Towards AI

LLM Inference Handbook 2026


Anubhav Mandarwal Follow 60 min read · Jun 8, 2026

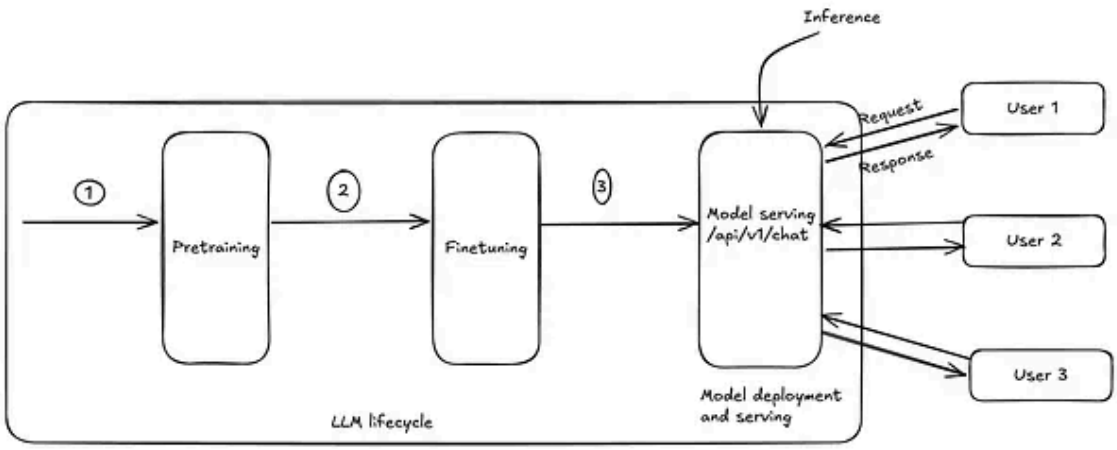
 25  2    

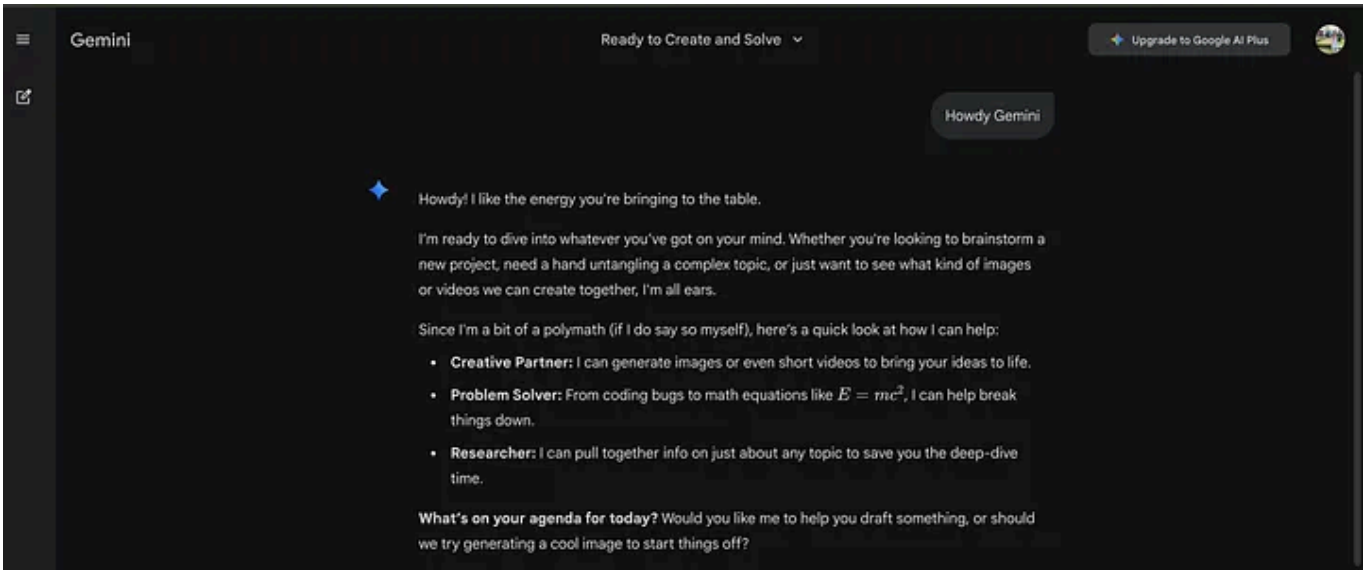
LLM inference is where system design meets AI engineering. In this blog, we will go from the basics of how LLMs generate tokens to advanced optimisation techniques and modern inference frameworks used in production systems in 2026.

INDEX

INTRODUCTION

Every time you send a message to ChatGPT, Claude, or Gemini, something remarkable happens in milliseconds. Somewhere in a data center, a model with billions of parameters wakes up, reads your words, and starts producing a response — one token at a time, at a cost that would have seemed absurd just five years ago.





Gemini deployed by Google and providing the output is example of model inference

Building an LLM requires a researcher's brain to put the right architecture in place, where to use Attention and where to use Mamba layers or any other if building Hybrid Models. Should we create a MoE model (Mixture of Experts) or should all the weights be active, etc.?

The Model deployment on GPU clusters or maintenance after building the model so it can serve real user queries is governed by Engineering.

So LLM Inference is less of research and more of an engineering piece of Art.

And behind that one word inference , sits an entire engineering discipline — one that determines whether your AI product is fast or slow, cheap or expensive, scalable or fragile. Getting inference right is the difference between a demo that impresses and a product that survives.

Training a model happens once. Inference happens billions of times a day. Every inefficiency compounds. Every wasted GPU cycle costs money. Every extra millisecond of latency costs users. This is why the world's best AI teams — at Google, Meta, Anthropic, Mistral — have entire engineering organizations dedicated to nothing but making inference faster, cheaper, and more scalable.

This handbook is your entry point into that world.

We will start from the very beginning — how a model actually produces tokens, what happens during prefill and decode, and why attention is both the most powerful and most expensive part of the transformer. From there we will build up layer by layer: KV cache, quantization, PagedAttention, FlashAttention, speculative decoding, parallelism strategies, and finally the

production frameworks — vLLM, TensorRT-LLM, and SGLang — that tie all of it together.

No step will be skipped. Every concept will be built from first principles, with real numbers, real examples, and real benchmark results from models I have personally run.

By the end, you will not just understand what these optimizations are. You will understand **why** they exist, **what problem** each one solves, and **how** they interact with each other in a real serving system.

LLM inference is less of research and more of an engineering art. Let's learn it properly.

Current state of art model are decoder-based autoregressive models, which means they predict the next token based on the previous tokens. So during pre-training, the model learns to predict the next token based on the previous token.

A token is the basic unit of text or data that a Large Language Model (LLM) processes and understands. Instead of reading individual letters, the model breaks text into smaller chunks called tokens. A token can be a whole word, a part of a word (a subword), or a single punctuation mark.

I have explained the entire process of tokenisation in a blog on Sampling; check it out — https://medium.com/@VrityaCodeRishi/before-attention-what-really-happens-to-your-prompt-inside-an-llm-23ae874f195a

Prefill and Decode

When we send a prompt to an LLM, the model first processes the entire prompt in one forward pass. This stage is called **prefill**. After reading the full prompt, the model predicts the **first output token**. The time from sending the request to receiving the first generated token is called **TTFT(Time to First Token)**.

After that, the model enters the **decode phase**. It generates one token at a time. The first generated token becomes part of the context, then the model predicts the second token. The first and second tokens become part of the context, then it predicts the third token, and so on. This continues until the model reaches `max_tokens` or generates an **EOS**, meaning End of Sequence.

Only after these two processes, Prefill and Decode, do we get our final response from the LLM.

Prompt:

```
The capital of France
```

The model first reads the full prompt and predicts:

```
is
```

So the context now becomes

```
The capital of France is
```

The model again reads the full context and predicts next token:

```
Paris
```

Now the context becomes:

```
The capital of France is Paris
```

Then it predicts the next token:

```
.
```

Then it predicts

```
<EOS>
```

Now the final output becomes:

```
The capital of France is Paris.
```

Now, for every token, the attention layer in the model calculates the K and V vectors.

A Simple Example of Attention: Query, Key, and Value

Before moving further, let's quickly take an example of how attention works and what the Query(Q), Key(K) and Value(V) vectors are.

Before the model predicts the next token, ***every attention layer creates three vectors for each token***:

```
Query  (Q): What is this token looking for?
Key    (K): What information does this token contain?
Value  (V): What information should this token pass forward?
```

Let's take a very small prompt:

```
I love AI
```

Assume the model creates the following simple Key and Value vectors for each token:

```
| Token | Key Vector K | Value Vector V |
| ----- | ------------ | -------------- |
| I     | [1, 0]       | [1, 0]         |
| love  | [0, 1]       | [0, 2]         |
| AI    | [1, 1]       | [3, 1]         |
```

Now, suppose the current token is **AI**, and its Query vector is:

```
Q_AI = [1, 1]
```

The attention layer compares this Query vector with the Key vector of every token using a dot product.

```
Q X K^t, where K^t means transpose of K vector
```

```
Q_AI · K_I    = [1, 1] · [1, 0] = 1
Q_AI · K_love = [1, 1] · [0, 1] = 1
Q_AI · K_AI   = [1, 1] · [1, 1] = 2
```

So the raw attention scores are:

```
I     → 1
love  → 1
AI    → 2
```

These scores are then passed through a softmax function so they become probabilities:

```
I     → 0.21
love  → 0.21
AI    → 0.58
```

This means the token **AI** is paying:

```
21% attention to "I"
21% attention to "love"
58% attention to "AI"
```

Now the model uses these attention weights to mix the Value vectors:

```
Attention Output =
0.21 × [1, 0] +
0.21 × [0, 2] +
0.58 × [3, 1]
```

So the final output becomes approximately:

```
Attention Output ≈ [1.95, 1.00]
```

This final vector is the attention output for the token **AI**.

In simple words, attention allows the model to look at all relevant tokens, decide how important each token is, and then combine their information.

So attention can be understood as:

```
Compare Query with Keys --> Get attention weights --> Mix the Values
```

This mixed information then moves through the next layers of the model and eventually helps predict the next token.

Attention calculates K and V vectors for every token in context.

Now, imagine during decode, as you know, we keep adding output tokens to context , so for every new output token, we again need to compute the K and V vectors of every previous token in context.

Now, taking the same example imagine we give this prompt to the model:

```
The capital of France is
```

Let’s say this prompt has 5 tokens:

```
[The] [capital] [of] [France] [is]
```

During the first forward pass (prefill phase), the model calculates K and V vectors for all 5 input tokens.

```
Token 1: The      → K, V
Token 2: capital  → K, V
Token 3: of       → K, V
Token 4: France   → K, V
Token 5: is       → K, V
```

After this, the model predicts the first output token:

```
Paris
```

Prefill ends here and Decode starts now.

Now, the context becomes:

```
The capital of France is Paris
```

The model would again process the full context and recalculate K and V vectors for all tokens:

```
Token 1: The      → K, V  computed again
Token 2: capital  → K, V  computed again
Token 3: of       → K, V  computed again
Token 4: France   → K, V  computed again
Token 5: is       → K, V  computed again
Token 6: Paris    → K, V  computed for the first time
```

Then the model predicts the next token:

```
.
```

Now the context becomes:

```
The capital of France is Paris.
```

The model would recompute K and V for the full context:

```
Token 1: The      → K, V  computed again
Token 2: capital  → K, V  computed again
Token 3: of       → K, V  computed again
Token 4: France   → K, V  computed again
Token 5: is       → K, V  computed again
Token 6: Paris    → K, V  computed again
Token 7: .        → K, V  computed for the first time
```

and will predict

```
<EOS>
```

The K and V vectors for the previous tokens will remain the same so computing them again and again after every ***decode*** step is wasted work.

Now, think from first-principles engineering: how should this problem be solved?

If something is expensive to compute, and the result will be reused again and again, what should we do?

We store it.

KV Cache

Imagine a user opens a website.

The website has a logo, CSS files, JavaScript files, fonts, and images.

Without caching, every time the user visits a new page, the browser would download the same logo, same CSS, same JavaScript, and same fonts again and again from the server.

That would be wasteful.

The logo has not changed.
The CSS file has not changed.
The JavaScript file has not changed.

So why download them again?

Instead, browsers use a **cache**.

The same principle is also applied here; we store the computer K and V vectors for a token in a cache.

During the P***refill*** phase, as the model is working hard to calculate the first token, as it already has access to the entire input prompt, it caches the K and V vectors for prompt tokens.

KV cache is like browser caching, but instead of caching images, CSS, and JavaScript files, the model caches Key and Value vectors for previous tokens.

This avoids repeated computation and makes token-by-token generation much faster.

Let's use the same prompt to understand

```
The capital of France is
```

During the first forward pass, the model computes the **K and V vectors** for all 5 tokens.

```
The      → K1, V1
capital  → K2, V2
of       → K3, V3
France   → K4, V4
is       → K5, V5
```

The model stores these K and V vectors in memory.

This stored memory is called the **KV cache**.

Now the model predicts the first output token

```
Paris
```

Due to the KV cache

```
The      → use cached K1, V1
capital  → use cached K2, V2
of       → use cached K3, V3
France   → use cached K4, V4
is       → use cached K5, V5
Paris    → compute new K6, V6
```

So instead of recomputing K and V for all 6 tokens, the model only computes K and V for the new token.

Then the model predicts the next token:

```
.
```

Now the context becomes:

```
The capital of France is Paris.
```

So only the new token `.` needs new K and V computation:

```
The      → use cached K1, V1
capital  → use cached K2, V2
of       → use cached K3, V3
France   → use cached K4, V4
is       → use cached K5, V5
Paris    → use cached K6, V6
.        → compute new K7, V7
```

Now compare this with the previous example, where for each token output we again needed to recompute the entire K and V vectors for each token along with the output token, but with the KV cache, we are only calculating the K and V vectors for the output token.

In the ***Prefill*** phase, we are calculating the K and V vectors and storing them in the KV cache. So this operation is **compute-bound**.

In the ***Decode*** phase, *we are reading values from the KV cache more than we are computing the K and V vectors for a token*.

Say if the output is 6 tokens, then for 5 tokens we read the K and V vectors from KV cache and only calculate the vectors for the 6th token. So this operation is **memory-bound**.

Also one thing worth sharing that KV cache can also be ***offloaded to a secondary memory*** to free up some VRAM. It is one of inference optimisations just sharing it here rather than giving it a separate topic when discussing inference optimisation techniques.

Do check out this script I created and ran to see the use of KV cache and how it helps reduce VRAM usage. I found around a 1.43x speedup for a 7b parameter model on L40S

LLM-inferencing/KV-cache/kv-cache.py at main · VrityaCodeRishi/LLM-inferencing

Contribute to VrityaCodeRishi/LLM-inferencing development by creating an account on GitHub.

github.com

Here is my LinkedIn post for results — https://www.linkedin.com/posts/anubhav-mandarwal_kv-cache-stores-the-key-and-value-vectors-activity-7426159586597871616-cGjR?utm_source=share&utm_medium=member_desktop&rcm=ACoAACvsMEQB79txviYP56p0j_vX1eObTiN8284

Key metrics in inference

Congratulations you have understood the fundamental concept involved in inference.

