Understanding the CacheView Class: Key-Value Cache Management for Transformer Models

Code Analysis with Numerical Examples

June 11, 2025

1 Introduction

The `CacheView` class is a sophisticated cache management system designed for efficient key-value storage and retrieval in transformer-based models. This system is particularly crucial for optimizing inference performance by avoiding redundant computations of attention keys and values that have already been processed.

2 Class Overview

The `CacheView` class manages four main components:

- `cache_k`: Cached key tensors
- `cache_v`: Cached value tensors
- `kv_seqlens`: Sequence lengths for each batch element
- `metadata`: Additional metadata for cache operations

3 Detailed Method Analysis

3.1 Constructor (`__init__`)

```
def __init__(
    self,
    cache_k: torch.Tensor,
    cache_v: torch.Tensor,
    metadata: CacheInputMetadata,
    kv_seqlens: torch.Tensor,
):
    self.cache_k = cache_k
    self.cache_v = cache_v
    self.kv_seqlens = kv_seqlens
    self.metadata = metadata
```

The constructor initializes the cache with pre-allocated tensors and metadata.

Numerical Example: Let's assume we have a batch of 2 sequences with the following dimensions:

$$\text{cache\_k} \in \mathbb{R}^{2\times512\times8\times64} \tag{1}$$

$$\text{cache\_v} \in \mathbb{R}^{2\times512\times8\times64} \tag{2}$$

$$\text{kv\_seqlens} = [128, 256] \tag{3}$$

Where:

- Batch size = 2
- Maximum sequence length = 512
- Number of KV heads = 8
- Head dimension = 64
- Current sequence lengths are 128 and 256 tokens respectively

3.2 Update Method

```
def update(self, xk: torch.Tensor, xv: torch.Tensor) -> None:
    """
    to_cache_mask masks the last [max_seq_len] tokens in each sequence
    """
    n_kv_heads, head_dim = self.cache_k.shape[-2:]
    flat_cache_k = self.cache_k.view(-1, n_kv_heads, head_dim)
    flat_cache_v = self.cache_v.view(-1, n_kv_heads, head_dim)

    flat_cache_k.index_copy_(0, self.metadata.cache_positions,
                             xk[self.metadata.to_cache_mask])
    flat_cache_v.index_copy_(0, self.metadata.cache_positions,
                             xv[self.metadata.to_cache_mask])
```

This method updates the cache with new key-value pairs using efficient indexing operations.

Step-by-Step Analysis:

1. **Extract dimensions:**

$$n\_kv\_heads = 8, \quad head\_dim = 64$$

2. **Flatten cache tensors:**

$$\text{flat\_cache\_k} : \mathbb{R}^{2\times512\times8\times64} \to \mathbb{R}^{1024\times8\times64} \tag{4}$$

$$\text{flat\_cache\_v} : \mathbb{R}^{2\times512\times8\times64} \to \mathbb{R}^{1024\times8\times64} \tag{5}$$

3. **Index-based copying:** The `index_copy_` operation efficiently places new key-value pairs at specific positions in the flattened cache.

Numerical Example: Suppose we have new tokens to cache:

$$xk \in \mathbb{R}^{10\times8\times64} \text{ (10 new tokens)} \tag{6}$$

$$\text{to\_cache\_mask} = [True, True, False, True, \ldots] \text{ (mask for 5 tokens)} \tag{7}$$

$$\text{cache\_positions} = [128, 129, 131, 132, 133] \text{ (positions in flat cache)} \tag{8}$$

The operation `flat_cache_k.index_copy_(0, cache_positions, xk[to_cache_mask])` will:

- Take the masked tokens from `xk` (5 tokens)
- Place them at positions [128, 129, 131, 132, 133] in the flattened cache

3.3 Interleave KV Method

```python
def interleave_kv(self, xk: torch.Tensor, xv: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    This is a naive implementation and not optimized for speed.
    """
    assert xk.ndim == xv.ndim == 3  # (B * T, H, D)
    assert xk.shape == xv.shape

    if all([s == 0 for s in self.metadata.seqlens]):
        # No cache to interleave
        return xk, xv

    # Make it a list of [(T, H, D)]
    xk: Tuple[torch.Tensor] = torch.split(xk, self.metadata.seqlens)
    xv: Tuple[torch.Tensor] = torch.split(xv, self.metadata.seqlens)
    assert len(xk) == len(self.kv_seqlens), f"Batch size is {len(self.kv_seqlens)}, got {len(xk)}"

    # Order elements in cache by position by unrotating
    cache_k = [unrotate(t, s) for t, s in zip(self.cache_k, self.kv_seqlens)]
    cache_v = [unrotate(t, s) for t, s in zip(self.cache_v, self.kv_seqlens)]

    interleaved_k = interleave_list(cache_k, list(xk))
    interleaved_v = interleave_list(cache_v, list(xv))

    return torch.cat(interleaved_k, dim=0), torch.cat(interleaved_v, dim=0)
```

This method combines cached key-value pairs with new ones, handling sequence-level operations.

Detailed Process:

1. **Input Validation:** Ensures input tensors are 3-dimensional with shape $(B \times T, H, D)$

2. **Sequence Splitting:**

$$xk \to [(T_1, H, D), (T_2, H, D), \ldots] \tag{9}$$
$$xv \to [(T_1, H, D), (T_2, H, D), \ldots] \tag{10}$$

3. **Cache Unrotation:** The `unrotate` function reorders cached elements by their original positions.

4. **Interleaving:** Combines cached and new key-value pairs in the correct sequential order.

Numerical Example: Consider a batch with 2 sequences:

$$\text{Input: } xk \in \mathbb{R}^{6\times8\times64} \text{ (total 6 new tokens)} \tag{11}$$
$$\text{seqlens} = [3, 3] \text{ (3 tokens per sequence)} \tag{12}$$
$$\text{kv\_seqlens} = [128, 256] \text{ (cached lengths)} \tag{13}$$

After splitting:

$$xk[0] \in \mathbb{R}^{3\times8\times64} \text{ (new tokens for seq 1)} \tag{14}$$
$$xk[1] \in \mathbb{R}^{3\times8\times64} \text{ (new tokens for seq 2)} \tag{15}$$

After interleaving:

$$\text{Final sequence 1: } \mathbb{R}^{(128+3)\times8\times64} = \mathbb{R}^{131\times8\times64} \tag{16}$$
$$\text{Final sequence 2: } \mathbb{R}^{(256+3)\times8\times64} = \mathbb{R}^{259\times8\times64} \tag{17}$$

3.4 Properties

The class provides several useful properties:

```python
@property
def max_seq_len(self) -> int:
    return self.cache_k.shape[1]

@property
def key(self) -> torch.Tensor:
    return self.cache_k[: len(self.kv_seqlens)]

@property
def value(self) -> torch.Tensor:
    return self.cache_v[: len(self.kv_seqlens)]

@property
def prefill(self) -> bool:
    return self.metadata.prefill

@property
def mask(self) -> AttentionBias:
    return self.metadata.mask
```

Property Explanations:

- `max_seq_len`: Returns the maximum sequence length the cache can handle
- `key/value`: Returns only the active portion of the cache (up to current batch size)
- `prefill`: Indicates whether this is a prefill operation
- `mask`: Returns the attention bias/mask for the current operation

4 Use Case and Benefits

4.1 Efficiency Gains

The `CacheView` class provides several efficiency benefits:

1. **Memory Reuse:** Avoids recomputing attention keys and values for previously processed tokens 2. **Batch Processing:** Handles multiple sequences simultaneously 3. **Flexible Updates:** Supports incremental updates without full recomputation

4.2 Typical Usage Pattern

1. **Initialization:** Create cache with pre-allocated tensors
2. **Update:** Add new key-value pairs as tokens are processed
3. **Interleave:** Combine cached and new data for attention computation
4. **Access:** Use properties to get current cache state

5 Mathematical Representation

Let's represent the cache operations mathematically:

5.1 Cache Update Operation

Given new keys $K_{new} \in \mathbb{R}^{T_{new} \times H \times D}$ and cache positions $P \in \mathbb{N}^{T_{cache}}$:

$$C_k[P[i]] \leftarrow K_{new}[M[i]] \quad \forall i \text{ where } M[i] = \text{True}$$

Where M is the `to_cache_mask` and C_k is the flattened cache.

