

Optimizing the KV cache size. The way the KV cache is managed is critical in mitigating the memory bottleneck during inference and enabling a larger batch size, especially for applications with long context. Many techniques are actively being developed to reduce and manage the KV cache.

One of the fastest growing inference frameworks, vLLM, gained popularity for introducing PagedAttention, which optimizes memory management by dividing the KV cache into non-contiguous blocks, reducing fragmentation, and enabling flexible memory sharing to improve LLM serving efficiency (Kwon et al., 2023).

Other techniques include KV cache quantization (Hooper et al., 2024; Kang et al., 2024), adaptive KV cache compression (Ge et al., 2023), and selective KV cache (Liu et al., 2024).

Writing kernels for attention computation. Instead of changing the mechanism design or optimizing the storage, this approach looks into how attention scores are computed and finds ways to make this computation more efficient. This approach is the most effective when it takes into account the hardware executing the computation. The code optimized for a specific chip is called a kernel. Kernel writing will be discussed further in the next section.

One of the most well-known kernels optimized for attention computation is FlashAttention (Dao et al., 2022). This kernel fused together many operations commonly used in a transformer-based model to make them run faster, as shown in Figure 9-13.

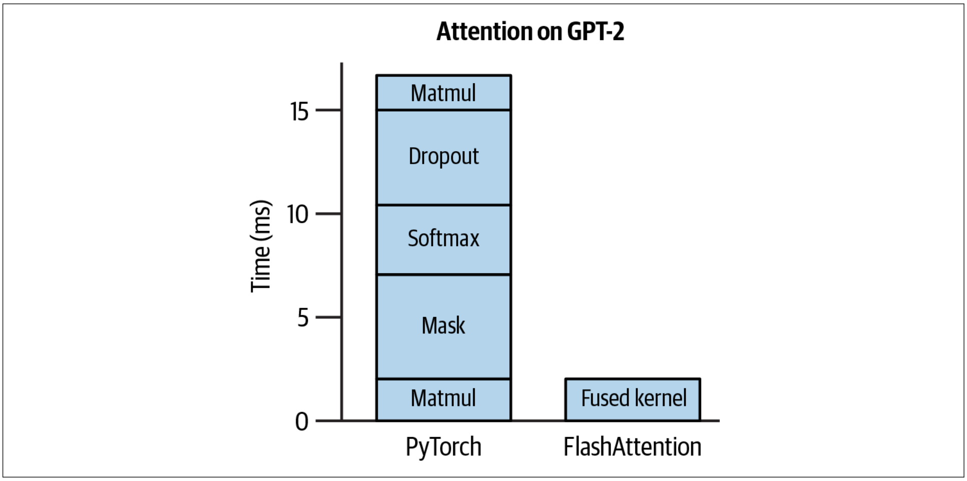


Figure 9-13. FlashAttention is a kernel that fuses together several common operators. Adapted from an original image licensed under BSD 3-Clause.

Kernels and compilers

Kernels are specialized pieces of code optimized for specific hardware accelerators, such as GPUs or TPUs. They are typically written to perform computationally intensive routines that need to be executed repeatedly, often in parallel, to maximize the performance of these accelerators.

Common AI operations, including matrix multiplication, attention computation, and convolution operation, all have specialized kernels to make their computation more efficient on different hardware.[23]

Writing kernels requires a deep understanding of the underlying hardware architecture. This includes knowledge about how the memory hierarchy is structured (such as caches, global memory, shared memory, and registers) and how data is accessed and moved between these different levels.

Moreover, kernels are typically written in lower-level programming languages like CUDA (for NVIDIA GPUs), Triton (a language developed by OpenAI for writing custom kernels), and ROCm (for AMD GPUs). These languages allow fine-grained control over thread management and memory access but are also harder to learn than the languages that most AI engineers are familiar with, like Python.

Due to this entry barrier, writing kernels used to be a dark art practiced by a few. Chip makers like NVIDIA and AMD employ optimization engineers to write kernels to make their hardware efficient for AI workloads, whereas AI frameworks like

[23] Convolution operations are often used in image generation models like Stable Diffusion.

PyTorch and TensorFlow employ kernel engineers to optimize their frameworks on different accelerators.

