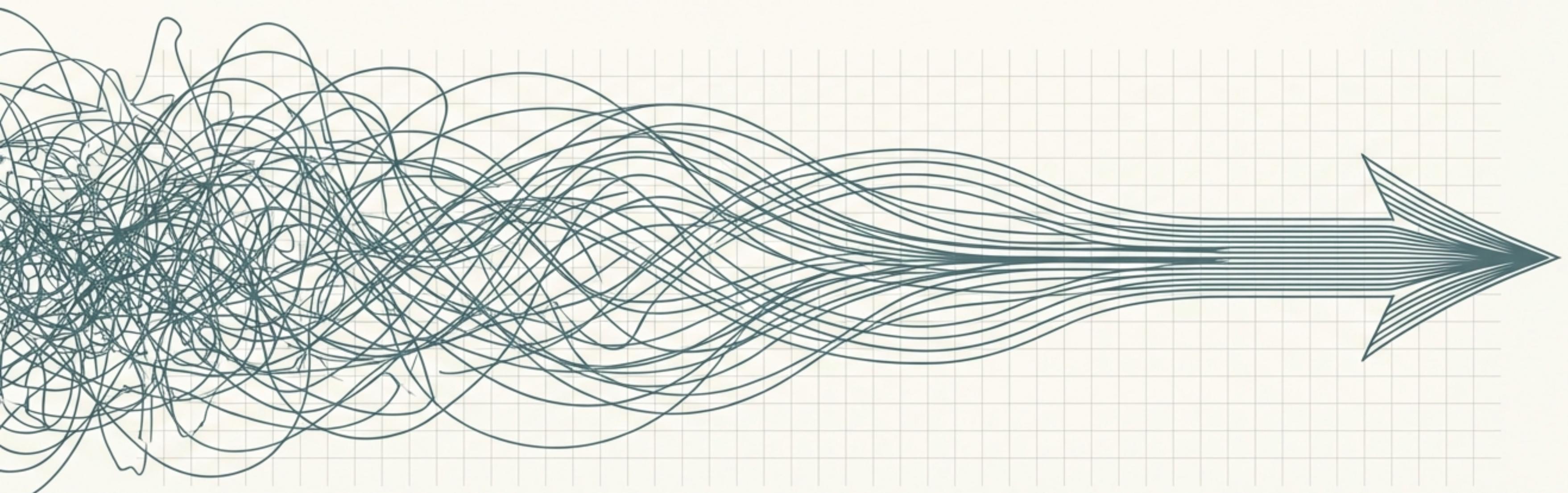


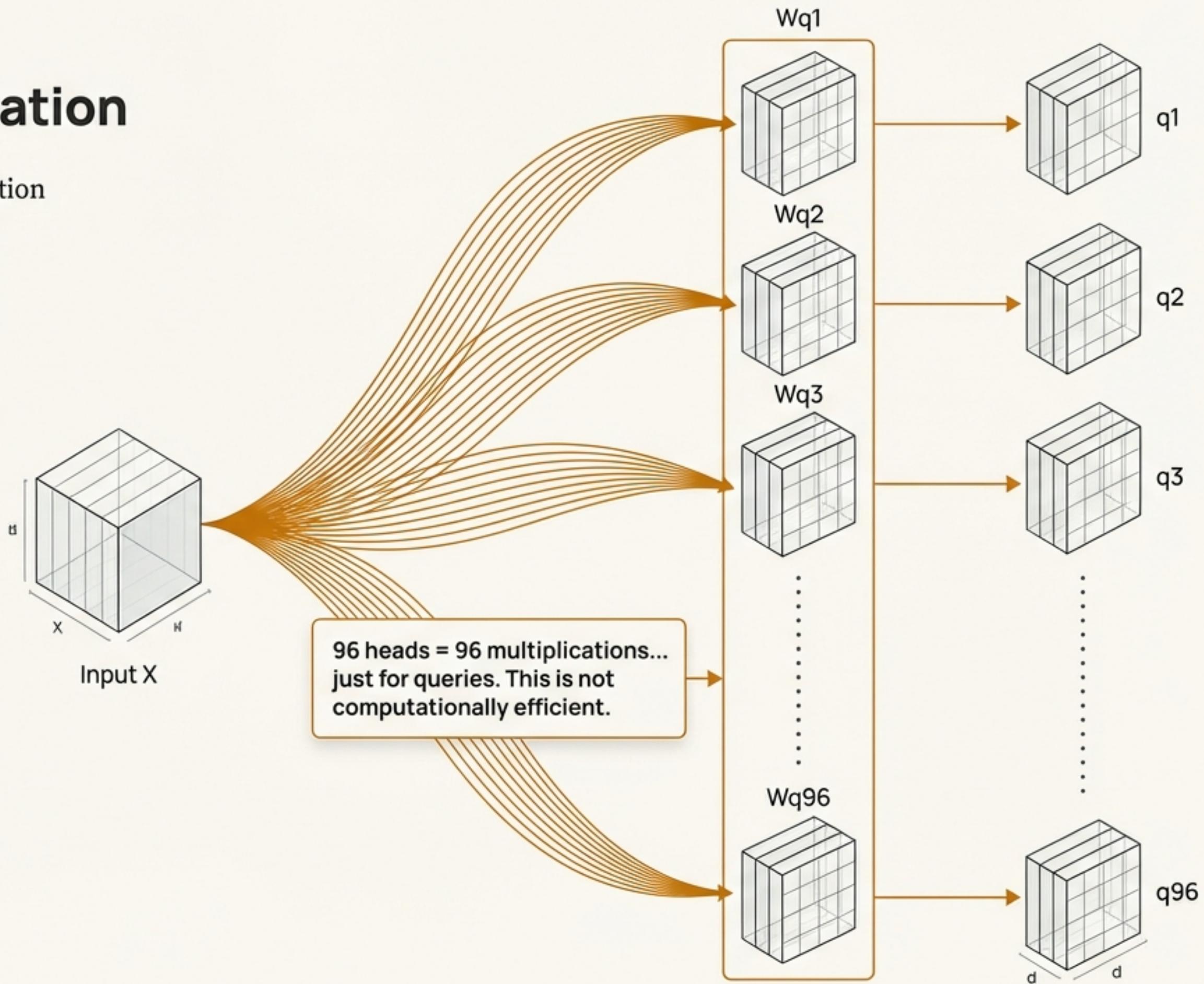
The Architecture of Attention: From Brute Force to Efficiency

A step-by-step deconstruction of the modern, optimized multi-head attention mechanism.