5.2 Interleaving Operation

For sequence s with cached length L_s and new length T_s:

$$K_{final}^{(s)} = \text{concat}(\text{unrotate}(C_k^{(s)}, L_s), K_{new}^{(s)})$$

The final concatenated result across all sequences:

$$K_{batch} = \text{concat}(K_{final}^{(1)}, K_{final}^{(2)}, \ldots, K_{final}^{(B)})$$

6 Conclusion

The `CacheView` class represents a sophisticated approach to managing key-value caches in transformer models. By providing efficient update mechanisms, flexible interleaving operations, and clean property access, it enables significant performance improvements in inference scenarios where maintaining state across multiple forward passes is crucial.

The mathematical operations, while complex, are designed to minimize computational overhead while maintaining the semantic correctness of attention computations. This makes it particularly valuable for applications requiring real-time or near-real-time transformer inference.

7 Comprehensive Real-World Example: Chatbot Inference

Let's walk through a complete real-world scenario where a chatbot is processing multiple conversations simultaneously. This example will demonstrate every aspect of the `CacheView` class with concrete numbers.

7.1 Scenario Setup

Context: A customer service chatbot is handling 3 simultaneous conversations:

- **Conversation 1:** Technical support (currently 15 tokens processed)
- **Conversation 2:** Product inquiry (currently 8 tokens processed)
- **Conversation 3:** Billing question (currently 22 tokens processed)

Model Configuration:

$$\text{Batch size (B)} = 3 \tag{18}$$
$$\text{Max sequence length} = 128 \tag{19}$$
$$\text{Number of KV heads (H)} = 4 \tag{20}$$
$$\text{Head dimension (D)} = 32 \tag{21}$$
$$\text{New tokens per step} = 1 \text{ (autoregressive generation)} \tag{22}$$

7.2 Initial Cache State

The cache tensors are initialized as:

$$\text{cache\_k} \in \mathbb{R}^{3\times128\times4\times32} \tag{23}$$
$$\text{cache\_v} \in \mathbb{R}^{3\times128\times4\times32} \tag{24}$$
$$\text{kv\_seqlens} = [15, 8, 22] \tag{25}$$

The cache structure looks like this:

Sequence	Used Positions	Free Positions	Total
Conv 1	0-14 (15 tokens)	15-127	128
Conv 2	0-7 (8 tokens)	8-127	128
Conv 3	0-21 (22 tokens)	22-127	128

7.3 Step 1: Processing New User Messages

New user inputs arrive:

- **Conv 1:** "What" (token ID: 2841)
- **Conv 2:** "How" (token ID: 1374)
- **Conv 3:** "When" (token ID: 4829)

Input tensors:

$$xk \in \mathbb{R}^{3\times4\times32} \text{ (3 new key vectors)} \tag{26}$$
$$xv \in \mathbb{R}^{3\times4\times32} \text{ (3 new value vectors)} \tag{27}$$

Metadata configuration:

$$\text{to\_cache\_mask} = [True, True, True] \tag{28}$$
$$\text{cache\_positions} = [15, 8, 22] \text{ (next available positions)} \tag{29}$$
$$\text{seqlens} = [1, 1, 1] \text{ (one new token each)} \tag{30}$$

7.4 Step 2: Cache Update Operation

The `update` method processes the new tokens:

Dimension extraction:

$$n\_kv\_heads = 4, \quad head\_dim = 32$$

Cache flattening:

$$\text{flat\_cache\_k} : \mathbb{R}^{3\times128\times4\times32} \to \mathbb{R}^{384\times4\times32} \tag{31}$$
$$\text{flat\_cache\_v} : \mathbb{R}^{3\times128\times4\times32} \to \mathbb{R}^{384\times4\times32} \tag{32}$$

Position mapping in flattened cache:

$$\text{Conv 1, pos 15} \to \text{flat index } 0 \times 128 + 15 = 15 \tag{33}$$
$$\text{Conv 2, pos 8} \to \text{flat index } 1 \times 128 + 8 = 136 \tag{34}$$
$$\text{Conv 3, pos 22} \to \text{flat index } 2 \times 128 + 22 = 278 \tag{35}$$

Index copy operation:

```
# Pseudocode for the actual operation
flat_cache_k[15] = xk[0]     # Conv 1's new key
flat_cache_k[136] = xk[1]    # Conv 2's new key
flat_cache_k[278] = xk[2]    # Conv 3's new key

flat_cache_v[15] = xv[0]     # Conv 1's new value
flat_cache_v[136] = xv[1]    # Conv 2's new value
flat_cache_v[278] = xv[2]    # Conv 3's new value
```

Updated sequence lengths:

$$\text{kv\_seqlens} = [16, 9, 23]$$

7.5 Step 3: Generating Next Tokens

The model now generates responses. For the attention computation, we need to combine cached keys/values with any new computation keys/values.

New computation inputs (for attention):

$$xk_{attn} \in \mathbb{R}^{3\times4\times32} \text{ (current step's keys)} \tag{36}$$
$$xv_{attn} \in \mathbb{R}^{3\times4\times32} \text{ (current step's values)} \tag{37}$$

7.6 Step 4: Interleaving Operation

The `interleave_kv` method combines cached and new data:

Input validation:

$$xk_{attn}.shape = (3, 4, 32) \quad \text{3D tensor} \tag{38}$$
$$xv_{attn}.shape = (3, 4, 32) \quad \text{matching shapes} \tag{39}$$

Sequence splitting: Since each sequence contributes 1 token:

$$xk_{split} = [(1, 4, 32), (1, 4, 32), (1, 4, 32)] \tag{40}$$
$$xv_{split} = [(1, 4, 32), (1, 4, 32), (1, 4, 32)] \tag{41}$$

Cache extraction and unrotation:

$$\text{cache\_k}[0] \in \mathbb{R}^{128\times4\times32} \text{ (Conv 1 cache)} \tag{42}$$
$$\text{cache\_k}[1] \in \mathbb{R}^{128\times4\times32} \text{ (Conv 2 cache)} \tag{43}$$
$$\text{cache\_k}[2] \in \mathbb{R}^{128\times4\times32} \text{ (Conv 3 cache)} \tag{44}$$

After `unrotate` with sequence lengths [16, 9, 23]:

$$\text{unrotated\_cache\_k}[0] \in \mathbb{R}^{16\times4\times32} \text{ (active part)} \tag{45}$$
$$\text{unrotated\_cache\_k}[1] \in \mathbb{R}^{9\times4\times32} \text{ (active part)} \tag{46}$$
$$\text{unrotated\_cache\_k}[2] \in \mathbb{R}^{23\times4\times32} \text{ (active part)} \tag{47}$$

Interleaving process: For each conversation, combine cached tokens with new token:

$$\text{final\_k}[0] = \text{concat}(\text{unrotated\_cache\_k}[0], xk_{split}[0]) \tag{48}$$
$$\in \mathbb{R}^{(16+1)\times 4\times 32} = \mathbb{R}^{17\times 4\times 32} \tag{49}$$
$$\text{final\_k}[1] = \text{concat}(\text{unrotated\_cache\_k}[1], xk_{split}[1]) \tag{50}$$
$$\in \mathbb{R}^{(9+1)\times 4\times 32} = \mathbb{R}^{10\times 4\times 32} \tag{51}$$
$$\text{final\_k}[2] = \text{concat}(\text{unrotated\_cache\_k}[2], xk_{split}[2]) \tag{52}$$
$$\in \mathbb{R}^{(23+1)\times 4\times 32} = \mathbb{R}^{24\times 4\times 32} \tag{53}$$

Final concatenation:

$$K_{final} = \text{concat}(\text{final\_k}[0], \text{final\_k}[1], \text{final\_k}[2]) \tag{54}$$
$$\in \mathbb{R}^{(17+10+24)\times 4\times 32} = \mathbb{R}^{51\times 4\times 32} \tag{55}$$

7.7 Step 5: Attention Computation

The model now has complete key-value pairs for attention:

- **Conv 1:** 17 tokens (16 cached + 1 new)
- **Conv 2:** 10 tokens (9 cached + 1 new)
- **Conv 3:** 24 tokens (23 cached + 1 new)

The attention computation uses these complete sequences while maintaining conversation boundaries through appropriate masking.