However, with the rising demand for inference optimization and the ubiquity of accelerators, more AI engineers have taken an interest in writing kernels. There are many great online tutorials for kernel writing. Here, I'll cover four common techniques often used to speed up computation:

Vectorization

Given a loop or a nested loop, instead of processing one data element at a time, simultaneously execute multiple data elements that are contiguous in memory. This reduces latency by minimizing data I/O operations.

Parallelization

Divide an input array (or n-dimensional array) into independent chunks that can be processed simultaneously on different cores or threads, speeding up the computation.

Loop tiling

Optimize the data accessing order in a loop for the hardware's memory layout and cache. This optimization is hardware-dependent. An efficient CPU tiling pattern may not work well on GPUs.

Operator fusion

Combine multiple operators into a single pass to avoid redundant memory access. For example, if two loops operate over the same array, they can be fused into one, reducing the number of times data is read and written.

While vectorization, parallelization, and loop tiling can be applied broadly across different models, operator fusion requires a deeper understanding of a model's specific operators and architecture. As a result, operator fusion demands more attention from optimization engineers.

Kernels are optimized for a hardware architecture. This means that whenever a new hardware architecture is introduced, new kernels need to be developed. For example, FlashAttention (Dao et al., 2022) was originally developed primarily for NVIDIA A100 GPUs. Later on, FlashAttention-3 was introduced for H100 GPUs (Shah et al., 2024).

A model script specifies a series of operations that need to be performed to execute that model. To run this code on a piece of hardware, such as a GPU, it has to be converted into a language compatible with that hardware. This process is called *lowering*. A tool that *lowers* code to run a specific hardware is called a compiler. Compilers bridge ML models and the hardware they run on. During the lowering process, whenever possible, these operations are converted into specialized kernels to run faster on the target hardware.

Inference Optimization Case Study from PyTorch

Figure 9-14 shows how much throughput improvement the PyTorch team could give to Llama-7B through the following optimization steps (PyTorch, 2023):

1. Call torch.compile to compile the model into more efficient kernels.
2. Quantize the model weights to INT8.
3. Further quantize the model weights to INT4.
4. Add speculative decoding.

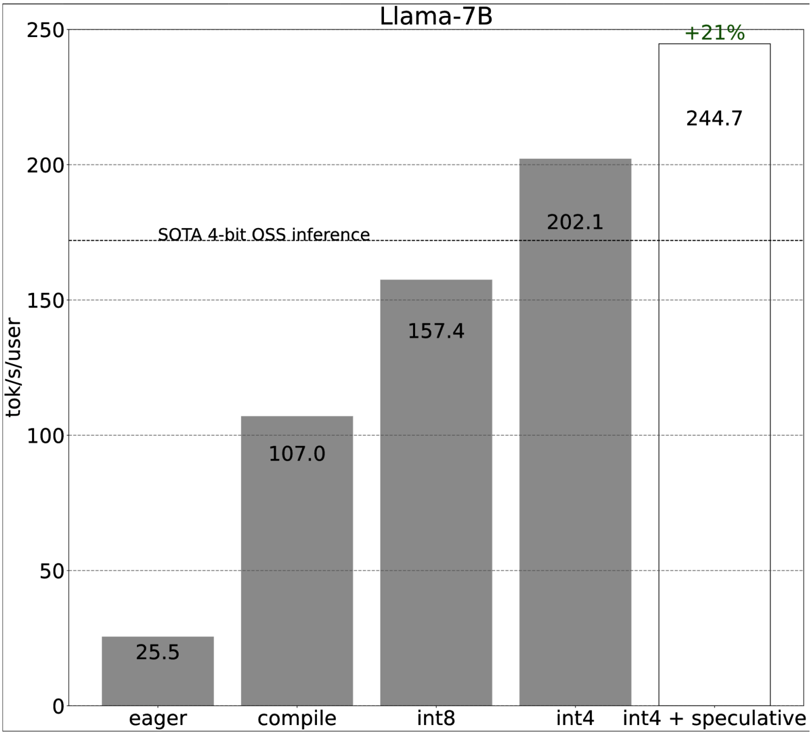


Figure 9-14. Throughput improvement by different optimization techniques in PyTorch. Image from PyTorch (2023).

