LLMs - Memory and Computational requirements

Aditya Chatterjee

Deploying Large Language Models (LLMs) efficiently requires a clear understanding of the memory and computational limits that affect inference performance. LLMs, due to their sheer size and complexity, introduce unique challenges, particularly during the inference process. Let’s break down the two key stages of LLM inference—Prefill and Decode—and then explore how tools like the Roofline Model and LLM Viewer can help optimize performance. We'll also use real-world examples, "Alice and Bob", to illustrate how these concepts work in practice.

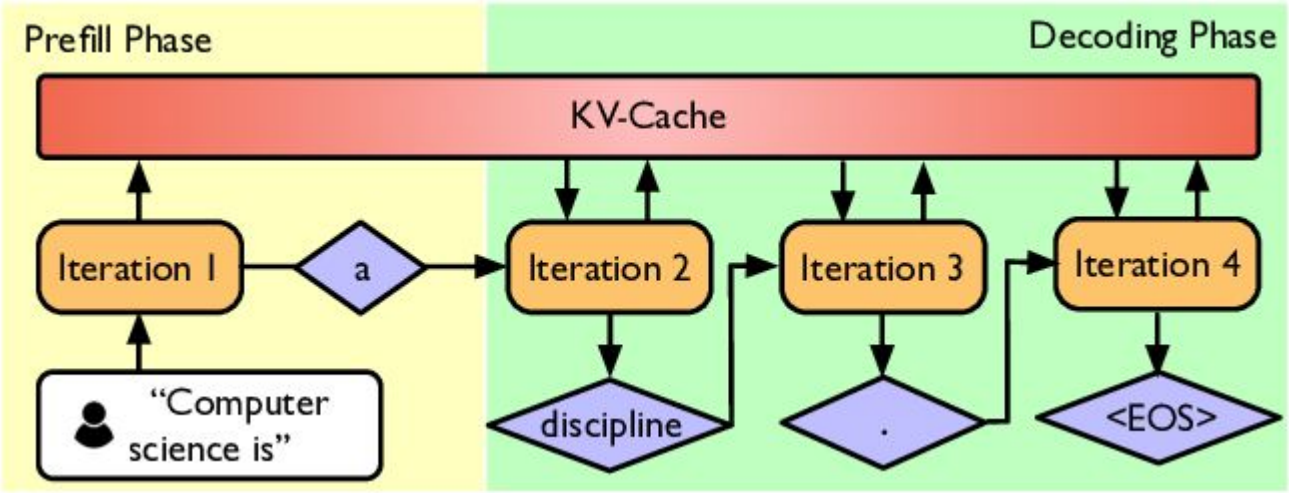
Paper used as reference - https://arxiv.org/abs/2402.16363

LLM-Viewer - http://llm-viewer.com/

Introduction to Prefill Stage

The Prefill Stage is the first phase in the inference process of Large Language Models (LLMs).
It processes the input sequence (prompt) and creates a key-value (KV) cache for each transformer layer.

The Prefill Stage is crucial as it sets up the cache needed to generate the next tokens in the Decode Stage.

