Private Cloud Compute

Jiale Guan

2024-10-08

Outline

Outline

1.1 Requirements

Stateless computation

Private Cloud Compute must use the personal user data that it receives exclusively for the purpose of fulfilling the user's request. This data must never be available to anyone other than the user, not even to Apple staff, not even during active processing. And **this data must not be retained**, including via logging or for debugging, after the response is returned to the user. In other words, we want a strong form of stateless data processing where **personal data leaves no trace** in the PCC system.

1.1 Requirements

Enforceable guarantees

Security and privacy guarantees are strongest when they are entirely technically enforceable, which means it must be possible to **constrain and analyze all the components** that critically contribute to the guarantees of the overall Private Cloud Compute system. To use our example from earlier, it's very difficult to reason about what a TLS-terminating load balancer may do with user data during a debugging session. Therefore, PCC must not depend on such external components for its core security and privacy guarantees. Similarly, operational requirements such as collecting server metrics and error logs must be supported with mechanisms that do not undermine privacy protections.

1.1 Requirements

No privileged runtime access

Private Cloud Compute **must not contain privileged interfaces** that would enable Apple's site reliability staff to bypass PCC privacy guarantees, even when working to resolve an outage or other severe incident. This also means that PCC must not support a mechanism by which the privileged access envelope could be enlarged at runtime, such as by loading additional software.

1.1 Requirements

Non-targetability

An attacker should not be able to attempt to compromise personal data that belongs to specific, targeted Private Cloud Compute users without attempting a broad compromise of the entire PCC system. This must hold true even for exceptionally sophisticated attackers who can attempt physical attacks on PCC nodes in the supply chain or attempt to obtain malicious access to PCC data centers. In other words, a limited PCC compromise must not allow the attacker to **steer requests from specific users to compromised nodes**; targeting users should require a wide attack that's likely to be detected. To understand this more intuitively, contrast it with a traditional cloud service design where every application server is provisioned with database credentials for the entire application database, so a compromise of a single application server is sufficient to access any user's data, even if that user doesn't have any active sessions with the compromised application server.

1.1 Requirements

Verifiable transparency

Security researchers need to be able to verify, with a high degree of confidence, that our privacy and security guarantees for Private Cloud Compute match our public promises. We already have an earlier requirement for our guarantees to be enforceable. Hypothetically, then, if security researchers had sufficient access to the system, they would be able to verify the guarantees. But this last requirement, verifiable transparency, goes one step further and does away with the hypothetical: **security researchers must be able to verify the security and privacy guarantees of Private Cloud Compute**, and they must be able to verify that the software that's running in the PCC production environment is the same as the software they inspected when verifying the guarantees.

1.2 Taxonomy

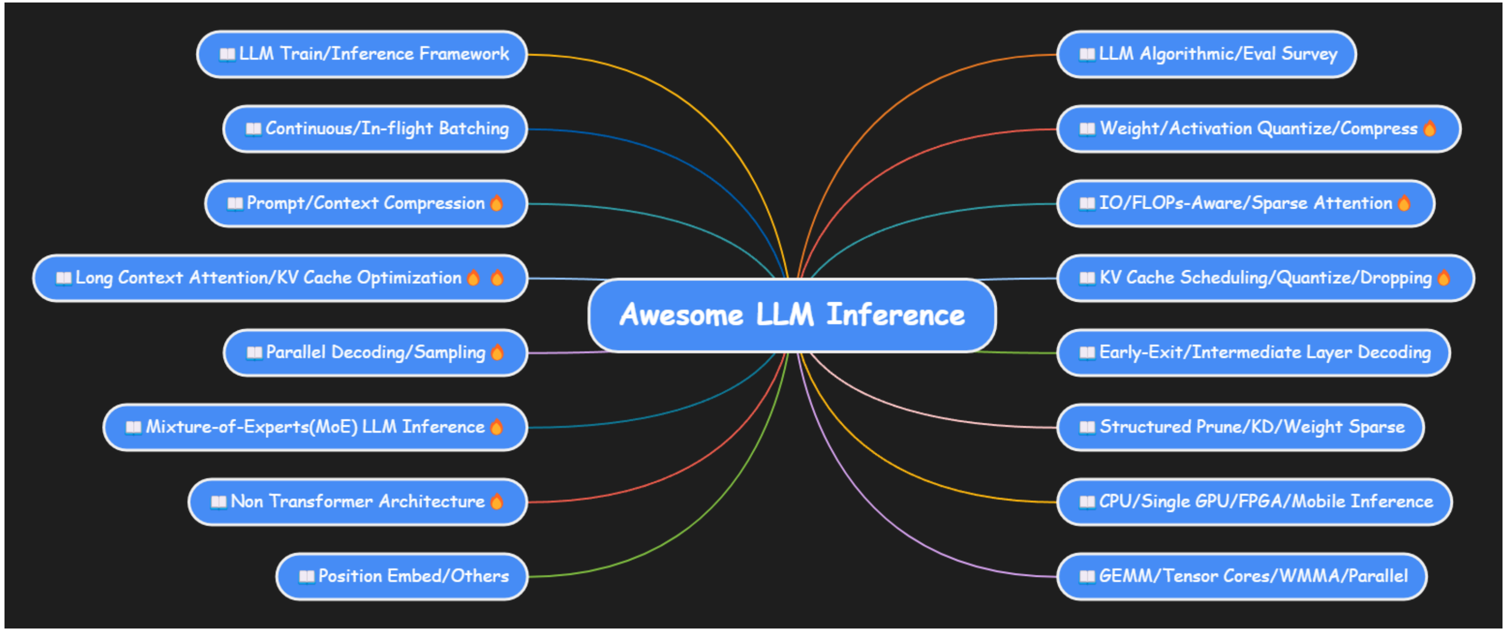
Category	Requirements	Threats	Guarantees
Technical	Stateless computation	Trace of data after processing Example: Logging, debugging	(Purpose) Only use user data to perform requested operations (Transient) Delete the data after fulfilling the request (Scope) Not available to even Apple staff
	Non-targetability	Targeted attack Example: Steer request to compromised nodes	(Hardware) Hardened supply chain (Scheduler) Requests cannot be user/content-specific routed (Anonymity) OHTTP Relay, RSA Blind Signature (Scope) No system-wide encryption
Ecosystem	Enforceable guarantees	Technical enforceability Example: External TLS-terminating load balancer	(Hardware) Secure Enclave, Secure Boot (System) Signed System Volume, Swift on Server (Software) Code Signing, Sandboxing
	No privileged runtime access	Privileged interfaces Example: Shell access by SREs	No remote shell. Only pre-specified, structured, and audited logs/metrics can leave the node User data is reviewed by multiple indepedent layers
	Verifiable transparency	Uninspected code	Every production build of PCC publicly available

Outline

1 Private Cloud Compute 2
1.1 Requirements 3
1.2 Taxonomy 8
2 LLM Serving Systems 9
2.1 LLM Inference 10
2.2 Phases 11
2.3 Challenges 12
3 Optimizations 13
3.1 Memory Management 14
3.2 Batch Processing 23
3.3 Parallel Processing 24
3.4 Speculative Inference 25
3.5 Summary 27
4 Threats 30
4.1 Stateless Computation 31
4.2 Non-Targetability 36
4.3 Ecosystem 38
5 Summary 39
5.1 Academic Systems 40
5.2 Industrial Systems 42
6 Appendix 45

2.1 LLM Inference

Most of the popular decoder-only LLMs (GPT-3, for example) are pretrained on the causal modeling objective, essentially as next-word predictors. These LLMs take a series of tokens as inputs, and generate subsequent tokens autoregressively until they meet a stopping criteria.