The experiment was run on an A100 GPU with 80 GB of memory. It was unclear how these optimization steps impact the model's output quality.

Compilers can be standalone tools, such as Apache TVM and MLIR (Multi-Level Intermediate Representation) or integrated into ML and inference frameworks, like `torch.compile` (a feature in PyTorch), XLA (Accelerated Linear Algebra, originally developed by TensorFlow, with an open source version called OpenXLA), and the compiler built into the TensorRT, which is optimized for NVIDIA GPUs. AI companies might have their own compilers, with their proprietary kernels designed to speed up their own workloads.[24]

Inference Service Optimization

Most service-level optimization techniques focus on resource management. Given a fixed amount of resources (compute and memory) and dynamic workloads (inference requests from users that may involve different models), the goal is to efficiently allocate resources to these workloads to optimize for latency and cost. Unlike many model-level techniques, service-level techniques don't modify models and shouldn't change the output quality.

Batching

One of the easiest ways to reduce your cost is batching. In production, your inference service might receive multiple requests simultaneously. Instead of processing each request separately, batching the requests that arrive around the same time together can significantly reduce the service's throughput. If processing each request separately is like everyone driving their own car, batching is like putting them together on a bus. A bus can move more people, but it can also make each person's journey longer. However, if you do it intelligently, the impact on latency can be minimal.

The three main techniques for batching are: static batching, dynamic batching, and continuous batching.

The simplest batching technique is *static batching*. The service groups a fixed number of inputs together in a batch. It's like a bus that waits until every seat is filled before departing. The drawback of static batching is that all requests have to wait until the batch is full to be executed. Thus the first request in a batch is delayed until the batch's last request arrives, no matter how late the last request is.

[24] Many companies consider their kernels their trade secrets. Having kernels that allow them to run models faster and cheaper than their competitors is a competitive advantage.

Dynamic batching, on the other hand, sets a maximum time window for each batch. If the batch size is four and the window is 100 ms, the server processes the batch either when it has four requests or when 100 ms has passed, whichever happens first. It's like a bus that leaves on a fixed schedule or when it's full. This approach keeps latency under control, so earlier requests aren't held up by later ones. The downside is that batches may not always be full when processed, possibly leading to wasted compute. Static batching and dynamic batching are visualized in Figure 9-15.

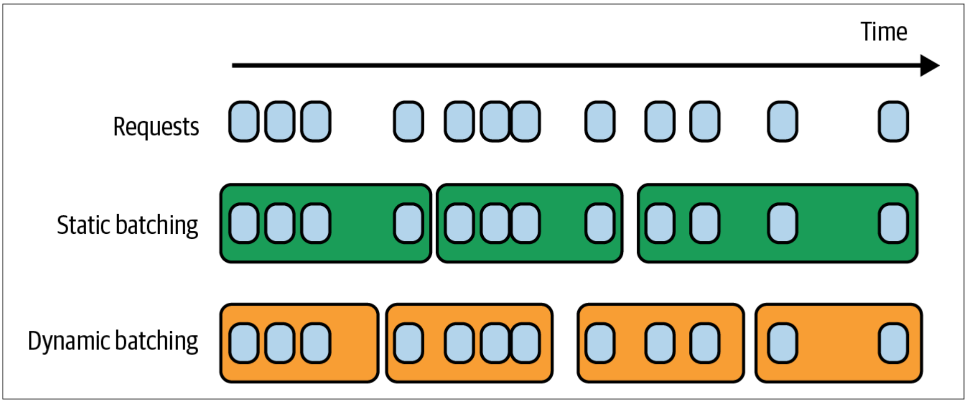


Figure 9-15. Dynamic batching keeps the latency manageable but might be less compute-efficient.

In naive batching implementations, all batch requests have to be completed before their responses are returned. For LLMs, some requests might take much longer than others. If one request in a batch generates only 10 response tokens and another request generates 1,000 response tokens, the short response has to wait until the long response is completed before being returned to the user. This results in unnecessary latency for short requests.

Continuous batching allows responses in a batch to be returned to users as soon as they are completed. It works by selectively batching operations that don't cause the generation of one response to hold up another, as introduced in the paper Orca (Yu et al., 2022). After a request in a batch is completed and its response returned, the service can add another request into the batch in its place, making the batching continuous. It's like a bus that, after dropping off one passenger, can immediately pick up another passenger to maximize its occupancy rate. Continuous batching, also called *in-flight batching*, is visualized in Figure 9-16.

