Requset-level GPU Sharing on vLLM

Tao Li, Zhuoyuan Li, Chuanyi Liu

2024-11-27

Planning and Tracking List

USTC, CHINA
ADSLAB

	Task	Schedule	Task Owner	11.12 Status	Last Week Status	Current Status
Model level	Deploy serverlessLLM on k8s	11.4-8	Tao&Zhuoyuan	***Complete***	***Complete***	***Complete***
	Profile key metrics of LLMs under different configurations	11.4-8	Chuanyi	***Complete***	***Complete***	***Complete***
	Implement MPS on serverlessLLM	11.8-15	Chuanyi	On track	***Complete***	***Complete***
	Dynamic resource allocation based on model popularity	11.15-20	Tao& Zhuoyuan	On track	***Complete***	***Complete***
Request level	Independent scheduling for prefill and decode phases	11.19-26	Chuanyi Liu	-	-	On track
	Parameter-sharing for multiple requests of the same model	11.19-26	Tao& Zhuoyuan	-	-	***Complete*** ***(vLLM)***

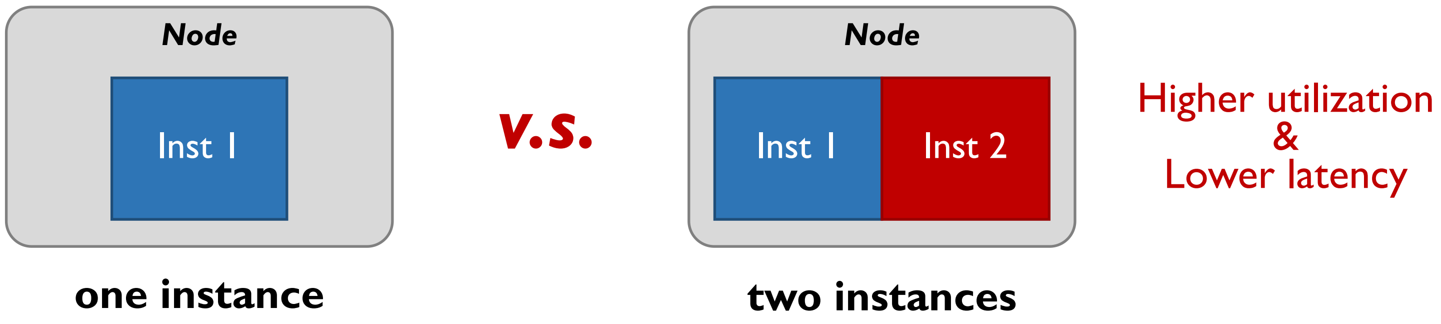
Part I: *Parameter-sharing for multiple requests of the same model*

Part1.1: GPU memory sharing among processes

Why Sharing Parameters?

USTC, CHINA ADSLAB

- Instances of one model reside on the same GPU
  - Scenario 1: Multiple instances can utilize GPU better than single one
  - Scenario 2: Disaggregate prefill and decode phases (*Part 2*)



- Multiple replicas of parameters bring about memory waste
  - Each redundant replica wastes space of ~GB

Underlying support: CUDA Runtime

USTC, CHINA ADSLAB

- Mechanism
  - *IpcMemHandle* from CUDA Runtime
  - Allows different processes to access the same memory
- Usage
  - Owner: Expose an address as a handle

```
handle = cudart.cudaIpcGetMemHandle(data_ptr1)
```

  - User: Read the handle and restore it to memory address

```
data_ptr2 = cudart.cudaIpcOpenMemHandle(handle)
```

Underlying support: CUDA Runtime

USTC, CHINA ADSLAB

- An example
  - Owner

```
data_ptr1 = tensor.data_ptr()

status, handle = cudart.cudaIpcGetMemHandle(data_ptr1)

memory_handle_str1 =
base64.b64encode(handle.reserved).decode('utf-8')

with open(memory_handle_file, 'w') as f:
    f.write(memory_handle_str)
```

```
# llm @ gpu-node in ~/sharemem [11:17:09]
$ python create_handler.py
tensor: tensor([[ 0.9282, -0.1506, -0.9697],
        [ 1.7256,  0.4512, -0.2925],
        [-0.7432,  0.9395,  0.0214]], device='cuda:0', dtype=torch.float16)
data ptr: 0x7f885e200000
```