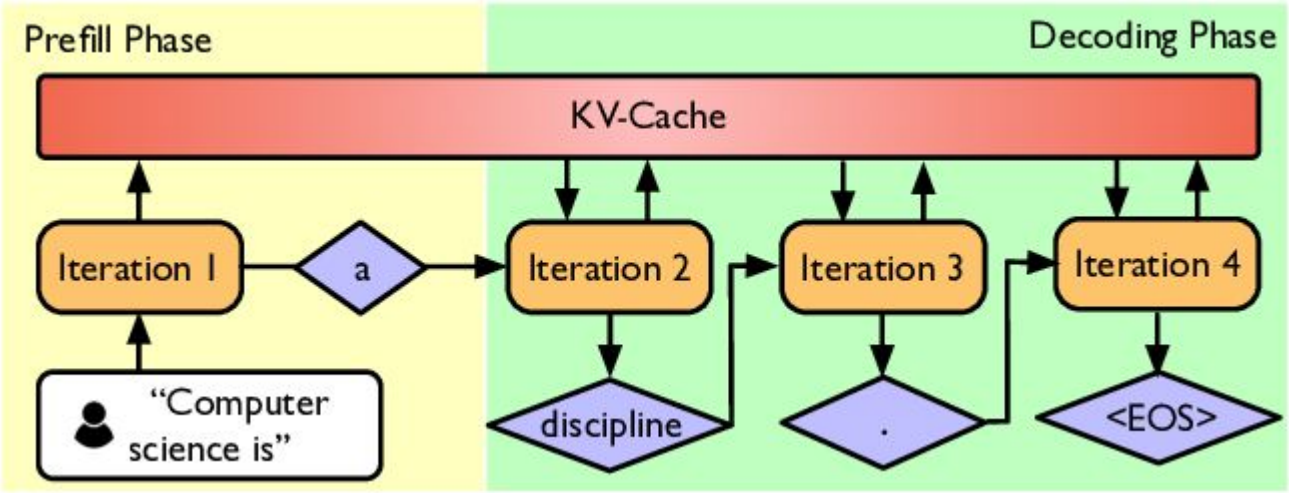


Memory and Compute Impact in Prefill Stage

The Prefill Stage generates a key-value (KV) cache for each transformer layer.

Memory impact: The size of the KV cache increases with both input sequence length and batch size.

Compute impact: The multi-head attention mechanism performs complex matrix operations, making this stage compute-bound.



Optimization Strategies for Prefill Stage

Reducing memory and compute load through:

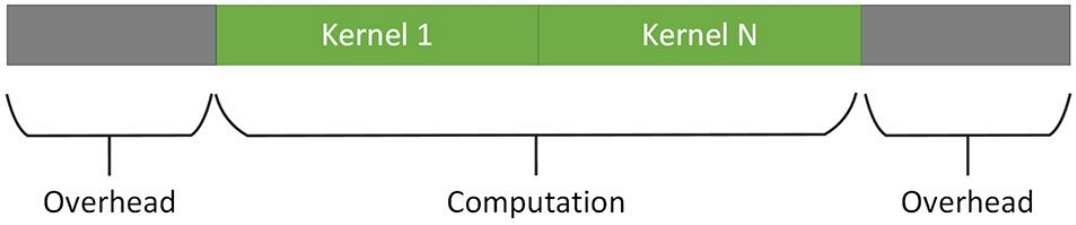
- Quantization: Lower precision data types (e.g., FP16, INT8) reduce memory usage.

- Kernel fusion: Combines operations to reduce compute load.

- Batch size adjustment: Smaller batch sizes can manage memory better.

Example: Bob's legal services firm optimizes the Prefill Stage to handle many short queries by focusing on computational throughput.

Quantization



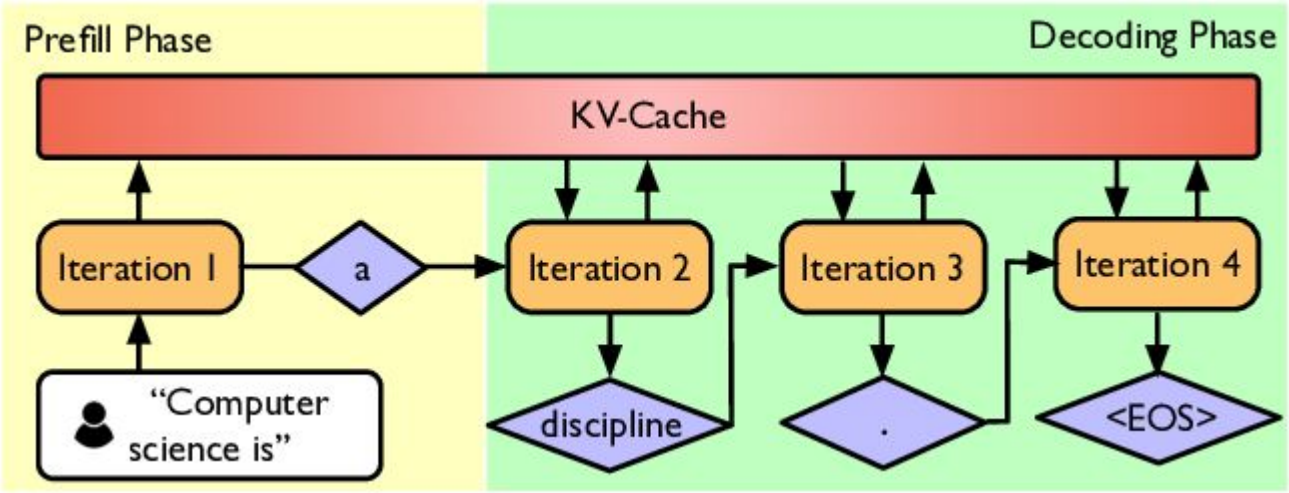
kernel fusion

Introduction to Decode Stage

The Decode Stage follows the Prefill Stage and is responsible for generating tokens one by one.

Each token depends on the previously generated tokens, making this stage sequential.

