: Architectural Deep Dive

Understanding a High Throughput LLM Inference System

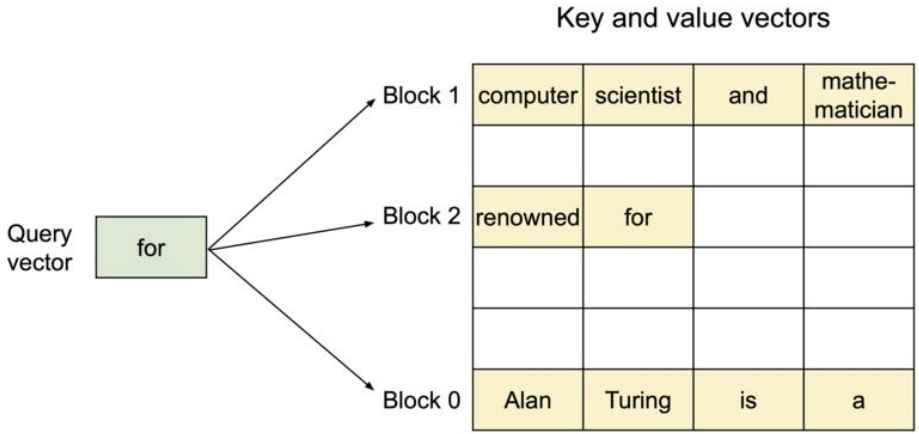
By: **Ayush Satyam**
Software Engineer at Red Hat
GitHub: **@ayushsatyam146**

Why vLLM is awesome!!

- Paged Attention
- Optimized K-V Caching
- Optimized CUDA kernels
- Speculative Decoding
- Chunked Prefill and many more

Paged Attention

An attention algorithm that allows for storing continuous keys and values in non-contiguous memory space.