  - User

```
with open(memory_handle_file, 'r') as f:
    cuda_memory_handle_b64 = f.read()

handle = cudart.cudaIpcMemHandle_t()

handle.reserved = base64.b64decode(cuda_memory_handle_b64)

data_ptr2 =
cudart.cudaIpcOpenMemHandle(handle,cudart.cudaIpcMemLazyEn
ablePeerAccess)

tensor = torch_tensor_module.create_gpu_tensor
(data_ptr2, dims, dtype)
```

```
# llm @ gpu-node in ~/sharemem [11:19:01]
$ python get_handler.py
tensor: tensor([[ 0.9282, -0.1506, -0.9697],
        [ 1.7256,  0.4512, -0.2925],
        [-0.7432,  0.9395,  0.0214]], device='cuda:0', dtype=torch.float16)
data ptr: 0x7fe7c2200000
```

Underlying support: CUDA Runtime

USTC, CHINA ADSLAB

- An example
  - Owner

```
data_ptr1 = tensor.data_ptr()

status, handle = cudart.cudaIpcGetMemHandle(data_ptr1)

memory_handle_str1 =
base64.b64encode(handle.reserved).decode('utf-8’)

with open(memory_handle_file, 'w') as f:
    f.write(memory_handle_str)
```

- User

```
with open(memory_handle_file, 'r') as f:
    cuda_memory_handle_b64 = f.read()

handle = cudart.cudaIpcMemHandle_t()

handle.reserved = base64.b64decode(cuda_memory_handle_b64)

data_ptr2 =
cudart.cudaIpcOpenMemHandle(handle,cudart.cudaIpcMemLazyEn
ablePeerAccess)

tensor = torch_tensor_module.create_gpu_tensor
(data_ptr2, dims, dtype)
```

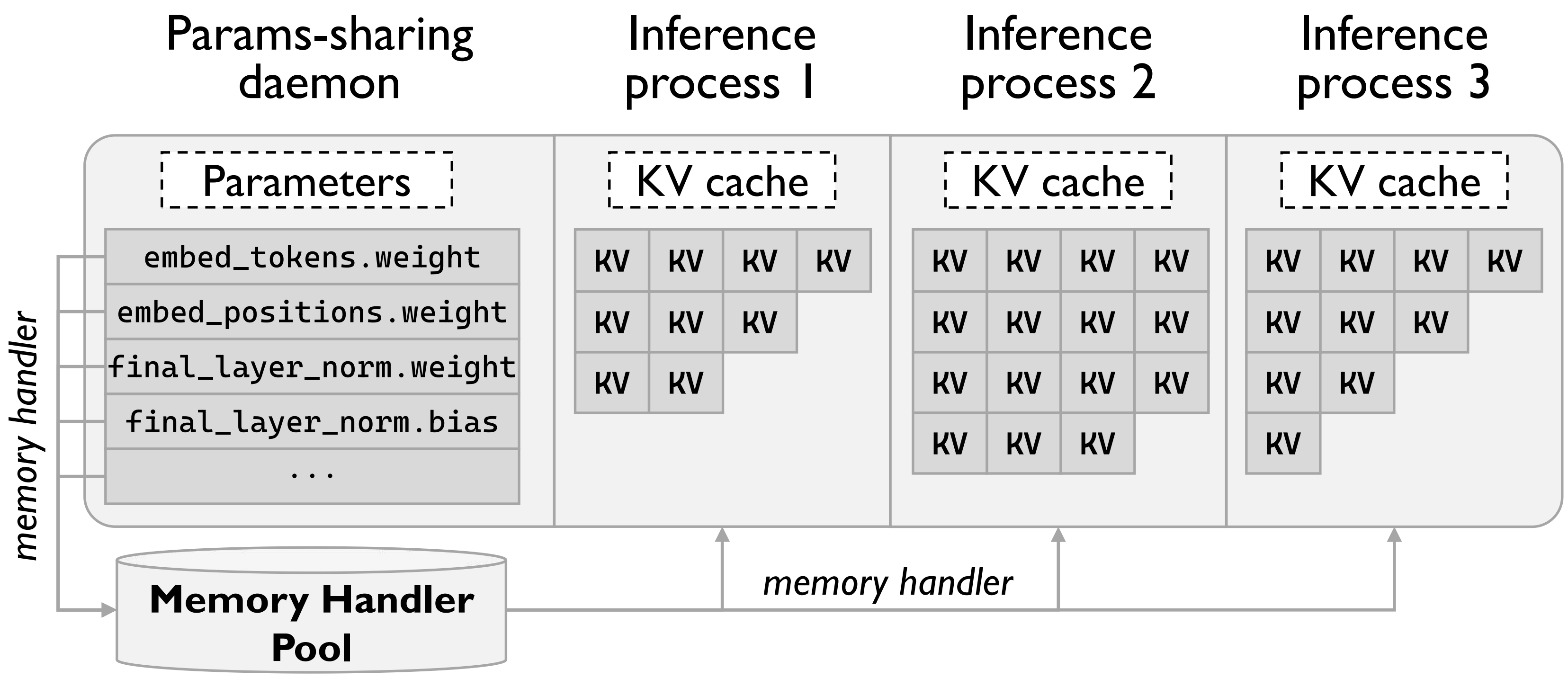
- Python tensor is not fit for pointer
  - Create tensor with C++ and integrate into python as **`torch_tensor_module`**