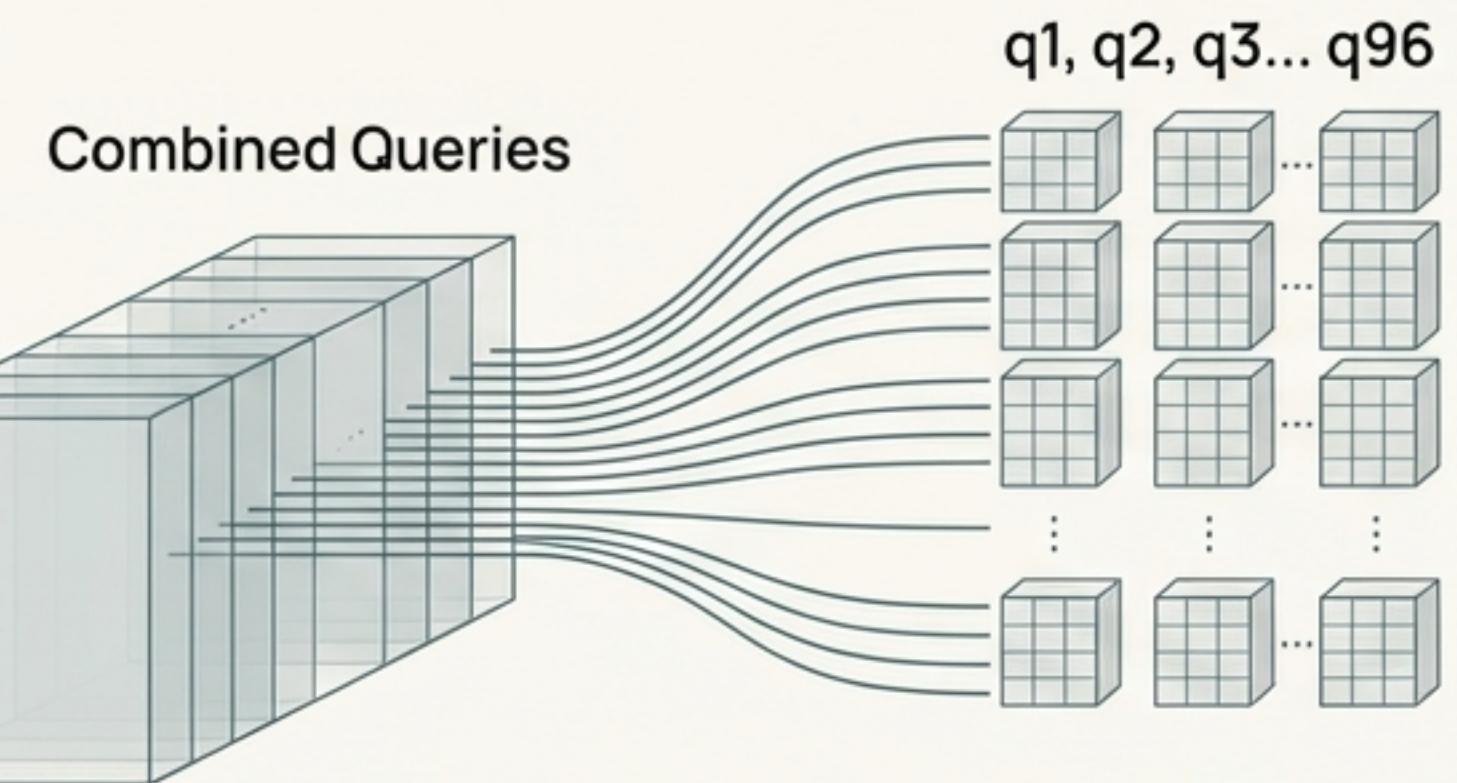
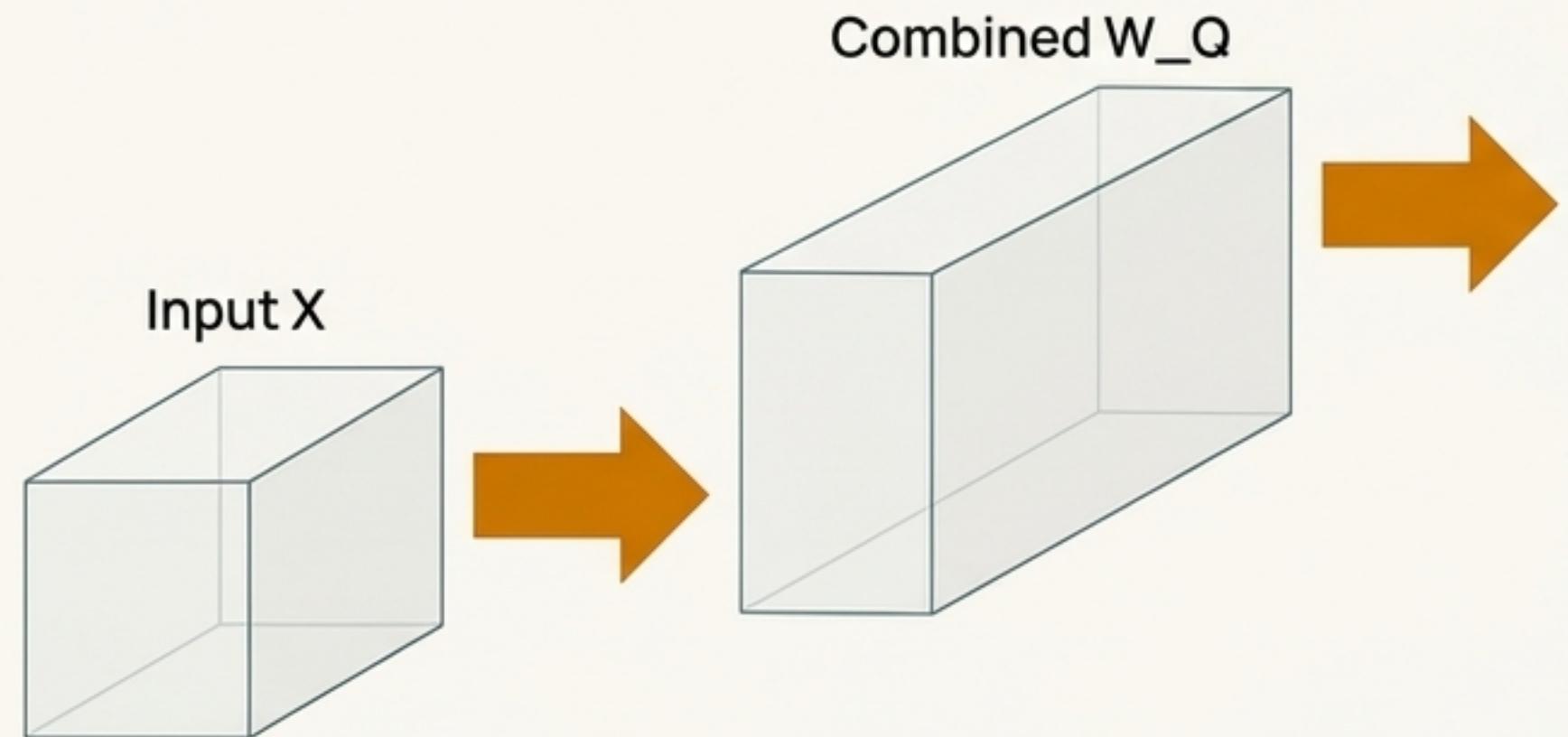
The Scaling Problem: One Head, One Multiplication