Now it’s time to look into various metrics like TTFT, ITL, Total Latency, Time To Publish, Goodput, Throughput, etc.

These metrics are core tests or benchmarks which work as an evaluation of how good the LLM inference setup or an LLM inference framework is.

TTFT

It stands for Time to First Token, and as its name implies, it means the time it takes for the model to output the first token.

TTFT is basically the time for the P*refill* phase, as the first output is given after the P*refill* phase is completed. So it depends on the input sequence length given to the model (prompt size).

ITL

It stands for ***Inter Token Latency***. Many textbooks also call it **TPOT (Time Per Output Token)**.

Once the first output token is generated after the P**refill phase**, the model enters the D**ecode phase**.

In decode phase, the model generates one token at a time. Each newly generated token is added back to the existing context, and this updated context is used to predict the next token.

The time taken between two consecutive output tokens during this decode phase is called **ITL**.

```
TTFT = Time taken to generate the first output token
ITL  = Time gap between each next output token
```

Reverting to our helpful prompt again,

```
The capital of France is
```

During the prefill phase, the model reads the full prompt and generates the first output token:

```
Paris
```

The time taken to generate `Paris` is **TTFT.**

Now the model enters the decode phase.

The context becomes:

```
The capital of France is Paris
```

Then the model generates the next token:

```
.
```

The time between generating `Paris` and `.` is **ITL.**

Another example with a longer output:

```
Prompt: AI is useful because
```

The model may generate:

```
it helps humans solve problems faster.
```

Token-by-token flow:

```
First output token: it        → TTFT
Next token: helps             → ITL
Next token: humans            → ITL
Next token: solve             → ITL
Next token: problems          → ITL
Next token: faster            → ITL
Next token: .                 → ITL
```

TTFT tells us how quickly the model starts responding, while ITL tells us how smoothly and quickly the model continues generating the rest of the response.

> *Note: As called out in AI Engineering by Chip Huyen the TTFT and TPOT can be different from model perspective and end user perspective as COT (Chain of*

> *Thought) models generate intermediate steps not shown to users. So some teams use **Time To Publish** as metric which is time for first token to be seen by the end user.*

Total Latency

It refers to the time taken by the model to generate all the output tokens. It is the total time taken since the input prompt is given to the model and the model generates all the output tokens.

Time it takes from the end user perspective to receive the final output token.

This is the time for the entire inference, so there must exist a relationship between this and TTFT and TPOT.

Total Latency can be defined mathematically

```
Total Latency = TTFT + TPOT X (Number of output tokens after first token)
```

We can also say

```
Total Latency = TTFT + TPOT X (Number of generated tokens - 1)
```

When comparing latencies across the different inference systems, it's better to use percentiles like p99 or p50 rather than average (median).

Goodput

SLO refers to ***Service Level Objective*** in software engineering. It defines the target reliability or performance level that a service is expected to meet.

For example, an application can define an SLO as 95% of API requests should complete within 2 seconds.

Goodput *in inference refers to the number of requests per second that satisfy the SLO.*

For example, suppose an LLM inference service has this SLO:

```
Requests should complete within 3 seconds.
```

Here we can define, for example, that TTFT should take 2 seconds at max and 0.5 seconds at most for TPOT.

Now imagine the system receives and processes 100 requests per second

But out of these 100 requests:

```
80 requests complete within 3 seconds
20 requests take more than 3 seconds
```

So Goodput is

```
Goodput    = 80 requests/second
```

Throughput

In inferencing ***Throughput refers*** to the number of requests completed per second or per minute.

```
Throughput = Total work done
Goodput    = Useful work done within the SLO
```

In the same example we take in ***Goodput,*** the model processed 100 requests per second is the ***Throughput.***

In inference systems, we usually want two things: low latency and high throughput.

Latency vs Throughput Tradeoff

Computer Science is full of tradeoffs, and AI is no exception in it.

Imagine an LLM server.

If we process requests one by one:

```
Batch size = 1
```

Each request may get a fast response because it does not wait for other requests.

```
Latency: low
Throughput: low
```

But GPU usage may be poor because the GPU is not fully utilised.

Now imagine we increase batching.

```
Batch size = 32
```

Now the GPU processes many requests together, so overall system throughput improves.

```
Throughput: high
GPU utilization: high
```

But each request may have to wait longer inside the batch queue, so latency becomes high.

Tradeoff is

```
Small batch size → lower latency, lower throughput
Large batch size → higher throughput, higher latency
```

So because of this, ***Throughput*** can be misleading

Coming back to the SLO example we took earlier

```
Each request should complete within 3 seconds.
```

Now compare two configurations.

Configuration A: Conservative batching

```
Total throughput = 80 requests/second
Requests meeting SLO = 78 requests/second
Goodput = 78 requests/second
```

This system is processing slightly fewer requests, but almost all of them are useful because they meet the latency target.

Configuration B: Aggressive batching

At first glance, Configuration B looks better because throughput is higher. But many requests are now too slow and violate the SLO.

From an end-user point of view, Configuration A is better because it gives more successful requests within the latency target.

***Goodput** tells us the useful capacity of the inference system, not just how much work was done, but how much work was done within an acceptable latency target.*

> *A good inference system is the one with maximum throughput **while still meeting the latency SLO**.*

AI Accelerators

Now I think it's the perfect time to discuss the hardware on which inference is performed. Later we will look into additional metrics like MBU, GPU Utilisation, MFU, etc.

An AI accelerator is a specialised hardware component designed to perform artificial intelligence and machine learning tasks much faster and more efficiently than a standard device like a CPU, for example.

Common AI accelerators are ***GPU*** and ***TPU***.

If you are interested in concepts behind how GPUs are utilised in AI, I have written a detailed blog post on that; check it out — https://medium.com/@VrityaCodeRishi/graphical-processing-unit-gpu-101-for-artificial-intelligence-3beb30029c28

We will understand inference with GPU in this blog, but remember there are certain chips like ***LPU*** entirely designed for LLM inference, but most commonly today inference is done on GPUs across the industry.

GPUs are powerfully designed to load terabytes of data and perform trillions of compute operations per second.

When we measure a GPU compute for inference, we measure it in terms of Tensor Core compute, as they are responsible for the Matrix Multiply and Accumulate (MMA). Again, if you are interested in understanding GPUs for AI, look at the blog I shared link above.

The MMA operation is of this kind.

```
D = (A X B) + C
```

Unlike CPU only have dozens of threads, GPUs have thousands of threads that can work concurrently and switch tasks in a single clock cycle.

GPU Memory

There are two types of RAM on any chip

1. **DRAM (Dynamic RAM):** General-purpose off-the-chip RAM

2. **SRAM (Static RAM):** Faster, more expensive on-chip memory.

VRAM is a type of DRAM, and cache in the GPU is a type of SRAM

L0 Cache: Instruction cache for a single tensor core in GPU.

L1: Shared memory per SM

L2: Global Cache in GPU

The total amount of VRAM will limit the size of the model you can infer on that GPU. VRAM holds model weights and also KV cache. If trying to infer a model larger than it can fit in VRAM, you will get the standard famous error OOM (Out of Memory).

Floating Point Formats

Compute is measured in FLOPS (floating-point operations per second), and GPUs are capable of trillions or quadrillions of FLOPS.

GPUs are designed to perform a massive number of mathematical operations in parallel. In AI workloads, most of these operations are matrix multiplications, especially inside attention layers and feed-forward layers.

The floating-point format decides **how many bits are used to store each parameter**.

*More bits usually mean better accuracy, but more memory usage and slower computatio*n. *Fewer bits usually mean faster computation and lower memory usage, but with some loss of precision.*

This is why GPUs support different numerical formats like **FP32, TF32, FP16, BF16, FP8, and FP4**. Modern NVIDIA Tensor Cores are specifically designed for mixed-precision computing, where lower precision is used for speed while trying to preserve model accuracy.

A floating-point number is usually divided into three parts:

```
Floating Point Number = Sign + Exponent + Mantissa
```

Think of it like scientific notation.

For example

```
123.45 = 1.2345 × 10²
```

In this number,

```
1.2345 → mantissa/significand
10²    → exponent
+      → sign
```

1. **FP32 — FP32, or 32-bit floating point,**

It is the standard high-precision format used in deep learning.

It uses 32 bits to store one number.

FP32 is useful when we need better numerical stability, especially during training or sensitive calculations. But for large LLMs, FP32 is expensive because model weights, activations, and intermediate tensors consume a lot of GPU memory.

```
If a model has 1 billion parameters:

FP32 memory = 1B × 4 bytes = 4 GB
```

So FP32 is accurate, but not always efficient for large-scale LLM inference.

```
FP32 = 1 sign bit + 8 exponent bits + 23 mantissa bits
```

2. TF32 — TF32, or TensorFloat-32

It is an NVIDIA format designed to speed up FP32-style matrix multiplication on Tensor Cores.

It keeps a similar exponent range to FP32, but uses fewer precision bits internally. This makes it faster than traditional FP32 while still being easier to use than FP16 in many workloads. NVIDIA TensorRT describes TF32 as one of the reduced precision formats that can significantly affect performance and accuracy tradeoffs.

It keeps a similar exponent range to FP32, but uses fewer precision bits internally. This makes it faster than traditional FP32 while still being easier to use than FP16 in many workloads.

3. FP16 — Half Precision

FP16, or 16-bit floating point, stores each number using 16 bits instead of 32 bits.

```
FP32 uses 4 bytes per number
FP16 uses 2 bytes per number
```

So FP16 can reduce memory usage by around half.

This is why FP16 became very popular in deep learning. GPUs can process FP16 matrix multiplications much faster using Tensor Cores.

NVIDIA's mixed precision documentation explains that models can use FP16 storage and Tensor Core math to accelerate matrix multiplication-heavy workloads.

The tradeoff is that FP16 has a smaller numerical range, so it can sometimes suffer from overflow or underflow during training.

```
FP16 = 1 sign bit + 5 exponent bits + 10 mantissa bits
```

4. BF16 — Brain Floating Point 16

It is also a 16-bit format, but it is designed differently from FP16.

The main idea is that BF16 keeps a large range like FP32, but with fewer precision bits

```
FP16 = more precision than BF16, but smaller range
BF16 = less precision than FP16, but larger range
```

For LLMs, BF16 is commonly used during training and inference because it gives a good balance between memory efficiency, speed, and stability.

TensorRT lists BF16 alongside FP16 and TF32 as reduced precision formats used to improve performance with an accuracy tradeoff

```
BF16 = 1 sign bit + 8 exponent bits + 7 mantissa bits
```

5. FP8 — **FP8** uses only 8 bits per number.

6. FP4 — it uses only 4 bits per number

Range vs Precision Tradeoff:

```
More exponent bits  → larger numerical range
More mantissa bits  → better numerical precision
```

For example:

```
FP16 has 5 exponent bits and 10 mantissa bits.
BF16 has 8 exponent bits and 7 mantissa bits.
```

So FP16 can represent numbers with more detail, but BF16 can represent much larger and much smaller values. That is why BF16 is often more stable for deep learning workloads.

Lower precision helps because:

```
1. Model weights take less GPU memory
2. Less data is moved from GPU memory
3. Tensor Cores can perform more operations per second
4. More tokens can be served with the same hardware
```

Example memory usage for 1 billion parameters:

```
FP32  → 1B × 4 bytes = 4 GB
FP16  → 1B × 2 bytes = 2 GB
BF16  → 1B × 2 bytes = 2 GB
```

```
FP8   → 1B × 1 byte   = 1 GB
FP4   → 1B × 0.5 byte = 0.5 GB
```

So, based on the numerical format, the VRAM needed for a model for inference will be calculated. The lower precision allows the model to be faster, but as we saw tradeoff; it will be less accurate

To simply understand this,

Consider a numerical floating-point number stored in memory

```
4.5676785
```

Now we can save space by rounding off this number to say 2 digits

```
4.57
```

So we saved memory space while storing this new rounded-off number but paid the price by losing the accuracy.

Ops: Byte Ratio

Each GPU has a specific compute speed and memory bandwidth.

```
Ops:Byte Ratio = Peak Compute / Memory Bandwidth
```

It tells us,

For every 1 byte loaded from memory, how many operations can the GPU perform.

Assume a GPU has:

```
Peak Compute = 100 TFLOPS or 100 trillion operations/second
Memory Bandwidth = 2 TB/s or 2 trillion bytes/second
```

Now,

```
Ops:Byte Ratio = 100 TFLOPS / 2 TB/s = 50 operations per byte
```

So this GPU can perform 50 operations for every 1 byte loaded from memory.

Now we will discuss arithmetic intensity, and only after that it will be clear why these things matter.

Arithmetic Intensity

It's also called operational intensity; it is the ratio between work and memory traffic.

```
Arithmetic intensity = work / memory traffic
```

It tells us,

How much computation are we doing for every byte of data moved from memory

Assume we want to do this operation:

```
C = A + B
where,
A = [1, 2, 3, 4]
B = [5, 6, 7, 8]
```

Now C will be

```
C = [6, 8, 10, 12]
```

Let's say 1 addition = 1 FLOP, so the total work done here is 4 FLOP

```
1 + 5 = 6
2 + 6 = 8
3 + 7 = 10
4 + 8 = 12

So we did 4 additions so 4 FLOP to get [6, 8, 10, 12]
```

Now memory traffic:

We need to read `A` and `B`, and write `C`. Assuming the numerical format used to store the numbers is FP32 (32 bits = 4 bytes)

```
Read A = 4 numbers × 4 bytes = 16 bytes
Read B = 4 numbers × 4 bytes = 16 bytes
Write C = 4 numbers × 4 bytes = 16 bytes
```

Total memory traffic:

```
Memory Traffic = 16 + 16 + 16 = 48 bytes
```

So arithmetic intensity is:

```
Arithmetic Intensity = 4 FLOPs / 48 bytes = 0.083 FLOPs/byte
```

So for every 1 byte moved from memory, we are doing 0.083 computation.

Now, as we have discussed both Ops: byte ratio and Arithmetic Intensity, let's understand their importance.

Suppose we have:

```
Arithmetic Intensity = 10 ops/byte
GPU Ops:Byte Ratio   = 50 ops/byte
```

This means the workload performs only **10 operations for every 1 byte moved from memory.**

But to fully utilise the GPU's compute capability, the workload needs around:

```
50 operations per byte
```

So the workload is not doing enough computation per byte of memory traffic.

The GPU is capable of doing much more computation, but it is waiting for data to arrive from memory.So the operation happening is **memory-bound.**

The opposite of this situation is that the operation is **compute-bound.**

```
Arithmetic Intensity < Ops:Byte Ratio → Memory-bound
Arithmetic Intensity > Ops:Byte Ratio → Compute-bound
```

*Now connect the dots with the idea that the **Prefill phase is compute-bound** and the **Decode phase is memory-bound.***

Calculating the Memory required for model inference

Now we will look into calculating how much VRAM an N-parameter model needs so we can deploy that model into that GPU.

Inference memory mainly comes from

```
Total VRAM ≈ Model weights + KV cache + activation/runtime overhead
```

If a model has `N` parameters, and each parameter uses a certain number of bytes, then:

```
Model memory = N × bytes per parameter
```

For example, a 7B parameter model inference in FP16 (2 bytes per parameter)

```
Model memory = 7 billion × 2 bytes = 14GB apporximately
```

As we have already learned during **Prefill**, the model stores the Key and Value vectors for previous tokens so it does not recompute them again and again.

```
KV Cache Memory =
```

```
2 × Batch Size × Sequence Length × Number of Layers × Number of KV Heads × Head
```

As is obvious, the reason we are multiplying by 2 is that we have two vectors (Key and Value both stored for each token during **Prefill**).