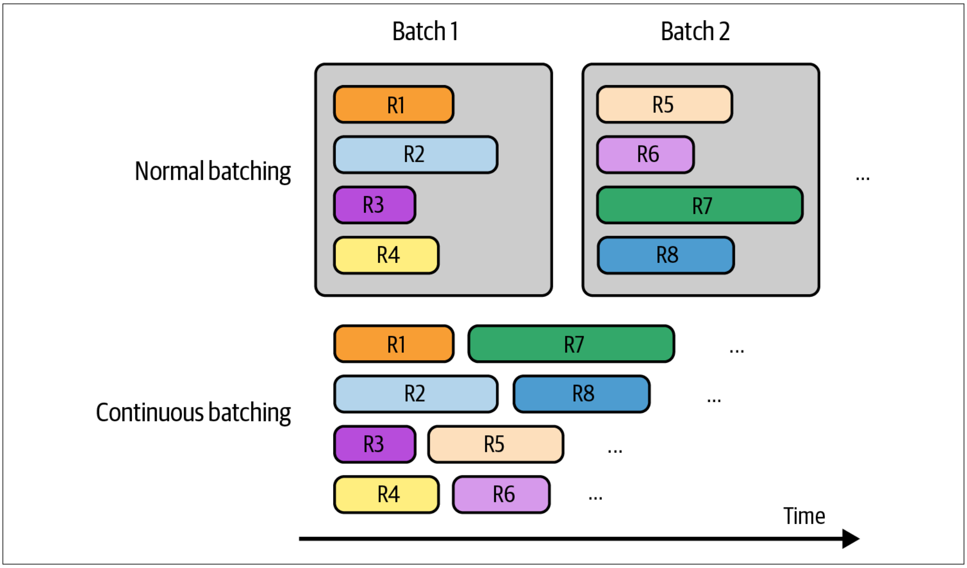


Figure 9-16. With continuous batching, completed responses can be returned immediately to users, and new requests can be processed in their place.

Decoupling prefill and decode

LLM inference consists of two steps: prefill and decode. Because prefill is compute-bound and decode is memory bandwidth-bound, using the same machine to perform both can cause them to inefficiently compete for resources and significantly slow down both TTFT and TPOT. Imagine a GPU that is already handling prefilling and decoding near its peak computational capacity. It might be able to handle another low computational job like decoding. However, adding a new query to this GPU means introducing a prefilling job along with a decoding job. This one prefilling job can drain computational resources from existing decoding jobs, slowing down TPOT for these requests.

One common optimization technique for inference servers is to disaggregate prefill and decode. "DistServe" (Zhong et al., 2024) and "Inference Without Interference" (Hu et al., 2024) show that for various popular LLMs and applications, assigning prefill and decode operations to different instances (e.g., different GPUs) can significantly improve the volume of processed requests while adhering to latency requirements. Even though decoupling requires transferring intermediate states from prefill instances to decode instances, the paper shows communication overhead is not substantial in modern GPU clusters with high-bandwidth connections such as NVLink within a node.

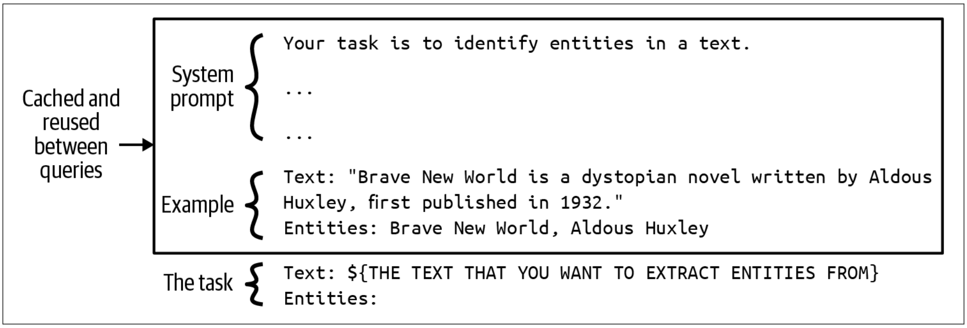
The ratio of prefill instances to decode instances depends on many factors, such as the workload characteristics (e.g., longer input lengths require more prefill compute) and latency requirements (e.g., whether you want lower TTFT or TPOT). For example, if input sequences are usually long and you want to prioritize TTFT, this ratio can be between 2:1 and 4:1. If input sequences are short and you want to prioritize TPOT, this ratio can be 1:2 to 1:1.[25]

