# vLLM

INTRODUCTION TO LLM
INFERENCE SERVING SYSTEMS

CHUHONG YUAN

# Review Of Attention Algorithm

- $Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_h}}\right)V$

# Review Of Attention Algorithm

Attention Score

- $Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_h}}\right)V$

# Review Of Attention Algorithm

Attention Score

- $Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_h}}\right)V$
- $a_{ij} = \frac{\exp(q_i k_j^T / \sqrt{d_h})}{\sum_{t=1}^{i} \exp(q_i k_t^T / \sqrt{d_h})}$ for each token

# Review Of Attention Algorithm

Attention Score

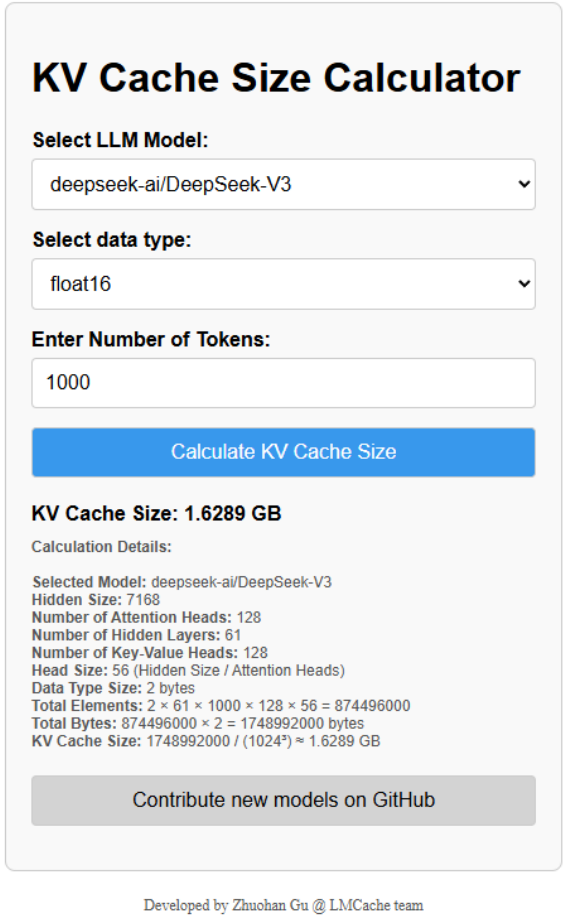
- $Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_h}}\right)V$
- $a_{ij} = \frac{\exp(q_i k_j^T / \sqrt{d_h})}{\sum_{t=1}^{i} \exp(q_i k_t^T / \sqrt{d_h})}$ for each token
- The tensors are stored in contiguous memory in prior works

# Memory Challenges In Decoding

- KV Cache
  - Store the previous tokens' K and V for accelerating decoding
  - Memory-costly
- Diverse memory usage patterns
- Unpredictable output length

# Memory Challenges In Decoding

- KV Cache
  - Store the previous tokens' K and V for accelerating decoding
  - Memory-costly
  - https://lmcache.ai/kv_cache_calculator.html
  - B200 192GB memory
  - Prior work requires the tensors to be stored in contiguous memory
- Diverse memory usage patterns
- Unpredictable output length



# Memory Challenges In Decoding

- KV Cache
- Diverse memory usage patterns
  - Requests of different sizes – unified memory allocation is inappropriate
  - Requests of the same prefixes – memory can be shared
- Unpredictable output length