7.8 Step 6: Next Iteration

The model generates responses:

- **Conv 1:** "about" (token ID: 1234)
- **Conv 2:** "much" (token ID: 5678)
- **Conv 3:** "will" (token ID: 9012)

These new tokens will be cached at positions [16, 9, 23] respectively, and the process repeats.

Updated state after this iteration:

$$\text{kv\_seqlens} = [17, 10, 24] \tag{56}$$
$$\text{cache\_positions} = [17, 10, 24] \text{ (for next update)} \tag{57}$$

7.9 Performance Analysis

Memory efficiency:

- Without caching: Would need to recompute $17 + 10 + 24 = 51$ token embeddings
- With caching: Only compute 3 new token embeddings
- **Speedup:** $\frac{51}{3} = 17\times$ fewer computations

Memory usage:

$$\text{Cache memory} = 3 \times 128 \times 4 \times 32 \times 2 \text{ (k+v)} \times 4 \text{ bytes (float32)} \tag{58}$$

$$= 196,608 \text{ bytes} = 192 \text{ KB per conversation batch} \tag{59}$$

This demonstrates how the `CacheView` class enables efficient real-time conversation handling by avoiding redundant computations while maintaining perfect attention semantics.

8 Ultra-Detailed Example: Complete Tensor Operations Walkthrough

To fully understand the inner workings of the `CacheView` class, let's examine a comprehensive example that shows actual tensor values, complete mathematical operations, and multiple cache update cycles.

8.1 Initial Setup with Concrete Tensor Values

Scenario: A small-scale transformer processing 2 sequences with the following specifications:

$$\text{Batch size} = 2 \tag{60}$$

$$\text{Max sequence length} = 8 \tag{61}$$

$$\text{Number of KV heads} = 2 \tag{62}$$

$$\text{Head dimension} = 4 \tag{63}$$

$$\text{Initial cached lengths} = [3, 2] \tag{64}$$

Initial cache state:

```
# cache_k shape: (2, 8, 2, 4)
# cache_v shape: (2, 8, 2, 4)
# kv_seqlens: [3, 2]

cache_k = [
    [ # Sequence 0
        [[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]],  # Position 0
        [[1.1, 2.1, 3.1, 4.1], [5.1, 6.1, 7.1, 8.1]],  # Position 1
        [[1.2, 2.2, 3.2, 4.2], [5.2, 6.2, 7.2, 8.2]],  # Position 2
```

```
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 3 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 4 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 5 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 6 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]]   # Position 7 (
    unused)
    ],
    [ # Sequence 1
        [[2.0, 3.0, 4.0, 5.0], [6.0, 7.0, 8.0, 9.0]],  # Position 0
        [[2.1, 3.1, 4.1, 5.1], [6.1, 7.1, 8.1, 9.1]],  # Position 1
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 2 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 3 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 4 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 5 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],  # Position 6 (
    unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]]   # Position 7 (
    unused)
    ]
]
```

8.2 Iteration 1: Adding New Tokens

New input tokens:

- Sequence 0: 2 new tokens
- Sequence 1: 1 new token

Input tensors:

```
# xk shape: (3, 2, 4) - total 3 new tokens across sequences
xk = [
    [[9.0, 8.0, 7.0, 6.0], [5.0, 4.0, 3.0, 2.0]],  # Seq 0, Token 1
    [[9.1, 8.1, 7.1, 6.1], [5.1, 4.1, 3.1, 2.1]],  # Seq 0, Token 2
    [[8.0, 7.0, 6.0, 5.0], [4.0, 3.0, 2.0, 1.0]]   # Seq 1, Token 1
]

# xv shape: (3, 2, 4) - corresponding value vectors
xv = [
    [[0.9, 0.8, 0.7, 0.6], [0.5, 0.4, 0.3, 0.2]],  # Seq 0, Token 1
    [[0.91, 0.81, 0.71, 0.61], [0.51, 0.41, 0.31, 0.21]], # Seq 0, Token 2
    [[0.8, 0.7, 0.6, 0.5], [0.4, 0.3, 0.2, 0.1]]   # Seq 1, Token 1
]
```

Metadata configuration:

```
metadata = CacheInputMetadata(
    to_cache_mask = [True, True, True],      # Cache all 3 tokens
    cache_positions = [3, 4, 2],            # Positions: [seq0_pos3,
   seq0_pos4, seq1_pos2]
    seqlens = [2, 1],                       # New tokens per sequence
    # ... other metadata fields
)
```

8.3 Update Operation Step-by-Step

Step 1: Extract dimensions

```
n_kv_heads, head_dim = cache_k.shape[-2:]  # n_kv_heads=2, head_dim=4
```

Step 2: Flatten cache tensors

```
# Original: (2, 8, 2, 4) -> Flattened: (16, 2, 4)
flat_cache_k = cache_k.view(-1, 2, 4)

# Mapping:
# flat_cache_k[0] = cache_k[0][0]  (seq 0, pos 0)
# flat_cache_k[1] = cache_k[0][1]  (seq 0, pos 1)
# ...
# flat_cache_k[8] = cache_k[1][0]  (seq 1, pos 0)
# flat_cache_k[9] = cache_k[1][1]  (seq 1, pos 1)
# ...
```

Step 3: Index copy operations

```
# cache_positions = [3, 4, 2] maps to flat indices:
# Position 3 in seq 0 -> flat_index = 0*8 + 3 = 3
# Position 4 in seq 0 -> flat_index = 0*8 + 4 = 4
# Position 2 in seq 1 -> flat_index = 1*8 + 2 = 10

# Copy operations:
flat_cache_k[3] = xk[0]  # [[9.0, 8.0, 7.0, 6.0], [5.0, 4.0, 3.0, 2.0]]
flat_cache_k[4] = xk[1]  # [[9.1, 8.1, 7.1, 6.1], [5.1, 4.1, 3.1, 2.1]]
flat_cache_k[10] = xk[2] # [[8.0, 7.0, 6.0, 5.0], [4.0, 3.0, 2.0, 1.0]]

flat_cache_v[3] = xv[0]  # [[0.9, 0.8, 0.7, 0.6], [0.5, 0.4, 0.3, 0.2]]
flat_cache_v[4] = xv[1]  # [[0.91, 0.81, 0.71, 0.61], [0.51, 0.41, 0.31,
   0.21]]
flat_cache_v[10] = xv[2] # [[0.8, 0.7, 0.6, 0.5], [0.4, 0.3, 0.2, 0.1]]
```