Managing KV cache: Lessons from OS

Request A

Alan	Turing	is	a
computer	scientist	and	mathema tician
renowned			

Logical KV blocks

Request B

Artificial	Intelli- gence	is	the
future	of	tech- nology	

Block Table

Block Table

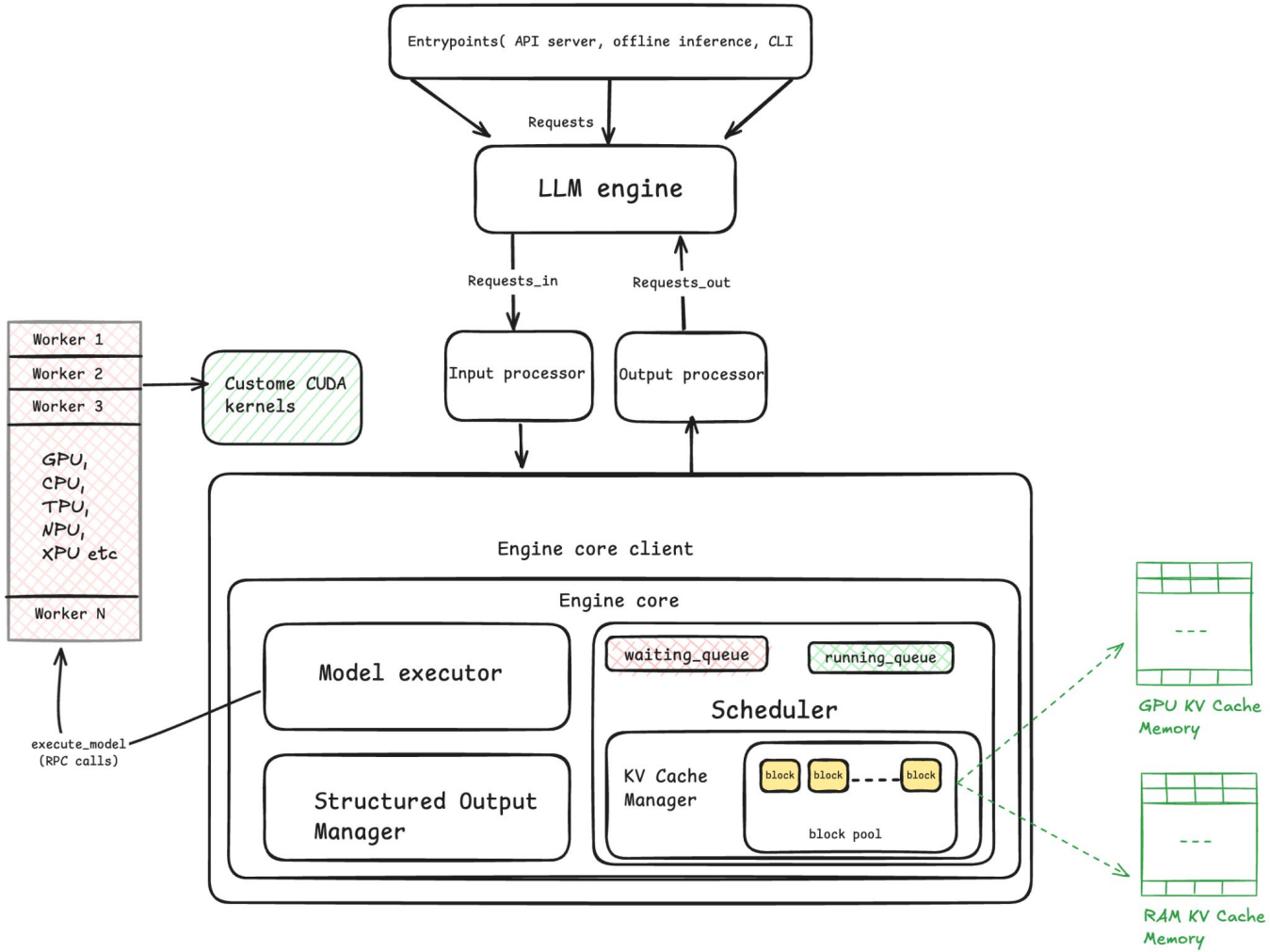
Physical KV blocks

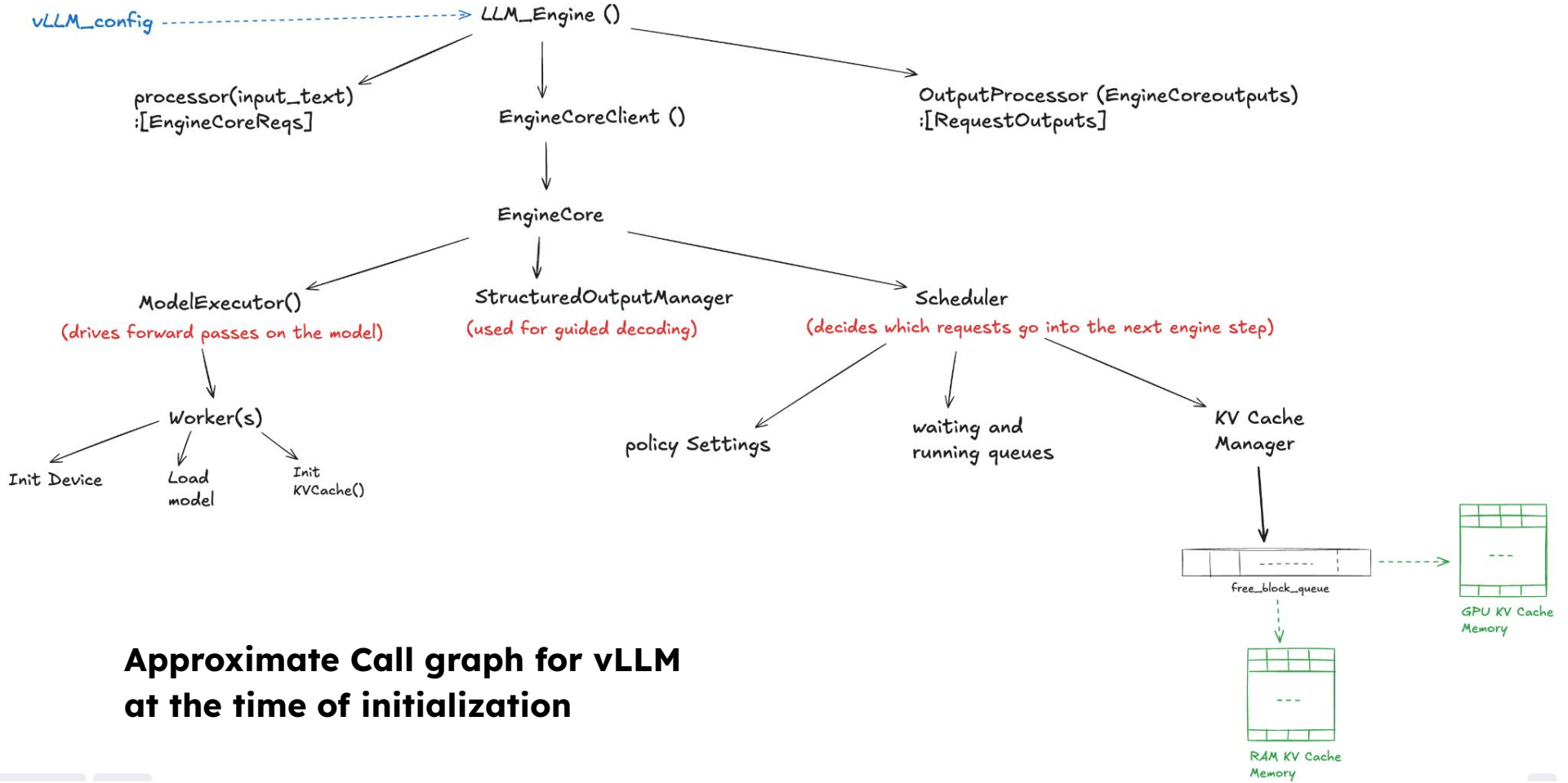
computer	scientist	and	mathe- matician
Artificial	Intelli- gence	is	the
renowned			
future	of	tech- nology	
Alan	Turing	is	a