Now, let's say for our 7B parameter model

```
Batch size B        = 1
Sequence length S   = 4096 tokens
Layers L            = 32
KV heads H_kv       = 32
Head dimension      = 128
Data type           = FP16 = 2 bytes
```

So putting the values in the formula

```
KV Cache = 2 × 1 × 4096 × 32 × 32 × 128 × 2 = 2,147,483,648 bytes
```

So we get approximately ~2GB

Now, in our hypothetical example

```
Model weights ≈ 14 GB
KV cache      ≈ 2 GB
```

```
So total is 14GB + 2GB that is 16GB.
```

Then we still need some extra memory for activations, CUDA kernels, framework overhead, fragmentation, and temporary buffers, which generally is 10–25% extra overhead.

So, assuming 10% in our example, 10% of 16GB will be 1.6GB.

```
Total VRAM needed ≈ 16 GB + 1.6 GB = 17.6 GB
```

So this implies that to infer our 7B model, we need a GPU with at least ~18GB VRAM.

Inference Optimisation Techniques

As we concluded our AI accelerator lesson, now let's move on to how inference can be optimised and what techniques are generally employed.

Quantization

Quantization is an inference optimisation technique in which we represent model weights, activations, or the KV cache using lower-precision formats. This reduces memory usage and memory bandwidth pressure, making LLM inference faster and cheaper, usually with a small tradeoff in model accuracy.

How do we shrink model numbers from FP16/FP32 into FP8/INT8/FP4 without destroying accuracy?

The key idea here is the **scale factor.**

Suppose original tensor values are:

```
[0.12, 0.45, -0.91, 1.20]
```

Now imagine we want to store them in INT4, which has very few values available.

So we cannot directly store decimal values like 0.12.

We first **compress** them using a *scale factor*

Tensor Level Quantization

One scale factor for entire tensor

Our tensor is

```
[0.1, 0.5, -1.2, 0.9]
```

The largest absolute value is 1.2

INT4 represents from -8 to 7. So the largest positive value INT4 can represent is 7.

So scale factor becomes:

```
scale = 1.27 / 7 =~ 0.171
```

Now we quantize

```
0.1/0.171≈1
0.5/0.171≈3
-1.2/0.171≈-7
```

So now new stored tensor becomes

```
[1, 3, -7, 5]
```

Now during inference

```
dequantized value=I NT4 value × scale

eg.  3 × 0.171 = 0.513 , To get value close to original value.
```

Problem with Tensor Level Quantization

Imagine tensor:

```
[0.001, 0.003, 0.005, 1000]
```

Now largest value is:

```
10001000
```

Scale becomes huge.

Tiny values become:

```
0.001/huge scale =~ 0
```

So small information gets destroyed because of outliers.

Channel Level Quantization

Instead of one scale for the entire tensor, use separate scales for different channels/features.

Suppose our tensor is like this

```
[0.1, 0.2, 0.15, 0.18, 50, 60, 55, 65]
```

Notice:

- first half = very small values
- second half = very large values

We can divide the tensor into two channels

Channel 1

```
[0.1, 0.2, 0.15, 0.18]
```

Channel 2

```
[50, 60, 55, 65]
```

Now we will calculate scale independently for each channel.

For Channel 1,

```
Largest value: 0.2
Scale: 0.2/7 =~ 0.028

Quantized:
0.1/0.028  =~ 4
0.2/0.028  =~ 7
0.15/0.028 =~ 5
```

For Channel 2,

```
Largest: 65

Scale: 65/7 =~ 9.28

Quantized:
50/9.28 =~ 5
65/9.28 =~ 7
```

Final Quantized vector

```
[4, 7, 5, 6, 5, 6, 6, 7]
```

Small values survived because they got their own scale.

Block level Quantization

Even finer and more granular than channel-level quantization

Suppose tensor

```
[0.1, 0.2, 0.15, 0.18, 50, 60, 55, 65]
```

Block 1

```
[0.1, 0.2]
```

Block 2

```
[0.15, 0.18]
```

Block 3

```
[50, 60]
```

Block 4

```
[55, 65]
```

Here we will apply the scale independently for each block. We are not going to show the calculation again, as by now you already know how to scale the value.

Final quantized tensor will be

```
[4, 7, 6, 7, 6, 7, 6, 7]
```

Now, quantization can happen during or after training

Suppose the model has weights:

```
[0.12, -0.83, 1.45, 0.67]
```

Originally these are stored in FP32.

Now we want:

- smaller VRAM
- faster inference
- better bandwidth

So we want INT8/FP8/INT4.

When we compress numbers:

```
0.12 -> 0
1.45 -> 1
```

information is lost.

This can hurt accuracy, so how can we reduce this damage

That's where these two approaches come in.

1. **Post Training Quantization (PTQ)**
2. **Quantization-Aware Training (QAT)**

In **Post Training Quantisation (PTQ)**, *training happens normally first, and then we quantize the model.*

First, the training finishes in FP32 or FP16, for example and then we convert the weights to INT8 or INT4.

Engineers run sample prompts through the model.

Example:

- summarization
- reasoning
- coding prompts

Then they observe:

- which layers lose accuracy
- where outliers happen
- best scales

This process is also called calibration.

A few examples of PTQ are

- GPTQ
- AWQ
- GGUF quantization
- bitsandbytes

The downside is the model was NOT trained expecting low precision, so some information loss happens, especially in INT4 and FP4.

In Quantization-Aware Training (QAT),

Instead of quantizing after training, the model trains while simulating quantization.

Suppose an athlete knows: ***Competition will happen wearing heavy shoes.***

So they train wearing heavy shoes themselves. Once competition starts, they are already adapted.

During training, say, the true weight is 0.83

In the forward pass, the model will pretend it becomes 1 due to quantization.

Training gradually adjusts weights to survive this rounding.

Some QAT examples

- NVIDIA NVFP4
- MXFP4
- some GPT-OSS releases
- production mobile AI models

So,

```
PTQ = Train first, compress later

QAT = Train model while teaching it to survive compression
```

Layers quantized in the model

So far, we have talked about how quantization is done and its types. We know we quantize model weights, but weights from which layer of the model?

We apply quantization in the transformer layers, but what parts of the transformer can safely be quantized, and what parts are dangerous to quantize?

A simplified transformer block looks like:

```
Input
  ↓
Linear layers (Q,K,V projections)
  ↓
Attention math (QKᵀ, softmax)
  ↓
KV Cache storage
  ↓
Output projection
  ↓
MLP / FFN linear layers
  ↓
Activations (GELU/SwiGLU)
```

Different parts have different sensitivity to quantization.

The safest weights to quantize are

- **Q projection weights**
- **K projection weights**
- **V projection weights**
- **MLP weights**
- **output projection weights**

They are safe because:

- billions of weights exist
- small rounding error averages out

- redundancy is huge

These layers are what most quantization methods quantize. They generally quantize the linear layers

Examples:

- GPTQ
- AWQ
- GGUF
- bitsandbytes

Now quantization of activations, which are nothing but temporary output during inference, is riskier than linear layers because activations change dynamically per token.

Some tokens produce:

- tiny values
- huge spikes (outliers)

KV cache can also be quantized, and many inference frameworks like vLLM / TensorRT-LLM / SGLang increasingly quantize KV cache because savings are enormous as KV cache is generally huge.

The most dangerous to quantize are the attention math

```
Attenton weight = softmax(QK^t)
```

As softmax is exponential, even tiny input differences can produce huge output differences.

Let’s understand the danger

Suppose scores:

```
[10, 9]
```

Softmax:

```
[0.73, 0.27]
```

Now quantization changes:

```
[10, 8]
```

Softmax becomes:

```
[0.88, 0.12]
```

Massive change from tiny rounding. Attention suddenly focuses on the wrong token.

Attention at token 500:

- depends on previous hidden states
- previous KV cache
- previous attention outputs

So quantization noise can snowball.

The same also applies to **Layer Norm**

Because normalisation is numerically sensitive.

Small errors affect:

- mean
- variance
- scaling

This destabilises the whole network.

So conclusion

```
Safe to quantize      = Storage-heavy components.

Dangerous to quantize = Math-heavy sensitive computations.
```

PagedAttention

PagedAttention is a perfect example of engineering meeting AI. Just like operating systems use paging to manage physical memory efficiently, PagedAttention uses pages to manage KV cache efficiently. The goal is the same: reduce memory waste, avoid fragmentation, and serve more requests with the same hardware.

You know by now KV cache is huge and takes a big chunk of memory, as stored as once continous giant block.

Now this can lead to fragmentation and waste of the GPU memory. During inference, the model weights, activations, and KV cache all need GPU memory. The actual computation happens on GPU compute units, but the data required for computation must be stored and moved through GPU VRAM.

So if VRAM is not managed efficiently, the inference system cannot serve many requests, even if the GPU still has compute capacity left.

Let’s understand this with the help of an example. For simplicity, assume every request is allocated memory based on its maximum possible sequence length.

Suppose three users send requests:

```
Request A → max length = 30 tokens
Request B → max length = 30 tokens
Request C → max length = 30 tokens
```

The system reserves memory like this:

```
GPU KV Cache Memory:

[A reserved 30 slots]
[B reserved 30 slots]
[C reserved 30 slots]
[Free 10 slots]
```

So out of 100 slots:

- 90 slots are reserved
- 10 slots are free

The problem is that LLM output length is unpredictable. One user may generate 10 tokens, another may generate 500 tokens, and another may stop early because of EOS. If we reserve memory for the maximum possible length, a lot of GPU memory remains unused.

Now suppose,

- Request A only generates 12 tokens.
- Request B only generates 18 tokens.

- Request C only generates 10 tokens.

Actual usage:

```
Request A → uses 12 out of 30 slots
Request B → uses 18 out of 30 slots
Request C → uses 10 out of 30 slots
```

So the real KV cache usage is:

```
Actual used memory = 12 + 18 + 10 = 40 slots

Wasted memory      = 90 - 40 = 50 slots
```

- A used 12 slots wasted 18.
- B used 18 slots, wasted 12.
- C used 10 slots, wasted 20.

Now imagine a new request comes in:

```
Request D needs 40 token slots
```

Logically, we have 50 wasted slots sitting inside the previous allocations. But those slots are trapped inside the reserved memory of requests A, B, and C.

The system may not be able to use them easily for request D.

So even though memory exists, it is not available in a useful form.

So I guess now the problem of fragmentation is clear to you. You can see how this is leading to poor GPU utilisation.

PagedAttention solves this by borrowing an old idea from operating systems: **paging**.

Instead of giving every request one large continuous memory block, it divides KV cache into smaller fixed-size blocks and allocates them only when needed.

So the idea changes from reserving one large block upfront to allocating small blocks as the sequence grows.

Now, taking the same example

Without paged attention

```
Actual used memory = 40 slots
Reserved memory    = 90 slots
Wasted memory      = 50 slots
```

Instead of giving each request one large continuous block, it divides KV cache memory into smaller fixed-size blocks.

For example, assume each block can store:

```
1 block = 10 token slots
```

Now the GPU memory is divided like this:

```
[Block 1]
[Block 2]
[Block 3]
[Block 4]
[Block 5]
[Block 6]
[Block 7]
[Block 8]
[Block 9]
[Block 10]
```

```
10 blocks × 10 token slots = 100 token slots
```

Request A uses 12 tokens.

```
Request A needs 2 blocks
So,
A → [Block 1][Block 2]
```

Request B uses 18 tokens:

```
Request B needs 2 blocks
So,
B → [Block 3][Block 4]
```

Request C uses 10 tokens:

```
Request C needs 1 block
So,
C → [Block 5]
```

Now memory looks like this:

```
A: [Block 1][Block 2] → 20 slots allocated, 12 used
B: [Block 3][Block 4] → 20 slots allocated, 18 used
C: [Block 5]          → 10 slots allocated, 10 used
```

Total memory:

```
Used tokens       = 12 + 18 + 10 = 40 slots
Allocated memory  = 20 + 20 + 10 = 50 slots
Wasted memory     = 10 slots
Free memory       = 50 slots
```

Compare this with without paged attention, where the wasted memory was 50 slots, so using paged attention we are able to reduce 80% of wastage.

```
Original wasted slots withotu Paged Attention: 50
New wasted slots in Paged Attention: 10
Slots saved in Paged Attention: 50 - 10 = 40 slots

So wastage reduced = (40 / 50) X 100 = 80%
```

Paged Attention reduces memory waste, avoids fragmentation, and allows the inference server to handle more concurrent requests with the same GPU VRAM.

PagedAttention was introduced by the vLLM paper in 2023. The idea was inspired by virtual memory and paging in operating systems, but vLLM applied this principle to KV cache management in LLM inference.

FlashAttention

FlashAttention is a kernel that fuses several operations to make attention faster.

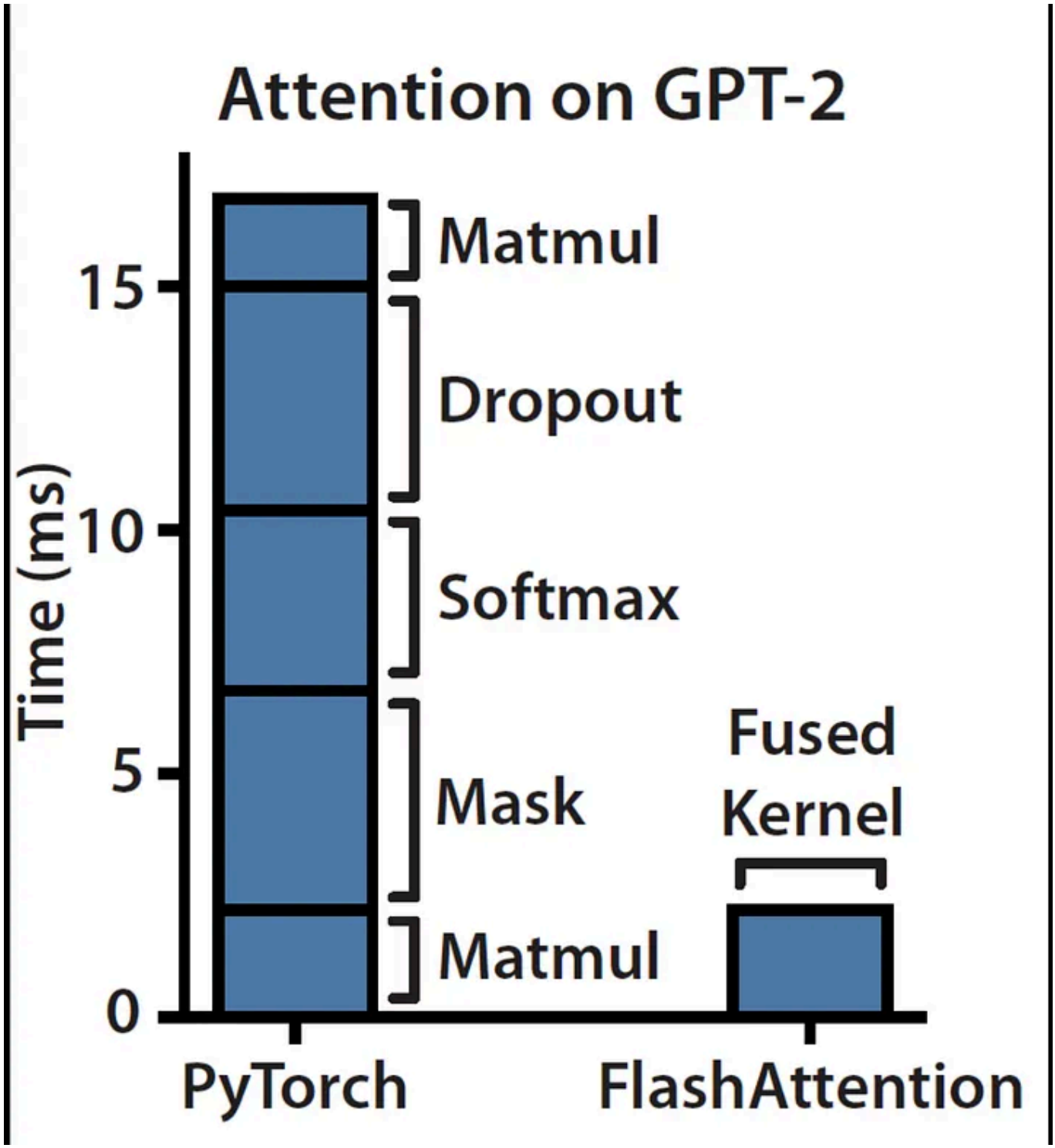


Image Credit: https://gordicaleksa.medium.com/eli5-flash-attention-5c44017022ad

Standard attention does this

```
Attention(Q, K, V) = softmax(QKᵀ / √d) V
```

In manual/eager attention, the model materializes the full attention score matrix:

```
[B, H, S, S]
Where,
B = batch size
H = number of attention heads
S = sequence length
```

The problem is that this matrix grows quadratically with sequence length. So if sequence length doubles, the attention matrix quadruples.