Updated cache state:

```
# After update, kv_seqlens = [5, 3]
# Sequence 0 now has 5 tokens (positions 0,1,2,3,4)
# Sequence 1 now has 3 tokens (positions 0,1,2)

updated_cache_k = [
    [ # Sequence 0 - now 5 tokens
        [[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]],        # Position 0
   (old)
        [[1.1, 2.1, 3.1, 4.1], [5.1, 6.1, 7.1, 8.1]],        # Position 1
   (old)
```

```
        [[1.2, 2.2, 3.2, 4.2], [5.2, 6.2, 7.2, 8.2]],        # Position 2 (old)
        [[9.0, 8.0, 7.0, 6.0], [5.0, 4.0, 3.0, 2.0]],        # Position 3 (NEW)
        [[9.1, 8.1, 7.1, 6.1], [5.1, 4.1, 3.1, 2.1]],        # Position 4 (NEW)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],        # Position 5 (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],        # Position 6 (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]]         # Position 7 (unused)
    ],
    [ # Sequence 1 - now 3 tokens
        [[2.0, 3.0, 4.0, 5.0], [6.0, 7.0, 8.0, 9.0]],        # Position 0 (old)
        [[2.1, 3.1, 4.1, 5.1], [6.1, 7.1, 8.1, 9.1]],        # Position 1 (old)
        [[8.0, 7.0, 6.0, 5.0], [4.0, 3.0, 2.0, 1.0]],        # Position 2 (NEW)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],        # Position 3 (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],        # Position 4 (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],        # Position 5 (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],        # Position 6 (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]]         # Position 7 (unused)
    ]
]
```

8.4 Interleaving Operation Detailed Walkthrough

Now let's trace through the interleaving operation for attention computation:

Input for interleaving:

```
# New tokens for current attention computation
xk_attention = [
    [[7.0, 6.0, 5.0, 4.0], [3.0, 2.0, 1.0, 0.0]],  # Seq 0 current token
    [[6.0, 5.0, 4.0, 3.0], [2.0, 1.0, 0.0, -1.0]]  # Seq 1 current token
]

# Metadata for current step
current_seqlens = [1, 1]  # One new token per sequence
```

Step 1: Input validation and splitting

```
# xk_attention.shape = (2, 2, 4)     3D tensor
# Split by sequence lengths [1, 1]:
xk_split = [
    [[[7.0, 6.0, 5.0, 4.0], [3.0, 2.0, 1.0, 0.0]]],  # Seq 0: (1, 2, 4)
    [[[6.0, 5.0, 4.0, 3.0], [2.0, 1.0, 0.0, -1.0]]]  # Seq 1: (1, 2, 4)
```

```
]
```

Step 2: Extract and unrotate cache

```
# Extract active cache portions based on kv_seqlens = [5, 3]
cache_k_active = [
    [ # Sequence 0: 5 tokens
        [[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]],        # Position 0
        [[1.1, 2.1, 3.1, 4.1], [5.1, 6.1, 7.1, 8.1]],        # Position 1
        [[1.2, 2.2, 3.2, 4.2], [5.2, 6.2, 7.2, 8.2]],        # Position 2
        [[9.0, 8.0, 7.0, 6.0], [5.0, 4.0, 3.0, 2.0]],        # Position 3
        [[9.1, 8.1, 7.1, 6.1], [5.1, 4.1, 3.1, 2.1]]         # Position 4
    ],
    [ # Sequence 1: 3 tokens
        [[2.0, 3.0, 4.0, 5.0], [6.0, 7.0, 8.0, 9.0]],        # Position 0
        [[2.1, 3.1, 4.1, 5.1], [6.1, 7.1, 8.1, 9.1]],        # Position 1
        [[8.0, 7.0, 6.0, 5.0], [4.0, 3.0, 2.0, 1.0]]         # Position 2
    ]
]

# unrotate() ensures proper positional ordering (assuming no rotation
    needed here)
unrotated_cache_k = cache_k_active  # Same in this example
```

Step 3: Interleave cached and new tokens

```
# Combine cached tokens with new tokens for each sequence
interleaved_k = [
    [ # Sequence 0: 5 cached + 1 new = 6 total tokens
        [[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]],        # Cached pos
    0
        [[1.1, 2.1, 3.1, 4.1], [5.1, 6.1, 7.1, 8.1]],        # Cached pos
    1
        [[1.2, 2.2, 3.2, 4.2], [5.2, 6.2, 7.2, 8.2]],        # Cached pos
    2
        [[9.0, 8.0, 7.0, 6.0], [5.0, 4.0, 3.0, 2.0]],        # Cached pos
    3
        [[9.1, 8.1, 7.1, 6.1], [5.1, 4.1, 3.1, 2.1]],        # Cached pos
    4
        [[7.0, 6.0, 5.0, 4.0], [3.0, 2.0, 1.0, 0.0]]         # NEW token
     ],
     [ # Sequence 1: 3 cached + 1 new = 4 total tokens
        [[2.0, 3.0, 4.0, 5.0], [6.0, 7.0, 8.0, 9.0]],        # Cached pos
    0
        [[2.1, 3.1, 4.1, 5.1], [6.1, 7.1, 8.1, 9.1]],        # Cached pos
    1
        [[8.0, 7.0, 6.0, 5.0], [4.0, 3.0, 2.0, 1.0]],        # Cached pos
    2
        [[6.0, 5.0, 4.0, 3.0], [2.0, 1.0, 0.0, -1.0]]        # NEW token
     ]
]
```

Step 4: Final concatenation

```
# Concatenate all sequences for batch attention computation
# Final shape: (10, 2, 4) - 10 total tokens across both sequences
final_k = [
```

```
    # Sequence 0 tokens (6 tokens):
    [[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]],
    [[1.1, 2.1, 3.1, 4.1], [5.1, 6.1, 7.1, 8.1]],
    [[1.2, 2.2, 3.2, 4.2], [5.2, 6.2, 7.2, 8.2]],
    [[9.0, 8.0, 7.0, 6.0], [5.0, 4.0, 3.0, 2.0]],
    [[9.1, 8.1, 7.1, 6.1], [5.1, 4.1, 3.1, 2.1]],
    [[7.0, 6.0, 5.0, 4.0], [3.0, 2.0, 1.0, 0.0]],
    # Sequence 1 tokens (4 tokens):
    [[2.0, 3.0, 4.0, 5.0], [6.0, 7.0, 8.0, 9.0]],
    [[2.1, 3.1, 4.1, 5.1], [6.1, 7.1, 8.1, 9.1]],
    [[8.0, 7.0, 6.0, 5.0], [4.0, 3.0, 2.0, 1.0]],
    [[6.0, 5.0, 4.0, 3.0], [2.0, 1.0, 0.0, -1.0]]
]
```

8.5 Iteration 2: Additional Cache Growth

Let's continue with another iteration to show cache evolution:

New input (iteration 2):

```
# Adding 1 token to sequence 0, 2 tokens to sequence 1
xk_iter2 = [
    [[10.0, 11.0, 12.0, 13.0], [14.0, 15.0, 16.0, 17.0]],  # Seq 0, Token 1
    [[20.0, 21.0, 22.0, 23.0], [24.0, 25.0, 26.0, 27.0]],  # Seq 1, Token 1
    [[20.1, 21.1, 22.1, 23.1], [24.1, 25.1, 26.1, 27.1]]   # Seq 1, Token 2
]

# Metadata for iteration 2
metadata_iter2 = CacheInputMetadata(
    to_cache_mask = [True, True],      # Cache both new tokens
    cache_positions = [5, 3],          # Next positions in cache
    seqlens = [1, 2],                  # One token for seq0, 2 tokens for seq1
    prefill = False                    # Generation phase
)
```