This stage is often the slowest and the most memory-intensive in the inference process of LLMs.

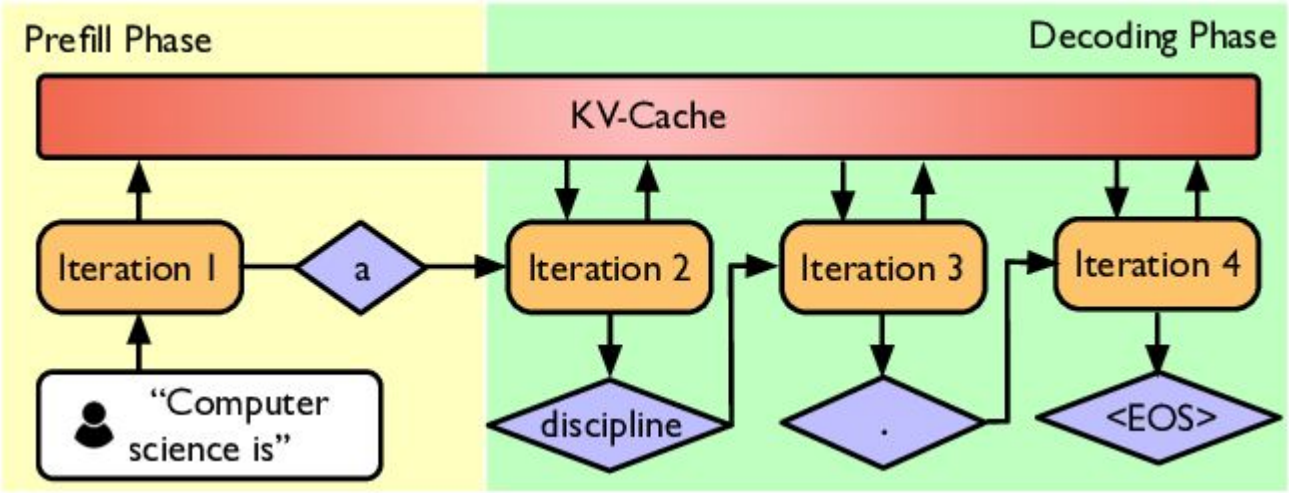


Memory-Bound Nature of Decode Stage

The Decode Stage is heavily memory-bound, meaning performance is limited by memory bandwidth rather than computation power.

During token generation, the model frequently accesses the key-value (KV) cache stored in memory.

Long input sequences or larger batch sizes place additional strain on memory, leading to slower token generation.



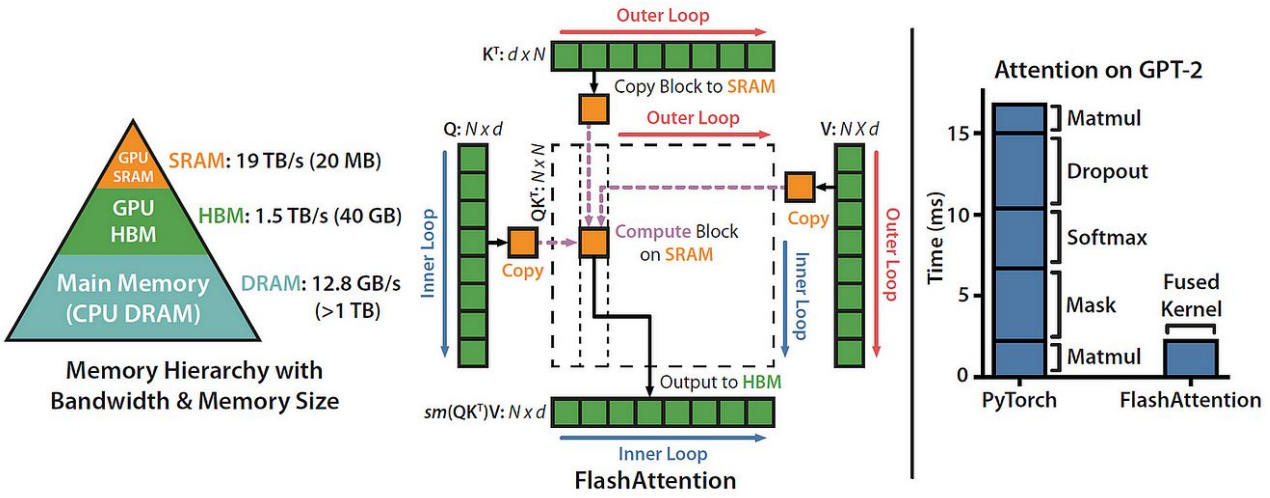
Optimization Strategies for Decode Stage

Improve memory bandwidth to enhance performance during token generation.