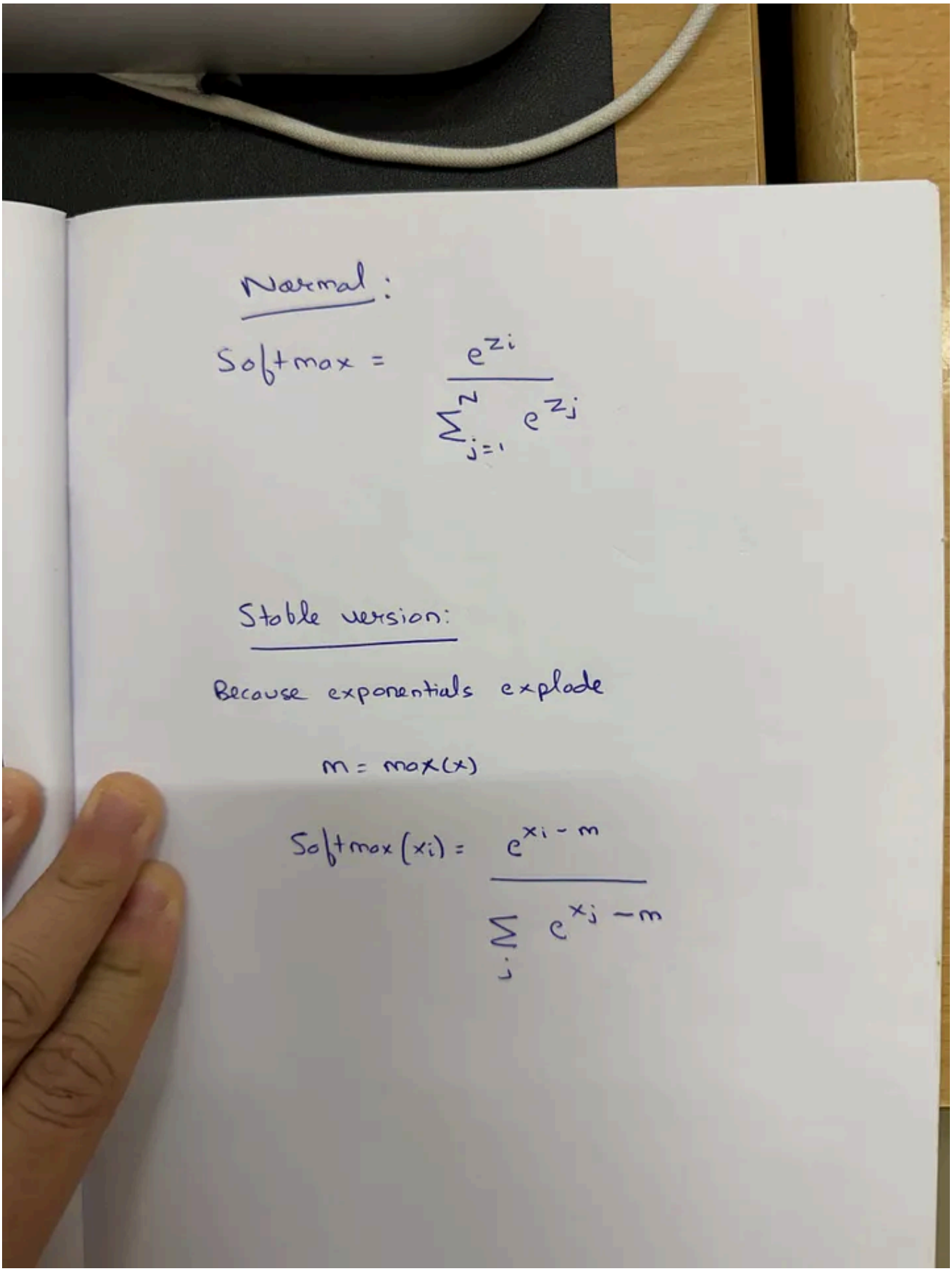
So for long sequence lengths, standard attention becomes heavily memory-bound.

Flash Attention does not change the attention formula. It changes how attention is executed on the GPU.

- Compute attention block by block
- Use SRAM / shared memory efficiently
- Apply online softmax
- Avoid storing the full S × S matrix in HBM

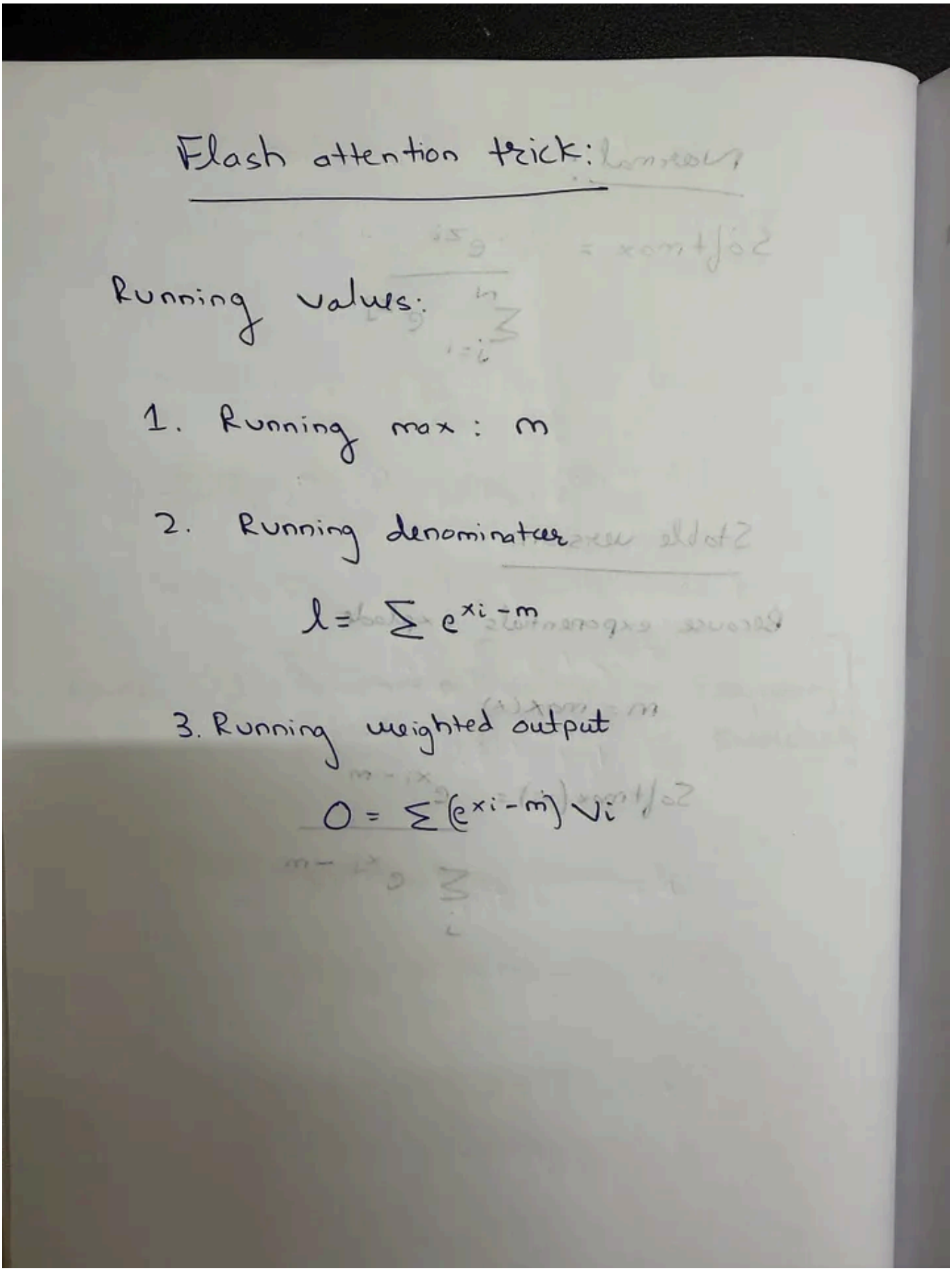
Formula used in FlashAttention

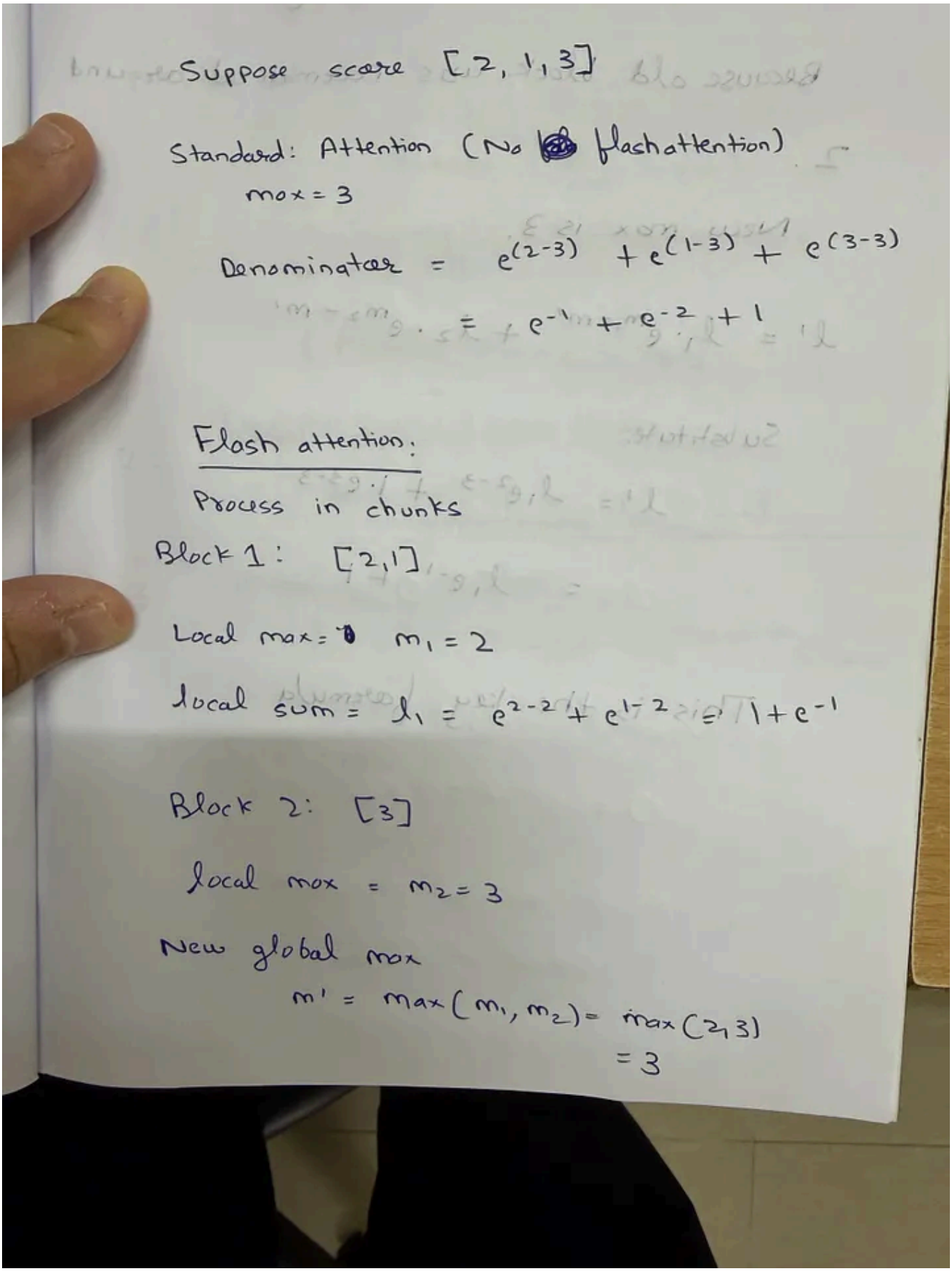
We use online softmax/ Streaming softmax.



Please mind my Handwriting if bad 😅

The Flash Attention trick is to compute attention scores block by block. I am going to pick up a notebook now and will explain the formula to you, as it is harder for me to type mathematical notation here.


Flash attention trick:
Running values:
1. Running max : m
2. Running denominator
$l = \sum e^{x_i - m}$
3. Running weighted output
$O = \sum (e^{x_i - m}) V_i$


Suppose score [2, 1, 3]
Standard: Attention (No flash attention)
max = 3
Denominator = $e^{(2-3)} + e^{(1-3)} + e^{(3-3)}$
$= e^{-1} + e^{-2} + 1$
Flash attention:
Process in chunks
Block 1: [2,1]
Local max = $m_1 = 2$
local sum = $l_1 = e^{2-2} + e^{1-2} = 1 + e^{-1}$
Block 2: [3]
local max = $m_2 = 3$
New global max
$m' = \max(m_1, m_2) = \max(2,3)$
$= 3$

Now rescale old denominator

Because old block was normalized around 2.

New max is 3.