10k feet view of the system

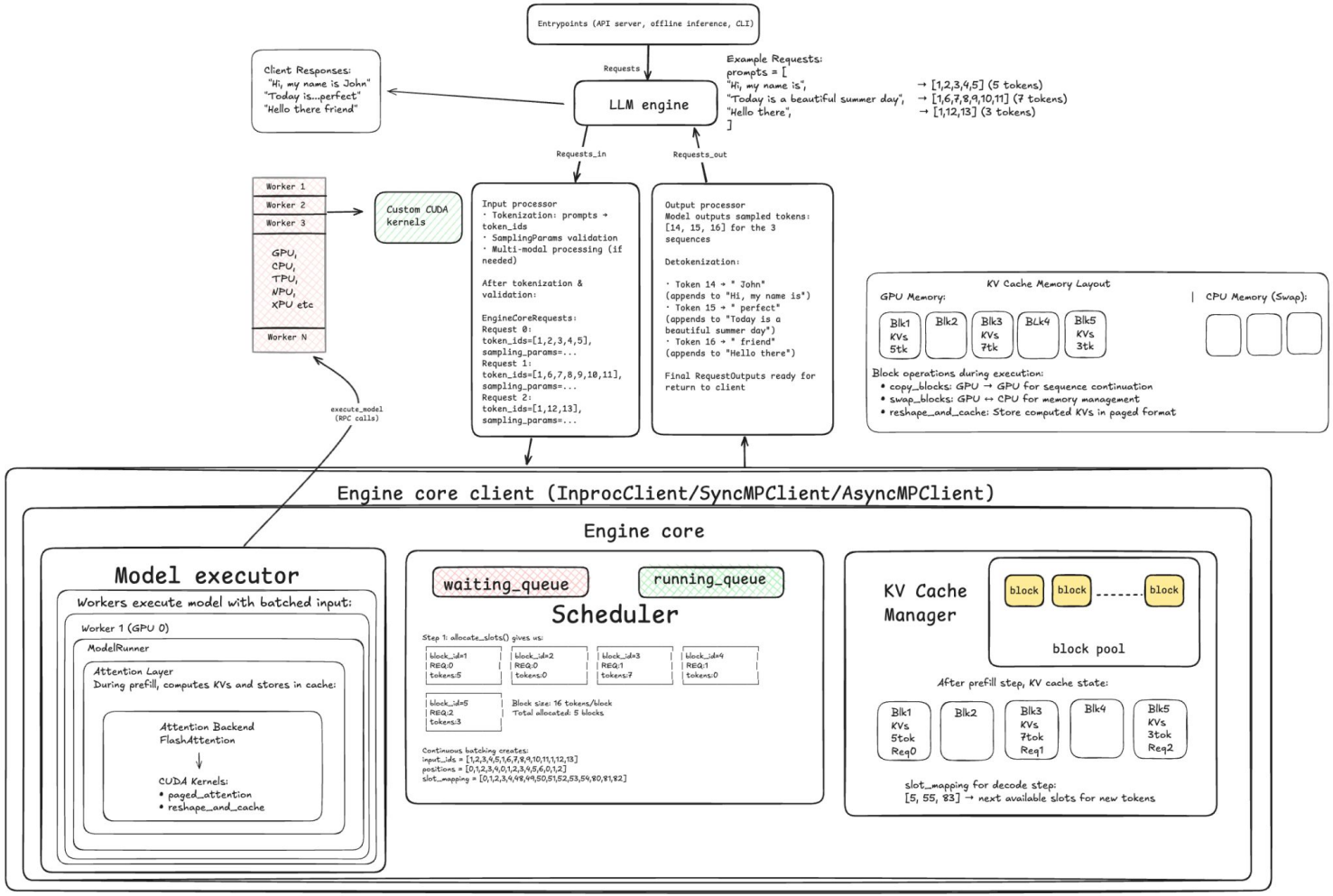
What are the core components of the system?