- Flash Attention: A technique that reduces memory access during attention computation, improving efficiency.

- Quantization: Using lower precision data types reduces memory usage.

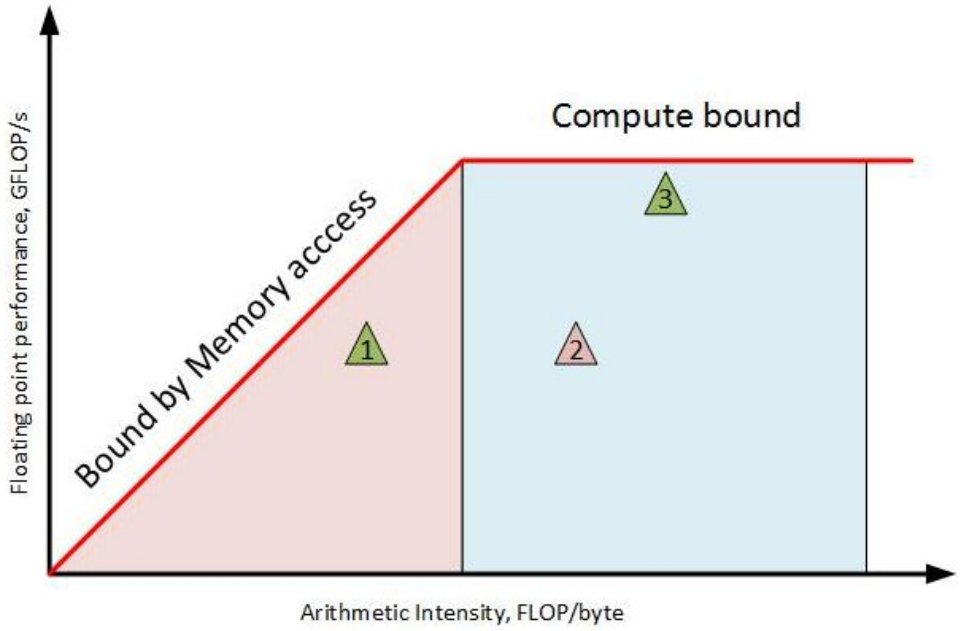
Example: Alice's case—as a writer, Alice optimizes the Decode Stage for long sequences by improving memory bandwidth and applying quantization.



Introduction to the Roofline Model

The Roofline Model is a performance analysis tool that combines both computation and memory bandwidth into a single framework.

It helps to identify whether a process is compute-bound or memory-bound. The model provides insights into how to optimize LLM stages for better performance by visualizing these constraints.



Applying the Roofline Model in LLMs

The Roofline Model helps visualize the performance bottlenecks in both the Prefill Stage and the Decode Stage.

- Compute-bound processes, like the Prefill Stage, require optimizations focused on hardware computation efficiency.
- Memory-bound processes, like the Decode Stage, benefit from improvements in memory bandwidth.

Examples: Bob focuses on compute-bound optimizations, while Alice focuses on memory-bound improvements.