Part1.2: *Parameter-sharing on vLLM*

Overview

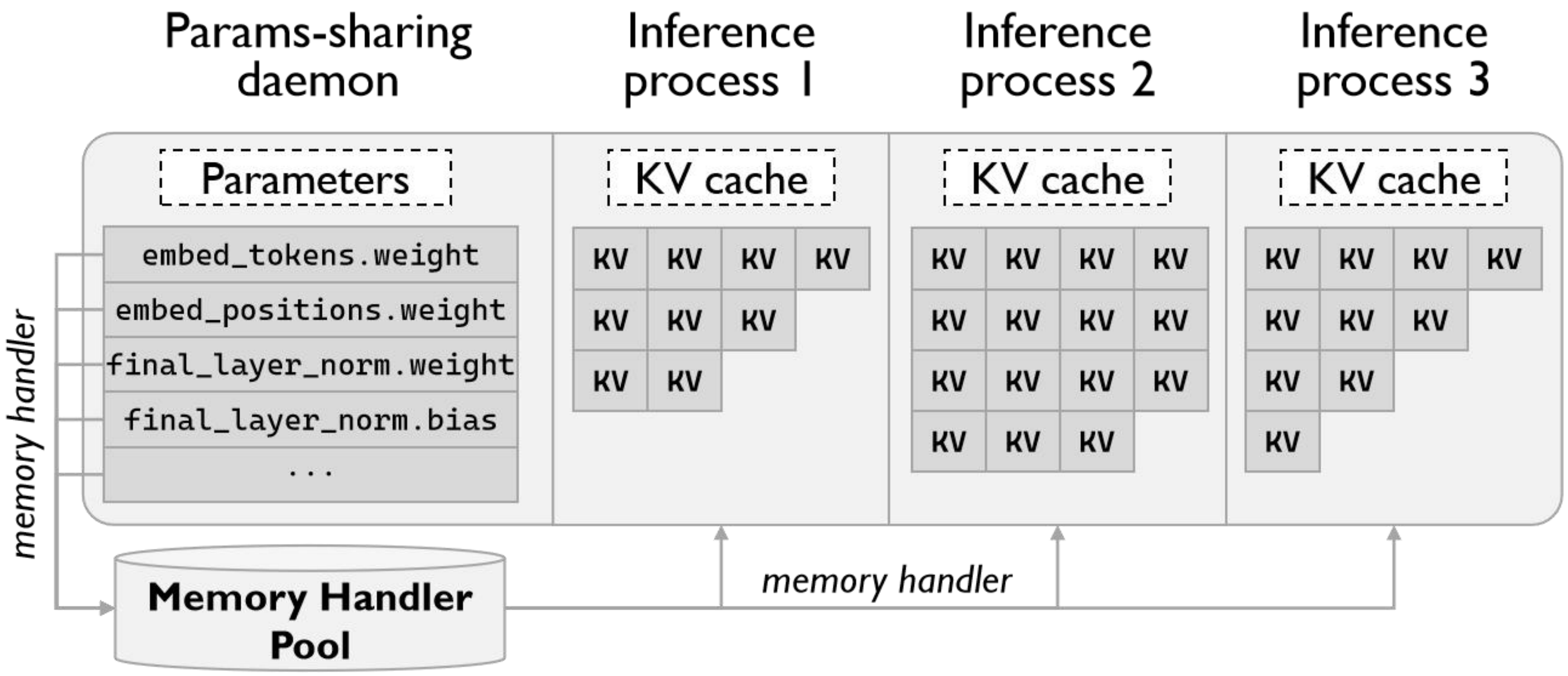
USTC, CHINA ADSLAB

- Params-sharing daemon and Inference processes
- Inference processes share parameters but have a separate kv cache address space



Overview

- ➢ Params-sharing daemon and Inference processes
- ➢ Inference processes share parameters but have a separate kv cache address space



Weights loader of vLLM

USTC, CHINA ADSLAB

➢ Weights initialization

```
#loader.py
_initialize_model(...)
```

```
#linear.py
create_weights(...)
```

```
#vocab_parallel_embed
#ding.py
create_weights(...)
```

```python
def create_weights(...):

    weight =
    Parameter(torch.empty(sum(output_partition_sizes),
    input_size_per_partition,dtype=params_dtype),
    requires_grad=False)
```

➢ Weights loading

```
#loader.py
load_weights(weights)
```

weights

```
#loader.py
_get_all_weights(...)
```

```python
#opt.py
load_weights(weights):