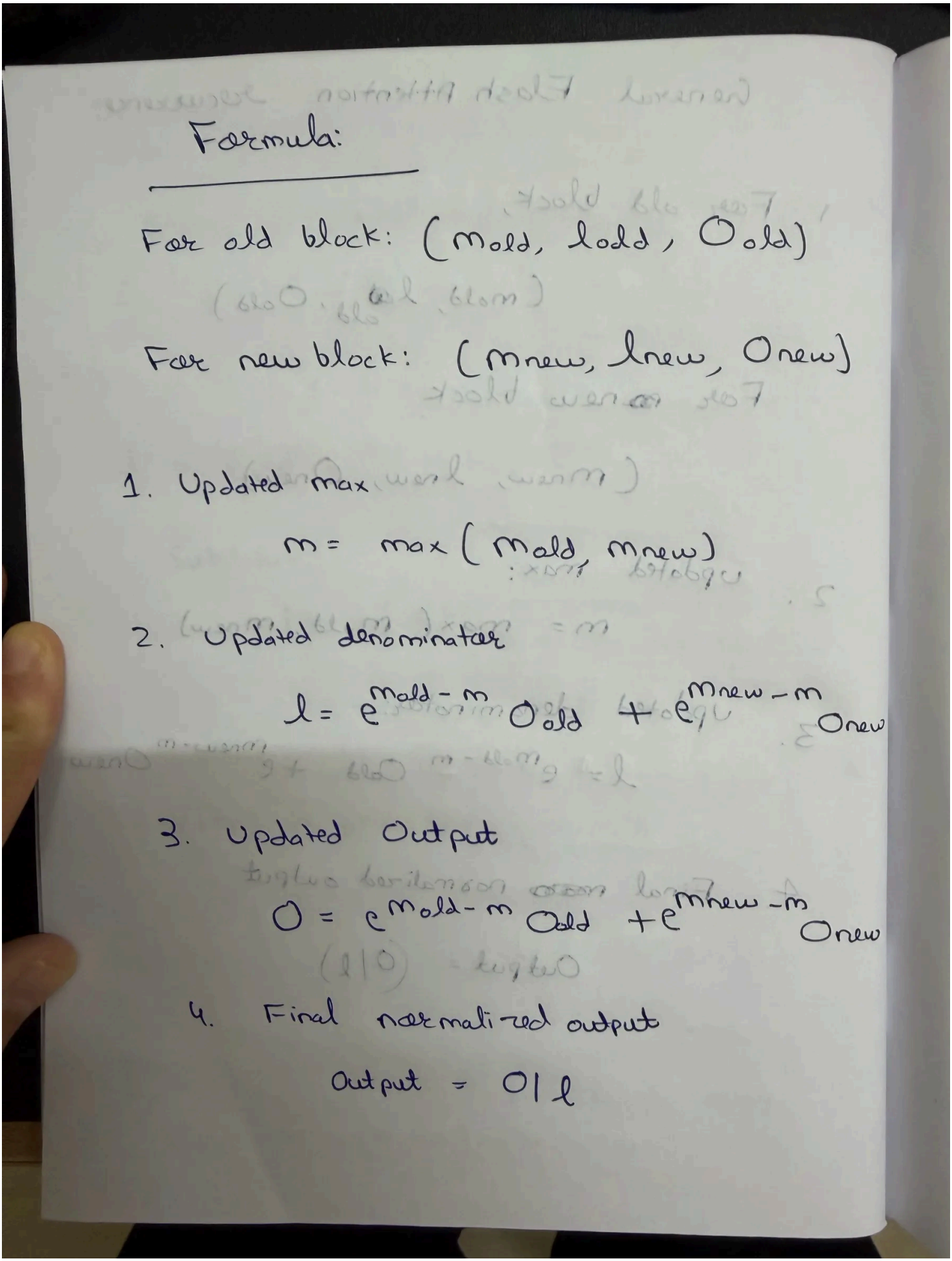
$$l' = l_1 \cdot e^{m_1 - m'} + l_2 \cdot e^{m_2 - m'}$$

Substitute:

$$l' = l_1 e^{2-3} + 1 \cdot e^{3-3}$$

$$= l_1 e^{-1} + 1$$

This is the key formula



Formula:

For old block: $(m_{old}, l_{old}, O_{old})$

For new block: $(m_{new}, l_{new}, O_{new})$

1. Updated max

$$m = \max(m_{old}, m_{new})$$

2. Updated denominator

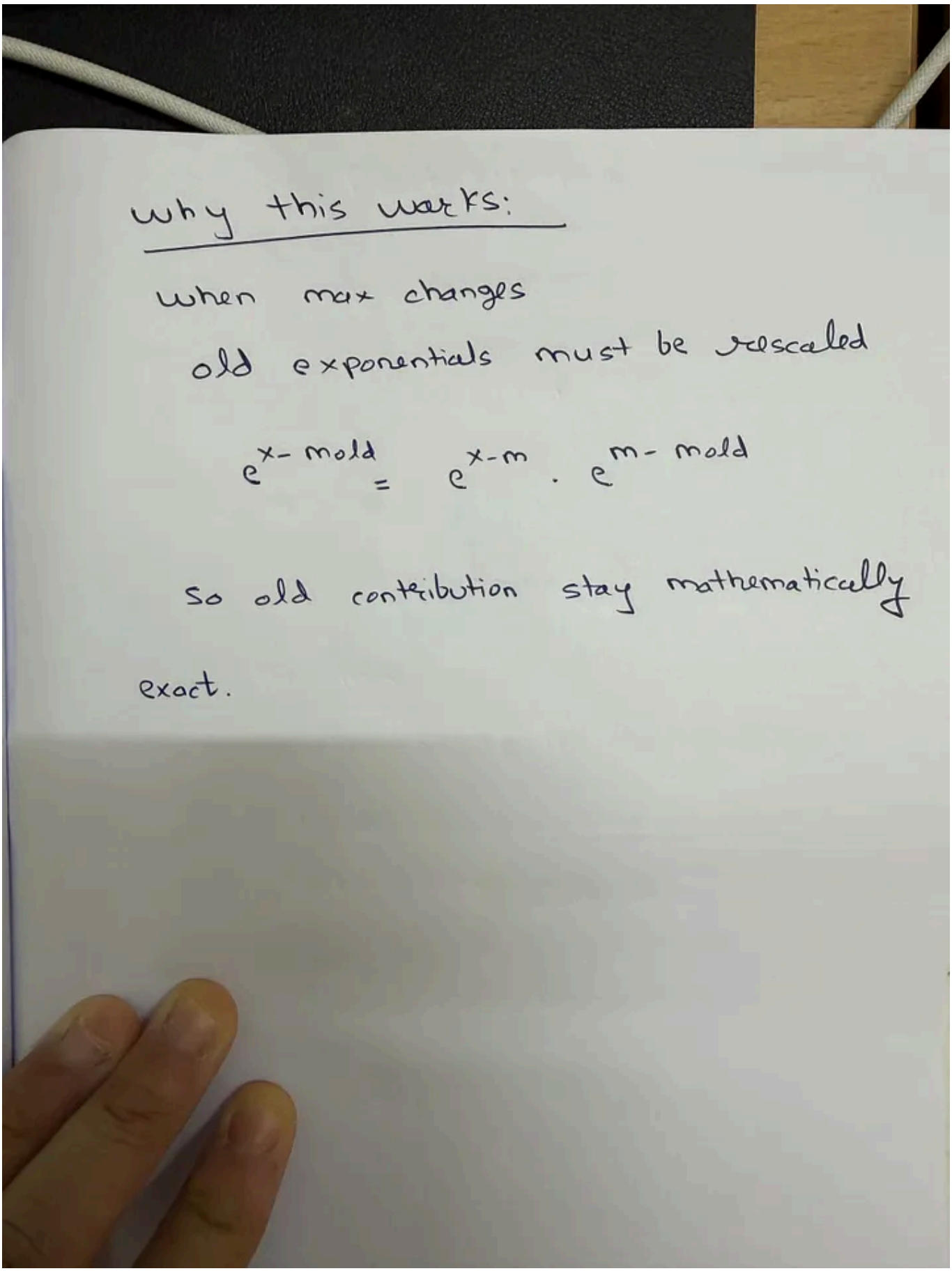
$$l = e^{m_{old} - m} O_{old} + e^{m_{new} - m} O_{new}$$

3. Updated Output

$$O = e^{m_{old} - m} O_{old} + e^{m_{new} - m} O_{new}$$

4. Final normalized output

$$\text{Output} = O / l$$


Why this works:
When max changes
old exponentials must be rescaled
$e^{x-m_{old}} = e^{x-m} \cdot e^{m-m_{old}}$
So old contribution stay mathematically
exact.

Let's take a very small example.

Assume we have **one query token** attending over **four key/value tokens.**

For simplicity, let:

```
Q = [1, 1]
```

Step 0: First Calculate Attention Scores Using Q and K

Let's take a very small example.

Assume we have **one query token** attending over **four key/value tokens.**

For simplicity, let:

```
Q = [1, 1]
```

And the Key matrix is:

```
K =
Token 1: [1, 0]
Token 2: [1, 1]
Token 3: [0, 0]
Token 4: [1, 2]
```

So in matrix form:

```
Q = [[1, 1]]

K =
[[1, 0],
 [1, 1],
 [0, 0],
 [1, 2]]
```

In attention, we calculate scores using:

```
Scores = QKᵀ
```

So first transpose `K`:

```
Kᵀ =
[[1, 1, 0, 1],
 [0, 1, 0, 2]]
```

Now multiply:

```
QKᵀ = [[1, 1]] × [[1, 1, 0, 1],
                  [0, 1, 0, 2]]
```

Now calculate each score:

```
Score for Token 1 = [1, 1] · [1, 0] = 1
Score for Token 2 = [1, 1] · [1, 1] = 2
Score for Token 3 = [1, 1] · [0, 0] = 0
Score for Token 4 = [1, 1] · [1, 2] = 3
```

So the attention scores are:

```
Scores = [1, 2, 0, 3]
```

In real transformers, we usually divide this by:

```
√d_head
```

But to keep this example simple, we will ignore that scaling factor.

Now assume the Value vectors are simple scalar values:

```
V = [10, 20, 30, 40]
```

In real LLMs, each value is a vector, but using one number makes the calculation easier.

So now we have:

```
Scores = [1, 2, 0, 3]
Values = [10, 20, 30, 40]
```

Now attention becomes:

```
Attention Output = softmax(Scores) × V
```

Normal Attention Calculation

First apply softmax on all scores together:

```
Scores = [1, 2, 0, 3]
```

Calculate exponentials:

```
exp(1) = 2.718
exp(2) = 7.389
```

```
exp(0) = 1.000
exp(3) = 20.086
```

Sum:

```
2.718 + 7.389 + 1.000 + 20.086 = 31.193
```

Now softmax weights:

```
Token 1 = 2.718 / 31.193 = 0.087
Token 2 = 7.389 / 31.193 = 0.237
Token 3 = 1.000 / 31.193 = 0.032
Token 4 = 20.086 / 31.193 = 0.644
```

So:

```
Attention Weights = [0.087, 0.237, 0.032, 0.644]
```

Now multiply these weights with values:

```
Output =
0.087 × 10 +
0.237 × 20 +
0.032 × 30 +
```

```
0.644 × 40


= 0.87 + 4.74 + 0.96 + 25.76 =~ 32.33
```

So normal attention gives:

```
Attention Output ≈ 32.33
```

Flash Attention Calculation

Flash Attention gives the same output, but it does not store the full attention matrix.

Instead of processing all scores at once:

```
[1, 2, 0, 3]
```

it processes them in smaller blocks.

Let’s split into two blocks:

```
Block 1:
Scores = [1, 2]
Values = [10, 20]
```

```
Block 2:
Scores = [0, 3]
Values = [30, 40]
```

Flash Attention keeps three running values:

```
m   = running maximum score
l   = running softmax denominator
o   = running weighted value sum
```

Final output is:

```
Output = o/ l
```

For block 1:

```
Scores = [1, 2]
Values = [10, 20]
```

Maximum score:

```
m1 = 2
```

Now subtract max for numerical stability:

```
exp(1 - 2) = exp(-1) = 0.368
exp(2 - 2) = exp(0)  = 1.000
```

Denominator:

```
l1 = 0.368 + 1.000 = 1.368
```

Weighted value sum:

```
o1 = 0.368 × 10 + 1.000 × 20 = 3.68 + 20 = 23.68
```

So after Block 1:

```
m  = 2
l  = 1.368
o1 = 23.68
```

Now Block 2:

```
Scores = [0, 3]
Values = [30, 40]
```

Maximum score:

```
m2 = 3
```

Subtract max:

```
exp(0 - 3) = exp(-3) = 0.050
exp(3 - 3) = exp(0)  = 1.000
```

Denominator:

```
l2 = 0.050 + 1.000 = 1.050
```

Weighted value sum:

```
o2 = 0.050 × 30 + 1.000 × 40 = 1.50 + 40 = 41.50
```

Merge Both blocks:

Block 1 used max:

```
m1 = 2
```

Block 2 used max:

```
m2 = 3
```

Global max becomes:

```
m_new = max(2, 3) = 3
```

Now rescale Block 1 because it was calculated using max `2`, but the new global max is `3`.

```
Rescale factor = exp(2 - 3) = exp(-1) = 0.368
```

Update denominator:

```
l_new = l1 × 0.368 + l2
      = 1.368 × 0.368 + 1.050
      = 0.503 + 1.050
      = 1.553
```

Update accumulated value:

```
o_new = o1 × 0.368 + o2
      = 23.68 × 0.368 + 41.50
      = 8.71 + 41.50
      = 50.21
```

Final output:

```
Output = o_new / l_new
       = 50.21 / 1.553
       =~ 32.33
```

Final Results :

Normal Attention:

```
Output ≈ 32.33
```

FlashAttention:

```
Output ≈ 32.33
```

Different memory strategy.

Normal attention computes and stores the full attention matrix first.

Flash Attention computes attention block by block, keeps only running values, and avoids storing the full `s × s` attention matrix in GPU memory.

Instead of building one giant table first, we read small chunks → process → discard. FlashAttention uses online softmax with running max, running normalisation sum, and running weighted output to compute exact softmax attention block-by-block without storing the full attention matrix.

Imagine you want to calculate a table of 2 up to 10:

```
2×1, 2×2, 2×3, ... 2×10
```

Normal attention approach:

First, you write all intermediate calculations on a big page:

```
2×1 = 2
2×2 = 4
2×3 = 6
2×4 = 8
...
2×10 = 20
```

Then you use that page to get the final answer. This is like standard attention. It creates and stores the full attention matrix.

FlashAttention Approach:

Now imagine you don’t write the full table on a big page.

You calculate one part:

```
2×1 = 2
```

Use it immediately.

Then discard it.

Then:

```
2×2 = 4
```

Use it immediately.

Then discard it.

You only keep the running useful result, not the whole page.

This is like FlashAttention. It computes attention in small blocks and uses the result immediately.

So now connect it with what we learned in compute-bound problem vs memory-bound problem.

So the main problem with attention was memory movement. GPUs are built for compute-bound problems; here, too much memory movement was a bottleneck.

Standard Attention

```
Load Q
Load K
Store QKᵀ
Reload
Store masked
Reload
Store softmax
Reload
```

Now FlashAttention

```
Load tile once
Do more math while data is hot
```

```
    Write final output
```

*So **FlashAttention converted this memory-bound attention problem into a compute-bound problem.** As the number of computations increased, like running max increase computations, it reduced memory movements. FlashAttention doesn't increase theoretical transformer FLOPs dramatically; it mostly reduces wasted I/O.*

To use FlashAttention in Pytorch code is something like

```
import torch
import torch.nn.functional as F
from torch.nn.attention import sdpa_kernel, SDPBackend

device = "cuda"

B, H, S, D = 2, 4, 128, 64

q = torch.randn(B, H, S, D, device=device, dtype=torch.float16)
k = torch.randn(B, H, S, D, device=device, dtype=torch.float16)
v = torch.randn(B, H, S, D, device=device, dtype=torch.float16)

with sdpa_kernel(SDPBackend.FLASH_ATTENTION):
    out = F.scaled_dot_product_attention(
        q,
        k,
        v,
        dropout_p=0.0,
        is_causal=True,
    )

print(out.shape)
```

I ran a benchmark on how much FlashAttention helps w.r.t standard attention

In terms of computation, we saw 5x to 8x improvement and around 22x memory reduction.

