PagedAttention Understanding - Summary with Load/Store Code

Problem: Memory Fragmentation

python
```
# Traditional approach wastes memory
Traditional:
  Seq1: Allocate 2048 tokens → Uses 256 → Wastes 1792 (87%)
  Seq2: Allocate 2048 tokens → Uses 512 → Wastes 1536 (75%)

PagedAttention:
  Seq1: Allocate 16 blocks (256 tokens) → Wastes 0-15 tokens (<6%)
  Seq2: Allocate 32 blocks (512 tokens) → Wastes 0-15 tokens (<3%)
```

Solution: Block-Based Allocation

python
```
# Blocks are physical memory units
block_size = 16  # tokens per block

# One physical block stores:
# - K and V for 16 tokens
# - For all layers
```

Virtual Memory Concept

python
```
# Logical view (sequence's perspective):
Sequence thinks it has: [Block 0, Block 1, Block 2]

# Physical reality (via block_table):
block_table = [7, 15, 2]
# Logical Block 0 → Physical Block 7
# Logical Block 1 → Physical Block 15
# Logical Block 2 → Physical Block 2

# Non-contiguous! Prevents fragmentation!
```

KV Cache Allocation

```
def allocate_kv_cache(self):
    """
    Allocate the entire KV cache pool at initialization.
    This is PHYSICAL memory that will be shared by all sequences.
    """
    config = self.config
    hf_config = config.hf_config

    # Calculate available GPU memory
    free, total = torch.cuda.mem_get_info()
    used = total - free
    peak = torch.cuda.memory_stats()["allocated_bytes.all.peak"]
    current = torch.cuda.memory_stats()["allocated_bytes.all.current"]

    # Dimensions
    num_kv_heads = hf_config.num_key_value_heads // self.world_size
    head_dim = hf_config.hidden_size // hf_config.num_attention_heads

    # Memory per block (K + V for all layers)
    block_bytes = (
        2                              # K and V
        * hf_config.num_hidden_layers  # All transformer layers
        * self.block_size              # Tokens per block (16)
        * num_kv_heads                 # Number of KV heads
        * head_dim                     # Dimension per head
        * hf_config.torch_dtype.itemsize  # Bytes (2 for fp16)
    )

    # How many blocks can we fit?
    available = total * config.gpu_memory_utilization - used - peak +
current
    num_blocks = int(available // block_bytes)

    assert num_blocks > 0, "Not enough memory for KV cache!"

    # Allocate the ENTIRE cache pool at once
    # Shape: [2, num_layers, num_blocks, block_size, num_heads, head_dim]
    self.kv_cache = torch.empty(
```

```
        2,                              # 0=Keys, 1=Values
        hf_config.num_hidden_layers,  # One cache per layer
        num_blocks,                     # Total physical blocks
        self.block_size,                # Tokens per block (16)
        num_kv_heads,                   # KV heads
        head_dim,                       # Dimension
        dtype=hf_config.torch_dtype,
        device='cuda'
    )

    # Give each attention layer a VIEW into this pool
    layer_id = 0
    for module in self.model.modules():
        if hasattr(module, "k_cache") and hasattr(module, "v_cache"):
            # Each layer gets its slice
            module.k_cache = self.kv_cache[0, layer_id]  # Keys
            module.v_cache = self.kv_cache[1, layer_id]  # Values
            layer_id += 1

    print(f"Allocated {num_blocks} KV cache blocks")
    print(f"Total KV cache memory: {num_blocks * block_bytes / 1e9:.2f}
GB")
```

Calculate Physical Memory Offset

```
def prepare_prefill(self, seqs: list[Sequence]):
    """
    Prepare slot_mapping for prefill phase.
    slot_mapping tells WHERE to write K, V in physical memory.
    """
    slot_mapping = []

    for seq in seqs:
        # For each block this sequence uses
        for i in range(seq.num_cached_blocks, seq.num_blocks):
            # Get physical block ID from block_table
            physical_block_id = seq.block_table[i]
```

```
            # Calculate physical memory start
            start = physical_block_id * self.block_size

            # Calculate end
            if i != seq.num_blocks - 1:
                # Full block
                end = start + self.block_size
            else:
                # Last block (might be partial)
                end = start + seq.last_block_num_tokens

            # Add all slots for this block
            slot_mapping.extend(list(range(start, end)))

    return torch.tensor(slot_mapping, dtype=torch.int32).cuda()


def prepare_decode(self, seqs: list[Sequence]):
    """
    Prepare slot_mapping for decode phase.
    Only one slot per sequence (next token position).
    """
    slot_mapping = []

    for seq in seqs:
        # Calculate where to write the NEXT token's K, V
        physical_block_id = seq.block_table[-1]  # Last block
        offset_in_block = seq.last_block_num_tokens - 1  # Current position

        # Physical offset = block_start + offset_within_block
        slot = physical_block_id * self.block_size + offset_in_block
        slot_mapping.append(slot)

    return torch.tensor(slot_mapping, dtype=torch.int32).cuda()
```

Prefill Phase

```
# Input: New tokens from prompt
q, k, v = compute from hidden_states  # New Q, K, V
```

```
# STORE: Save new K, V to cache
store_kvcache(k, v, k_cache, v_cache, slot_mapping)
# Memory write: k_cache[slots] = k, v_cache[slots] = v

# If prefix cached:
if context.block_tables is not None:
    # LOAD: Replace k, v with cached versions
    k, v = k_cache, v_cache
    # Now k, v point to CACHED values (prefix)

# COMPUTE: Attention with Q and (possibly cached) K, V
o = flash_attn_varlen_func(q, k, v, ...)
```

DECODE Phase:

```
python
# Input: One new token
q, k, v = compute from hidden_states  # New Q, K, V for 1 token

# STORE: Save new K, V to cache
store_kvcache(k, v, k_cache, v_cache, slot_mapping)
# Writes to position: block_table[-1] * block_size + offset

# LOAD & COMPUTE: FlashAttention loads ALL K, V from cache
o = flash_attn_with_kvcache(
    q.unsqueeze(1),      # New Q (1 token)
    k_cache, v_cache,    # ENTIRE cache (read using block_table)
    block_table=...      # Tells FlashAttn which blocks to read
)
# ^^^ Internally loads:
#     K_all = k_cache[block_table] using PagedAttention
#     V_all = v_cache[block_table]
#     Then computes: attention(Q, K_all, V_all)
```