Awesome LLM Inference
LLM Train/Inference Framework
Continuous/In-flight Batching
Prompt/Context Compression
Long Context Attention/KV Cache Optimization
Parallel Decoding/Sampling
Mixture-of-Experts(MoE) LLM Inference
Non Transformer Architecture
Position Embed/Others
LLM Algorithmic/Eval Survey
Weight/Activation Quantize/Compress
IO/FLOPs-Aware/Sparse Attention
KV Cache Scheduling/Quantize/Dropping
Early-Exit/Intermediate Layer Decoding
Structured Prune/KD/Weight Sparse
CPU/Single GPU/FPGA/Mobile Inference
GEMM/Tensor Cores/WMMA/Parallel

LLM Inference

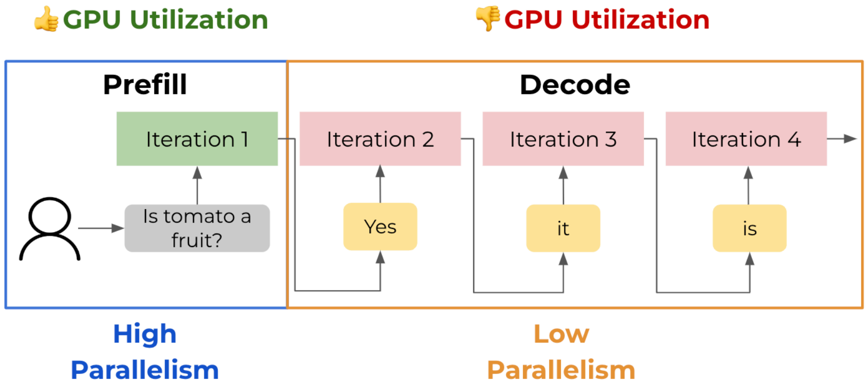
2.2 Phases

Prefill: Processing the input

In the prefill phase, the LLM processes the input tokens to compute the intermediate states (keys and values), which are used to generate the “first” new token. Each new token depends on all the previous tokens, but because the full extent of the input is known, at a high level this is a matrix-matrix operation that’s **highly parallelized**. It effectively **saturates GPU utilization**.

Decode: Generating the output

In the decode phase, the LLM generates output tokens autoregressively one at a time, until a stopping criteria is met. Each sequential output token needs to know all the previous iterations’ output states (keys and values). This is like a matrix-vector operation that underutilizes the GPU compute ability compared to the prefill phase. The speed at which the data (weights, keys, values, activations) is **transferred to the GPU from memory** dominates the latency, not how fast the computation actually happens. In other words, this is a **memory-bound operation**.



2.3 Challenges

Workload Heterogeneity

Universality and application diversity lead to heterogeneity of the inference requests, in terms of input lengths, output lengths, expected latencies, etc

- Queuing Delays, Preemptions, Interference

Execution Unpredictability

Unknown a priori how many tokens will be generated before the stopping criteria is met. As such, the execution time and the resource demand of a request are both unpredictable.

Multi-Tenant and Dynamic Environment

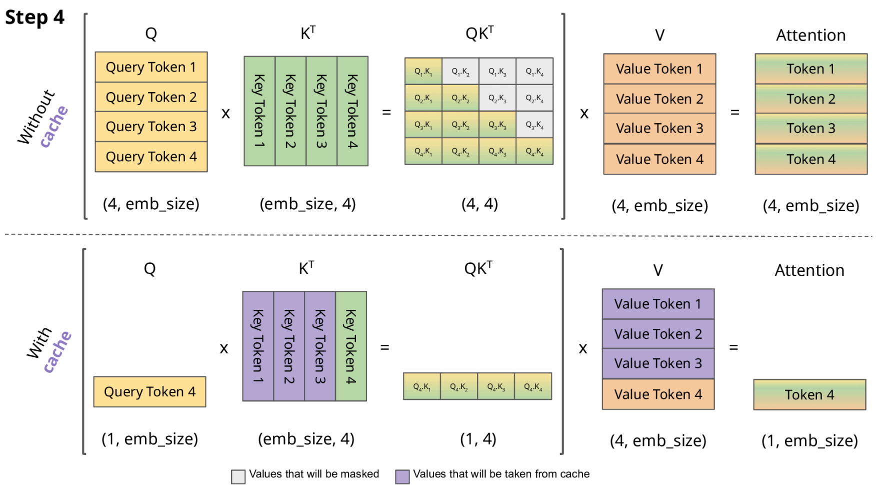
The system must scale to support multiple users and adapt to the dynamic nature of the environment.

Outline

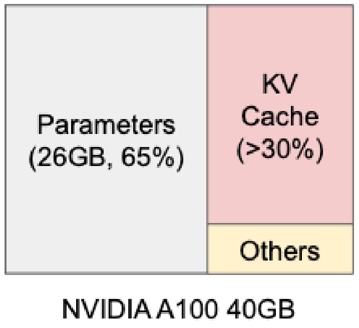
1 Private Cloud Compute 2
  1.1 Requirements 3
  1.2 Taxonomy 8
2 LLM Serving Systems 9
  2.1 LLM Inference 10
  2.2 Phases 11
  2.3 Challenges 12
3 Optimizations 13
  3.1 Memory Management 14
  3.2 Batch Processing 23
  3.3 Parallel Processing 24
  3.4 Speculative Inference 25
  3.5 Summary 27
4 Threats 30
  4.1 Stateless Computation 31
  4.2 Non-Targetability 36
  4.3 Ecosystem 38
5 Summary 39
  5.1 Academic Systems 40
  5.2 Industrial Systems 42
6 Appendix 45

3.1 Memory Management

KV Cache



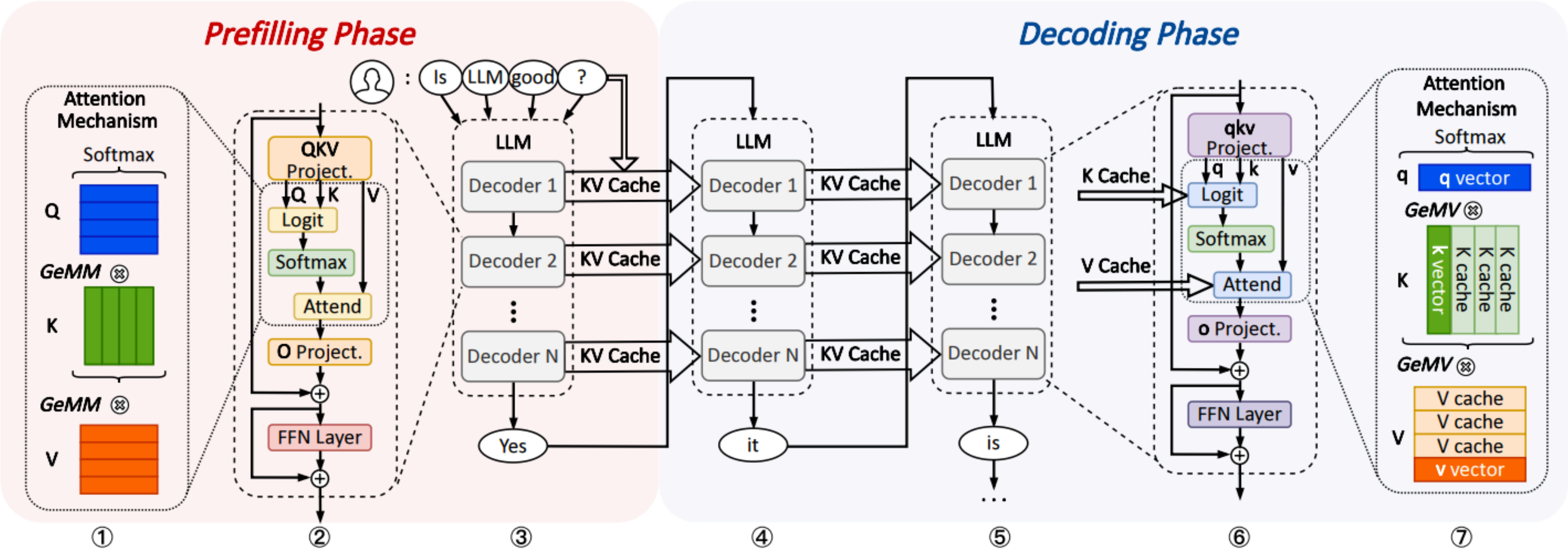
Transformers use attention mechanisms that compute attention scores between tokens. The KV Cache helps by storing previously computed key-value pairs, allowing the model to quickly access and reuse them for new tokens, avoiding redundant calculations.



Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemerally for activation.

3.1 Memory Management

LLM inference architecture primarily comprises multiple stacked decoder blocks, each consisting of a self-attention module and a Feed-Forward Neural Network (FFN) module.



3.1 Memory Management

Paged Attention

Paged Attention[*] is a technique that divides the attention matrix into smaller pages. This approach provides a near-perfect solution for mitigating fragmentation and hence, PagedAttention has become the de facto standard for dynamic memory allocation in LLM serving systems.

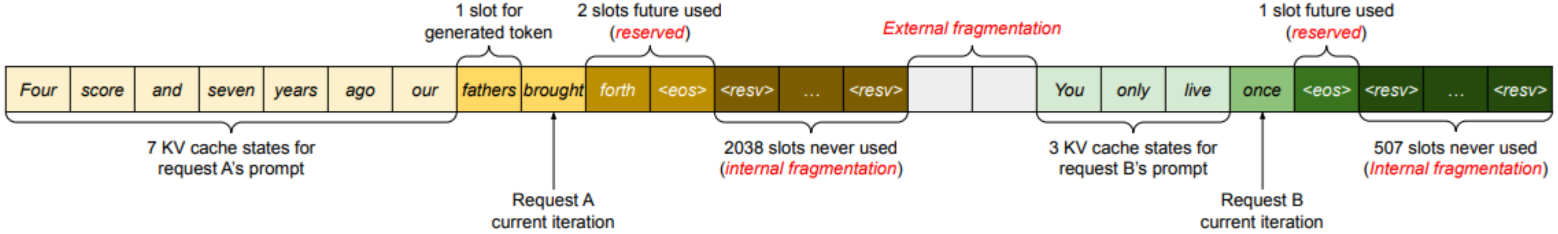
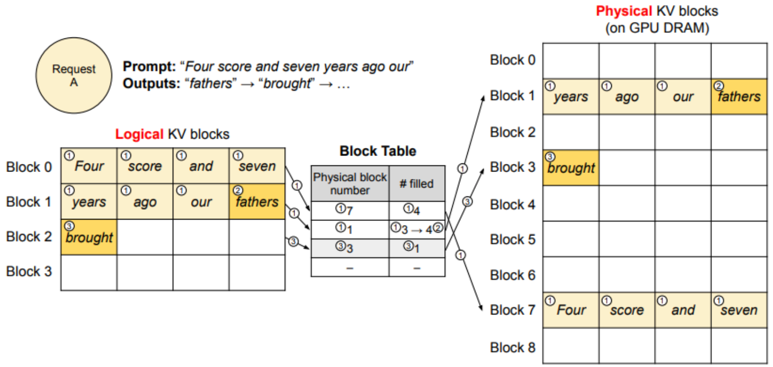

1 slot for generated token
2 slots future used (reserved)
External fragmentation
1 slot future used (reserved)
Four
score
and
seven
years
ago
our
fathers
brought
forth
<eos>
<resv>
...
<resv>
You
only
live
once
<eos>
<resv>
...
<resv>
7 KV cache states for request A's prompt
2038 slots never used (internal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

Figure 3. KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

[*] Efficient Memory Management for Large Language Model Serving with PagedAttention

3.1 Memory Management

Paged Attention

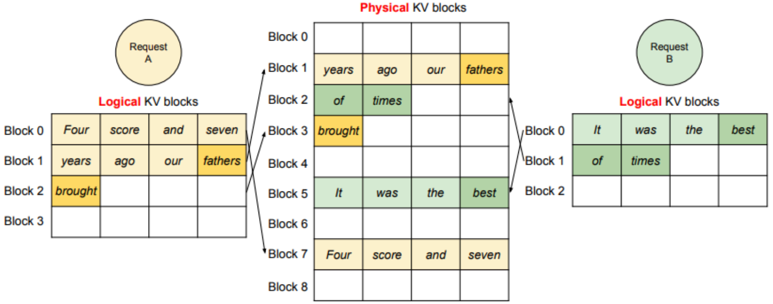

Request A
Prompt: "Four score and seven years ago our"
Outputs: "fathers" → "brought" → …
Logical KV blocks
Block Table
Physical KV blocks
(on GPU DRAM)

Logical KV blocks				
Block 0	Four	score	and	seven
Block 1	years	ago	our	fathers
Block 2	brought			
Block 3				

Physical block number	# filled
7	4
1	3 → 4
3	1
–	–

Physical KV blocks				
Block 0				
Block 1	years	ago	our	fathers
Block 2				
Block 3	brought			
Block 4				
Block 5				
Block 6				
Block 7	Four	score	and	seven
Block 8				

Figure 6. Block table translation in vLLM.



Request A Logical KV blocks				
Block 0	Four	score	and	seven
Block 1	years	ago	our	fathers
Block 2	brought			
Block 3				

Physical KV blocks				
Block 0				
Block 1	years	ago	our	fathers
Block 2	of	times		
Block 3	brought			
Block 4				
Block 5	It	was	the	best
Block 6				
Block 7	Four	score	and	seven
Block 8				

Request B Logical KV blocks				
Block 0	It	was	the	best
Block 1	of	times		
Block 2				

Figure 7. Storing the KV cache of two requests at the same time in vLLM.

Pitfalls[*]

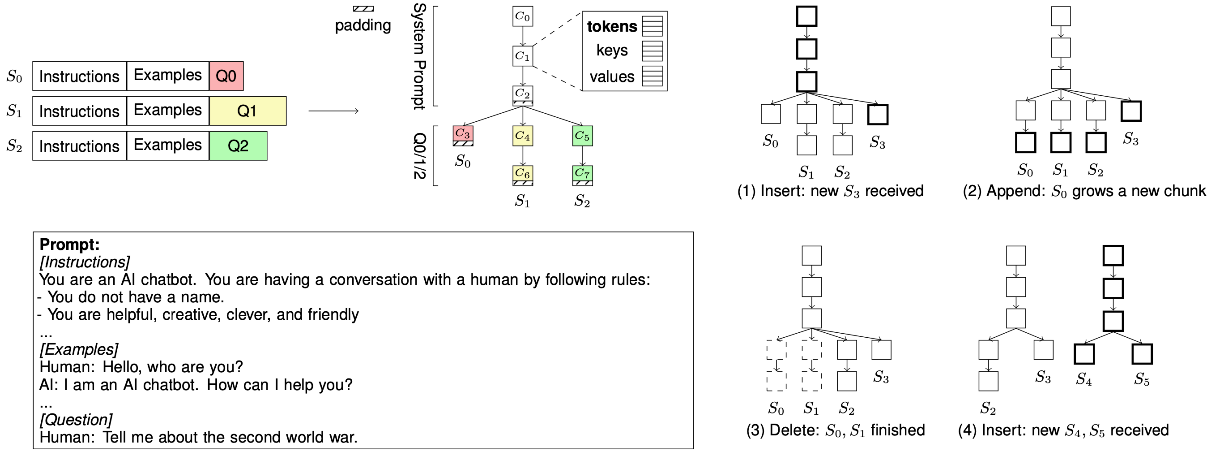
- Requires re-writing the attention kernel.
- Adds software complexity and redundancy (CPU code), can degrade throughput by 11%.
- Introduces performance overhead. 20-26% slower than original FasterTransformer kernel.