The straightforward approach to multi-head attention is to perform a separate matrix multiplication for each attention head. For a model like GPT-3 with 96 heads, this means 96 separate multiplications just to generate the Query vectors, another 96 for the Keys, and another 96 for the Values.



The Elegant Solution: One Multiplication, Many Heads

A far more efficient method performs one large matrix multiplication using a combined weight matrix. The resulting tensor is then split into the required number of heads. This reduces a large number of matrix multiplications into a single one, followed by a simple reshape operation.

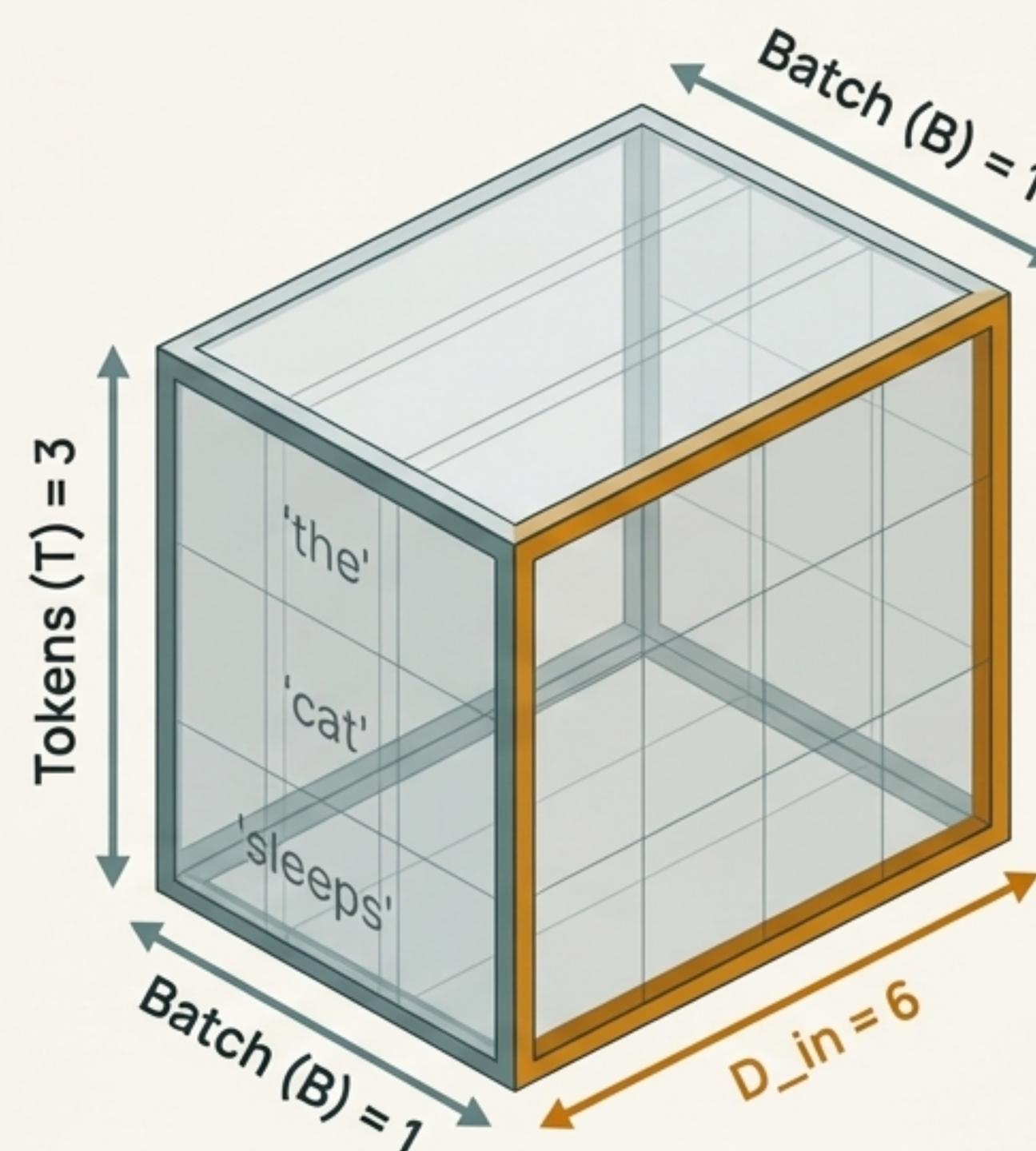


**1 large multiplication
+ 1 reshape operation.**

Our Example: Tracing a 3-Token Sequence

To understand the process, we will follow a **single input tensor** through all 11 steps.