# Memory Challenges In Decoding

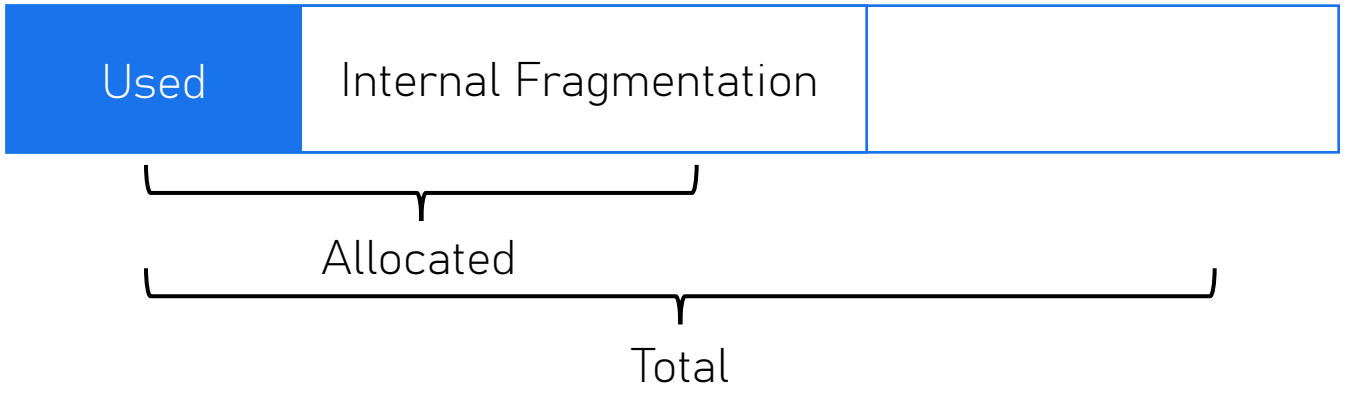
- KV Cache
- Diverse memory usage patterns
- Unpredictable output length
  - Outputs depend on the inputs, so the memory cost varies