[*] vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. 2024

3.1 Memory Management

Prefix Caching

Prefix Caching[*] is a technique that caches the intermediate states of the model during the prefill phase. These states are then reused during the decode phase to speed up inference.


padding
S_0 Instructions Examples Q0
S_1 Instructions Examples Q1
S_2 Instructions Examples Q2
System Prompt
Q0/1/2
C_0
C_1
C_2
C_3
C_4
C_5
C_6
C_7
tokens
keys
values
S_0
S_1
S_2
S_3
(1) Insert: new S_3 received
(2) Append: S_0 grows a new chunk
(3) Delete: S_0, S_1 finished
S_4
S_5
(4) Insert: new S_4, S_5 received

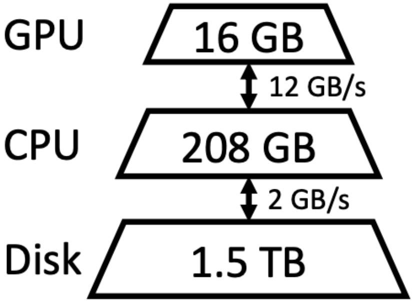
Prompt:
[Instructions]
You are an AI chatbot. You are having a conversation with a human by following rules:
- You do not have a name.
- You are helpful, creative, clever, and friendly
...
[Examples]
Human: Hello, who are you?
AI: I am an AI chatbot. How can I help you?
...
[Question]
Human: Tell me about the second world war.

[*] ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. 2024

3.1 Memory Management

KV Cache Offloading

The KV Cache Offloading technique moves the KV cache from the GPU to the CPU to free up GPU memory for other tasks.



3.1 Memory Management

Flash Attention

GPU: One kind of computation done on the input data at a time in sequence

Fusing: Fusing multiple layers together during the actual computation can enable minimizing the data access by GPUs.

FlashAttention[*] uses **tiling** to fully compute and write out a small part of the final matrix at once

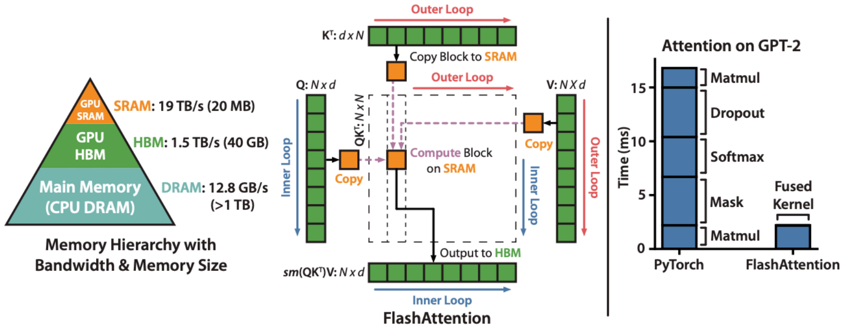


Figure 1: **Left:** FlashAttention uses tiling to prevent materialization of the large $N \times N$ attention matrix (dotted box) on (relatively) slow GPU HBM. In the outer loop (red arrows), FlashAttention loops through blocks of the **K** and **V** matrices and loads them to fast on-chip SRAM. In each block, FlashAttention loops over blocks of **Q** matrix (blue arrows), loading them to SRAM, and writing the output of the attention computation back to HBM. **Right:** Speedup over the PyTorch implementation of attention on GPT-2. FlashAttention does not read and write the large $N \times N$ attention matrix to HBM, resulting in an 7.6× speedup on the attention computation.

[*] FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

3.1 Memory Management

Group-Query Attention

- **Standard Attention**: Compute attention for each query separately. Complexity is $O(n^2)$.
- **Multi-Query Attention**: Reuse the same attention matrix for multiple queries. Queries are similar enough to share the same attention distribution.
- **Group-Query Attention**[*]: Divide queries into groups and compute attention for each group separately.

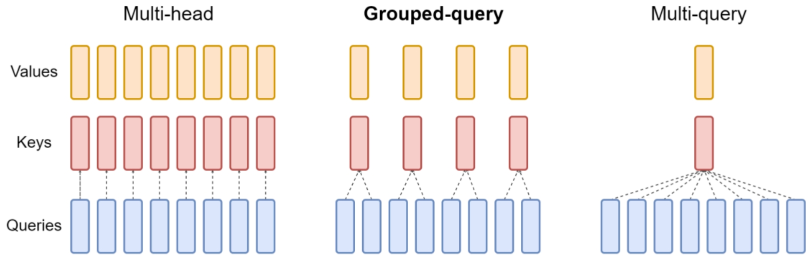
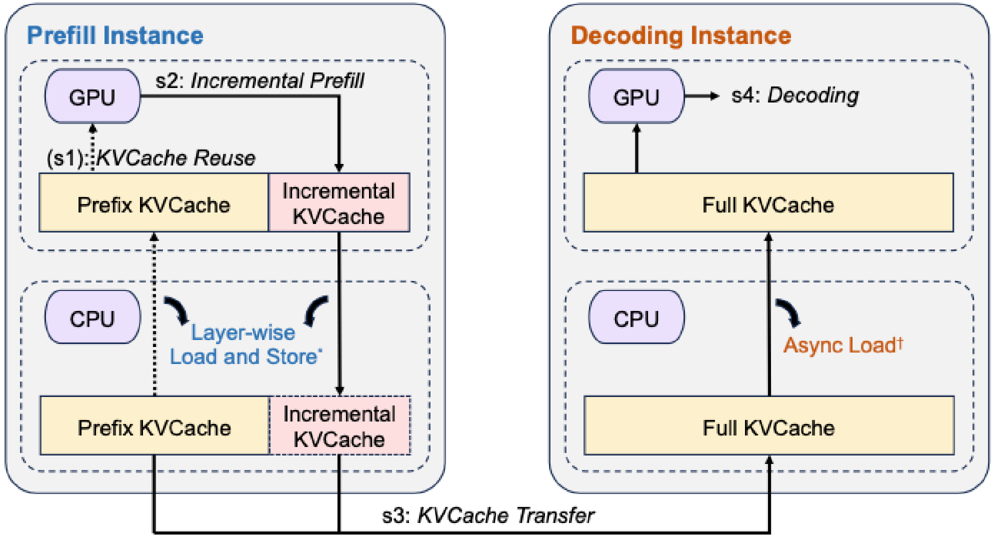