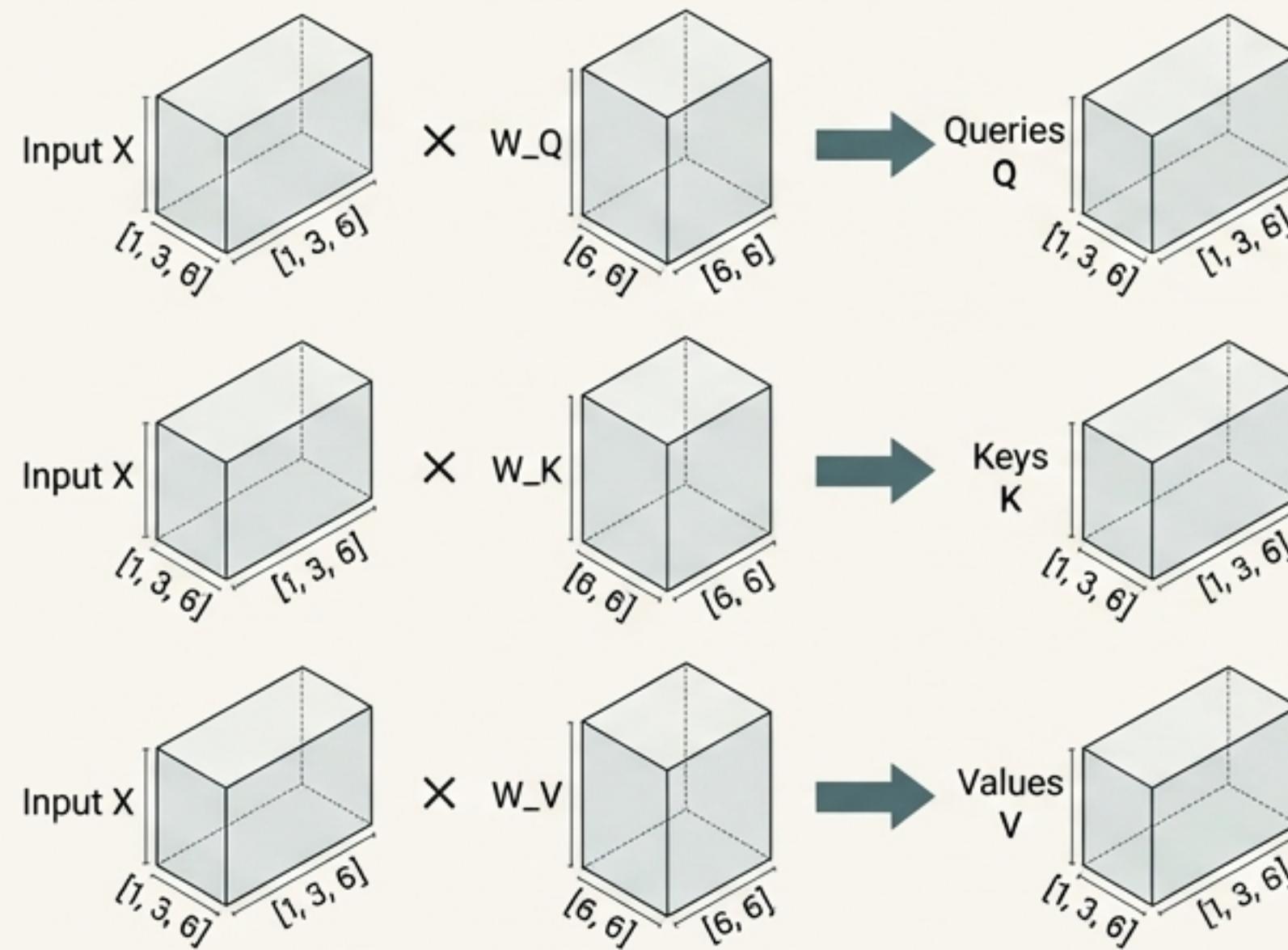
- **Input:** A batch of 1 sentence with 3 tokens (e.g., 'the', 'cat', 'sleeps').
- **Key Parameters:**
 - Output Dimension ('D_out'): 6 (same as 'D_in', common in GPT models)
 - Number of Heads ('num_heads'): 2
 - Head Dimension ('head_dim'): $'D_{out}' / 'num_heads' = 6 / 2 = 3$



Input Shape: [Batch, Tokens, D_in] = [1, 3, 6]

Phase 1: Projection - Creating the Query, Key, and Value Spaces

The input tensor is projected into three distinct representations (Query, Key, and Value) by multiplying it with three trainable weight matrices (W_Q , W_K , W_V).