Cache state after iteration 2:

```
# Updated kv_seqlens = [6, 5]
final_cache_k = [
    [ # Sequence 0 - now 6 tokens (positions 0-5)
        [[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]],          # Position 0
        [[1.1, 2.1, 3.1, 4.1], [5.1, 6.1, 7.1, 8.1]],          # Position 1
        [[1.2, 2.2, 3.2, 4.2], [5.2, 6.2, 7.2, 8.2]],          # Position 2
        [[9.0, 8.0, 7.0, 6.0], [5.0, 4.0, 3.0, 2.0]],          # Position 3
        [[9.1, 8.1, 7.1, 6.1], [5.1, 4.1, 3.1, 2.1]],          # Position 4
        [[10.0, 11.0, 12.0, 13.0], [14.0, 15.0, 16.0, 17.0]], # Position 5 (NEW)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],          # Position 6 (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]]           # Position 7 (unused)
```

```
    ],
    [ # Sequence 1 - now 5 tokens (positions 0-4)
        [[2.0, 3.0, 4.0, 5.0], [6.0, 7.0, 8.0, 9.0]],          # Position 0
        [[2.1, 3.1, 4.1, 5.1], [6.1, 7.1, 8.1, 9.1]],          # Position 1
        [[8.0, 7.0, 6.0, 5.0], [4.0, 3.0, 2.0, 1.0]],          # Position 2
        [[20.0, 21.0, 22.0, 23.0], [24.0, 25.0, 26.0, 27.0]], # Position 3
    (NEW)
        [[20.1, 21.1, 22.1, 23.1], [24.1, 25.1, 26.1, 27.1]], # Position 4
    (NEW)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],          # Position 5
    (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]],          # Position 6
    (unused)
        [[0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0]]           # Position 7
    (unused)
    ]
]
```

8.6 Performance Metrics for This Example

Cache efficiency analysis:

$$\text{Initial state: } 3 + 2 = 5 \text{ tokens cached} \tag{65}$$

$$\text{After iteration 1: } 5 + 3 = 8 \text{ tokens cached} \tag{66}$$

$$\text{After iteration 2: } 6 + 5 = 11 \text{ tokens cached} \tag{67}$$

$$\text{New computations: } 3 + 3 = 6 \text{ tokens total} \tag{68}$$

$$\text{Computation savings: } \frac{11}{6} = 83\% \text{ fewer calculations} \tag{69}$$

Memory utilization:

$$\text{Allocated memory: } 2 \times 8 \times 2 \times 4 \times 4 = 512 \text{ bytes} \tag{70}$$

$$\text{Used memory: } (6 + 5) \times 2 \times 4 \times 4 = 352 \text{ bytes} \tag{71}$$

$$\text{Utilization rate: } \frac{352}{512} = 68.75\% \tag{72}$$

This ultra-detailed walkthrough demonstrates the precise tensor operations, memory management, and computational efficiency achieved by the `CacheView` class. Every step shows concrete values, making the abstract cache operations tangible and verifiable.

9 Complete Transformer Forward Passes: 3-Pass Deep Dive

To understand how `CacheView` integrates into the complete transformer architecture, let's trace through 3 full forward passes from raw tokens to final outputs, showing every intermediate computation with concrete values.

9.1 Model Configuration and Initial Setup

Transformer specifications:

$$\text{vocab\_size} = 1000 \tag{73}$$
$$\text{d\_model} = 8 \text{ (embedding dimension)} \tag{74}$$
$$\text{n\_heads} = 2 \text{ (attention heads)} \tag{75}$$
$$\text{n\_kv\_heads} = 2 \text{ (key-value heads)} \tag{76}$$
$$\text{head\_dim} = 4 \text{ (per-head dimension)} \tag{77}$$
$$\text{seq\_len\_max} = 6 \tag{78}$$
$$\text{batch\_size} = 2 \tag{79}$$

Layer components:

- **Embedding layer:** Maps tokens to `d_model` dimensional vectors
- **Attention layer:** Multi-head self-attention with `CacheView`
- **Feed-forward layer:** Simple MLP transformation
- **Output projection:** Maps back to vocabulary space

9.2 Pass 1: Initial Processing (Prefill Phase)

Input tokens:

```
# Two sequences with different lengths
input_tokens = [
    [45, 123, 67],      # Sequence 0: "Hello world example"
    [89, 234]           # Sequence 1: "AI system"
]
# Flattened for processing: [45, 123, 67, 89, 234]
# seqlens = [3, 2]
```

Step 1: Token Embedding

```
# Embedding lookup (simplified with concrete values)
embeddings = [
    # Sequence 0 embeddings
    [1.0, 0.5, -0.2, 0.8, 1.2, -0.5, 0.3, 0.9],  # token 45: "Hello"
    [0.2, 1.1, 0.7, -0.3, 0.6, 1.0, -0.8, 0.4],  # token 123: "world"
    [-0.5, 0.8, 1.3, 0.2, -0.7, 0.5, 1.1, -0.2], # token 67: "example"
    # Sequence 1 embeddings
    [0.9, -0.4, 0.6, 1.1, 0.3, -0.9, 0.7, 0.8],  # token 89: "AI"
    [1.1, 0.3, -0.6, 0.4, 0.9, 0.2, -0.4, 1.2]   # token 234: "system"
]
# Shape: (5, 8) - 5 total tokens, 8-dimensional embeddings
```

Step 2: Positional Encoding

```
# Add positional encodings based on absolute positions
pos_encodings = [
    [0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0],    # Position 0
    [0.1, 0.9, 0.1, 0.9, 0.1, 0.9, 0.1, 0.9],    # Position 1
    [0.2, 0.8, 0.2, 0.8, 0.2, 0.8, 0.2, 0.8],    # Position 2
    [0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0],    # Position 0 (seq 1)
    [0.1, 0.9, 0.1, 0.9, 0.1, 0.9, 0.1, 0.9]     # Position 1 (seq 1)
]

# Add positional encodings to embeddings
input_embeddings = embeddings + pos_encodings
final_input = [
    [1.0, 1.5, -0.2, 1.8, 1.2, 0.5, 0.3, 1.9],   # "Hello" + pos_0
    [0.3, 2.0, 0.8, 0.6, 0.7, 1.9, -0.7, 1.3],   # "world" + pos_1
    [-0.3, 1.6, 1.5, 1.0, -0.5, 1.3, 1.3, 0.6],  # "example" + pos_2
    [0.9, 0.6, 0.6, 2.1, 0.3, 0.1, 0.7, 1.8],    # "AI" + pos_0
    [1.2, 1.2, -0.5, 1.3, 1.0, 1.1, -0.3, 2.1]   # "system" + pos_1
]
```

Step 3: Multi-Head Attention - Query, Key, Value Projections

```
# Weight matrices (simplified 8x8 matrices, shown as key elements)
W_q = [
    [0.1, 0.2, ..., 0.8],  # 8x8 query projection matrix
    ...
]
W_k = [
    [0.2, 0.1, ..., 0.7],  # 8x8 key projection matrix
    ...
]
W_v = [
    [0.3, 0.4, ..., 0.6],  # 8x8 value projection matrix
    ...
]

# Project to Q, K, V (showing calculated results)
Q = [
    # Head 0 (dim 0-3), Head 1 (dim 4-7)
    [2.1, 1.5, 0.8, 1.2, 1.8, 0.9, 1.1, 1.4],   # Query for "Hello"
    [1.8, 2.2, 1.1, 0.9, 1.5, 1.7, 0.8, 1.6],   # Query for "world"
    [1.4, 1.9, 1.3, 1.0, 1.2, 1.4, 1.0, 1.1],   # Query for "example"
    [2.0, 1.3, 0.9, 1.5, 1.6, 1.0, 1.3, 1.7],   # Query for "AI"
    [1.7, 1.6, 1.2, 1.1, 1.4, 1.3, 1.1, 1.5]    # Query for "system"
]

K = [
    [1.9, 1.4, 0.7, 1.1, 1.6, 0.8, 1.0, 1.3],   # Key for "Hello"
    [1.6, 2.0, 1.0, 0.8, 1.3, 1.5, 0.7, 1.4],   # Key for "world"
    [1.2, 1.7, 1.1, 0.9, 1.0, 1.2, 0.9, 1.0],   # Key for "example"
    [1.8, 1.2, 0.8, 1.3, 1.4, 0.9, 1.1, 1.5],   # Key for "AI"
    [1.5, 1.4, 1.0, 1.0, 1.2, 1.1, 1.0, 1.3]    # Key for "system"
]

V = [
    [2.2, 1.6, 0.9, 1.3, 1.9, 1.0, 1.2, 1.5],   # Value for "Hello"
```

```
    [1.9, 2.3, 1.2, 1.0, 1.6, 1.8, 0.9, 1.7],   # Value for "world"
    [1.5, 2.0, 1.4, 1.1, 1.3, 1.5, 1.1, 1.2],   # Value for "example"
    [2.1, 1.4, 1.0, 1.6, 1.7, 1.1, 1.4, 1.8],   # Value for "AI"
    [1.8, 1.7, 1.3, 1.2, 1.5, 1.4, 1.2, 1.6]    # Value for "system"
]
```