- Tokenizer
- LLM Engine
- KV cache manager
- Worker
- Scheduler
- Model executor….
- … Let’s **SEE** in detail


Entrypoints(API server, offline inference, CLI
Requests
LLM engine
Requests_in
Requests_out
Input processor
Output processor
Worker 1
Worker 2
Worker 3
GPU,
CPU,
TPU,
NPU,
XPU etc
Worker N
Custome CUDA kernels
Engine core client
Engine core
Model executor
Structured Output Manager
waiting_queue
running_queue
Scheduler
KV Cache Manager
block
block
block
block pool
execute_model
(RPC calls)
GPU KV Cache Memory
RAM KV Cache Memory



Approximate Call graph for vLLM at the time of initialization

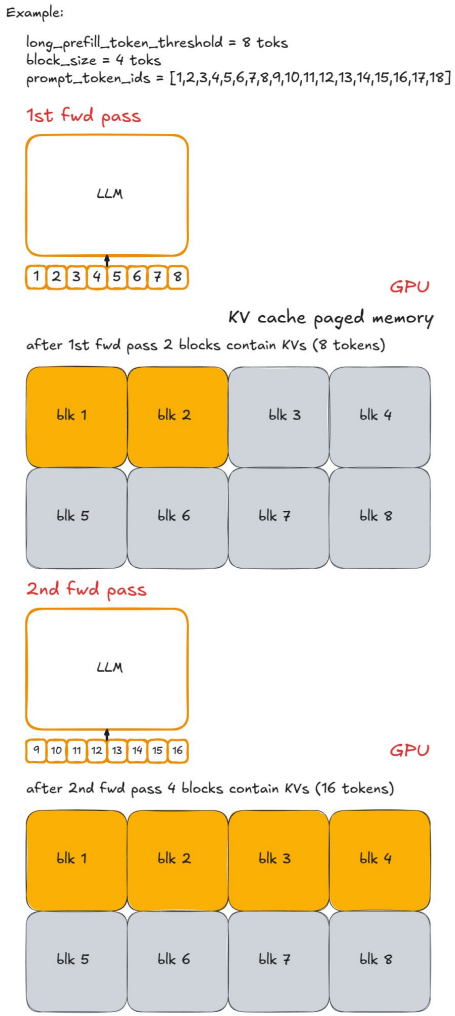

Entrypoints (API server, offline inference, CLI)
Requests
LLM engine
Client Responses:
"Hi, my name is John"
"Today is...perfect"
"Hello there friend"
Example Requests:
prompts = [
"Hi, my name is", → [1,2,3,4,5] (5 tokens)
"Today is a beautiful summer day", → [1,6,7,8,9,10,11] (7 tokens)
"Hello there", → [1,12,13] (3 tokens)
]
Requests_in
Requests_out
Worker 1
Worker 2
Worker 3
GPU,
CPU,
TPU,
NPU,
XPU etc
Worker N
Custom CUDA kernels
Input processor
· Tokenization: prompts → token_ids
· SamplingParams validation
· Multi-modal processing (if needed)
After tokenization & validation:
EngineCoreRequests:
Request 0:
token_ids=[1,2,3,4,5],
sampling_params=...
Request 1:
token_ids=[1,6,7,8,9,10,11],
sampling_params=...
Request 2:
token_ids=[1,12,13],
sampling_params=...
Output processor
Model outputs sampled tokens:
[14, 15, 16] for the 3 sequences
Detokenization:
· Token 14 → " John" (appends to "Hi, my name is")
· Token 15 → " perfect" (appends to "Today is a beautiful summer day")
· Token 16 → " friend" (appends to "Hello there")
Final RequestOutputs ready for return to client
KV Cache Memory Layout
GPU Memory:
CPU Memory (Swap):
Blk1 KVs 5tk
Blk2
Blk3 KVs 7tk
Blk4
Blk5 KVs 3tk
Block operations during execution:
• copy_blocks: GPU → GPU for sequence continuation
• swap_blocks: GPU ↔ CPU for memory management
• reshape_and_cache: Store computed KVs in paged format
execute_model (RPC calls)
Engine core client (InprocClient/SyncMPClient/AsyncMPClient)
Engine core
Model executor
Workers execute model with batched input:
Worker 1 (GPU 0)
ModelRunner
Attention Layer
During prefill, computes KVs and stores in cache:
Attention Backend
FlashAttention
CUDA Kernels:
• paged_attention
• reshape_and_cache
waiting_queue
running_queue
Scheduler
Step 1: allocate_slots() gives us:
block_id=1 REQ:0 tokens:5
block_id=2 REQ:0 tokens:0
block_id=3 REQ:1 tokens:7
block_id=4 REQ:1 tokens:0
block_id=5 REQ:2 tokens:3
Block size: 16 tokens/block
Total allocated: 5 blocks
Continuous batching creates:
input_ids = [1,2,3,4,5,1,6,7,8,9,10,11,1,12,13]
positions = [0,1,2,3,4,0,1,2,3,4,5,6,0,1,2]
slot_mapping = [0,1,2,3,4,48,49,50,51,52,53,54,80,81,82]
KV Cache Manager
block
block
block
block pool
After prefill step, KV cache state:
Blk1 KVs 5tok Req0
Blk2
Blk3 KVs 7tok Req1
Blk4
Blk5 KVs 3tok Req2
slot_mapping for decode step:
[5, 55, 83] → next available slots for new tokens

Let’s see some Advanced features

- Chunked Prefill
- Prefix Caching
- Guided decoding (FSM)
- Disaggregated PD
- and more…