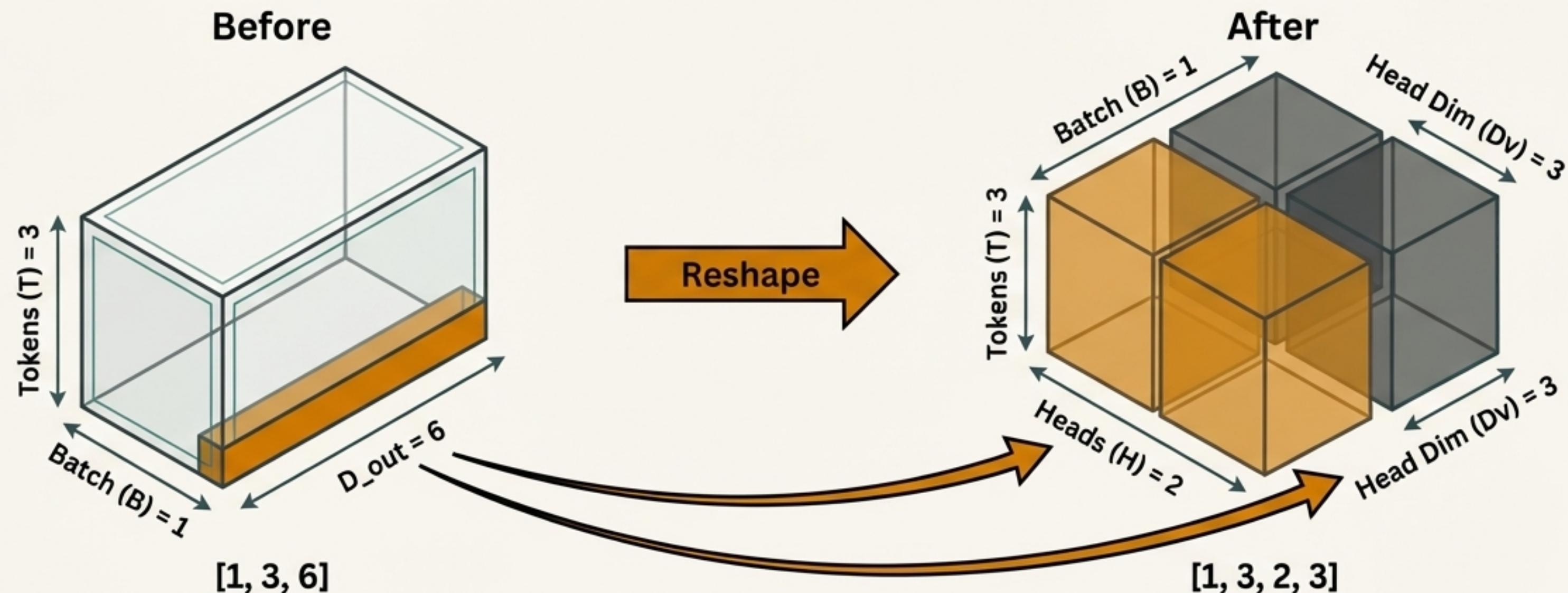


```
# D_in = 6, D_out = 6
self.w_query = nn.Linear(D_in, D_out, bias=False)
self.w_key  = nn.Linear(D_in, D_out, bias=False)
self.w_value = nn.Linear(D_in, D_out, bias=False)

queries = self.w_query(x)      # Shape: [1, 3, 6]
keys = self.w_key(x)           # Shape: [1, 3, 6]
values = self.w_value(x)       # Shape: [1, 3, 6]
```

Phase 2: The Core Trick - Step 5: Reshaping to Reveal the Heads

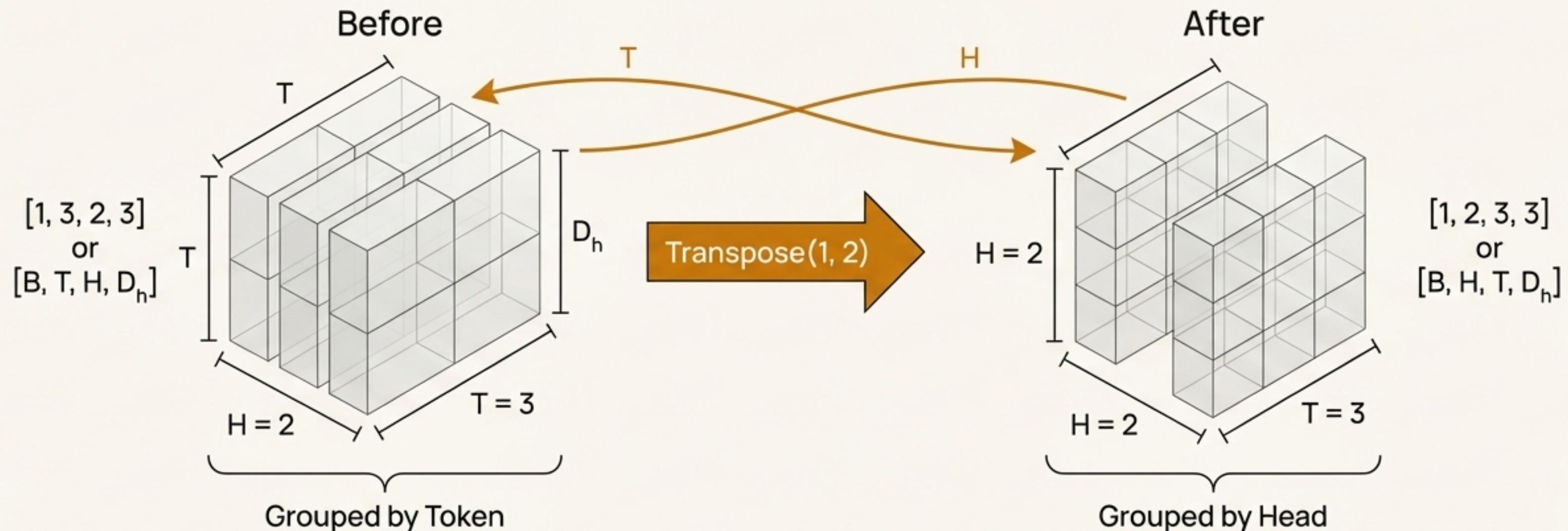
The D_{out} dimension of the Q, K, and V tensors is “unrolled” to explicitly represent the number of heads and the dimension of each head. The 3D tensor becomes a 4D tensor.



```
# B=1, num_tokens=3, num_heads=2, head_dim=3
queries = queries.view(B, num_tokens, self.num_heads, self.head_dim)
# Shape changes from [1, 3, 6] to [1, 3, 2, 3]
```

Phase 2: The Core Trick - Step 6: Transposing for Parallel Computation

To compute attention for all heads simultaneously, we must group the data by head. This is achieved by transposing the 'Tokens' (dim 1) and 'Heads' (dim 2) dimensions.