Step 4: Initialize CacheView (First Pass)

```
# Initialize cache for this layer
cache_k = torch.zeros(2, 6, 2, 4)  # (batch, max_seq, n_kv_heads, head_dim)
cache_v = torch.zeros(2, 6, 2, 4)
kv_seqlens = torch.tensor([0, 0])  # Initially empty

# CacheView setup for prefill
metadata = CacheInputMetadata(
    to_cache_mask = [True, True, True, True, True],  # Cache all tokens
    cache_positions = [0, 1, 2, 0, 1],               # Cache positions
    seqlens = [3, 2],                                # Sequence lengths
    prefill = True
)

cache_view = CacheView(cache_k, cache_v, metadata, kv_seqlens)
```

Step 5: Cache Update

```
# Reshape K, V for caching (split by heads)
K_reshaped = [
    # Sequence 0
    [[1.9, 1.4, 0.7, 1.1], [1.6, 0.8, 1.0, 1.3]],  # "Hello" - head 0, head 1
    [[1.6, 2.0, 1.0, 0.8], [1.3, 1.5, 0.7, 1.4]],  # "world"
    [[1.2, 1.7, 1.1, 0.9], [1.0, 1.2, 0.9, 1.0]],  # "example"
    # Sequence 1
    [[1.8, 1.2, 0.8, 1.3], [1.4, 0.9, 1.1, 1.5]],  # "AI"
    [[1.5, 1.4, 1.0, 1.0], [1.2, 1.1, 1.0, 1.3]]   # "system"
]

# Update cache using CacheView
cache_view.update(K_reshaped, V_reshaped)  # Similar structure for V

# Cache state after update:
# kv_seqlens = [3, 2]
# cache_k[0, 0:3] contains keys for sequence 0
# cache_k[1, 0:2] contains keys for sequence 1
```

Step 6: Attention Computation

```
# Compute attention scores (Q @ K^T)
# For sequence 0 (3x3 attention matrix for each head):
attention_scores_seq0_head0 = [
    # Q_hello @ [K_hello, K_world, K_example]^T
    [2.1*1.9 + 1.5*1.4 + 0.8*0.7 + 1.2*1.1,  # great->Hello
     2.1*1.6 + 1.5*2.0 + 0.8*1.0 + 1.2*0.8,  # great->world
     2.1*1.2 + 1.5*1.7 + 0.8*1.1 + 1.2*0.9], # great->example

    # Q_world @ [K_hello, K_world, K_example]^T
```

```
    [1.8*1.9 + 2.2*1.4 + 1.1*0.7 + 0.9*1.1,  # world->Hello
     1.8*1.6 + 2.2*2.0 + 1.1*1.0 + 0.9*0.8,  # world->world
     1.8*1.2 + 2.2*1.7 + 1.1*1.1 + 0.9*0.9], # world->example

    # Q_example @ [K_hello, K_world, K_example]^T
    [1.4*1.9 + 1.9*1.4 + 1.3*0.7 + 1.0*1.1,  # example->Hello
     1.4*1.6 + 1.9*2.0 + 1.3*1.0 + 1.0*0.8,  # example->world
     1.4*1.2 + 1.9*1.7 + 1.3*1.1 + 1.0*0.9]  # example->example
]
# = [9.15, 8.80, 8.45, 8.93]

# Apply causal mask (lower triangular)
masked_scores_seq0_head0 = [9.15, 8.80, 8.45, 8.93]  # Can attend to all previous
softmax_scores_seq0_head0 = [0.28, 0.24, 0.21, 0.27]       # Normalized attention

# Weighted combination of all cached values
great_output = 0.28*V_hello + 0.24*V_world + 0.21*V_example + 0.27*V_great
```

Step 7: Apply Attention to Values

```
# Weighted sum of values using attention scores
attention_output_seq0_head0 = [
    # Hello output (only attends to itself)
    1.0 * [2.2, 1.6, 0.9, 1.3] = [2.2, 1.6, 0.9, 1.3],

    # World output (0.27 * Hello + 0.73 * world)
    0.27 * [2.2, 1.6, 0.9, 1.3] + 0.73 * [1.9, 2.3, 1.2, 1.0]
    = [0.594, 0.432, 0.243, 0.351] + [1.387, 1.679, 0.876, 0.730]
    = [1.981, 2.111, 1.119, 1.081],

    # Example output (0.15*Hello + 0.52*world + 0.33*example)
    0.15 * [2.2, 1.6, 0.9, 1.3] + 0.52 * [1.9, 2.3, 1.2, 1.0] + 0.33 * [1.5, 2.0, 1.4, 1.1]
    = [0.33, 0.24, 0.135, 0.195] + [0.988, 1.196, 0.624, 0.520] + [0.495, 0.66, 0.462, 0.363]
    = [1.813, 2.096, 1.221, 1.078]
]

# Similar computation for head 1 and sequence 1...
```

Step 8: Final Output Projection

```
# Concatenate heads and project back to d_model
attention_output_concatenated = [
    # Sequence 0
    [2.2, 1.6, 0.9, 1.3, 1.8, 0.9, 1.1, 1.4],     # Hello (head0 + head1)
    [1.981, 2.111, 1.119, 1.081, 1.6, 1.8, 0.9, 1.7], # world
    [1.813, 2.096, 1.221, 1.078, 1.4, 1.6, 1.2, 1.3], # example
    # Sequence 1
    [2.1, 1.4, 1.0, 1.6, 1.7, 1.1, 1.4, 1.8],     # AI
    [1.9, 1.7, 1.2, 1.4, 1.5, 1.3, 1.1, 1.5]      # system
]

# Feed-forward network processing (simplified)
```

```
ff_output = apply_feedforward(attention_output_concatenated)
pass1_output = [
    [2.5, 1.8, 1.1, 1.6, 2.0, 1.2, 1.4, 1.7],     # Processed Hello
    [2.3, 2.4, 1.3, 1.3, 1.8, 2.0, 1.1, 1.9],     # Processed world
    [2.1, 2.3, 1.4, 1.3, 1.6, 1.8, 1.4, 1.5],     # Processed example
    [2.4, 1.6, 1.2, 1.8, 1.9, 1.3, 1.6, 2.0],     # Processed AI
    [2.2, 1.9, 1.4, 1.6, 1.7, 1.5, 1.3, 1.7]      # Processed system
]
```

9.3 Pass 2: Incremental Generation

New tokens to generate:

```
# Generate next tokens for each sequence
new_tokens = [456, 789]  # "great", "works"
seqlens = [1, 1]         # One new token per sequence
# Current kv_seqlens = [3, 2] from previous pass
```