# Memory Fragmentation

- Internal fragmentation
- External fragmentation

# Memory Fragmentation

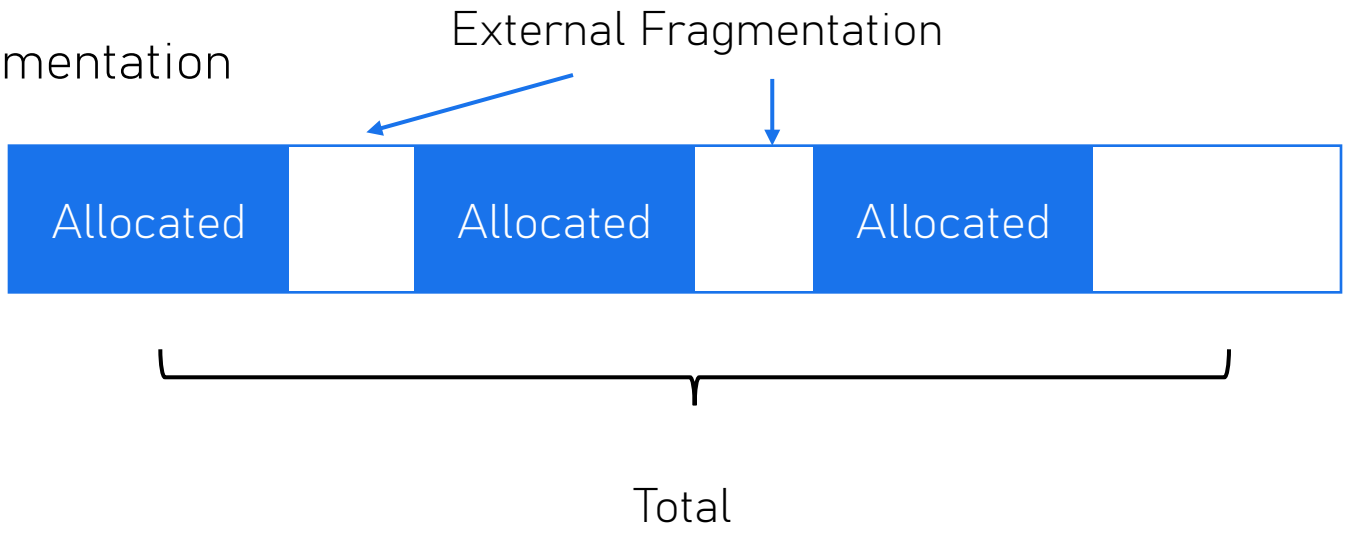
- Internal fragmentation




- External fragmentation

# Memory Fragmentation

- Internal fragmentation
- External fragmentation



# PagedAttention

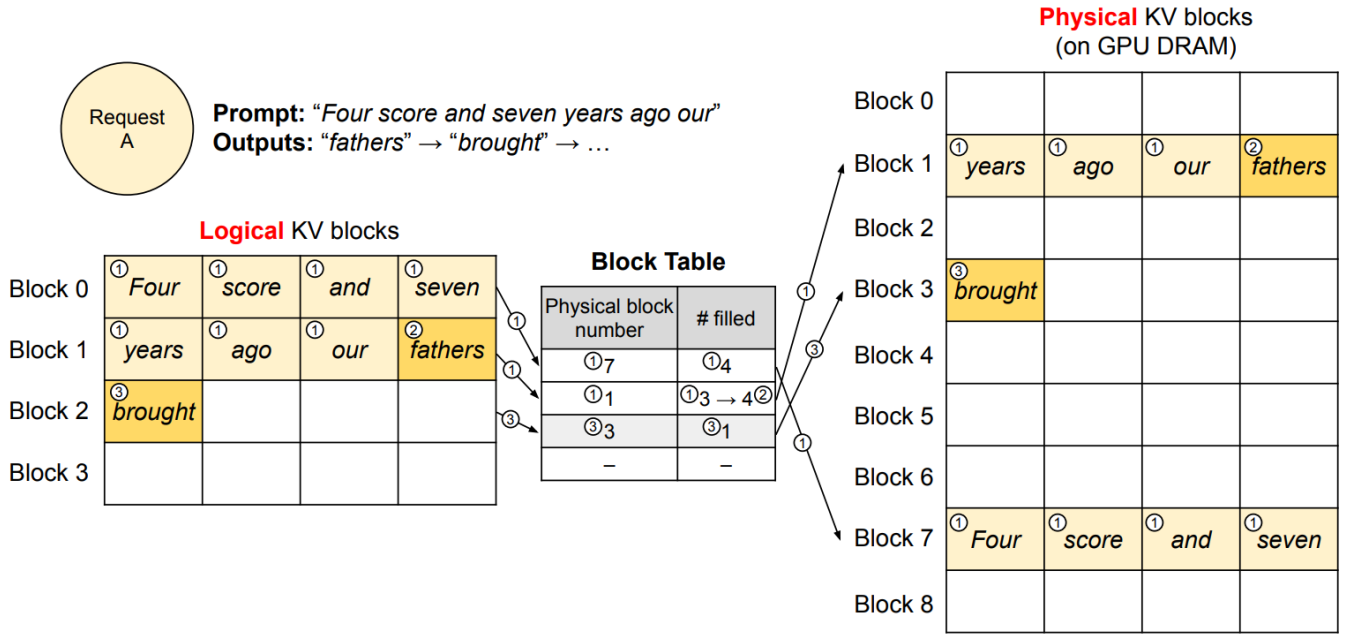
- Divide the KV Cache used in attention algorithm into KV Blocks
- $a_{ij} = \frac{\exp(q_i k_j^T / \sqrt{d_h})}{\sum_{t=1}^{i} \exp(q_i k_t^T / \sqrt{d_h})}$

# PagedAttention

- Divide the KV Cache used in attention algorithm into KV Blocks
- $A_{ij} = \frac{\exp(q_i K_j^T / \sqrt{d_h})}{\sum_{t=1}^{\lfloor i/B \rfloor} \exp(q_i K_t^T / \sqrt{d_h})}$, B = the block size
- The result can be finally merged