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

[*] GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints

3.1 Memory Management

Real-World System: Mooncake

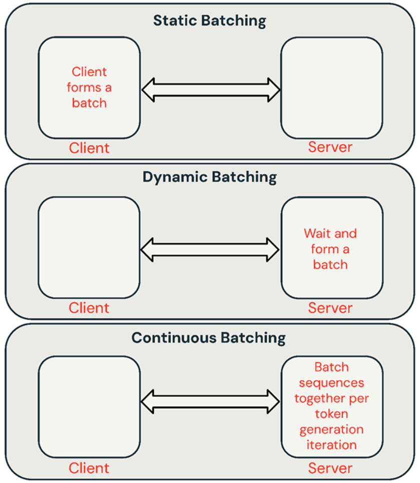


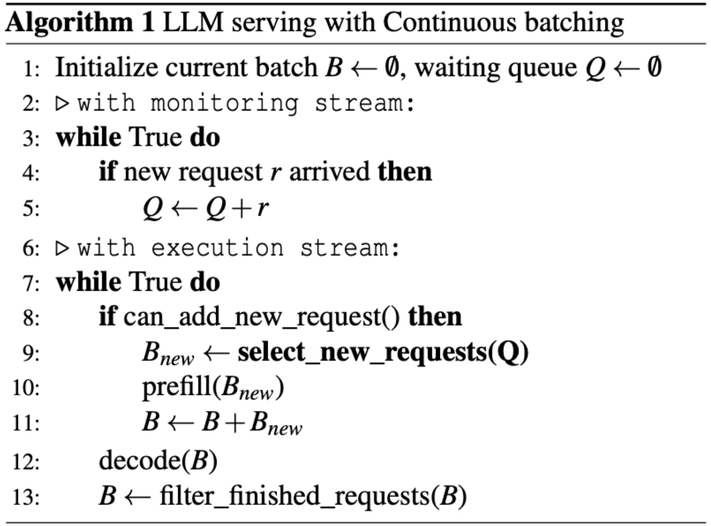
3.2 Batch Processing

Static batching: Client packs multiple prompts into requests and a response is returned after all sequences in the batch have been completed. Our inference servers support this but do not require it.

Dynamic batching: Prompts are batched together on the fly inside the server. Typically, this method performs worse than static batching but can get close to optimal if responses are short or of uniform length. Does not work well when requests have different parameters.

Continous Batching: A batch that is continuously processed, leveraging the opportunity by batching new requests once some old requests are finished





Algorithm 1 LLM serving with Continuous batching

```
Initialize current batch $B \leftarrow \emptyset$, waiting queue $Q \leftarrow \emptyset$
▷ with monitoring stream:
while True do
    if new request $r$ arrived then
        $Q \leftarrow Q + r$
▷ with execution stream:
while True do
    if can_add_new_request() then
        $B_{new} \leftarrow$ select_new_requests(Q)
        prefill($B_{new}$)
        $B \leftarrow B + B_{new}$
    decode($B$)
    $B \leftarrow$ filter_finished_requests($B$)
```

3.3 Parallel Processing

Data Parallelism (DP)

Each device processes a different batch of data. This allows multiple devices to work on independent data batches in parallel, improving throughput.

Sequence Parallelism (SP)

The input sequence is divided into chunks, with each chunk processed by a separate device. This enables parallel processing of sequence segments across devices.

Pipeline Parallelism (PP)

The model is split vertically into chunks, where each chunk contains a subset of layers. Each device handles a different set of layers, allowing different stages of the model to be executed in parallel across devices.

Tensor Parallelism (TP)

The model's parameters are sharded horizontally into chunks, with each chunk distributed across devices. This allows the computation within each layer to be split and processed in parallel.

3.4 Speculative Inference

Standard inference

Sequence generation is strictly sequential. Each token must be generated based on the previously generated token, which leads to high latency, especially for long-sequence tasks.

Speculative inference[*]

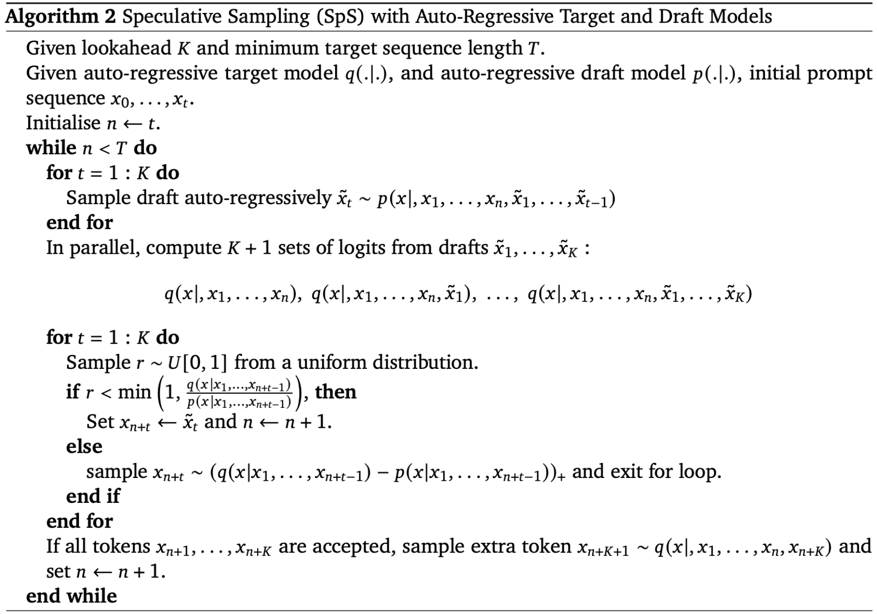
- **Predict multiple tokens ahead**: When generating the first token, the model simultaneously makes speculative predictions about the next several tokens.
- **Parallel processing**: These speculative predictions allow the model to process multiple possible outcomes in parallel, speeding up the inference.
- **Validate predicted paths**: If the speculative predictions are correct, the model can continue with these results, avoiding the need to recalculate. If the predictions are incorrect, the model adjusts and corrects the path.

[*] Blockwise Parallel Decoding for Deep Autoregressive Models

3.4 Speculative Inference

Algorithm*

- p is the smaller draft model, q is the larger target model.

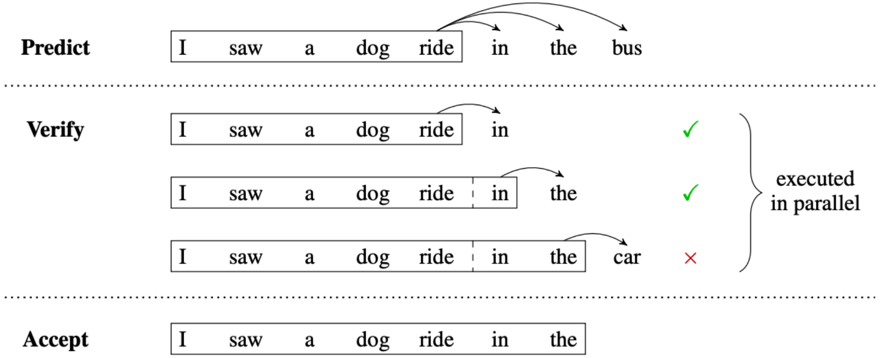


Algorithm 2 Speculative Sampling (SpS) with Auto-Regressive Target and Draft Models

```
Given lookahead $K$ and minimum target sequence length $T$.
Given auto-regressive target model $q(.|.)$, and auto-regressive draft model $p(.|.)$, initial prompt sequence $x_0, \ldots, x_t$.
Initialise $n \leftarrow t$.
while $n < T$ do
  for $t = 1 : K$ do
    Sample draft auto-regressively $\tilde{x}_t \sim p(x|, x_1, \ldots, x_n, \tilde{x}_1, \ldots, \tilde{x}_{t-1})$
  end for
  In parallel, compute $K + 1$ sets of logits from drafts $\tilde{x}_1, \ldots, \tilde{x}_K$ :

      $q(x|, x_1, \ldots, x_n),\ q(x|, x_1, \ldots, x_n, \tilde{x}_1),\ \ldots,\ q(x|, x_1, \ldots, x_n, \tilde{x}_1, \ldots, \tilde{x}_K)$

  for $t = 1 : K$ do
    Sample $r \sim U[0, 1]$ from a uniform distribution.
    if $r < \min\left(1, \frac{q(x|x_1,\ldots,x_{n+t-1})}{p(x|x_1,\ldots,x_{n+t-1})}\right)$, then
      Set $x_{n+t} \leftarrow \tilde{x}_t$ and $n \leftarrow n + 1$.
    else
      sample $x_{n+t} \sim (q(x|x_1, \ldots, x_{n+t-1}) - p(x|x_1, \ldots, x_{n+t-1}))_+$ and exit for loop.
    end if
  end for
  If all tokens $x_{n+1}, \ldots, x_{n+K}$ are accepted, sample extra token $x_{n+K+1} \sim q(x|, x_1, \ldots, x_n, x_{n+K})$ and set $n \leftarrow n + 1$.
end while
```