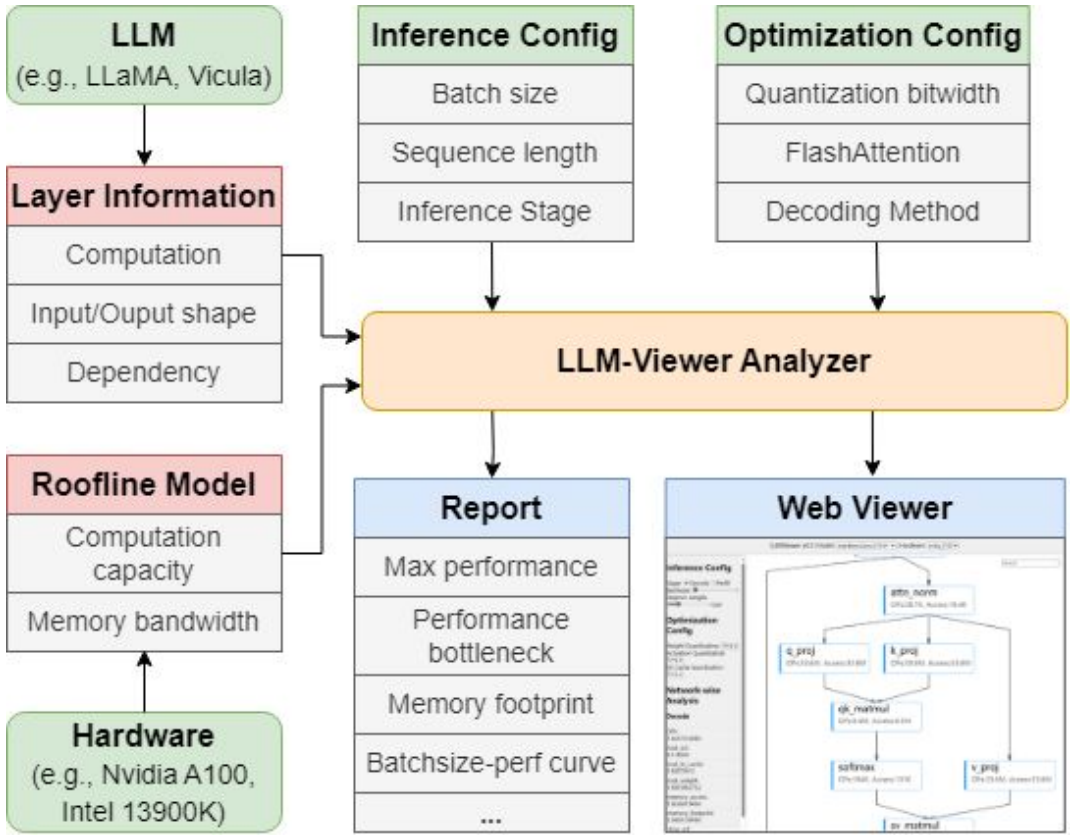
Layer Name	OPs	Memory Access	Arithmetic Intensity	Max Performance	Bound
			Prefill		
q_proj	69G	67M	1024	155T	compute
k_proj	69G	67M	1024	155T	compute
v_proj	69G	67M	1024	155T	compute
o_proj	69G	67M	1024	155T	compute
gate_proj	185G	152M	1215	155T	compute
up_proj	185G	152M	1215	155T	compute
down_proj	185G	152M	1215	155T	compute
qk_matmul	34G	302M	114	87T	memory
sv_matmul	34G	302M	114	87T	memory
softmax	671M	537M	1.25	960G	memory
norm	59M	34M	1.75	1T	memory
add	8M	34M	0.25	192G	memory
			Decode		
q_proj	34M	34M	1	768G	memory
k_proj	34M	34M	1	768G	memory
v_proj	34M	34M	1	768G	memory
o_proj	34M	34M	1	768G	memory
gate_proj	90M	90M	1	768G	memory
up_proj	90M	90M	1	768G	memory
down_proj	90M	90M	1	768G	memory
qk_matmul	17M	17M	0.99	762G	memory
sv_matmul	17M	17M	0.99	762G	memory
softmax	328K	262K	1.25	960G	memory
norm	29K	16K	1.75	1T	memory
add	4K	16K	0.25	192G	memory

Introduction to the LLM Viewer

The LLM Viewer is a tool designed to help users optimize their LLM inference processes layer by layer.

It provides detailed insights into the performance of each transformer layer, showing whether they are compute-bound or memory-bound.

The tool helps suggest optimizations, such as quantization or batch size adjustments, to improve overall performance.



LLM Viewer Real-World Applications

Bob can use the LLM Viewer to optimize batch size for multiple short queries in his legal services.

Alice can test different configurations to optimize long-sequence generation for her novel writing.

The LLM Viewer helps identify bottlenecks and provides actionable insights to improve memory and compute efficiency.

prefill
OPs: 13.8T
inference_time: 102.4ms
load_act: 8.5G
load_kv_cache: 537M
load_weight: 13.2G
memory_access: 30.9G
memory_consumption: 13.7G
memory_consumption_kv_cache: 537M
memory_consumption_tmp_act: 255M
memory_consumption_weight: 13.0G
store_act: 8.2G
store_kv_cache: 537M

decode
OPs: 13.6G
inference_time: 17.9ms
load_act: 8.3M
load_kv_cache: 537M
load_weight: 13.2G
memory_access: 13.8G
memory_consumption: 13.5G
memory_consumption_kv_cache: 537M
memory_consumption_tmp_act: 249K
memory_consumption_weight: 13.0G
store_act: 8.0M
store_kv_cache: 524K

Combining Tools for Optimal LLM Deployment

Efficient LLM deployment requires balancing memory and computation limits.

The Prefill Stage is often compute-bound, while the Decode Stage is typically memory-bound.

Tools like the Roofline Model and LLM Viewer provide valuable insights to optimize both stages for better performance.