Prompt caching

Many prompts in an application have overlapping text segments. A prompt cache stores these overlapping segments for reuse, so you only need to process them once. A common overlapping text segment in different prompts is the system prompt. Without a prompt cache, your model needs to process the system prompt with every query. With a prompt cache, the system prompt needs to be processed just once for the first query.

Prompt caching is useful for queries that involve long documents. For example, if many of your user queries are related to the same long document (such as a book or a codebase), this long document can be cached for reuse across queries. It's also useful for long conversations when the processing of earlier messages can be cached and reused when predicting future messages.

A prompt cache is visualized in Figure 9-17. It's also called a context cache or prefix cache.



```
Your task is to identify entities in a text.

...

...

Text: "Brave New World is a dystopian novel written by Aldous
Huxley, first published in 1932."
Entities: Brave New World, Aldous Huxley

Text: ${THE TEXT THAT YOU WANT TO EXTRACT ENTITIES FROM}
Entities:
```

Figure 9-17. With a prompt cache, overlapping segments in different prompts can be cached and reused.

[25] Talks mentioning the prefill to decode instance ratio include "Llama Inference at Meta" (Meta, 2024).

For applications with long system prompts, prompt caching can significantly reduce both latency and cost. If your system prompt is 1,000 tokens, and your application generates one million model API calls daily, a prompt cache will save you from processing approximately one billion repetitive input tokens a day! However, this isn't entirely free. Like the KV cache, prompt cache size can be quite large and take up memory space. Unless you use a model API with this functionality, implementing prompt caching can require significant engineering effort.

Since its introduction in November 2023 by Gim et al., the prompt cache has been rapidly incorporated into model APIs. As of this writing, Google Gemini offers this functionality, with cached input tokens given a 75% discount compared to regular input tokens, but you'll have to pay extra for cache storage (as of writing, $1.00/one million tokens per hour). Anthropic offers prompt caching that promises up to 90% cost savings (the longer the cached context, the higher the savings) and up to 75% latency reduction. The impact of prompt caching on the cost and latency of different scenarios is shown in Table 9-3.[26]

Table 9-3. Cost and latency reduced by prompt caching. Information from Anthropic (2024).

Use case	Latency w/o caching (time to first token)	Latency with caching (time to first token)	Cost reduction
Chat with a book (100,000-token cached prompt)	11.5 s	2.4 s (–79%)	–90%
Many-shot prompting (10,000-token prompt)	1.6 s	1.1 s (–31%)	–86%
Multi-turn conversation (10-turn convo with a long system prompt)	~10 s	~2.5 s (–75%)	–53%

Parallelism

Accelerators are designed for parallel processing, and parallelism strategies are the backbone of high-performance computing. Many new parallelization strategies are being developed. This section covers only a few of them for reference. Two families of parallelization strategies that can be applied across all models are data parallelism and model parallelism. A family of strategies applied specifically for LLMs is context and sequence parallelism. An optimization technique might involve multiple parallelism strategies.

[26] While llama.cpp also has prompt caching, it seems to cache only whole prompts and work for queries in the same chat session, as of this writing. Its documentation is limited, but my guess from reading the code is that in a long conversation, it caches the previous messages and processes only the newest message.

Replica parallelism is the most straightforward strategy to implement. It simply creates multiple replicas of the model you want to serve.[27] More replicas allow you to handle more requests at the same time, potentially at the cost of using more chips. Trying to fit models of different sizes onto different chips is a bin-packing problem, which can get complicated with more models, more replicas, and more chips.