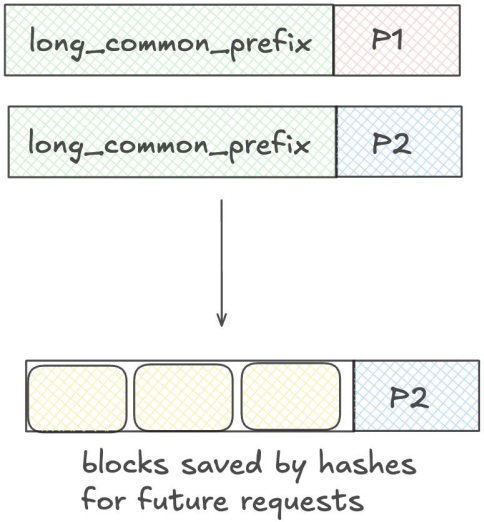
Chunked Prefill

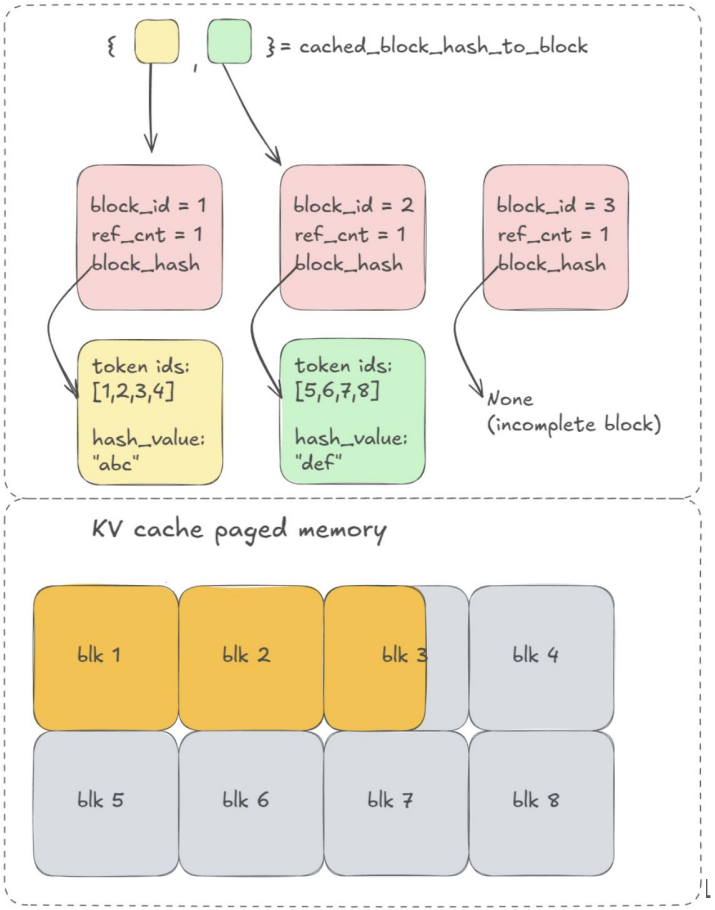
Chunked Prefill is a technique for handling long prompts by splitting their prefill step into smaller chunks



Prefix Caching

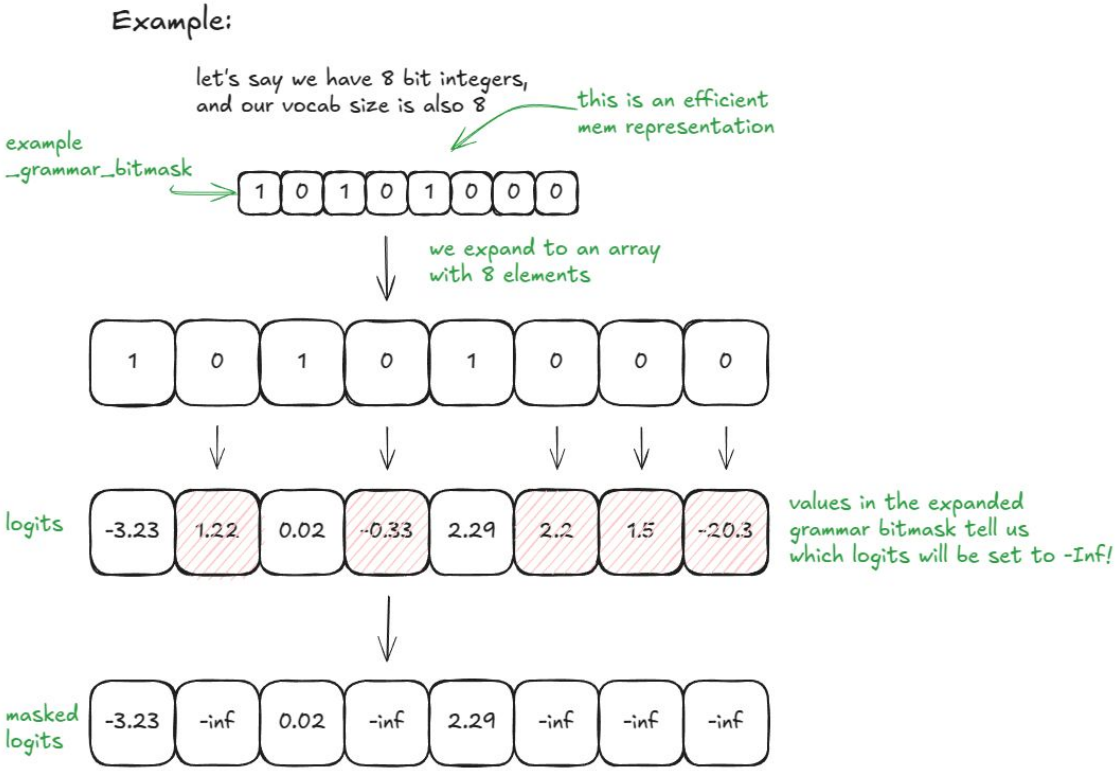
Prefix Caching avoids recomputing tokens that multiple prompts share at the beginning - hence prefix.