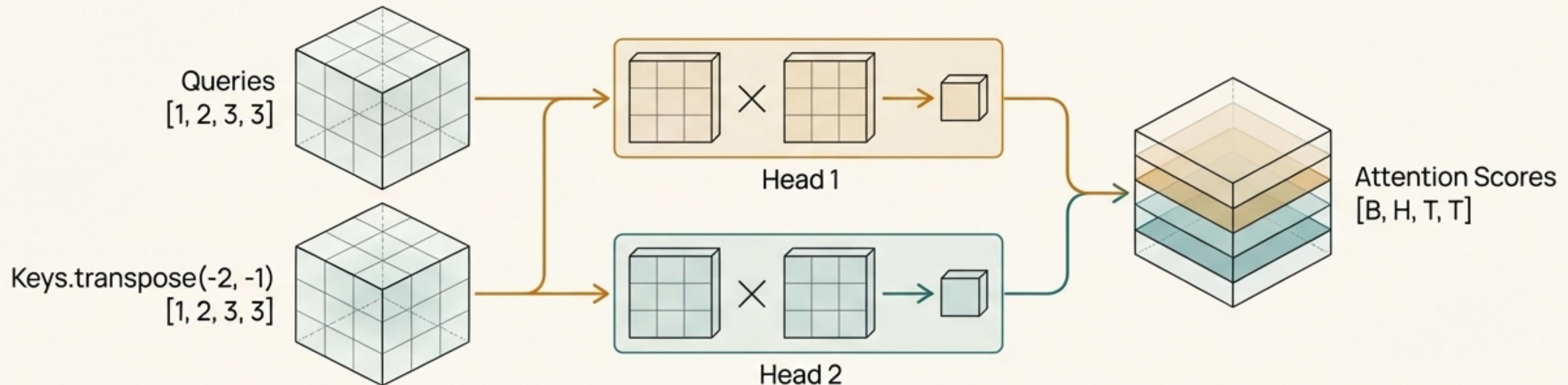


```
queries = queries.transpose(1, 2)
# Shape changes from [1, 3, 2, 3] to [1, 2, 3, 3]
```

Phase 3: Parallel Attention Calculation - Step 7: Calculating Attention Scores

With data grouped by head, we can now compute the dot product between Query and Key vectors for all heads in parallel. The result is an attention score matrix for each head, indicating token-to-token relevance.

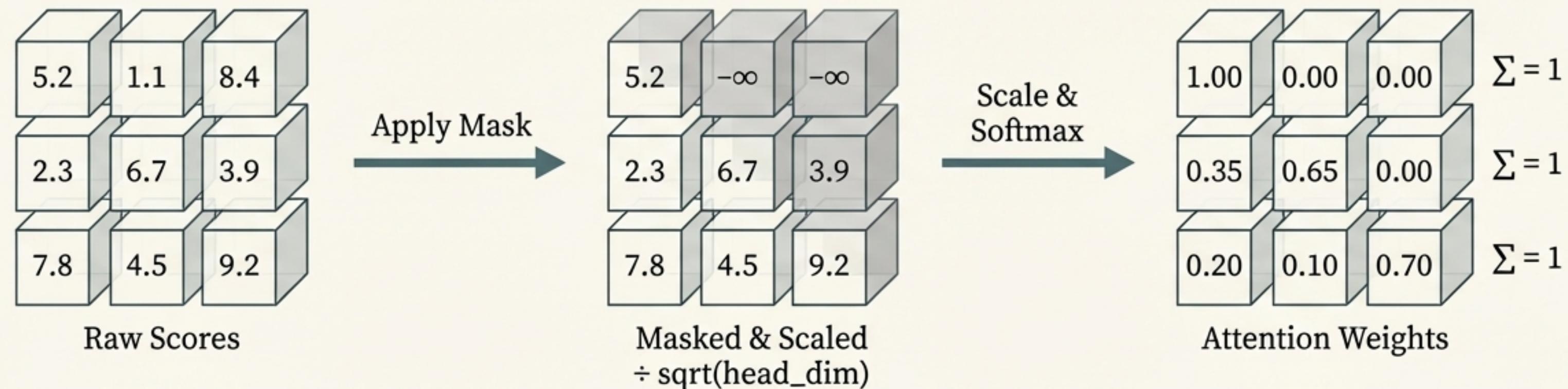
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q} @ \mathbf{K}^T}{\sqrt{d_k}}\right) @ \mathbf{V}$$



```
attn_scores = queries @ keys.transpose(-2, -1)  
# Shape: [1, 2, 3, 3]
```

Phase 3: Parallel Attention Calculation - Steps 8-9: Masking, Scaling, and Normalizing