Here are the full benchmark results and the script — https://github.com/VrityaCodeRishi/LLM-inferencing/blob/main/flash-attention-benchmark/benchmark-results.md

Speculative Decoding

It exploits the idea that verifying the already predicted output tokens is faster than predicting the output tokens themselves.

Speculative decoding involves not 1 but 2 models. One is a powerful model called the target model, and another is a weaker model of the same family as the target model called the draft model.

Step 1: Draft model generates tokens

Instead of asking the large model to generate one token at a time, we first ask the smaller draft model to generate K tokens quickly.

```
Prompt: The capital of France is

Draft model predicts 4 tokens:

[Paris] [and] [it] [is]
```

here K = 4

Step 2: Target model verifies the draft tokens

Now the large target model takes the original prompt plus the draft tokens and verifies them in a single forward pass.

```
Input to target model:

The capital of France is Paris and it is
```

Instead of generating only one token, the target model checks whether it agrees with the draft model's proposed tokens.

Step 3: Accept tokens from left to right

The target model accepts the draft tokens from left to right.

It keeps accepting tokens as long as they match what the target model would have generated.

```
Draft tokens:
[Paris] [and] [it] [is]

Target model agrees:
[Paris] Agree
[and]   Disagree
```

The target model accepts only the longest valid prefix:

```
Accepted tokens = [Paris]
```

So here:

```
j = 1
```

This means the target model accepted 1 draft token.

Step 4: Reject after first mismatch

Once the target model disagrees at a token, the remaining draft tokens are discarded.

So in this case:

```
Accepted:
[Paris]

Discarded:
[and] [it] [is]
```

Then generation continues from the accepted context and then the next token is predicted by target model in case draft token is rejected.

```
The capital of France is Paris
```

Without speculative decoding, the large model generates one token per forward pass:

```
Pass 1 → Paris
Pass 2 → .
Pass 3 → EOS
```

With speculative decoding, the small model proposes multiple tokens, and the large model verifies them together:

- Small model drafts K tokens quickly
- Large model verifies K tokens in one pass

So if many draft tokens are accepted, the large model effectively generates multiple tokens per expensive forward pass.

Okay, let's take a full example to understand how speculative decoding is faster

Our prompt

```
The capital of France is
```

A normal LLM generates one token at a time:

```
Step 1: Big model predicts " Paris"
Step 2: Big model sees "The capital of France is Paris" and predicts "."
Step 3: Big model sees "... Paris." and predicts "<EOS>"
```

So for 3 output tokens:

```
Paris . <end>
```

The big model had to run **3 sequential forward passes.**

That is slow because each token depends on the previous one.

```
Token 1 -> Token 2 -> Token 3
```

You cannot generate token 3 before token 2 exists.

Now introduce a small draft model.

Prompt

```
The capital of France is
```

The **draft model** quickly guesses multiple tokens:

```
Draft model guesses: Paris . <end>
```

Now we send the whole sequence to the big model:

```
The capital of France is Paris . <end>
```

The big model checks whether those tokens make sense.

If the big model agrees, we accept multiple tokens at once.

So instead of:

```
Big model call 1 -> Paris
Big model call 2 -> .
Big model call 3 -> <end>
```

We do:

```
Draft model quickly proposes: Paris . <end>
Big model verifies them in one forward pass
```

That is the core idea.

Now why can verification be parallelised?

Transformer attention is designed to compute predictions for all positions of a known sequence at once.

During normal decoding, the big model must do this:

```
Run 1:
Input: The capital of France is
Output: Paris

Run 2:
Input: The capital of France is Paris
Output: .

Run 3:
Input: The capital of France is Paris .
Output: <end>
```

That is sequential.

But during verification, we already have the candidate sequence:

```
The capital of France is Paris . <end>
```

So the big model can process the whole sequence in one forward pass.

Transformer attention already supports processing many positions at once. The target model computes logits for many positions simultaneously:

```
Position after "is"      -> predicts Paris
Position after "Paris"   -> predicts .
Position after "."       -> predicts <end>
```

So instead of this:

```
Big model forward pass
Big model forward pass
Big model forward pass
```

we do this:

```
One big model forward pass over the whole draft sequence
```

Example where draft makes a mistake

Now suppose the draft model proposes for our input prompt

```
Paris London <end>
```

This is bad because after Paris, the next token should probably be “ . ”.

```
The capital of France is
```

Target model says

Token	Probability
Paris	0.92
London	0.04
.	0.02
`<end>`	0.02

Draft token:

```
Paris
```

So accepted.

Next,

```
The capital of France is Paris
```

Target model says:

Token	Probability
.	0.80
`<end>`	0.10
Paris	0.05
London	0.05

Draft token:

```
London
```

Rejected.

Now speculative decoding stops accepting from this point. The target model then samples or chooses a replacement token from its own distribution.

Since the next likely replacement as per the target model is a “ . ”

so output becomes

```
Paris .
```

Then decoding continues from there again.

So it's the accept-until-first-mistake rule.

```
Draft proposes:   token1 token2 token3 token4
Target verifies:  yes    yes    no
```

```
Accepted: token1 token2
Corrected by target: token3
```

Then we accept token1 and token2. The token3 will be predicted now by the target model, and then the next speculative round starts.

*Speculative decoding is designed so that the final output distribution still follows the **target model**, not the draft model.*

Now the biggest confusion you might be having at this point is what savings we are getting by using the draft model and why it's cheap.

Don't worry you are not the only one; I also felt the same and explored how using a smaller model for prediction is faster than using the bigger model

itself.

A cheap draft model means a model that costs less to run per token.

Property	Big target model	Draft model
Parameters	14B	0.5B / 1.5B / 3B
Layers	more	fewer
Hidden size	larger	smaller
Attention heads	more	fewer
KV cache size	bigger	smaller
GPU memory bandwidth needed	higher	lower
Time per token	slower	faster

```
Target model: Qwen2.5-14B
Draft model:  Qwen2.5-0.5B
```

The draft model is cheap because one forward pass through it requires far fewer matrix multiplications and less memory movement.

The confusion among many textbooks and literature is that they try to give an impression draft model generates many next predicted tokens in parallel. It's not actually, and those textbooks and literature did not mean that. I am telling all these cause I also got confused here as to how can a model can generate next tokens in parallel, as almost all models today are autoregressive, so they predict the next token one at a time, as they need the previous token to predict the next tokens.

The draft model is also generating one token at a time; it's just that it generates tokens faster than it would have taken for our big target model to

generate the same token. Fewer parameters mean faster token generation, as fewer calculations.

For each generated token, the model must run through all transformer layers.

A 14B model has much larger matrices than a 0.5B model.

Roughly

```
14B model = much bigger matrix multiplications
0.5B model = much smaller matrix multiplications
```

So even if both generate one token at a time:

```
Draft token 1 -> Draft token 2 -> Draft token 3
```

each draft step is much faster for the 0.5B model than the 14B model.

Another simple example to take

Let’s say

```
Big model one token:     40 ms
```

```
Small model one token:    5 ms
```

So if draft generates 4 tokens:

```
4 draft tokens = 4 × 5 ms = 20 ms
```

Then target verifies all 4 in one pass:

```
target verification = maybe 45–60 ms
```

Total:

```
20 ms + 50 ms = 70 ms
```

Normal target decoding(only target model, no speculative decoding):

```
4 target tokens = 4 × 40 ms = 160 ms
```

So

```
Speculative decoding: 70 ms
Without speculative decoding: 160 ms
```

But remember something,

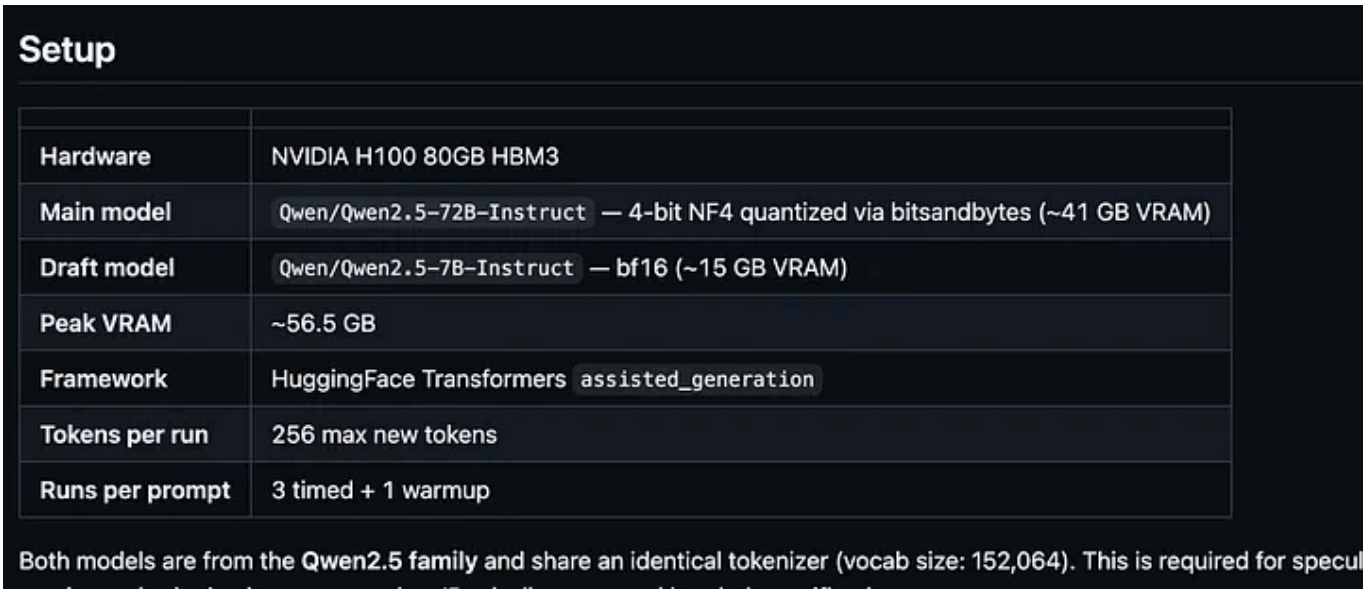
Speculative Decoding is only faster when the draft model is very small compared to the target model, and a higher token acceptance rate means draft tokens are frequently accepted.

The rule of thumb is generally to use a draft model around ~10x smaller than the target model to get the best benefit.

Speculative decoding will be slower if draft tokens are rejected more frequently than accepted, and it can even become slower than just using the target model for inference.

I ran a benchmark on how fast token generation and latency get when we use speculative decoding single model doing the inference.

Here are the results I found on H100



Setup

Hardware	NVIDIA H100 80GB HBM3
Main model	`Qwen/Qwen2.5-72B-Instruct` — 4-bit NF4 quantized via bitsandbytes (~41 GB VRAM)
Draft model	`Qwen/Qwen2.5-7B-Instruct` — bf16 (~15 GB VRAM)
Peak VRAM	~56.5 GB
Framework	HuggingFace Transformers `assisted_generation`
Tokens per run	256 max new tokens
Runs per prompt	3 timed + 1 warmup

Both models are from the **Qwen2.5 family** and share an identical tokenizer (vocab size: 152,064). This is required for specul

Results

Prompt	Baseline (tok/s)	Spec Decoding (tok/s)	Speedup	Latency saved
Transformer self-attention (technical)	11.69	17.31	**1.48x**	7.1s
Red-black tree in Python (code)	11.62	23.65	**2.04x**	11.2s
Turing test philosophy (open-ended)	11.99	12.90	**1.08x**	1.5s
mRNA vaccine process (factual)	11.49	13.43	**1.17x**	3.2s
2008 financial crisis (analytical)	11.62	13.41	**1.15x**	2.9s
Average	11.68	16.14	**1.38x**	—

Check out the full results and it's interpretation here: https://github.com/VrityaCodeRishi/LLM-inferencing/tree/main/speculative-decoding

Prefill Decode Disaggregation

This means we have separate GPUs for:

```
Prefill GPUs → process the input prompt
Decode GPUs  → generate output tokens one by one
```

Imagine an LLM inference server receiving two types of requests:

```
Request A: Long document summarization
Input prompt = 16,000 tokens
Output = 100 tokens

Request B: Chat request
Input prompt = 200 tokens
Output = 100 tokens
```

Imagine only a single GPU is doing both Prefill and Decoding.

Request A is **prefill-heavy** because the model needs to process a very long input prompt.

Request B is lightweight and should ideally get a quick first token quickly.

Suppose request A arrived first and GPU starts processing the 16,000-token prompt.

Now during this time Request B also arrived but it has to wait as request 1 is getting fulfilled currently so TTFT for request B increases even though it's just a small request.

The request B needs to wait not only for Prefill for request A to be over but also for Decode phase for the same to be over too for request A.

Long prompts can block the shorter prompt requests.

Prefill is compute-bound and Decode is memory-bound so they have different behaviour so ideally we can scale in and out both of them individually.

When both phases run on the same GPU, they compete for the same GPU resources. So a large prefill request can disturb ongoing decode requests and increase latency.

Now imagine we split the system into two GPU pools:

```
Prefill GPU pool → handles prompt processing
Decode GPU pool  → handles token generation
```

Now Request A comes in with large 16,000 token prompt and now prefill phase is happening.

Request B comes in and waits for Prefill to be over on Prefill GPU pool. Once Prefill is done for request A and KV cache is created and transferred to the Decode GPU pool.

Now Prefill GPU pool is free so it takes request B Prefill phase meanwhile Decode phase of request A also runs parallely. Once Decode phase of request A ends then Request B decode is picked up and completed.So this helps in speedup.

Now we only took 2 requests examples but consider 100 of such requests then the real benefits of this disaggregation start coming into the picture.

Prefix Caching

So far, we have seen that **KV cache** helps inside a single request.

During decoding, the model does not recompute the K and V vectors for old tokens again and again. It stores them and reuses them.

But now let's take this idea one step further.

What if many requests have the same starting prompt?

For example, in production LLM systems, many requests may share the same:

```
System prompt
Tool definitions
Few-shot examples
RAG instruction template
Long document prefix
Conversation history
```

If these tokens are exactly the same across requests, why should the model recompute their KV cache again and again?

Prefix caching is an inference optimization technique where the inference engine stores the KV cache for a previously processed prompt prefix and reuses it when a new request starts with the same prefix.

If two requests start with the same tokens, reuse the KV cache for the shared starting part.

Imagine we have a chatbot with the same system prompt for every user:

```
You are a helpful AI assistant.
Always answer clearly and concisely.
```

Now User A sends:

```
System: You are a helpful AI assistant.
Always answer clearly and concisely.

Question: Explain KV cache.
```

During prefill, the model computes KV cache for the full prompt.

```
System prompt → KV cache created
User question → KV cache created
```

Now User B sends another request:

```
System: You are a helpful AI assistant.
Always answer clearly and concisely.

Question: Explain Flash Attention.
```

The user question is different, but the system prompt is the same.