# KV Cache Manager

- Page-like KV Cache management

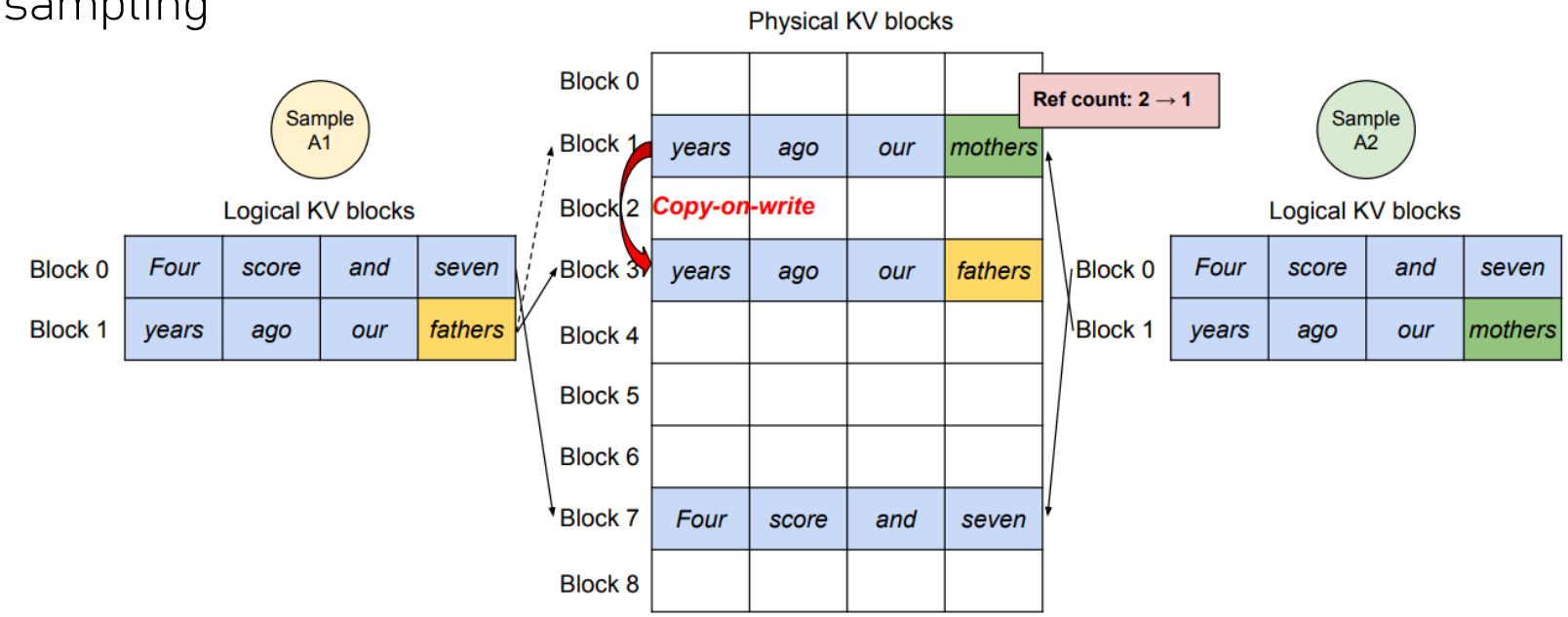


# KV Cache Manager

- Page-like KV Cache management
- How does this method solve the challenges in the memory management?
  - Memory utilization – better utilization rate
  - Memory fragmentation – mitigate both internal and external fragmentations
  - Diverse memory usage patterns and unpredictable output lengths – dynamic memory allocation