    # get initialized params
    params_dict = dict(self.named_parameters(...))
    # load weights
    for name, loaded_weight in weights:
        weight_loader(...)
```

Weights loader of params-sharing daemon

USTC, CHINA ADSLAB

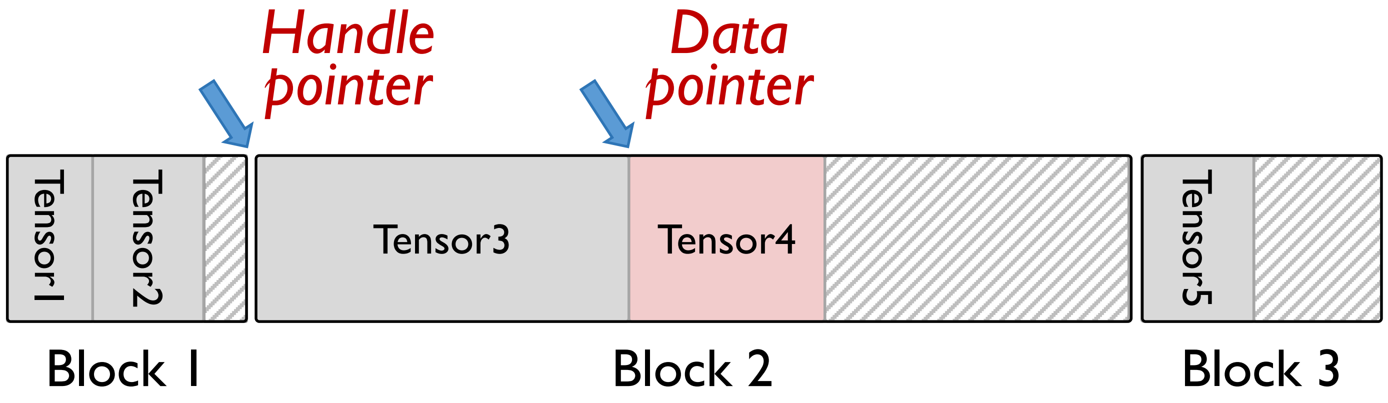
- Save memory handles when loading the weights

```
#opt.py
load_weights(weights):
    # get initialized params
    params_dict = dict(self.named_parameters(...))
    # load weights
    for name, loaded_weight in weights:
        weight_loader(...)
        device_buffer_ptr = params_dict[name].data_ptr()
        err, ipc_mem_handle =
        cudart.cudaIpcGetMemHandle(device_buffer_ptr)
           handler_dict[name] = {
                                 "handler": ipc_mem_handle,
                                 "offset": offset,
                                 "dims": dims,
                                 "dtype": "float16"
                              }
```