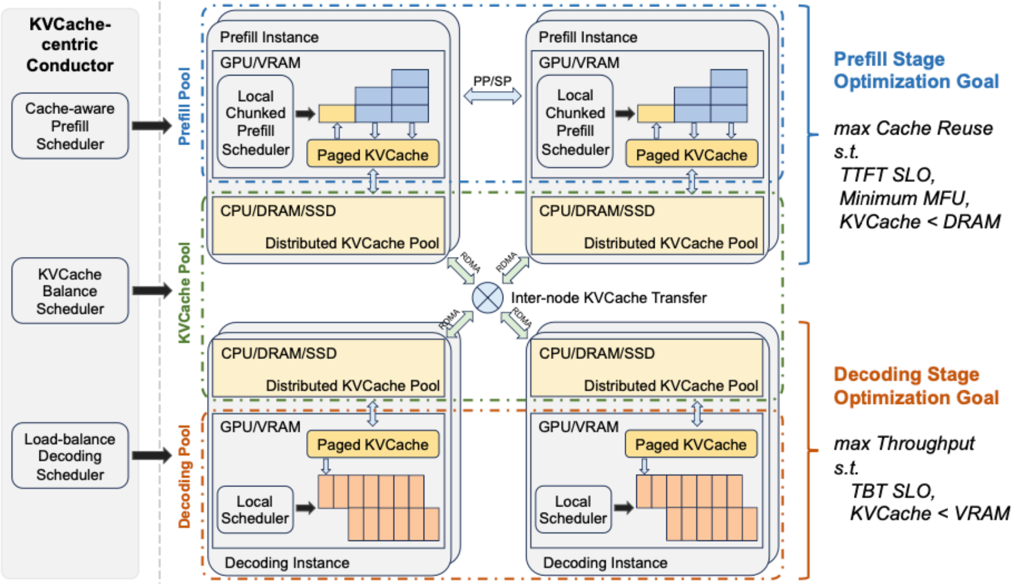


[*] Accelerating Large Language Model Decoding with Speculative Sampling, 2023

3.5 Summary

Real-World System: Mooncake

Gray: Control plane



3.5 Summary

Category	Optimization	GPU Resources			Optimization Goal		
		Compute	Memory	Transmission	Throughput	TTFT	TBT
Memory	Paging	-	+		+		
	Prefix Caching	-	+		+		
	Disk Offloading		+	-	+		
	Group-Query Attention		+		+	+	+
Tranmission	Duplication	+	-	+	+	+	+
	Pulling	+	-	+	+	+	+
	Request Migration	+	+	-	+	+	+
	Disaggregated Arch	+	+	+	+	-	-
Batch	Continuous Batch	+		+	+	-	-
	Chunked Prefill	+			+	+	+
	Prepack Prefill	+			+	-	
Parallelism	Pipeline Parallelism	+		-	+	-	?
	Tensor Parallelism	+		-	+	-	+
	Sequence Parallelism	+		-	+	+	?
	Speculative Inference	+	-		+		+

3.5 Summary

Trends

Category	Trend	Examples	Conflict
Memory	Improve reusability of KV Cache **with finer granularity**	Paging Token-Level Optimization	**S**
Transmission	Minimizing transmission **latency**	Data Duplication Prefetching/Pulling PD Disaggregation	**T**
Scheduling	Customized scheduling for **specific scenarios**	Request-level Predictions Machine-Level Scheduling Global profiling	**STP**
Parallelism	Optimizing parallelism for resource reuse and efficiency	Pipeline Parallelism Tensor Parallelism Sequence Parallelism Speculative Inference	**S**

S: Stateless computation E: Enforceable guarantees T: Non-targetability P: No privileged runtime access V: Verifiable transparency

Outline

4.1 Stateless Computation

Requirement

The system does not maintain any state between requests and the system does not store any information about previous requests.

Attacker's Capabilities

- **Weak**: An attacker gains access to the system's storage mechanism, potentially compromising databases or disk storage. They can also query the system to infer the presence of sensitive information.
- **Strong**: An attacker gains control over specific nodes, allowing them to request or intercept data within the system. However, they are unable to directly access the original prompt due to model sharding or other security measures.

Overview of Attacker's Capabilities

Capabilities	Goal	Query	Access to Storage	Knowledge of Model	Access to Specific Nodes	Control over Node
Weak	Reconstructing User Inputs and Contexts	✓	✓	✓		
Strong	Reconstructing User Inputs and Contexts	✓	✓	✓	✓	✓

4.1 Stateless Computation

Prompt & KV Cache

While the KV cache is related to the input prompt, it is not possible to directly infer the original prompt from it due to the complex and non-reversible nature of the transformations involved in generating the cache.

Precompute KV Cache

Precompute the KV cache for a set of known sensitive prefixes. For instance, if the system is used for medical queries, precompute the KV cache for common medical terms.

State

In LLM inference, once KVs are computed for a sequence, their values do not change.

Input

$X = [x_1, x_2, ..., x_n].$

Embedded Input

$$\begin{aligned} E &= \text{Embed}(X) + \text{Positional Encoding}(X) \\ &= [E_1, E_2, ..., E_n]. \end{aligned}$$

K Cache

$K = [E_1 W^k, E_2 W^k, ..., E_n W^k].$

V Cache

$V = [E_1 W^v, E_2 W^v, ..., E_n W^v].$

4.1 Stateless Computation

Weak Attacker

Inference Attack

An attacker could analyze cached data to infer patterns or user behaviors. Even without full query access, understanding what prefixes are frequently cached might reveal the types of queries being made to the system.

Example: If the cache contains **frequent prefixes related to medical inquiries**, the attacker could infer that the LLM is being used in a healthcare context.

Replay Attack

Attacker can craft new queries that match existing keys in the cache. This allows them to replay or trigger cached computations, potentially extracting sensitive information based on model completions.

Example: If the key in the KV cache is "My bank account number is…", the attacker can craft similar queries to attempt to elicit a continuation that reveals more about the original input.

4.1 Stateless Computation

Strong Attacker

Targeted Data Extraction

An attacker controlling a node could craft specific inputs designed to trigger the retrieval of cached prefixes. By doing this repeatedly, they can extract sensitive information from the cache based on the model's responses.

Example: By submitting inputs like "My Social Security Number is…" and monitoring responses, they could **infer whether such a prefix was previously cached and what context or continuation it triggers**.

Cache Content Mapping

The attacker maps key tensors to actual input tokens using controlled nodes, thereby obtaining the full sequence of user inputs. The attacker can reconstruct the complete input sequence for high-frequency queries or commonly used phrases, directly exposing user input such as personal queries or confidential data.