Let's say you have a mixture of models of different sizes (e.g., 8B, 13B, 34B, and 70B parameters) and access to GPUs of different memory capabilities (e.g., 24 GB, 40 GB, 48 GB, and 80 GB). For simplicity, assume that all models are in the same precision, 8 bits:

- If you have a fixed number of chips, you need to decide how many replicas to create for each model and what GPUs to use for each replica to maximize your metrics. For example, should you place three 13B models on a 40 GB GPU, or should you reserve this GPU for one 34B model?
- If you have a fixed number of model replicas, you need to decide what chips to acquire to minimize the cost. This situation, however, rarely occurs.

Often, your model is so big that it can't fit into one machine. *Model parallelism* refers to the practice of splitting the same model across multiple machines. Fitting models onto chips can become an even more complicated problem with model parallelism.

There are several ways to split a model. The most common approach for inference is *tensor parallelism*, also known as *intra-operator parallelism*. Inference involves a sequence of operators on multidimensional tensors, such as matrix multiplication. In this approach, tensors involved in an operator are partitioned across multiple devices, effectively breaking up this operator into smaller pieces to be executed in parallel, thus speeding up the computation. For example, when multiplying two matrices, you can split one of the matrices columnwise, as shown in Figure 9-18.

Tensor parallelism provides two benefits. First, it makes it possible to serve large models that don't fit on single machines. Second, it reduces latency. The latency benefit, however, might be reduced due to extra communication overhead.

[27] During training, the same technique is called data parallelism.

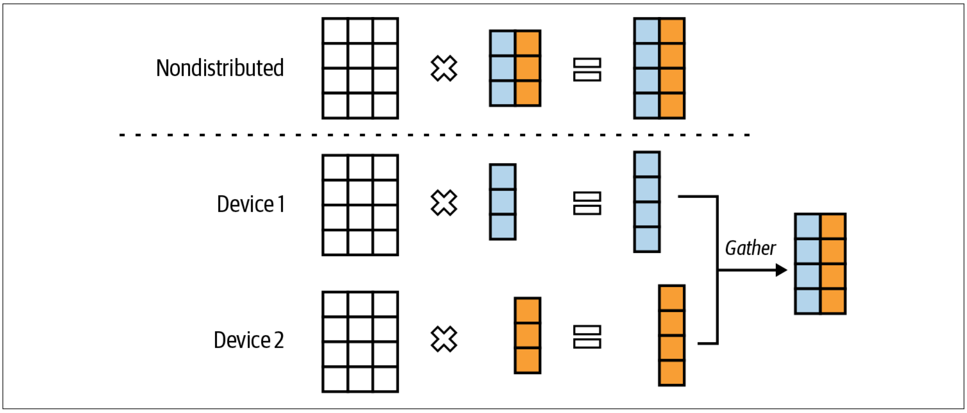


Figure 9-18. Tensor parallelism for matrix multiplication.

Another way to split a model is *pipeline parallelism*, which involves dividing a model's computation into distinct stages and assigning each stage to a different device. As data flows through the model, each stage processes one part while others process subsequent parts, enabling overlapping computations. Figure 9-19 shows what pipeline parallelism looks like on four machines.

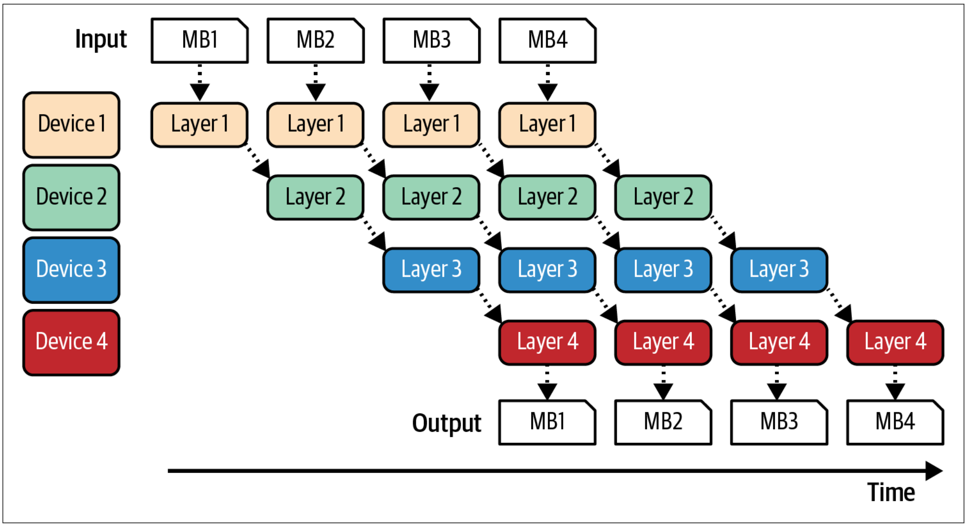


Figure 9-19. Pipeline parallelism enables model splits to be executed in parallel.