PyTorch memory management and allocation

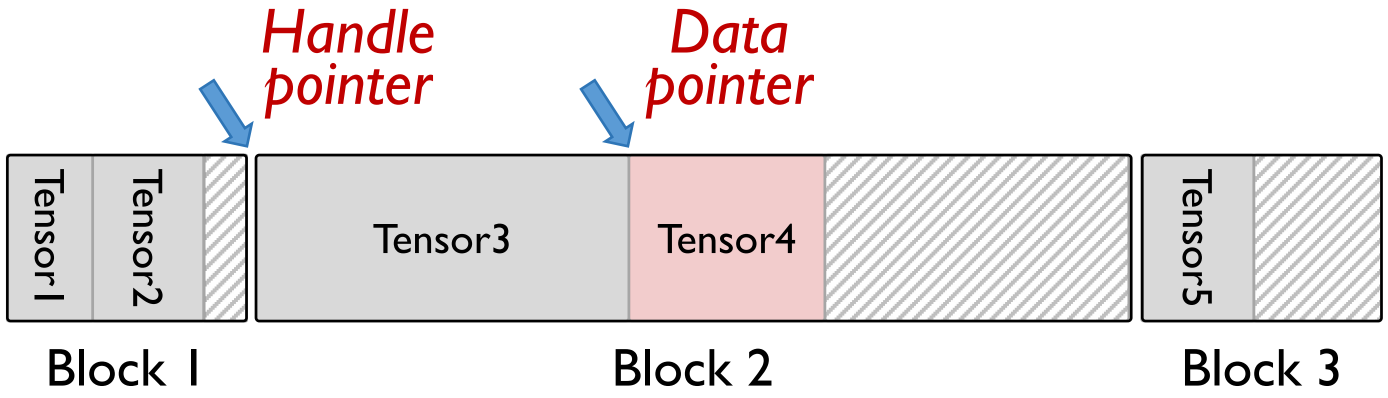
USTC, CHINA ADSLAB

- PyTorch memory management
  - At block granularity
  - `cudart.cudaIpcGetMemHandle(data_ptr)` returns block base address, not data address
- PyTorch memory allocation
  - Allocate 2MB for size less than 1MB;
  - Allocate 20MB for size 1MB ~ 10MB;
  - Allocate { size rounded up to a multiple of 2MB } MB for size >= 10MB



Record offset

- Record the address of the handle: *addr(handle)*
- offset = *addr(data)* - *addr(handle)*



Weights loader of inference processes

USTC, CHINA ADSLAB

- Weights initialization
  - **weight = empty()**

```
#loader.py
_initialize_model(...)
```

```
#linear.py
create_weights(...)
```

```
#vocab_parallel_embed
#ding.py
create_weights(...)
```

```
def create_weights(...):

    weight =
    Parameter(torch.empty(sum(output_partition_sizes),
    input_size_per_partition,dtype=params_dtype),
    requires_grad=False)
```

weight <= empty()

```
#loader.py
_initialize_model(...)
```

```
#linear.py
create_weights(...)
```

```
#vocab_parallel_embed
#ding.py
create_weights(...)
```

```
def create_weights(...):

    weight =Parameter()
```