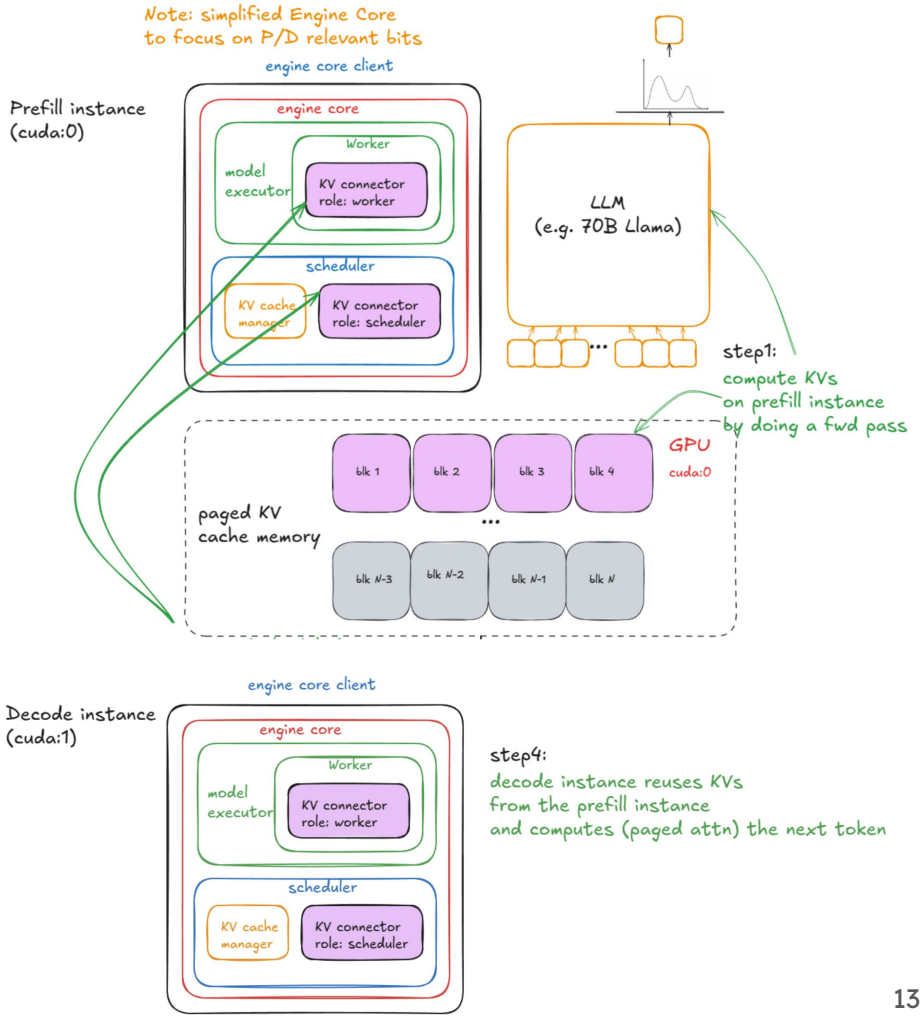
Guided Decoding

Guided Decoding is a technique where, at each decoding step, the logits are constrained by a grammar-based finite state machine.



Disaggregated PD

Prefill and decode have very different performance profiles (compute-bound vs. memory-bandwidth-bound), so separating their execution is a sensible design. It gives tighter control over latency — both TFTT (time-to-first-token) and ITL (inter-token latency



References:

- Anatomy of vLLM Blog
- vLLM codebase (Must read!)
- Talk by WooSuk Kwon & Zhuohan Li
- Modal notebook here
- vLLM paper here

THANK YOU

Reach out to me on:

@ayushsatyam146

@ayushsatyam146