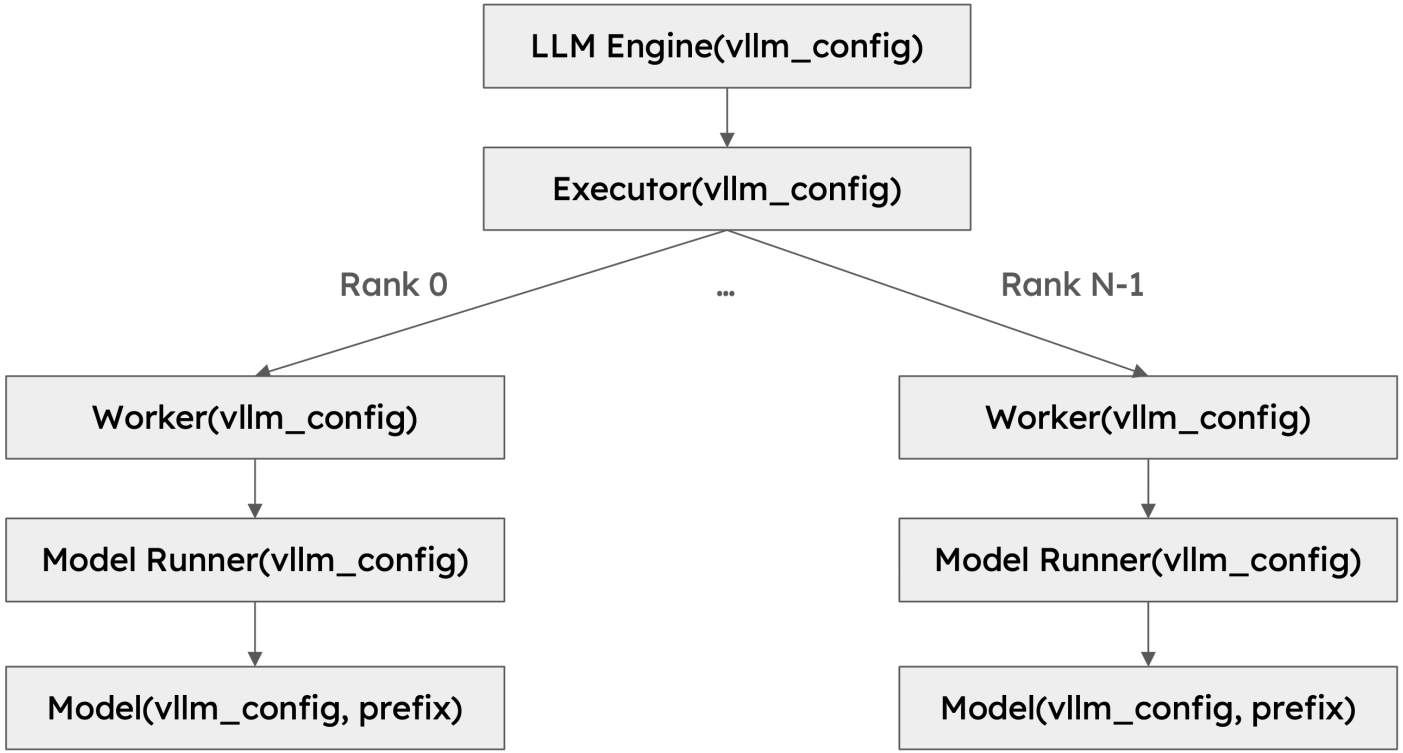
# Possible Applications

- Parallel sampling



# vLLM Usage

# vLLM Class Hierarchy



# LLM Entry Point

```python
class LLM:
    """An LLM for generating texts from given prompts and sampling parameters.…

    DEPRECATE_LEGACY: ClassVar[bool] = True
    """A flag to toggle whether to deprecate the legacy generate/encode API."""

    DEPRECATE_INIT_POSARGS: ClassVar[bool] = True
    """
    A flag to toggle whether to deprecate positional arguments in
    :meth:`LLM.__init__`.
    """

    @classmethod
    @contextmanager
    def deprecate_legacy_api(cls):…

    @deprecate_args(…
    def __init__(
        self,
        model: str,
        tokenizer: Optional[str] = None,
        tokenizer_mode: str = "auto",
        skip_tokenizer_init: bool = False,
        trust_remote_code: bool = False,
        allowed_local_media_path: str = "",
        tensor_parallel_size: int = 1,
        dtype: str = "auto",
        quantization: Optional[str] = None,
        revision: Optional[str] = None,
        tokenizer_revision: Optional[str] = None,
        seed: Optional[int] = None,
        gpu_memory_utilization: float = 0.9,
        swap_space: float = 4,
        cpu_offload_gb: float = 0,
        enforce_eager: Optional[bool] = None,
        max_seq_len_to_capture: int = 8192,
        disable_custom_all_reduce: bool = False,
        disable_async_output_proc: bool = False,
        hf_token: Optional[Union[bool, str]] = None,
        hf_overrides: Optional[HfOverrides] = None,
        mm_processor_kwargs: Optional[dict[str, Any]] = None,
        # After positional args are removed, move this right below `model`
        task: TaskOption = "auto",
        override_pooler_config: Optional[PoolerConfig] = None,
        compilation_config: Optional[Union[int, dict[str, Any]]] = None,
        **kwargs,
    ) -> None:
        '''
```