Weights loader of inference processes

USTC, CHINA ADSLAB

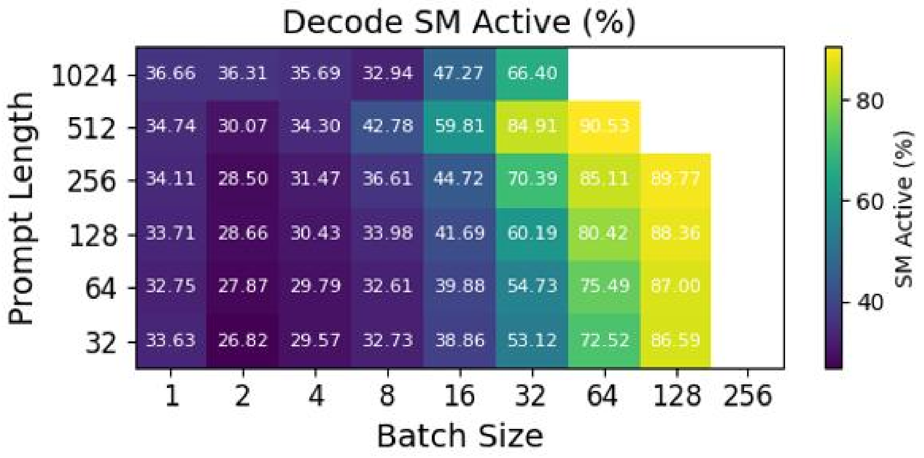
➢ Weights loading: load weights from memory handles

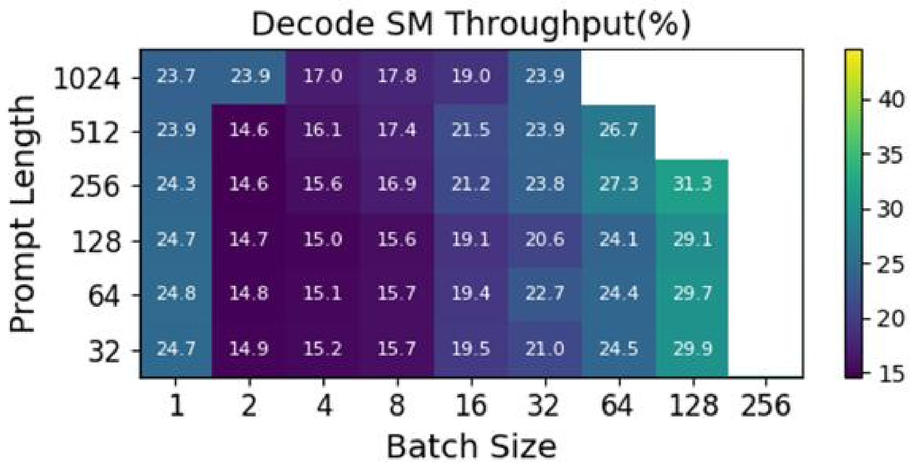
```
#opt.py
load_weights(weights):
    # get initialized params
    params_dict = dict(self.named_parameters(...))
    # load weights
    for name, shared_weight in handles:
        # Gets memory pointers from handles
        err, devPtr =  cudart.cudaIpcOpenMemHandle(
        shared_weight['handler'], cudart.cudaIpcMemLazyEnablePeerAccess)

        # Get weights by memory pointers
        params_dict[name].data =
            torch_tensor_module.create_gpu_tensor(devPtr +
            shared_weight["offset"], shared_weight['dims'],
            shared_weight['dtype'])
```

***Part2*: Prefill and decode phases disaggregating**

Overview

- Why do we need to disaggregate prefill and decode stages?
  - Address low GPU utilization caused by mismatched resource demands
  - Leverage GPU Sharing to further raise resource utilization