4.1 Stateless Computation

Optimization	Stored States	Location	Mitigation	Weak Attacker		Strong Attacker	
				Inference	Replay	Extraction	Mapping
Prefix Caching	KV Cache	GPU Memory CPU Memory	Cache Expiry Isolation			✓	✓
Disk Offloading	KV Cache	Disk Storage (SSD, Hard Drive)	Encryption	✓	✓	✓	
Pulling	KV Cache	GPU Memory CPU Memory	Randomized Scheduler			✓	✓
Database-based Speculative Inference	Token	GPU Memory CPU Memory	Differential Priavacy			✓	

4.2 Non-Targetability

Non-Targetability: An attacker should not be able to attempt to compromise personal data that belongs to specific, targeted Private Cloud Compute users without attempting a broad compromise of the entire PCC system.

Definition: Let $S = \{S_1, S_2, ..., S_n\}$ denote the set of all servers in the system, with the capability of each server S_i represented by $C(S_i)$. The set of requests handled by these servers is denoted as $R(S) = \{R(S_1), R(S_2), ..., R(S_n)\}$. The system is considered non-targetable if, for any subset $T = \{T_1, T_2, ..., T_m\} \subseteq S$ of servers, the probability of compromising the data of a specific user u is given by:

$$P(u \in R(T)) = \frac{\sum_{i=1}^{m} C(T_i)}{\sum_{i=1}^{n} C(S_i)}$$

4.2 Non-Targetability

$$P(u \in R(T)) = \frac{\sum_{i=1}^{m} C(T_i)}{\sum_{i=1}^{n} C(S_i)}$$

Goal

- Increase Hit Count: $\sum C(T_i)$
- Decrease Miss Count: $\sum C(S_i) - \sum C(T_i)$

Optimization	Increase Hit Rate	Decrease Miss Rate
Duplication	✓ Duplicate targeted cache to other victims	✓ Duplicate untargeted cache to non-victims
Pulling	✓ Pull targeted cache from other victims	
Priority-Based Scheduling	✓ Prioritize targeted requests	✓ Deprioritize untargeted requests
Request-Level Prediction	✓ Prioritize targeted requests	✓ Deprioritize untargeted requests
Instance Flip	✓ Flip victim(s) to prefill instance	

4.3 Ecosystem

No Privileged Runtime Access

Definition

The system must not contain privileged interfaces that would enable site reliability staff to bypass PCC privacy.

Violations

- Global Profiling

Enforceable Guarantees

Definition

The system must provide guarantees that can be enforced by the system itself. These guarantees must be technically enforceable and not rely on external components or human intervention.

Violations

- Disk Offloading
- Pulling
- Machine-level Scheduler

Verifiable Transparency

Definition

Security researchers must be able to verify the security and privacy guarantees of Private Cloud Compute, and they must be able to verify that the software that's running in the PCC production environment is the same as the software they inspected when verifying the guarantees.

Violations

- Non open-source systems

Outline

5.1 Academic Systems

Category	Optimization	Threat	Orca 2206	FlexGen 2303	FastServe 2305	SpecInfer 2305	vLLM 2309	REST 2311	Splitwise 2311	SGLang 2312	Lookahead 2312	Sarathi 23-24	InfiniteLLM 2401	DistServe 2401	Medusa 2401	TetriInfer 2401	AttentionStore 2403	LoongServe 2404	Andes 2405	Llumnix 2406	Preble 2407	MInference 2407	TR 2408
Memory	Paging						Initial			🟩✓		🟩✓				🟩✓							
	Prefix Caching	**SE**								🟥✓											🟥✓		
	Disk Offloading	**SE**		🟥✓													🟥✓						
Tranmission	Duplication	**T**																					
	Pulling	**SET**												🟥✓									
	Request Migration																	🟩✓		🟩✓			
	Disaggregated Arch								🟩✓					🟩✓		🟩✓							
Batch	Continuous Batch		Initial		🟩✓	🟩✓	🟩✓					🟩✓		🟩✓		🟩✓							
	Chunked Prefill											Initial				🟩✓					🟩✓		
	Prepack Prefill													🟩✓		🟩✓							
Parallelism	Speculation					🟩✓		🟩✓		🟩✓	🟩✓				🟩✓								
	Context-Based Speculation	**S**						🟥✓															
	Database-Based Speculation	**S**									🟥✓												🟥✓
	Tensor Parallelism					🟩✓																	
	Sequence Parallelism																	🟩✓					
Scheduling	Priority-Based	**T**			🟥✓					🟥✓		🟥✓				🟥✓			🟥✓	🟥✓	🟥✓		
	Request-Level Prediction	**T**			🟥✓	🟥✓										🟥✓							
	Machine-level Scheduler	**E**			🟥✓				🟥✓				🟥✓			🟥✓		🟥✓			🟥✓		
	Instance Flip	**T**							🟥✓							🟥✓							
	Global Profiling	**P**		🟥✓					🟥✓					🟥✓								🟥✓	
Verification	Non Open-Source	**V**														🟥✓							🟥✓

Miscellaneous

Title	Keywords	Contributions
Prompt Cache	Prefill, Memory	Reuse attention states across different LLM prompts. Parse the prompt and use reusable text segments(snippet)
Layer-wise Transmission	Transmission	Transmit each layer's output to the next layer in the pipeline, instead of transmitting the entire model's output
LightLLM	Interface	Use http as the interface to the system
SkyPilot	Cross Region & Cloud	Given a job and its resource requirements (CPU/GPU/TPU), SkyPilot automatically figures out which locations (zone/region/cloud) have the compute to run the job, then sends it to the cheapest one to execute
MLC LLM	Efficient Execution	Enable efficient execution of large language models across a wide range of hardware platforms, including mobile devices, edge devices, and even web browsers
vAttention	Virtual Memory	stores KV-cache in contiguous virtual memory and leverages OS support for on-demand allocation of physical memory
MemServe	API, Framework	an elastic memory pool API managing distributed memory and KV caches across serving instances
CacheGen	Network, Streaming	CacheGen uses a custom tensor encoder, leveraging KV cache's distributional properties to encode a KV cache into more compact bitstream representations
DynamoLLM	Energy	It exploits heterogeneity in inference compute properties and fluctuations in inference workloads to save energy
MInference	Prefill, Long Context	Addresses the expensive computational cost and the unacceptable latency of the attention calculations in the pre-filling stage of long-context LLMs by leveraging dynamic sparse attention with spatial aggregation patterns
Shared Attention	Attention	directly sharing pre-computed attention weights across multiple layers in LLMs
SnapKV	Compression	Observing that specific tokens within prompts gain consistent attention from each head during generation, our methodology not only retrieve crucial information but also enhances processing efficiency

5.2 Industrial Systems

Category	Optimization	Threat	vLLM Open Source	LightLLM Open Source	FlexFlow Open Source	SGLang Open Source	Mooncake Moonshot	DeepSpeed Microsoft	TensorRT NVIDIA	TGI Hugging Face	Llama Intel	LMDeploy Shanghai AI lab	fastllm Open Source	rtp-llm Alibaba	MindIE Huawei
Memory	Paging		✓			✓	✓	✓		✓				✓	
	Token Attention			✓											
	Prefix Caching	S	✓						✓						
	Disk Offloading	SE				✓	✓		✓						✓
	Multi-Query Attention								✓						
Tranmission	Duplication	T					✓								
	Pulling	SET													
	Request Migration														
	Disaggregated Arch		✓				✓								✓
Batch	Continuous Batch		✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	
	Chunked Prefill		✓				✓	✓							
	Prepack Prefill														
Parallelism	Speculation		✓		✓	✓			✓	✓				✓	✓
	Tensor Parallelism									✓					✓
	Sequence Parallelism														
Scheduling	Priority-Based	T				✓	✓	✓							
	Request-Level Prediction	T		✓		✓									
	Machine-level Scheduler	E				✓	✓								
	Instance Flip	T													
	Global Profiling	P					✓								
Verification	Non Open-Source	V					✓		✓						