Step 1: Process New Token Embeddings

```
# New embeddings + positional encoding
new_embeddings = [
    [1.3, 0.7, -0.4, 0.9, 1.1, -0.3, 0.5, 1.0],  # "great" + pos_3
    [0.8, 1.2, 0.6, -0.2, 0.7, 1.1, -0.6, 0.9]   # "works" + pos_2
]

# Project to Q, K, V
new_Q = [
    [2.3, 1.7, 1.0, 1.4, 1.9, 1.1, 1.3, 1.6],    # Query for "great"
    [1.9, 1.5, 1.1, 1.2, 1.6, 1.3, 1.0, 1.4]     # Query for "works"
]

new_K = [
    [2.0, 1.5, 0.9, 1.2, 1.7, 1.0, 1.1, 1.4],    # Key for "great"
    [1.7, 1.3, 1.0, 1.1, 1.4, 1.2, 0.9, 1.3]     # Key for "works"
]

new_V = [
    [2.4, 1.8, 1.1, 1.5, 2.0, 1.2, 1.4, 1.7],    # Value for "great"
    [2.0, 1.6, 1.3, 1.3, 1.7, 1.4, 1.1, 1.5]     # Value for "works"
]
```

Step 2: Update Cache with New Tokens

```
# Update metadata for incremental generation
metadata_gen = CacheInputMetadata(
    to_cache_mask = [True, True],      # Cache both new tokens
    cache_positions = [3, 2],          # Next positions in cache
    seqlens = [1, 1],                  # One token each
    prefill = False                    # Generation phase
)

# Update cache using CacheView
new_K_reshaped = [
    [[2.0, 1.5, 0.9, 1.2], [1.7, 1.0, 1.1, 1.4]],  # "great"
```

```
    [[1.7, 1.3, 1.0, 1.1], [1.4, 1.2, 0.9, 1.3]]   # "works"
]

cache_view.update(new_K_reshaped, new_V_reshaped)
# kv_seqlens now = [4, 3]
```

Step 3: Attention with Cached Context

```
# Use interleave_kv to combine cached and new keys/values
K_full, V_full = cache_view.interleave_kv(new_K_reshaped, new_V_reshaped)

# K_full now contains:
# Sequence 0: [K_hello, K_world, K_example, K_great] (4 tokens)
# Sequence 1: [K_AI, K_system, K_works] (3 tokens)

# Compute attention for new queries against full context
attention_scores_great = [
    # Q_great @ [K_hello, K_world, K_example, K_great]^T
    [2.3*1.9 + 1.7*1.4 + 1.0*0.7 + 1.4*1.1,  # great->Hello
     2.3*1.6 + 1.7*2.0 + 1.0*1.0 + 1.4*0.8,  # great->world
     2.3*1.2 + 1.7*1.7 + 1.0*1.1 + 1.4*0.9,  # great->example
     2.3*2.0 + 1.7*1.5 + 1.0*0.9 + 1.4*1.2]  # great->great
]
# = [9.15, 8.80, 8.45, 8.93]

# Apply causal mask and softmax
masked_scores_great = [9.15, 8.80, 8.45, 8.93]  # Can attend to all
   previous
softmax_great = [0.28, 0.24, 0.21, 0.27]       # Normalized attention

# Weighted combination of all cached values
great_output = 0.28*V_hello + 0.24*V_world + 0.21*V_example + 0.27*V_great
```

9.4 Pass 3: Continued Generation

Generate one more token:

```
# Generate final tokens
final_tokens = [512]  # "!" for sequence 0 only
seqlens = [1, 0]      # Only sequence 0 gets new token

# Current kv_seqlens = [4, 3]
# After this pass: kv_seqlens = [5, 3]
```

Complete processing similar to Pass 2, showing cache growth:

```
# Final cache state:
# Sequence 0: [Hello, world, example, great, !] (5 tokens cached)
# Sequence 1: [AI, system, works] (3 tokens cached)

# Cache utilization:
# cache_k shape: (2, 6, 2, 4)
# Used slots: 5 + 3 = 8 out of 12 total slots = 67% utilization

# Total tokens processed: 3 + 2 + 1 + 1 + 1 = 8 tokens
```

```
# Cached tokens available: 5 + 3 = 8 tokens
# New computations needed: Only for new tokens (3 in total across passes)
# Computation savings: 8 cached / 3 new = 267% efficiency gain
```

9.5 Performance Summary Across 3 Passes

Memory efficiency evolution:

$$\text{Pass 1 (prefill): Cache 5 tokens, compute 5 tokens} \tag{80}$$
$$\text{Pass 2 (generation): Use 5 cached + cache 2 new = 7 total} \tag{81}$$
$$\text{Pass 3 (generation): Use 7 cached + cache 1 new = 8 total} \tag{82}$$
$$\tag{83}$$
$$\text{Total computations without cache: } 5 + 7 + 8 = 20 \text{ token-computations} \tag{84}$$
$$\text{Total computations with cache: } 5 + 2 + 1 = 8 \text{ token-computations} \tag{85}$$
$$\text{Efficiency gain: } \frac{20}{8} = 2.5\times \text{ speedup} \tag{86}$$

This comprehensive 3-pass example demonstrates how `CacheView` seamlessly integrates into the complete transformer pipeline, enabling efficient incremental generation while maintaining perfect attention semantics across all computation phases.

10 Visual Understanding: The Core Insight Behind CacheView

The fundamental insight that makes `CacheView` possible is beautifully illustrated in the inference optimization diagram. Let's break down this concept with concrete examples that directly relate to our cache implementation.

10.1 The Key Insight: Reusing Computed Attention Components

The Problem: During autoregressive generation, we repeatedly compute attention for tokens we've already processed.

The Solution: Cache the Key (K) and Value (V) matrices since they don't change for previously processed tokens.

10.2 Detailed Walkthrough: From Theory to CacheView Implementation

Let's trace through the exact scenario shown in the diagram using our `CacheView` implementation:

Setup: Generating token 4 in a sequence that already has tokens 1, 2, and 3.

```
# Initial state: we have processed 3 tokens
existing_tokens = ["The", "cat", "sat"]  # tokens 1, 2, 3
current_seqlens = [3]  # sequence length so far
vocab_size = 50000
d_model = 4096
n_heads = 32
head_dim = 128  # d_model / n_heads
```

Step 1: The Inefficient Approach (Without Caching)

```
# Without caching, we would recompute everything for token 4
all_tokens = ["The", "cat", "sat", "on"]  # including new token 4
seqlen = 4

# Compute Q, K, V for ALL tokens (wasteful!)
Q_all = compute_queries(all_tokens)      # Shape: (4, 4096)
K_all = compute_keys(all_tokens)         # Shape: (4, 4096)
V_all = compute_values(all_tokens)       # Shape: (4, 4096)

# Attention computation
attention_scores = Q_all @ K_all.T      # Shape: (4, 4)
# Only the last row (token 4's attention) is actually needed!
```

Step 2: The CacheView Approach (Efficient)

```
# Initialize CacheView with pre-allocated cache
cache_k = torch.zeros(1, 8, 32, 128)  # (batch=1, max_seq=8, heads=32,
    head_dim=128)
cache_v = torch.zeros(1, 8, 32, 128)
kv_seqlens = torch.tensor([3])         # We have 3 tokens cached

# Previous tokens' K, V are already cached from earlier computations
# cached_k contains: [K_"The", K_"cat", K_"sat"] at positions [0, 1, 2]
# cached_v contains: [V_"The", V_"cat", V_"sat"] at positions [0, 1, 2]

# For token 4 "on", we only compute NEW Q, K, V
new_token = "on"
Q_new = compute_queries([new_token])     # Shape: (1, 4096) - only 1 token
    !
K_new = compute_keys([new_token])        # Shape: (1, 4096) - only 1 token
    !
V_new = compute_values([new_token])      # Shape: (1, 4096) - only 1 token
    !

# Update cache with new K, V
metadata = CacheInputMetadata(
    to_cache_mask=[True],                # Cache the new token
    cache_positions=[3],                 # Store at position 3
    seqlens=[1],                        # 1 new token
    prefill=False                       # Generation phase
)

cache_view = CacheView(cache_k, cache_v, metadata, kv_seqlens)
K_new_reshaped = K_new.view(1, 32, 128)  # Reshape for heads
V_new_reshaped = V_new.view(1, 32, 128)
cache_view.update(K_new_reshaped, V_new_reshaped)
```