Without prefix caching, the model recomputes everything:

```
System prompt → recomputed again
User question → computed
```

With prefix caching, the model reuses the already computed KV cache for the shared prefix:

- System prompt → reused from prefix cache
- User question → computed only for the new suffix

So the work changes from recompute common prefix every time to reuse common prefix and compute only the new part.

If the large shared prompt is already cached in KV cache then it does not need to be recomputed for each request thus for each request TTFT improves.

Prefix caching is most useful when many requests share the same beginning tokens. So this is very useful in agentic AI applications like chatbots, RAG apps with same instruction template etc.

SGLang is famous for optimising the speed for Agentic AI applications using Prefix caching. We will dive into it’s details later like how it’ uses a data structure called RadixTree to use Prefix Caching effectively.

BTW personally I am a fan of SGLang.

Static,Dynamic and Continous Batching

Batching is one of the most important optimization techniques in LLM inference.

Idea is instead of processing one request at a time, group multiple requests together and process them in parallel on the GPU.

This improves GPU utilization because GPUs are very good at parallel computation.

But batching in LLMs is tricky because every request can have:

- Different input length
- Different output length
- Different arrival time
- Different stopping point

In static batching, the batch is fixed before execution starts.

For example, suppose the server receives these requests:

```
Request A → output length = 10 tokens
Request B → output length = 50 tokens
Request C → output length = 20 tokens
```

The system creates one batch:

```
Batch = [A, B, C]
```

Now all three requests are processed together.

At every decode step, the model generates one token for each active request.

But there is one problem

Suppose,

Request A finishes after 10 tokens.

```
Request A finished at token 10
```

Request C finishes after 20 tokens.

```
Request C finished at token 20
```

But Request B still needs 50 tokens.

So the batch keeps running until Request B finishes.

```
Request B finishes at token 50
```

The issue is that after A and C finish, their batch slots become useless.

```
Token 1-10:  A, B, C active
Token 11-20: B, C active, A slot wasted
Token 21-50: B active, A and C slots wasted
```

This is called **static batching.**

So static batching wastes GPU capacity when requests have different output lengths.

Simple way to understand it:

```
Static batching = fixed batch from start to end
```

Dynamic batching improves on this by forming batches from requests that arrive around the same time.

Instead of running each request immediately, the server waits for a small batching window.

For example

```
Batching window = 20 ms
```

If multiple requests arrive within this window, the server groups them together.

Example

```
Time 0 ms  → Request A arrives
Time 5 ms  → Request B arrives
Time 12 ms → Request C arrives
```

Since all three requests arrived within 20 ms, the server creates one batch:

```
Batch = [A, B, C]
```

This is better than processing each request separately because the GPU can process multiple requests together.

But the batch is still mostly fixed once it starts running. So if Request A finishes early, its slot may still remain unused until the batch completes.

Example

```
Request A → 10 output tokens
Request B → 50 output tokens
Request C → 20 output tokens
```

The same issue appears:

```
A finishes early.
C finishes early.
B keeps running.
Some GPU capacity becomes wasted.
```

So dynamic batching helps with request arrival timing, but it does not fully solve the problem of variable output lengths.

So we can say,

```
Dynamic batching = collect nearby requests and batch them together
```

The last one is called **Continuous batching or also called in-flight batching** is the batching technique used by modern LLM inference systems like vLLM.

The idea is do not wait for the whole batch to finish. At every decode step, remove completed requests and insert new waiting requests.

This makes batching much more efficient for LLM serving.

Let's use the same example.

```
Request A → output length = 10 tokens
Request B → output length = 50 tokens
Request C → output length = 20 tokens
```

Initial batch:

```
Batch = [A, B, C]
```

Now decoding starts.

After 10 tokens:

```
Request A finishes
```

In static or dynamic batching, A's slot may remain wasted.

But in continuous batching, the system immediately removes A and adds a new waiting request.

Suppose Request D is waiting:

```
Request D → output length = 30 tokens
```

Now the batch becomes:

```
Batch = [D, B, C]
```

After 20 tokens:

```
Request C finishes
```

Suppose Request E is waiting.

The batch becomes:

```
Batch = [D, B, E]
```

So the GPU keeps working on active requests instead of wasting slots.

So very intuitive way to understand this is

```
Step 1-10:   A   B   C
Step 11-20:  D   B   C
Step 21-50:  D   B   E
```

Here, when one request finishes, another request takes its place. So GPU utilization stays high.

So to summarise

```
Static batching      = fixed batch
Dynamic batching     = batch requests that arrive together
Continuous batching  = keep updating the batch while generation is running
```

Parallelism Strategies

In the blog we have discussed how to calculate the memory required for the model and also about AI accelerators.

You know a GPU has a fixed VRAM and we can only fit a model whose memory requirement fit within the VRAM size of the GPU.

As models become larger, a single GPU is often not enough to store the model or run it efficiently.

So we split the work across multiple GPUs.

Data Parallelism

In data parallelism, each GPU gets a full copy of the model, but processes different data. So we replicate and add the model on each GPU.

```
GPU 1 → same model → Request A
GPU 2 → same model → Request B
GPU 3 → same model → Request C
GPU 4 → same model → Request D
```

Each GPU has the full model loaded.

This is useful when the model fits on one GPU, but we want to serve more requests in parallel.

```
Data parallelism = replicate the model across GPUs to increase request throughpu
```

If one GPU can serve 20 requests/second 4 GPU can serve

```
4 × 20 = 80 requests/second
```

The only requirement is model should be completely fit in a GPU VRAM.

So data parallelism does not help if the model is too large for one GPU.

Tensor Parallelism

In **tensor parallelism**, we split individual model tensors across multiple GPUs. Instead of putting the full layer on one GPU, we divide parts of the layer across GPUs.

This is useful when the model is too large or too compute-heavy for one GPU.

Suppose a model layer has a large matrix multiplication:

```
Y = XW
```

Where W is a huge weight matrix.

With tensor parallelism, we split W across GPUs:

```
GPU 1 → stores part of W
GPU 2 → stores part of W
GPU 3 → stores part of W
GPU 4 → stores part of W
```

Each GPU computes part of the result, and then the results are combined.

Simple way to understand:

```
Tensor Parallelism = split one layer across multiple GPUs
```

So without tensor parallelism

```
GPU 1 stores full layer
```

With tensor parallelism

```
GPU 1 stores 25% of layer
GPU 2 stores 25% of layer
GPU 3 stores 25% of layer
GPU 4 stores 25% of layer
```

In tensor parallelism, a single layer's computation is split across multiple GPUs. Each GPU holds only a shard of the tensor, performs its part of the matrix multiplication, and then the partial results are combined using GPU communication operations like all-reduce or all-gather to produce the final layer output. This helps when model is too large.

The main limitation is there is lot of intercommunication happens between each GPU so tensor parallelism works best when GPUs have fast interconnects like ***NVLink*** *or* ***NVSwitch****.*

Pipeline Parallelism

In **pipeline parallelism**, we split the model by layers across GPUs.

Instead of every GPU holding every layer, each GPU holds only some layers.

Example: Suppose a model has 40 transformer layers.

We can split them across 4 GPUs:

```
GPU 1 → Layers 1–10
GPU 2 → Layers 11–20
GPU 3 → Layers 21–30
GPU 4 → Layers 31–40
```

The input passes through GPU 1 first, then GPU 2, then GPU 3, then GPU 4.

```
Pipeline Parallelism = split the model vertically by layers
```

Main benefit is that model that cannot fit on one GPU can be split across multiple GPUs. But this also leads to something called **pipeline bubbles**.

A **pipeline bubble** means some GPUs may sit idle while waiting for work from previous GPUs.

Let’s say one request enters the pipeline.

```
Time Step 1:
GPU 1 → working
GPU 2 → idle
GPU 3 → idle
GPU 4 → idle
```

After GPU 1 finishes its part, the request moves to GPU 2.

```
Time Step 2:
GPU 1 → idle
GPU 2 → working
GPU 3 → idle
GPU 4 → idle
```

Then it moves to GPU 3.

```
Time Step 3:
GPU 1 → idle
GPU 2 → idle
GPU 3 → working
GPU 4 → idle
```

Then GPU 4.

```
Time Step 4:
GPU 1 → idle
GPU 2 → idle
GPU 3 → idle
GPU 4 → working
```

So for a single request, most GPUs are idle most of the time. That idle time is the pipeline bubble.

The most common way to reduce pipeline bubbles is **microbatching**.

Instead of sending one large batch through the pipeline, we split it into smaller pieces called **microbatches.**

For example, suppose we have a batch of 8 requests.

Instead of sending all 8 together as one batch we split it into 4 microbatches:

```
Microbatch 1 = 2 requests
Microbatch 2 = 2 requests
Microbatch 3 = 2 requests
Microbatch 4 = 2 requests
```

Now we send these microbatches one after another into the pipeline.

This keeps all GPUs busy.

Assume we have 4 GPUs and 4 microbatches:

```
M1, M2, M3, M4
```

The pipeline now looks like this:

```
Time →      T1      T2      T3      T4      T5      T6      T7
GPU 1       M1      M2      M3      M4      -       -       -
GPU 2       -       M1      M2      M3      M4      -       -
```

```
GPU 3       -       -       M1      M2      M3      M4      -
GPU 4       -       -       -       M1      M2      M3      M4
```

Now after the pipeline is filled, all GPUs are working at the same time.

So if we lok at T4

```
GPU 1 → M4
GPU 2 → M3
GPU 3 → M2
GPU 4 → M1
```

So instead of one request slowly moving through the pipeline while other GPUs wait, multiple microbatches are flowing through different stages at the same time.

This is how microbatching reduces pipeline bubbles.

Expert Parallelism

Expert parallelism is mainly used in **Mixture of Experts — MoE** models.

In a dense model, every token usually goes through the same feed-forward network.

But in an MoE model, there are many expert networks, and only a few experts are activated for each token.

```
Token: "Paris"

Router chooses:
Expert 2 and Expert 7
```

So instead of sending the token through all experts, the model sends it only to selected experts.

With expert parallelism, different experts are placed on different GPUs.

```
GPU 1 → Expert 1, Expert 2
GPU 2 → Expert 3, Expert 4
GPU 3 → Expert 5, Expert 6
GPU 4 → Expert 7, Expert 8
```

If a token needs Expert 7, it is routed to GPU 4. This is useful because MoE models can have huge total parameter count, but only a small fraction of parameters are active per token.

```
Expert Parallelism = split MoE experts across GPUs
```

So MoE gives the model more capacity without using all parameters for every token.

```
Total model size = 100B parameters
Active per token = 10B parameters
```

The main overhead is due to routing and communication.

Hybrid Parallelism

In real large-scale systems, we usually do not use only one type of parallelism.

We combine multiple types.

This is called **hybrid parallelism**.

Suppose we have 16 GPUs.

We may use:

- Tensor parallelism inside a node
- Pipeline parallelism across model stages
- Data parallelism across replicas
- Expert parallelism for MoE expert

Example:

```
8 GPUs → one model replica using tensor parallelism
```

```
8 GPUs → another model replica using tensor parallelism
```

Here we are using

```
Tensor Parallelism → split each model replica across 8 GPUs
Data Parallelism   → run multiple replicas for more throughput.
```

Offline batching inference

Offline batching inference is used when we do not need an immediate response.

Instead of serving one user request instantly, we collect many inputs together and run them as a large batch on the GPU.

Imagine

```
We have 10,000 customer reviews.
We want the LLM to summarize or classify all of them.
```

Since this is not real-time chat, we can batch many reviews together:

```
Batch 1 → 128 reviews
Batch 2 → 128 reviews
Batch 3 → 128 reviews
...
```

So in simple words

```
Online inference  = user is waiting, latency matters
Offline inference = user is not waiting, throughput matters
```

Generally inference API for batch API is different from the online inference we use day to day.

This is used in use cases where we don't need results immediately and the input prompts are very large.

Inference With Reference

Sometimes, the output generated by an LLM heavily overlaps with some reference text already available in the input.

For example, suppose we ask a model to modify this code:

```
def add(a, b):
    return a + b
```

Prompt will be like :

```
Add type hints to this function:
```

```
def add(a, b):
    return a + b
```

Output might be:

```
def add(a: int, b: int) -> int:
    return a + b
```

Notice that most of the output is copied from the reference code and only small part changed.

Without optimization, the model generates every token one by one:

```
def → add → ( → a → : → int → ...
```

But many of these tokens already exist in the reference.

So in this optimisation the inference engine looks at the reference text and selects a span that may appear in the output.

For example, from the original code:

```
return a + b
```

The system copies these tokens as a candidate continuation.

Then the LLM verifies in one forward pass whether these copied tokens are valid as the next output tokens.

```
Normal inference:
Generate every output token one by one.

Inference with reference:
Copy likely tokens from the reference,
verify them in parallel,
and only generate new tokens when needed.
```

Inference with Reference speeds up generation by reusing text that already exists in the reference, instead of making the model regenerate the same tokens one by one.

Inference Frameworks

So far, we have discussed the core building blocks of LLM inference: P**refill, Decode, KV cache, batching, quantization, parallelism, speculative decoding, and memory optimization**.

But in production, we usually do not implement all these optimizations from scratch.

This is where **inference frameworks** come in.

An inference framework is a system designed to serve LLMs efficiently on GPUs.

It handles the complex engineering required to make model serving fast, scalable, and cost-efficient.

A good inference framework usually takes care of:

- Model loading
- Request scheduling
- KV cache management
- Continuous batching
- Quantization support
- Tensor/pipeline parallelism
- GPU memory optimization
- Streaming responses
- Throughput and latency tradeoffs

If you are from coding background you will understand the word frameworks it's same principle just like we have Nextjs,Django etc.

We have frameworks for LLM inferencing which will take care of most of the optimisation techniques we discussed.

We will discuss 3 top frameworks currently used in industry as of June 2026.

vLLM

This is the most popular and most used framework across the industry as of now.

If you are looking for inference your model your first line of thought will go to this framework.