# LLM Entry Point

```python
def generate(
    self,
    prompts: Union[Union[PromptType, Sequence[PromptType]],
                   Optional[Union[str, list[str]]]] = None,
    sampling_params: Optional[Union[SamplingParams,
                                    Sequence[SamplingParams]]] = None,
    prompt_token_ids: Optional[Union[list[int], list[list[int]]]] = None,
    use_tqdm: bool = True,
    lora_request: Optional[Union[list[LoRARequest], LoRARequest]] = None,
    prompt_adapter_request: Optional[PromptAdapterRequest] = None,
    guided_options_request: Optional[Union[LLMGuidedOptions,
                                           GuidedDecodingRequest]] = None,
    priority: Optional[list[int]] = None,
) -> list[RequestOutput]:
    """Generates the completions for the input prompts. ···
    runner_type = self.llm_engine.model_config.runner_type
    if runner_type not in ["generate", "transcription"]: ···

    if prompt_token_ids is not None: ···
    else: ···

    if isinstance(guided_options_request, dict): ···

    if sampling_params is None: ···

    self._validate_and_add_requests(
        prompts=parsed_prompts,
        params=sampling_params,
        lora_request=lora_request,
        prompt_adapter_request=prompt_adapter_request,
        guided_options=guided_options_request,
        priority=priority)

    outputs = self._run_engine(use_tqdm=use_tqdm)
    return self.engine_class.validate_outputs(outputs, RequestOutput)
```

```python
def _add_request(
    self,
    prompt: PromptType,
    params: Union[SamplingParams, PoolingParams],
    lora_request: Optional[LoRARequest] = None,
    prompt_adapter_request: Optional[PromptAdapterRequest] = None,
    priority: int = 0,
) -> None:
    request_id = str(next(self.request_counter))
    self.llm_engine.add_request(
        request_id,
        prompt,
        params,
        lora_request=lora_request,
        prompt_adapter_request=prompt_adapter_request,
        priority=priority,
    )
```

# LLM Entry Point

```python
def generate(
    self,
    prompts: Union[Union[PromptType, Sequence[PromptType]],
                   Optional[Union[str, list[str]]]] = None,
    sampling_params: Optional[Union[SamplingParams,
                                    Sequence[SamplingParams]]] = None,
    prompt_token_ids: Optional[Union[list[int], list[list[int]]]] = None,
    use_tqdm: bool = True,
    lora_request: Optional[Union[list[LoRARequest], LoRARequest]] = None,
    prompt_adapter_request: Optional[PromptAdapterRequest] = None,
    guided_options_request: Optional[Union[LLMGuidedOptions,
                                           GuidedDecodingRequest]] = None,
    priority: Optional[list[int]] = None,
) -> list[RequestOutput]:
    """Generates the completions for the input prompts. ···
    runner_type = self.llm_engine.model_config.runner_type
    if runner_type not in ["generate", "transcription"]: ···

    if prompt_token_ids is not None: ···
    else: ···

    if isinstance(guided_options_request, dict): ···

    if sampling_params is None: ···

    self._validate_and_add_requests(
        prompts=parsed_prompts,
        params=sampling_params,
        lora_request=lora_request,
        prompt_adapter_request=prompt_adapter_request,
        guided_options=guided_options_request,
        priority=priority)

    outputs = self._run_engine(use_tqdm=use_tqdm)
    return self.engine_class.validate_outputs(outputs, RequestOutput)
```

```python
def _run_engine(
        self, *, use_tqdm: bool
) -> list[Union[RequestOutput, PoolingRequestOutput]]:
    # Initialize tqdm.
    if use_tqdm: ···

    # Run the engine.
    outputs: list[Union[RequestOutput, PoolingRequestOutput]] = []
    total_in_toks = 0
    total_out_toks = 0
    while self.llm_engine.has_unfinished_requests():
        step_outputs = self.llm_engine.step()
        for output in step_outputs:
            if output.finished:
                outputs.append(output)
                if use_tqdm:
                    if isinstance(output, RequestOutput):
                        # Calculate tokens only for RequestOutput
                        n = len(output.outputs)
                        assert output.prompt_token_ids is not None
                        total_in_toks += len(output.prompt_token_ids) * n
                        in_spd = total_in_toks / pbar.format_dict["elapsed"]
                        total_out_toks += sum(
                            len(stp.token_ids) for stp in output.outputs)
                        out_spd = (total_out_toks /
                                   pbar.format_dict["elapsed"])
                        pbar.postfix = (
                            f"est. speed input: {in_spd:.2f} toks/s, "
                            f"output: {out_spd:.2f} toks/s")
                        pbar.update(n)
                    else:
                        pbar.update(1)
```

# LLM Engine

```python
def add_request(
        self,
        request_id: str,
        prompt: Optional[PromptType] = None,
        params: Optional[Union[SamplingParams, PoolingParams]] = None,
        arrival_time: Optional[float] = None,
        lora_request: Optional[LoRARequest] = None,
        trace_headers: Optional[Mapping[str, str]] = None,
        prompt_adapter_request: Optional[PromptAdapterRequest] = None,
        priority: int = 0,
        *,
        inputs: Optional[PromptType] = None,  # DEPRECATED
) -> None:
    """Add a request to the engine's request pool.

    The request is added to the request pool and will be processed by the
    scheduler as `engine.step()` is called. The exact scheduling policy is
    determined by the scheduler.
```