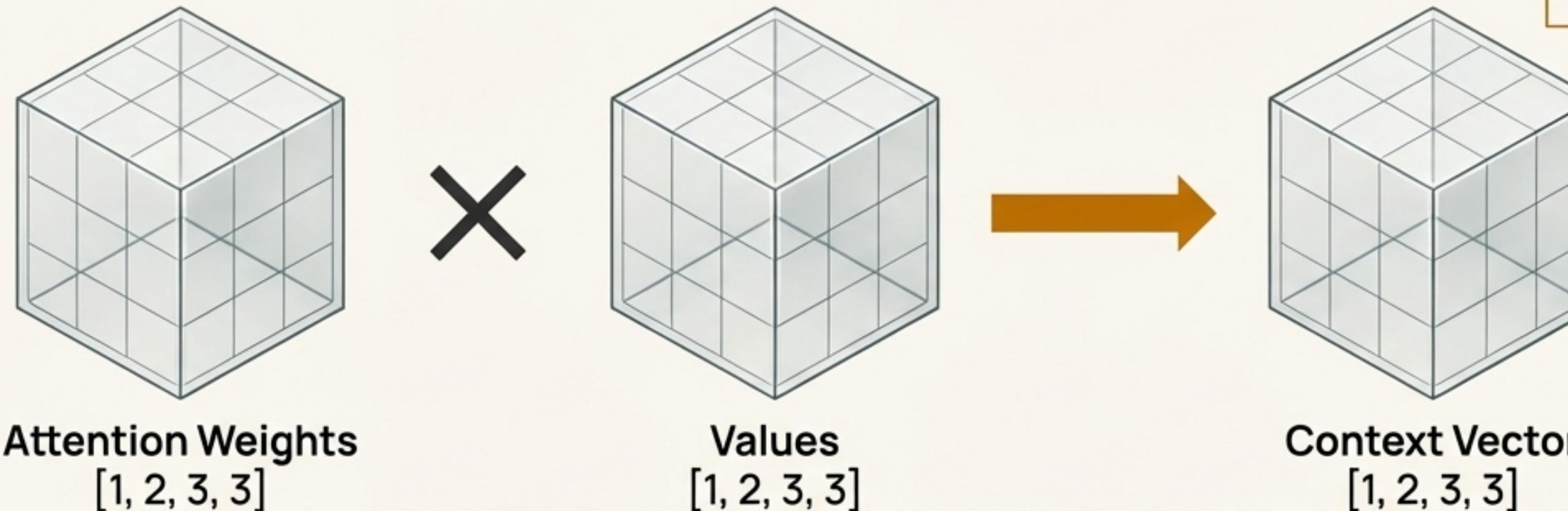
The raw scores are refined. First, a causal mask is applied to prevent tokens from “seeing” future tokens. Then, scores are scaled for stability. Finally, **softmax** converts scores into attention weights, where each row sums to 1.



```
# Apply mask
scores.masked_fill_(mask == 0, float("-inf"))
# Scale and apply softmax
attn_weights = F.softmax(scores / keys.shape[-1]**0.5, dim=-1)
```

Phase 3: Parallel Attention Calculation - Step 10: Computing Weighted Context Vectors

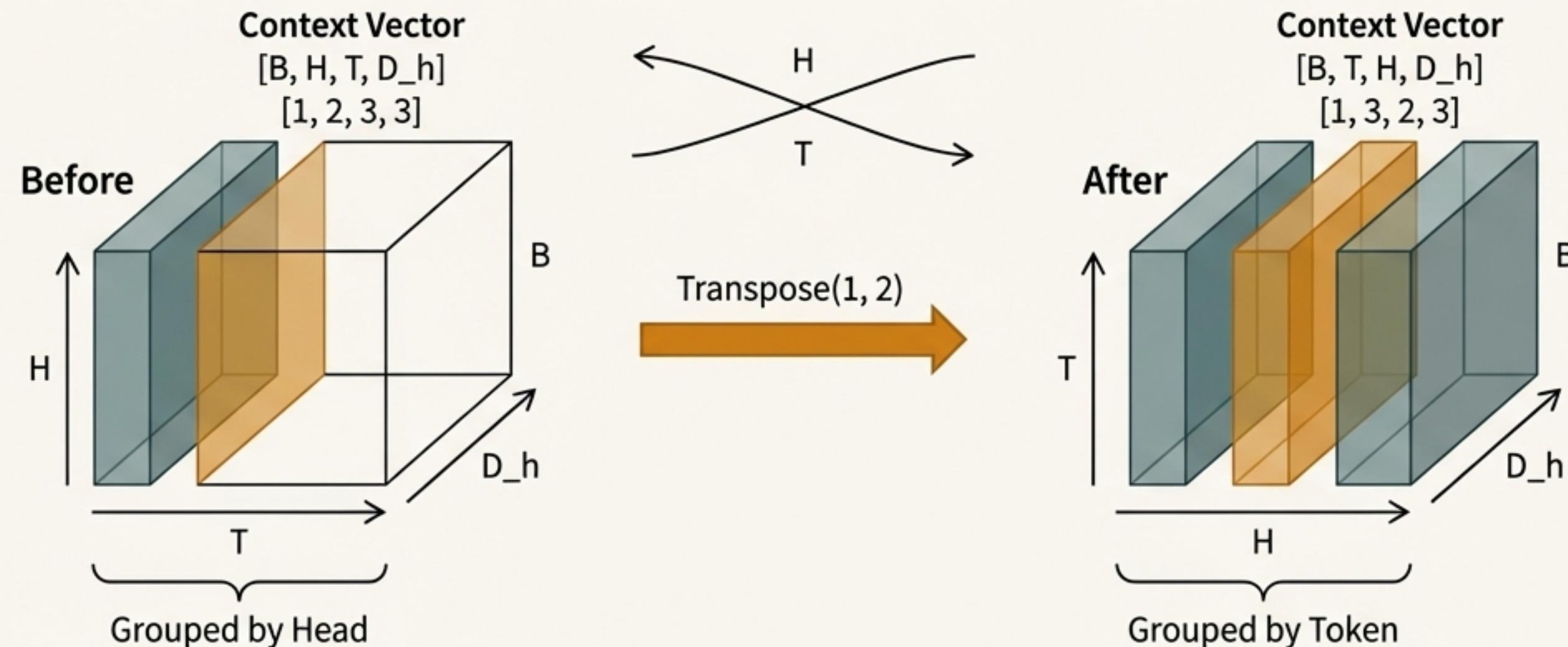
The attention weights are multiplied by the Value tensor. This produces a new context vector for each token, which is a weighted combination of all value vectors in the sequence, encoding the learned relationships.



```
context_vec = attn_weights @ values  
# Shape: [1, 2, 3, 3]
```

Phase 4: Reassembly - Step 11: Reformatting the Context Vectors

To combine the results from all heads, we must first reverse the earlier transpose operation. We swap the Heads and Tokens dimensions back, re-grouping the context vectors by token.