Step 3: Efficient Attention Computation

```
# Get complete K, V using interleave_kv
K_complete, V_complete = cache_view.interleave_kv(K_new_reshaped,
    V_new_reshaped)
# K_complete now contains: [K_"The", K_"cat", K_"sat", K_"on"]
# V_complete now contains: [V_"The", V_"cat", V_"sat", V_"on"]

# Compute attention ONLY for the new token's query
Q_new_reshaped = Q_new.view(1, 32, 128)
attention_scores = Q_new_reshaped @ K_complete.transpose(-2, -1)  # Shape:
    (1, 32, 4)

# Apply causal mask (token 4 can attend to tokens 1, 2, 3, 4)
causal_mask = torch.tril(torch.ones(1, 4))  # Lower triangular
masked_scores = attention_scores.masked_fill(causal_mask == 0, float('-inf
    '))

# Softmax and weighted sum
attention_weights = torch.softmax(masked_scores / math.sqrt(128), dim=-1)
output = attention_weights @ V_complete  # Final attention output for
    token 4
```

10.3 Mapping to the Visual Diagram

Let's directly map our implementation to each numbered insight in the diagram:

Insight 1: "We already computed these dot products. Can we cache them?"

```
# In our CacheView implementation:
# - Previous Q @ K^T computations are implicitly "cached" by caching K
# - When we need Q_new @ K_old^T, we retrieve K_old from cache
# - No need to recompute K for tokens 1, 2, 3

# Example: Token 4 attending to token 2
# Instead of recomputing: Q_"on" @ K_"cat"^T
# We use: Q_"on" @ cached_K[1]^T  (where cached_K[1] is K_"cat")
```

Insight 2: "Since the model is causal, we don't care about attention of a token with its successors"

```
# CacheView respects causality automatically
# When generating token 4, the attention computation is:

attention_pattern = [
    # Token 4 can attend to:
    [True,  True,  True,  True],   # tokens 1, 2, 3, 4 (all previous +
    self)
    # Tokens 1, 2, 3 don't need to attend to token 4 (future)
    # We don't compute their attention - they're already finalized!
]

# This is why we only compute Q for the NEW token
# Previous tokens' outputs are already computed and cached
```

Insight 3: "We don't care about these, as we want to predict the next token"

```
# In generation, we only need the output for the LAST token
# Previous tokens' outputs are already computed

# CacheView optimization:
# - Don't recompute attention for tokens 1, 2, 3
# - Only compute attention for token 4
# - Previous hidden states are already available from earlier forward passes
```

Insight 4: "We are only interested in this last row!"

```
# This is exactly what CacheView enables:
# Instead of computing full (4 x 4) attention matrix:
full_attention = Q_all @ K_all.T  # Shape: (4, 4) - wasteful!

# We compute only the last row:
last_row_attention = Q_new @ K_complete.T  # Shape: (1, 4) - efficient!

# The last row is: [Q_"on" @ K_"The"^T, Q_"on" @ K_"cat"^T, Q_"on" @ K_"sat"^T, Q_"on" @ K_"on"^T]
# Where K_"The", K_"cat", K_"sat" come from cache, and K_"on" is newly computed
```

10.4 Concrete Performance Comparison

Without CacheView (Naive Approach):

$$\text{Token 4 generation:} \tag{87}$$
$$\text{Q computation: 4 tokens} \times 4096 \times 4096 = 67M \text{ operations} \tag{88}$$
$$\text{K computation: 4 tokens} \times 4096 \times 4096 = 67M \text{ operations} \tag{89}$$
$$\text{V computation: 4 tokens} \times 4096 \times 4096 = 67M \text{ operations} \tag{90}$$
$$\text{Attention: } 4 \times 4 \times 4096 = 65K \text{ operations} \tag{91}$$
$$\text{Total: } \approx 201M \text{ operations} \tag{92}$$

With CacheView (Optimized):

$$\text{Token 4 generation:} \tag{93}$$
$$\text{Q computation: 1 token} \times 4096 \times 4096 = 17M \text{ operations} \tag{94}$$
$$\text{K computation: 1 token} \times 4096 \times 4096 = 17M \text{ operations} \tag{95}$$
$$\text{V computation: 1 token} \times 4096 \times 4096 = 17M \text{ operations} \tag{96}$$
$$\text{Attention: } 1 \times 4 \times 4096 = 16K \text{ operations} \tag{97}$$
$$\text{Cache operations: } \approx 1K \text{ operations} \tag{98}$$
$$\text{Total: } \approx 51M \text{ operations} \tag{99}$$

Speedup: $\frac{201M}{51M} = 3.9\times$ faster!

10.5 Real-World Example: Chatbot Response Generation

Let's see how this applies to a practical chatbot scenario:

```
# User input: "What is machine learning?"
# Model generates: "Machine learning is a subset of artificial
    intelligence..."

conversation_so_far = [
    "What", "is", "machine", "learning", "?"  # User input (5 tokens)
]

# Without CacheView: For each generated token, recompute everything
generated_tokens = []
for i in range(10):  # Generate 10 response tokens
    # Inefficient: recompute Q, K, V for ALL tokens so far
    all_tokens = conversation_so_far + generated_tokens
    # Cost grows quadratically: O(n  ) where n is sequence length

# With CacheView: Incremental generation
cache_view = initialize_cache_view()

# Cache user input once
cache_user_input(conversation_so_far, cache_view)

# Generate response incrementally
for i in range(10):
    # Efficient: only compute Q, K, V for NEW token
    new_token = generate_next_token(cache_view)
    cache_view.update(new_token)  # Add to cache
    # Cost is constant: O(1) per token

# Result: 10x faster response generation for long conversations!
```

10.6 The Mathematical Beauty

The core insight can be expressed mathematically:

Traditional Approach:

$$\text{Attention}_t = \text{softmax}\left(\frac{Q_{1:t}K_{1:t}^T}{\sqrt{d_k}}\right)V_{1:t}$$

Where we recompute $Q_{1:t}$, $K_{1:t}$, $V_{1:t}$ for every new token t.

CacheView Approach:

$$\text{Attention}_t = \text{softmax}\left(\frac{[Q_t][\text{cached}(K_{1:t-1}), K_t]^T}{\sqrt{d_k}}\right)[\text{cached}(V_{1:t-1}), V_t]$$

Where: - Q_t is computed only for the new token - $K_{1:t-1}$ and $V_{1:t-1}$ are retrieved from cache - Only K_t and V_t are newly computed

This transforms the complexity from $O(t^2)$ to $O(t)$ for sequence generation!

10.7 Conclusion: From Insight to Implementation

The visual diagram captures the essential insight that makes modern transformer inference efficient. `CacheView` is the practical implementation of this insight, providing:

- **Automatic caching** of K and V matrices
- **Efficient retrieval** during attention computation
- **Seamless integration** with existing transformer code
- **Dramatic speedups** for autoregressive generation

The beauty lies in the simplicity: by recognizing that causal attention only needs the "last row" of the attention matrix, we can cache all previous computations and achieve massive efficiency gains without any loss in model quality.