Figure 9-19 shows a batch can be split into smaller micro-batches. After a micro-batch is processed on one machine, its output is passed onto the next part of the model on the next machine.

While pipeline parallelism enables serving large models on multiple machines, it increases the total latency for each request due to extra communication between pipeline stages. Therefore, for applications with strict latency requirements, pipeline parallelism is typically avoided in favor of replica parallelism. However, pipeline parallelism is commonly used in training since it can help increase throughput.

Two techniques that are less common but might warrant a quick mention to illustrate the diversity of techniques are *context parallelism* and *sequence parallelism*. They were both developed to make long input sequence processing more efficient, including context parallelism and sequence parallelism.

In *context parallelism*, the input sequence itself is split across different devices to be processed separately. For example, the first half of the input is processed on machine 1 and the second half on machine 2.

In *sequence parallelism*, operators needed for the entire input are split across machines. For example, if the input requires both attention and feedforward computation, attention might be processed on machine 1 while feedforward is processed on machine 2.

Summary

A model's usability depends heavily on its inference cost and latency. Cheaper inference makes AI-powered decisions more affordable, while faster inference enables the integration of AI into more applications. Given the massive potential impact of inference optimization, it has attracted many talented individuals who continually come up with innovative approaches.

Before we start making things more efficient, we need to understand how efficiency is measured. This chapter started with common efficiency metrics for latency, throughput, and utilization. For language model-based inference, latency can be broken into time to first token (TTFT), which is influenced by the prefilling phase, and time per output token (TPOT), which is influenced by the decoding phase. Throughput metrics are directly related to cost. There's a trade-off between latency and throughput. You can potentially reduce cost if you're okay with increased latency, and reducing latency often involves increasing cost.

How efficiently a model can run depends on the hardware it is run on. For this reason, this chapter also provided a quick overview of AI hardware and what it takes to optimize models on different accelerators.

The chapter then continued with different techniques for inference optimization. Given the availability of model APIs, most application developers will use these APIs with their built-in optimization instead of implementing these techniques themselves. While these techniques might not be relevant to all application developers, I believe that understanding what techniques are possible can be helpful for evaluating the efficiency of model APIs.

This chapter also focused on optimization at the model level and the inference service level. Model-level optimization often requires changing the model itself, which can lead to changes in the model behaviors. Inference service-level optimization, on the other hand, typically keeps the model intact and only changes how it's served.

Model-level techniques include model-agnostic techniques like quantization and distillation. Different model architectures require their own optimization. For example, because a key bottleneck of transformer models is in the attention mechanism, many optimization techniques involve making attention more efficient, including KV cache management and writing attention kernels. A big bottleneck for an autoregressive language model is in its autoregressive decoding process, and consequently, many techniques have been developed to address it, too.

Inference service-level techniques include various batching and parallelism strategies. There are also techniques developed especially for autoregressive language models, including prefilling/decoding decoupling and prompt caching.

The choice of optimization techniques depends on your workloads. For example, KV caching is significantly more important for workloads with long contexts than those with short contexts. Prompt caching, on the other hand, is crucial for workloads involving long, overlapping prompt segments or multi-turn conversations. The choice also depends on your performance requirements. For instance, if low latency is a higher priority than cost, you might want to scale up replica parallelism. While more replicas require additional machines, each machine handles fewer requests, allowing it to allocate more resources per request and, thus, improve response time.

However, across various use cases, the most impactful techniques are typically quantization (which generally works well across models), tensor parallelism (which both reduces latency and enables serving larger models), replica parallelism (which is relatively straightforward to implement), and attention mechanism optimization (which can significantly accelerate transformer models).

Inference optimization concludes the list of model adaptation techniques covered in this book. The next chapter will explore how to integrate these techniques into a cohesive system.