# LLM Engine

```python
def step(self) -> List[Union[RequestOutput, PoolingRequestOutput]]:
    """Performs one decoding iteration and returns newly generated results.

    .. figure:: https://i.imgur.com/sv2HssD.png
        :alt: Overview of the step function
        :align: center

        Overview of the step function.

    Details:
        - Step 1: Schedules the sequences to be executed in the next
          iteration and the token blocks to be swapped in/out/copy.

            - Depending on the scheduling policy,
              sequences may be `preempted/reordered`.
            - A Sequence Group (SG) refer to a group of sequences
              that are generated from the same prompt.

        - Step 2: Calls the distributed executor to execute the model.
        - Step 3: Processes the model output. This mainly includes:

            - Decodes the relevant outputs.
            - Updates the scheduled sequence groups with model outputs
              based on its `sampling parameters` (`use_beam_search` or not).
            - Frees the finished sequence groups.

        - Finally, it creates and returns the newly generated results.
```

```python
if not scheduler_outputs.is_empty():

    # Check if we have a cached last_output from the previous iteration.
    # For supporting PP this is probably the best way to pass the
    # sampled_token_ids, as a separate broadcast over all the PP stages
    # will cause one virtual engine's microbatch to block the pipeline.
    last_sampled_token_ids = \
        self._get_last_sampled_token_ids(virtual_engine)

    execute_model_req = ExecuteModelRequest(…

    if allow_async_output_proc:…

    try:
        outputs = self.model_executor.execute_model(
            execute_model_req=execute_model_req)
        self._skip_scheduling_next_step = False
    except InputProcessingError as e:
        # The input for this request cannot be processed, so we must
        # abort it. If there are remaining requests in the batch that
        # have been scheduled, they will be retried on the next step.
        invalid_request_id = e.request_id
        self._abort_and_cache_schedule(
            request_id=invalid_request_id,
            virtual_engine=virtual_engine,
            seq_group_metadata_list=seq_group_metadata_list,
            scheduler_outputs=scheduler_outputs,
            allow_async_output_proc=allow_async_output_proc)
        # Raise so the caller is notified that this request failed
        raise
```

# Executor

```python
class ExecutorBase(ABC):
    """Base class for all executors.

    An executor is responsible for executing the model on one device,
    or it can be a distributed executor
    that can execute the model on multiple devices.
    """
```

```python
class UniProcExecutor(ExecutorBase):

    uses_ray: bool = False

    def _init_executor(self) -> None: ...

    def collective_rpc(self,
                       method: Union[str, Callable],
                       timeout: Optional[float] = None,
                       args: Tuple = (),
                       kwargs: Optional[Dict] = None) -> List[Any]:
        if kwargs is None:
            kwargs = {}
        answer = run_method(self.driver_worker, method, args, kwargs)
        return [answer]
```

```python
def execute_model(
    self, execute_model_req: ExecuteModelRequest
) -> Optional[List[Union[SamplerOutput, PoolerOutput]]]:
    output = self.collective_rpc("execute_model",
                                 args=(execute_model_req, ))
    return output[0]
```

# Worker

```python
class LocalOrDistributedWorkerBase(WorkerBase):
    """
    Partial implementation of WorkerBase that has a default `execute_model`
    definition to perform metadata transfer between workers when in distributed
    mode. Subclasses of this interface should use model runners that inherit
    from ModelRunnerBase, and should only need to implement worker-local logic.
    If custom control plane logic is needed to transfer metadata, or if the
    model runner cannot inherit from ModelRunnerBase, use WorkerBase instead.
    """
    is_driver_worker: bool
    model_runner: ModelRunnerBase
    observability_config: Optional[ObservabilityConfig] = None
```

```python
def execute_model(
    self,
    execute_model_req: Optional[ExecuteModelRequest] = None,
) -> Optional[List[SamplerOutput]]:
    """Executes at least one model step on the given sequences, unless no
    sequences are provided."""
    start_time = time.perf_counter()

    inputs = self.prepare_input(execute_model_req)
    if inputs is None:...

    model_input, worker_input, kwargs = inputs
    num_steps = worker_input.num_steps
    if (execute_model_req is not None and execute_model_req.spec_step_idx):...

    self.execute_worker(worker_input)

    # If there is no input, we don't need to execute the model.
    if worker_input.num_seq_groups == 0:...

    intermediate_tensors = None
    orig_model_execute_time = 0.0
    if not get_pp_group().is_first_rank:...

    output = self.model_runner.execute_model(
        model_input=model_input,
        kv_caches=self.kv_cache[worker_input.virtual_engine]
        if self.kv_cache is not None else None,
        intermediate_tensors=intermediate_tensors,
        num_steps=num_steps,
        **kwargs,
    )
```