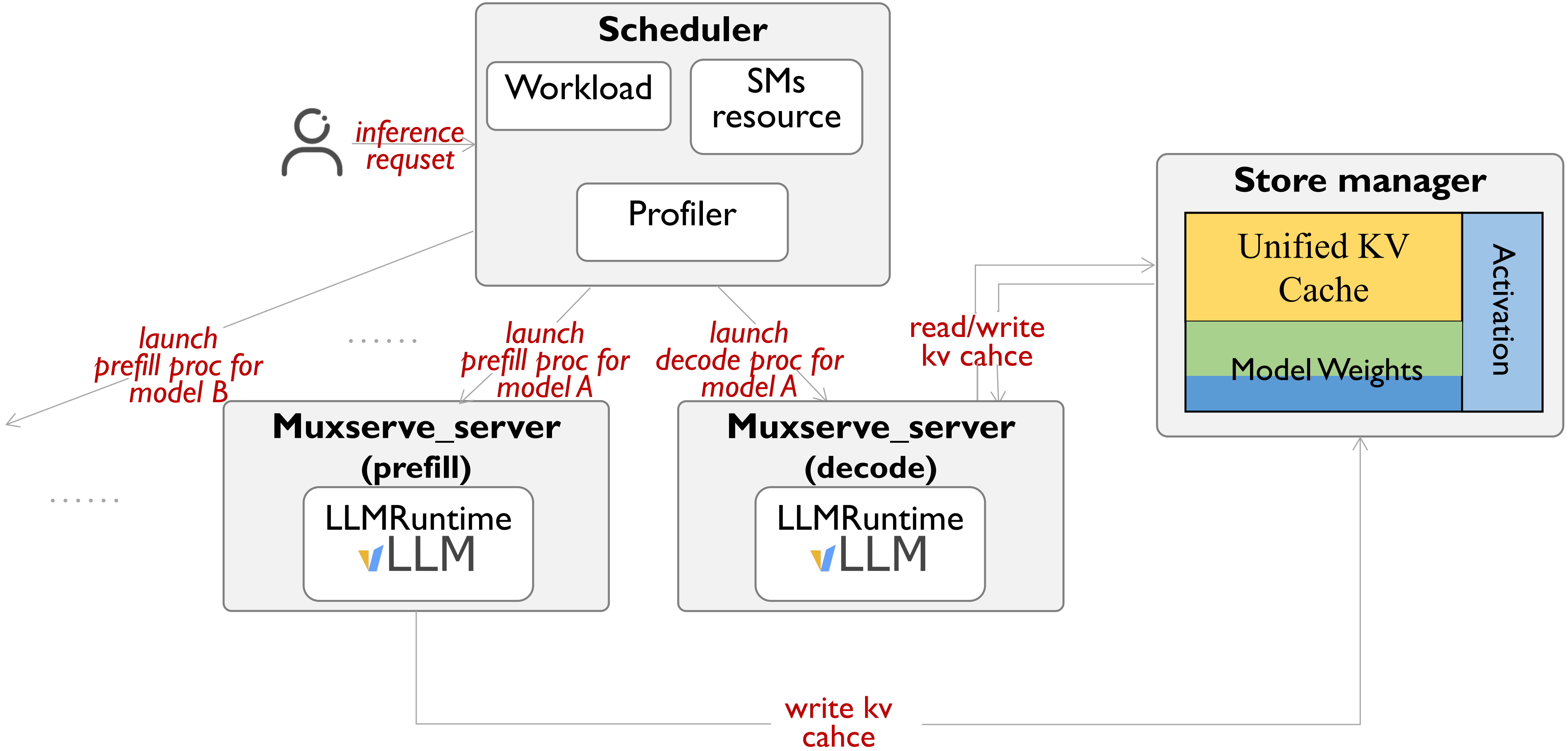
- Challenges
  - How to combine prefill-decode disaggregation with MPS?
  - How to disaggregate prefill and decode stages in vLLM?

[1] DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. OSDI'24
[2] MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. ICML'24

Disaggregate prefill and decode

USTC, CHINA ADSLAB

- Architecture of Muxserve[1]



[1] MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. ICML'24

Disaggregate prefill and decode

USTC, CHINA ADSLAB

- Current idea
  - Launch two subprocesses for prefilling and decoding to utilize MPS for each request → **More flexible dynamic allocation**
  - Do not maintain a unified KV cache, decoding process shares memory with prefilling process or deepcopy → **Fine-grained memory management**
  - Write our own LLMEngine atop vLLM → **To combine with vLLM**
- TODO
  - Implement the schduling algorithm for prefill and decode stages
  - Share KV cache between prefill and decode processes

Planning and Tracking List

USTC, CHINA ADSLAB

Task	Schedule	Task Owner
Implement prefill-decode disaggregated instances	11.26-12.03	Chuanyi
Independent scheduling policy for prefill and decode phases	11.26-12.03	Tao
Implement parameter-sharing on serverlessLLM	11.26-12.03	Zhuoyuan

THANKS


USTC
Thanks for your attention