It is very simple to use and this is the first framework which bring **PagedAttention** to industry.

vLLM also supports important production-serving features other than PagedAttention like:

- Model loading
- Request scheduling
- KV cache management
- Continuous batching
- Quantization support
- Tensor/pipeline parallelism
- GPU memory optimization
- Streaming responses
- Throughput and latency tradeoffs

To use vLLM

1. We can use the native installation

```
pip install vllm openai
```

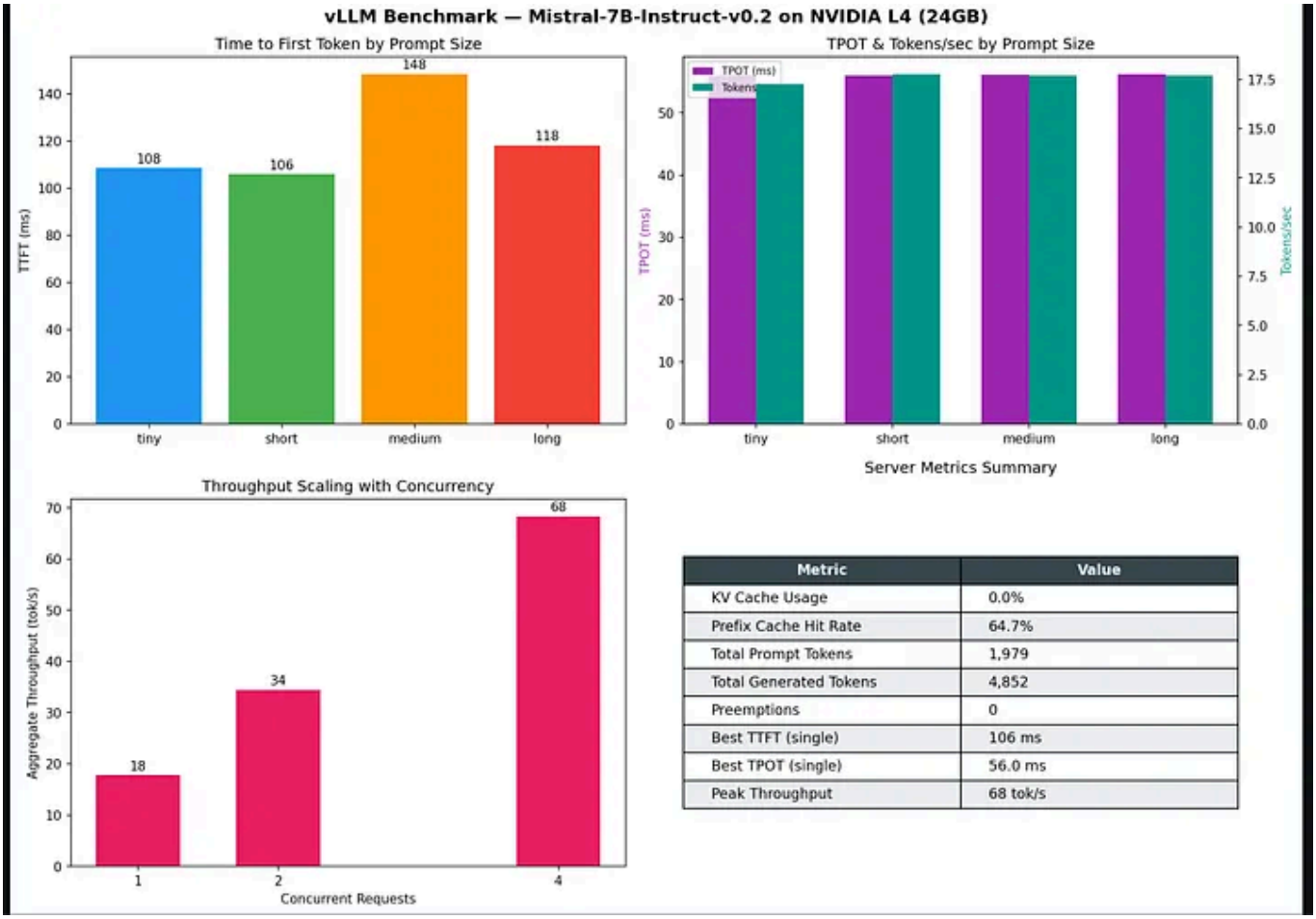
```
vllm serve mistralai/Mistral-7B-Instruct-v0.2 \
  --dtype bfloat16 \
  --max-model-len 8192 \
  --enable-prefix-caching \
  --gpu-memory-utilization 0.90 \
  --host 0.0.0.0 \
  --port 8000
```

2. We can use docker container

Here is the **docker-compose file** I used when experimenting with vLLM

```
services:
  vllm:
    image: vllm/vllm-openai:v0.14.1-cu130 # Change the image as per your driver.
    command:
      [
        "mistralai/Mistral-7B-Instruct-v0.2",
        "--dtype",
        "bfloat16",
        "--max-model-len",
        "8192"
      ]
    network_mode: host # On the Brev instance docker bridge was not resolving ho
    environment:
      - HF_TOKEN=${HF_TOKEN:-}
    volumes:
      - ${HOME}/.cache/huggingface:/root/.cache/huggingface
      - ./gai.conf:/etc/gai.conf:ro
    ipc: host
    gpus: all
    restart: unless-stopped
```

I inference a mistral 7B model on L4 GPU instance and captured the metrics



You can look more into **my repo** with more details

LLM-inferencing/vLLM at main · VrityaCodeRishi/LLM-inferencing

Contribute to VrityaCodeRishi/LLM-inferencing development by creating an account on GitHub.

github.com

TensorRT-LLM

TensorRT-LLM is NVIDIA's inference optimization framework for running LLMs efficiently on NVIDIA GPUs.

TensorRT-LLM includes optimizations such as:

- Optimized CUDA kernels
- Paged KV cache
- In-flight batching
- Tensor parallelism
- Pipeline parallelism
- Expert parallelism
- FP8 / FP4 / INT8 / INT4 quantization
- Speculative decoding
- Chunked prefill
- Disaggregated serving

If vLLM is known for making LLM serving easy and efficient with PagedAttention and continuous batching, TensorRT-LLM is known for going deep into NVIDIA GPU-level optimization: custom kernels, TensorRT engines, quantization, parallelism, and production-grade runtime execution. NVIDIA describes TensorRT-LLM as an open-source library for accelerating and optimizing LLM inference on NVIDIA GPUs, with support for single-GPU, multi-GPU, and multi-node deployments

Triton Inference Server is NVIDIA's production model-serving server.

TensorRT-LLM optimizes and runs the LLM efficiently, but Triton helps expose that model as a production service.

```
TensorRT-LLM = optimized LLM inference engine
Triton       = production serving layer
```

Triton provides the serving infrastructure around the model:

- HTTP/gRPC APIs
- Request routing
- Model repository
- Dynamic batching
- Concurrent model execution
- Health checks
- Metrics
- Kubernetes-friendly deployment
- Multiple backend support

In production, TensorRT-LLM and Triton are often used together.

A simple flow looks like this

```
User request
   ↓
Triton Inference Server
   ↓
Preprocessing / tokenization
   ↓
TensorRT-LLM backend
   ↓
```

```
Optimized LLM inference on NVIDIA GPU
   ↓
Postprocessing / detokenization
   ↓
Response
```

I really wanted to go deep into TensorRT-LLM and Triton server **but I actually already added a blog on there internal working already.** I will highly encourage you to check it out. This also contains how to setup and use TensorRT-LLM with Triton server.

https://medium.com/@VrityaCodeRishi/llm-inferencing-with-tensorrt-llm-triton-inference-server-cb25bdb259ec

I will also encourage to check my repo out

LLM-inferencing/tensorrt-llm-with-triton-server at main · VrityaCodeRishi/LLM-inferencing

Contribute to VrityaCodeRishi/LLM-inferencing development by creating an account on GitHub.

github.com

SGLang

SGLang is an open-source framework for high-performance LLM and multimodal model serving. It is designed for low-latency and high-throughput inference, from a single GPU setup to large distributed clusters

The main idea behind SGLang is that modern LLM applications are not always simple one-prompt-one-response calls. Many real applications involve:

- Multi-turn chat
- RAG pipelines
- Agent workflows
- Tool calling
- Structured JSON generation
- Multiple chained LLM calls

SGLang is built to make these workloads faster and easier to control.

Its most important optimization is **RadixAttention.**

RadixAttention automatically reuses KV cache across requests when they share common prefixes. This is very useful for workloads where many prompts start with the same system prompt, tool definitions, few-shot examples, conversation history, or RAG template.

Instead of recomputing the same prefix again and again, SGLang stores and reuses the KV cache using a radix tree structure

```
Request 1:
System prompt + tool definitions + Question A

Request 2:
System prompt + tool definitions + Question B
```

The beginning of both requests is the same.

Without SGLang-style KV reuse:

```
System prompt + tool definitions → recomputed again and again
```

With RadixAttention:

```
Shared prefix → reused from cache
Only unique question part → computed
```

This reduces **Prefill** computation and improves **TTFT**, especially for agentic and RAG workloads.

SGLang also focuses heavily on **structured generation**. It supports controlled outputs like JSON, regex, and schema-based generation, which is useful when the LLM output must follow a strict format for downstream systems. it does it using Finite State Machine (FSM). When you define a structured output (e.g., using a Pydantic schema), SGLang converts that schema into a finite state machine. During generation, it masks out any tokens that would violate the structure. This means the model can only produce outputs that are structurally valid so no more malformed JSON, no broken formats. Instead of hoping the model follows instructions, the structure is mathematically enforced

In multi-GPU setups, most systems use round-robin routing which ignores where the cache lives. SGLang uses cache aware router which routes requests to the GPU that already has the relevant prefix cached. This

dramatically improves cache hit rates and reduces redundant computation while still balancing load to avoid hotspots.

GPUs wait for the CPU to prepare the next batch of requests. SGLang pipelines this process , While the GPU is computing batch N, the CPU prepares batch N+1 in parallel. This removes idle gaps and keeps GPU utilization extremely high even under heavy load

The SGLang paper describes its runtime optimizations including RadixAttention for KV cache reuse and compressed finite-state machines for faster structured output decoding

I inference a mistral 7B model with SGLang on L40S GPU.

Checkout the repository here https://github.com/VrityaCodeRishi/LLM-inferencing/tree/main/SGLang

You can start the SGLang like this

```
uv pip install sglang

python3 -m sglang.launch_server \
  --model-path qwen/qwen2.5-0.5b-instruct \
  --host 0.0.0.0 \
  --port 30000
```

We can then send the inference request on localhost

```
import requests

url = "http://localhost:30000/generate"

response = requests.post(url, json={
    "text": "How Hinata in Haikyuu like passion,motivation and hard workcan help
    "sampling_params": {
        "max_new_tokens": 150,
        "temperature": 0.8,
        "top_p": 0.95
    }
})

result = response.json()
print("Generated text:", result["text"])
print("Tokens generated:", result["meta_info"]["completion_tokens"])
```

In SGLang, there are two parts:

1. Frontend Language
2. backend runtime

SGLang is not only an inference runtime. It also provides a **frontend language** for writing structured LLM programs.

The frontend is a Python-embedded language that makes it easier to write complex LLM workflows involving:

```
import sglang as sgl

backend = sgl.RuntimeEndpoint("http://localhost:30000")
sgl.set_default_backend(backend)

@sgl.function
```

```
def simple_qa(s, question):
    s += sgl.system("You are a helpful AI assistant.")
    s += sgl.user(question)
    s += sgl.assistant(sgl.gen("answer", max_tokens=200))


state = simple_qa.run(question="What makes SGLang fast?")
print(state["answer"])
```

We can also use this for multi turn conversations

```
import sglang as sgl

backend = sgl.RuntimeEndpoint("http://localhost:30000")
sgl.set_default_backend(backend)

@sgl.function
def multi_turn(s, topic):
    s += sgl.system("You are an expert teacher.")

    # Turn 1: Introduction
    s += sgl.user(f"Explain {topic} to me like I'm a beginner.")
    s += sgl.assistant(sgl.gen("intro", max_tokens=150))

    # Turn 2: Follow-up
    s += sgl.user("Can you give me a practical example?")
    s += sgl.assistant(sgl.gen("example", max_tokens=150))

    # Turn 3: Summary
    s += sgl.user("Summarize the key points in 3 bullet points.")
    s += sgl.assistant(sgl.gen("summary", max_tokens=100))

state = multi_turn.run(topic="prefix caching in LLM inference")

print("=== Introduction ===")
print(state["intro"])
print("\n=== Example ===")
print(state["example"])
print("\n=== Summary ===")
print(state["summary"])
```

Here,

```
backend = sgl.RuntimeEndpoint("http://localhost:30000")
sgl.set_default_backend(backend)
```

This connects your SGLang frontend code to the running SGLang backend server.

```
@sgl.function
def multi_turn(s, topic):
```

This defines an SGLang function.

Think of this as a programmable LLM workflow. Instead of sending one prompt manually, we define a multi-turn conversation as code.

```
s += sgl.system("You are an expert teacher.")
```

Adds a **system message**.

This tells the model how it should behave during the conversation.

```
s += sgl.user(f"Explain {topic} to me like I'm a beginner.")
```

Adds the first user message.

```
s += sgl.assistant(sgl.gen("intro", max_tokens=150))
```

This asks the model to generate an assistant response. The generated output is stored under the key intro.

We can create second turns too

```
s += sgl.user("Can you give me a practical example?")
s += sgl.assistant(sgl.gen("example", max_tokens=150))
```

The model sees the previous conversation and generates a practical example.

The last piece of puzzle is

```
state = multi_turn.run(topic="prefix caching in LLM inference")
```

It executes all three turns using the backend model server.

So we can summarize for our better understanding this way

```
SGLang frontend code defines the conversation flow.
SGLang backend server runs the model.
sgl.gen() generates text.
The generated text is stored using names like intro, example, and summary.
```

Bonus: Ollama/llama.cpp

llama.cpp is a lightweight C/C++ inference engine for running LLMs locally with minimal setup. Its main goal is to make LLM inference run efficiently on a wide range of hardware, from local machines to cloud environments.

It is especially popular for:

- Local LLM inference
- CPU inference
- Apple Silicon inference
- Consumer GPU inference
- Edge/on-device deployment
- Quantized models

The biggest reason llama.cpp became popular is that it made it practical to run LLMs outside expensive data center GPUs.

It supports quantized model formats like **GGUF**, which allow large models to run with much lower memory usage.

Ollama is a developer-friendly tool built to make running local LLMs simple. It provides an easy CLI and API experience for downloading, running, and interacting with open models.

Ollama's official site positions it as an easy way to build with open models locally

It is as simple to use as

```
ollama run llama3.1
```

If you are looking for local inference try out first Ollama as it is most simplest to use. I find it very easy to run models locally on my Mac.

Ollama

Ollama is the easiest way to automate your work using open models, while keeping your data safe.

ollama.com

Conclusion

LLM Inference is where the rubber meets the road. Research gives us the model; engineering makes it useful.

In this handbook, we went from the very basics — how a model produces tokens one by one through the Prefill and Decode phases — all the way to the systems and frameworks that power production AI today. We understood why KV cache exists (because recomputing the same vectors is wasteful), why PagedAttention matters (because memory fragmentation kills GPU utilization), and why FlashAttention works (because the real bottleneck was memory movement, not compute). We saw how speculative decoding cleverly exploits the asymmetry between generating and verifying, how prefix caching saves repeated work across requests, and how batching strategies evolved from static to dynamic to continuous to keep GPUs busy.

On the hardware side, we learned that inference is not just about model accuracy — it's about arithmetic intensity, ops-to-byte ratios, floating point formats, and knowing when you're compute-bound vs memory-bound. These are the same engineering principles used in traditional systems, just applied to a new domain.

We also saw that inference at scale is a multi-dimensional problem. Quantization trades a little precision for a lot of memory savings. Parallelism strategies — tensor, pipeline, data, expert — let you scale models across hardware that no single GPU could hold. And frameworks like vLLM, TensorRT-LLM, and SGLang package all of this into systems you can actually deploy.

The biggest takeaway: **LLM inference is not a black box. It is engineering.** Every optimization we discussed has a clear motivation rooted in first principles — *memory is expensive, compute is fast, GPUs hate idling, and wasted tokens cost real money*.

The field is still evolving fast. Disaggregated Prefill-Decode, new quantization formats like FP4, smarter routing in multi-GPU clusters, structured generation with FSMs — these are still being refined. But the foundation you've built here gives you the mental model to understand any new development that comes along.

If you made it this far, you're ready to think like an inference engineer.

Artificial Intelligence + LLM + Llm Inference + Computer Science + Software Engineering +