# ModelRunner

```python
@torch.inference_mode()
def execute_model(
    self,
    model_input: ModelInputForGPUWithSamplingMetadata,
    kv_caches: List[torch.Tensor],
    intermediate_tensors: Optional[IntermediateTensors] = None,
    num_steps: int = 1,
    **kwargs,
) -> Optional[Union[List[SamplerOutput], IntermediateTensors]]:
```

```python
if not bypass_model_exec:
    with set_forward_context(model_input.attn_metadata,
                             self.vllm_config, virtual_engine):
        hidden_or_intermediate_states = model_executable(
            input_ids=model_input.input_tokens,
            positions=model_input.input_positions,
            intermediate_tensors=intermediate_tensors,
            **MultiModalKwargs.as_kwargs(multi_modal_kwargs,
                                         device=self.device),
            **seqlen_agnostic_kwargs,
            **model_kwargs,
        )

if (self.observability_config is not None
```

```python
logits = self.model.compute_logits(hidden_or_intermediate_states,
                                   model_input.sampling_metadata)

if not self.is_driver_worker:…

if model_input.async_callback is not None:…

# Sample the next token.
output: SamplerOutput = self.model.sample(
    logits=logits,
    sampling_metadata=model_input.sampling_metadata,
)
if (self.observability_config is not None
        and self.observability_config.collect_model_forward_time
        and output is not None):…

if self.return_hidden_states:…

return [output]
```

# Model

```python
@support_torch_compile
class LlamaModel(nn.Module):

    def __init__(self, ···

    def get_input_embeddings(self, input_ids: torch.Tensor) -> torch.Tensor:
        return self.embed_tokens(input_ids)

    def forward(
        self,
        input_ids: Optional[torch.Tensor],
        positions: torch.Tensor,
        intermediate_tensors: Optional[IntermediateTensors],
        inputs_embeds: Optional[torch.Tensor] = None,
    ) -> Union[torch.Tensor, IntermediateTensors]:
        if get_pp_group().is_first_rank:
            if inputs_embeds is not None:
                hidden_states = inputs_embeds
            else:
                hidden_states = self.get_input_embeddings(input_ids)
            residual = None
        else:
            assert intermediate_tensors is not None
            hidden_states = intermediate_tensors["hidden_states"]
            residual = intermediate_tensors["residual"]

        for layer in self.layers[self.start_layer:self.end_layer]:
            hidden_states, residual = layer(positions, hidden_states, residual)

        if not get_pp_group().is_last_rank:
            return IntermediateTensors({
                "hidden_states": hidden_states,
                "residual": residual
            })

        hidden_states, _ = self.norm(hidden_states, residual)
        return hidden_states
```

```python
def compute_logits(
    self,
    hidden_states: torch.Tensor,
    sampling_metadata: SamplingMetadata,
) -> Optional[torch.Tensor]:
    logits = self.logits_processor(self.lm_head, hidden_states,
                                   sampling_metadata)
    return logits

def sample(self, logits: torch.Tensor,
           sampling_metadata: SamplingMetadata) -> Optional[SamplerOutput]:
    next_tokens = self.sampler(logits, sampling_metadata)
    return next_tokens
```

# Homework - SGLang

- In this homework, read the paper of SGLang, answer the following questions.
  - 1. How does RadixAttention work? Use an example different from the one in the paper to explain.
  - 2. What benchmarks does the paper use for evaluating SGLang? What kinds of workloads do they represent? You can discuss several benchmarks together if they represent the same kind of workload. Hint: check the `benchmark` directory of SGLang source code.

- Related links:
  - https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdfLinks to an external site.
  - https://github.com/sgl-project/sglang

# Q & A