5.2 Industrial Systems

The roadmap of the vLLM project includes the following features:

Version	Date	Memory	Transmission	Batch	Parallelism	Scheduling	Model
v0.1	2306	Paging		Continuous Batching			MQA, GQA
v0.2	2309				Better TP & EP Support		AWQ
v0.3	2401	Prefix Caching					GPTQ
v0.4	2404		Optimize Distributed Communication	Chunked Prefill	Speculative Inference		
v0.5	2407	CPU Offloading			Support PP	Schedule multiple GPU steps in advances	FP8
v0.6	2409					Asynchronous output processor	

5.2 Industrial Systems

Trends

Category	Trend	Examples	Conflict
Memory	Improve reusability of KV Cache **with finer granularity**	Paging Token-Level Optimization	**S**
Transmission	Minimizing transmission **latency**	Data Duplication Prefetching/Pulling PD Disaggregation	**T**
Scheduling	Customized scheduling for **specific scenarios**	Request-level Predictions Machine-Level Scheduling Global profiling	**STP**
Parallelism	Optimizing parallelism for resource reuse and efficiency	Pipeline Parallelism Tensor Parallelism Sequence Parallelism Speculative Inference	**S**

S: Stateless computation E: Enforceable guarantees T: Non-targetability P: No privileged runtime access V: Verifiable transparency

Thanks

Outline

6 Appendix

Attention

https://zh.d2l.ai/chapter_attention-mechanisms/attention-cues.html

Inference Optimization

https://github.com/DefTruth/Awesome-LLM-Inference

https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/

Parallelism

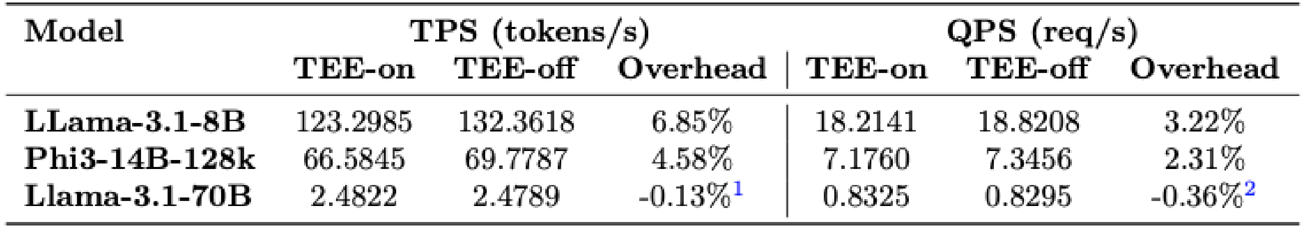
https://developer.nvidia.com/blog/demystifying-ai-inference-deployments-for-trillion-parameter-large-language-models/

Utilities

https://github.com/Trusted-AI/adversarial-robustness-toolbox

6 Appendix

Confidential Computing on nVIDIA H100 GPU: A Performance Benchmark Study

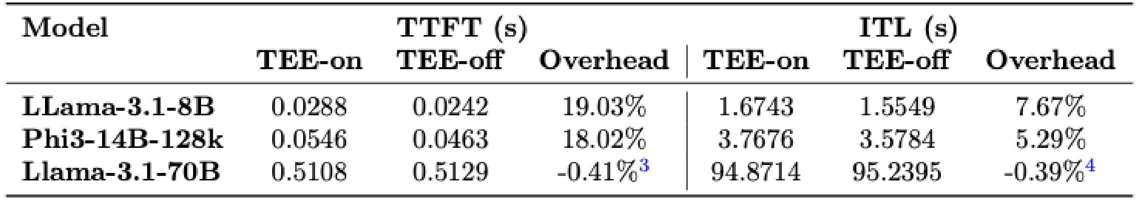


Model	TPS (tokens/s)			QPS (req/s)		
	TEE-on	TEE-off	Overhead	TEE-on	TEE-off	Overhead
LLama-3.1-8B	123.2985	132.3618	6.85%	18.2141	18.8208	3.22%
Phi3-14B-128k	66.5845	69.7787	4.58%	7.1760	7.3456	2.31%
Llama-3.1-70B	2.4822	2.4789	-0.13%[1]	0.8325	0.8295	-0.36%[2]

Table 1: Performance comparison of TEE-on and TEE-off modes for various models in terms of TPS (tokens per second) and QPS (queries per second).

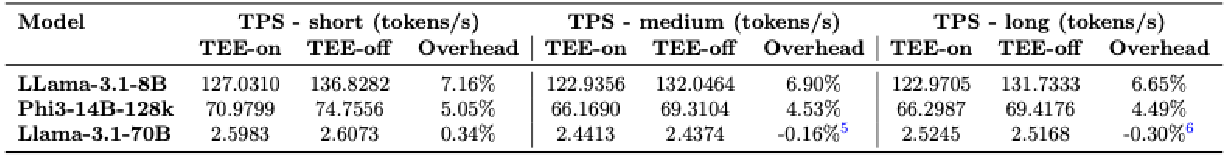
6 Appendix

Confidential Computing on nVIDIA H100 GPU: A Performance Benchmark Study



Model	TTFT (s)			ITL (s)		
	TEE-on	TEE-off	Overhead	TEE-on	TEE-off	Overhead
LLama-3.1-8B	0.0288	0.0242	19.03%	1.6743	1.5549	7.67%
Phi3-14B-128k	0.0546	0.0463	18.02%	3.7676	3.5784	5.29%
Llama-3.1-70B	0.5108	0.5129	-0.41%[3]	94.8714	95.2395	-0.39%[4]

Table 2: Comparison of TTFT (Time to First Token) and ITL (Inter Output Token Latency) for TEE-on and TEE-off modes across models.



Model	TPS - short (tokens/s)			TPS - medium (tokens/s)			TPS - long (tokens/s)		
	TEE-on	TEE-off	Overhead	TEE-on	TEE-off	Overhead	TEE-on	TEE-off	Overhead
LLama-3.1-8B	127.0310	136.8282	7.16%	122.9356	132.0464	6.90%	122.9705	131.7333	6.65%
Phi3-14B-128k	70.9799	74.7556	5.05%	66.1690	69.3104	4.53%	66.2987	69.4176	4.49%
Llama-3.1-70B	2.5983	2.6073	0.34%	2.4413	2.4374	-0.16%[5]	2.5245	2.5168	-0.30%[6]

Table 3: Performance comparison of TEE-on and TEE-off modes across different sequence lengths in terms of TPS (tokens per second). Short sequences are no longer than 100 tokens. Medium sequences are no longer than 500 tokens. Long sequences are between 501 and 1500 tokens.

6 Appendix

Stateful Inference Systems

Static state States in traditional systems can be modified after creation and require various consistency and coherence mechanisms to support parallelism. In LLM inference, once KVs are computed for a sequence, their values do not change.

Regular computation patterns LLMs' transformer computation is regular. Its computing and memory consumption is determined by the model size, the prompt length, and the output generation length. The model size and a request's prompt length are known before execution, and output is generated one token per iteration. Thus, we can estimate the computing and memory consumption for every iteration.

6 Appendix

Quantization

Quantization is the process of reducing the precision of a model's weights and activations.

Sparsity

Sparsity is the process of setting a portion of the model's weights to zero. Then the model can be expressed as a sparse matrix.

Distillation

Distillation is the process of training a smaller model to mimic the behavior of a larger model.