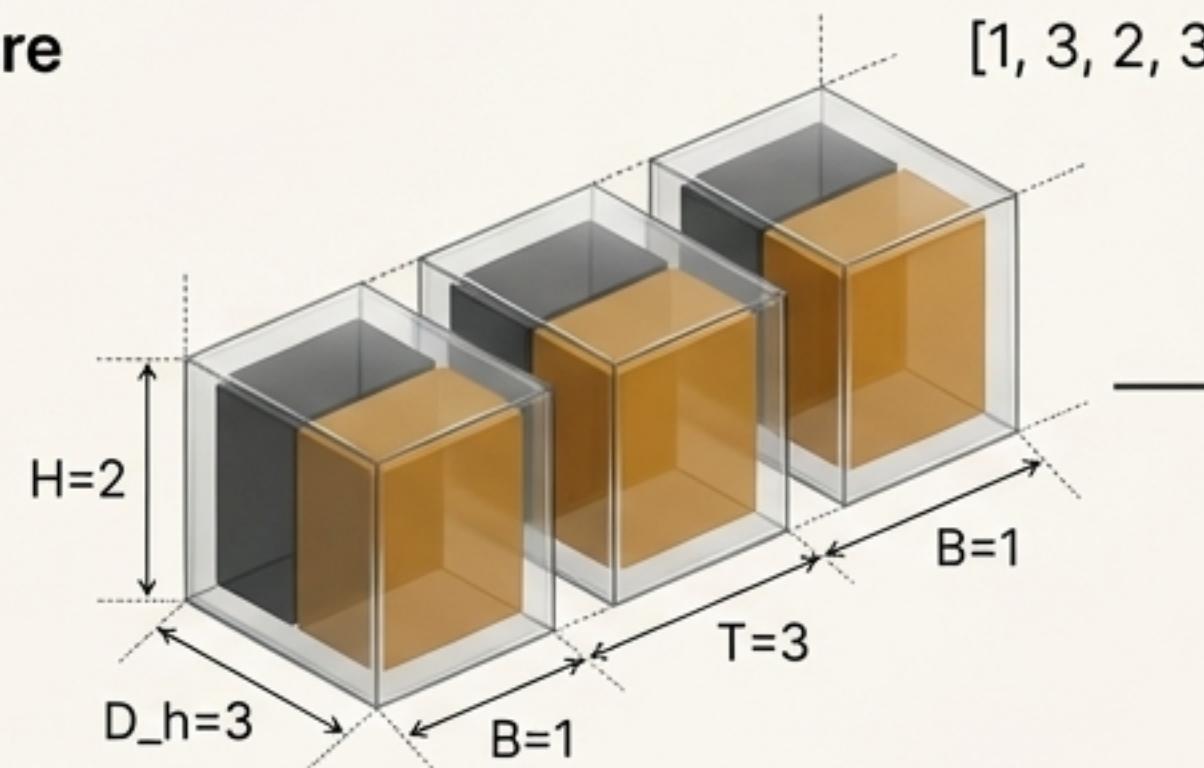


```
context_vec = context_vec.transpose(1, 2)
# Shape changes from [1, 2, 3, 3] to [1, 3, 2, 3]
```

Phase 4: Reassembly - Step 12: Merging the Heads

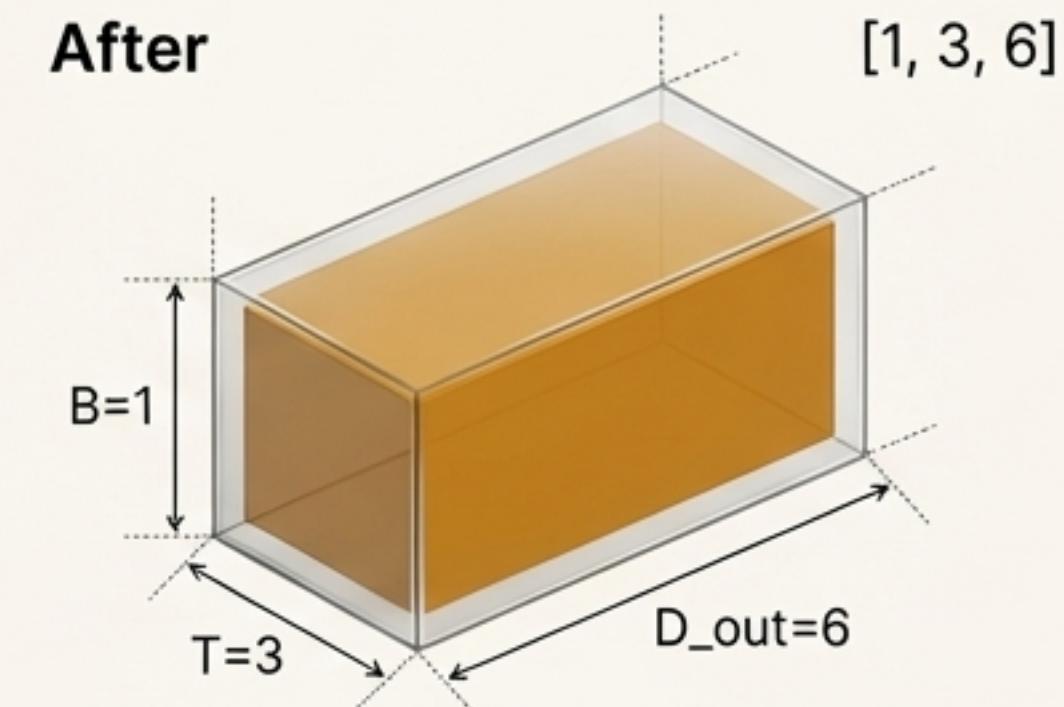
The final step is to merge the results from the individual heads. The `num_heads` and `head_dim` dimensions are flattened back into the single `D_out` dimension, producing the final output tensor.

Before



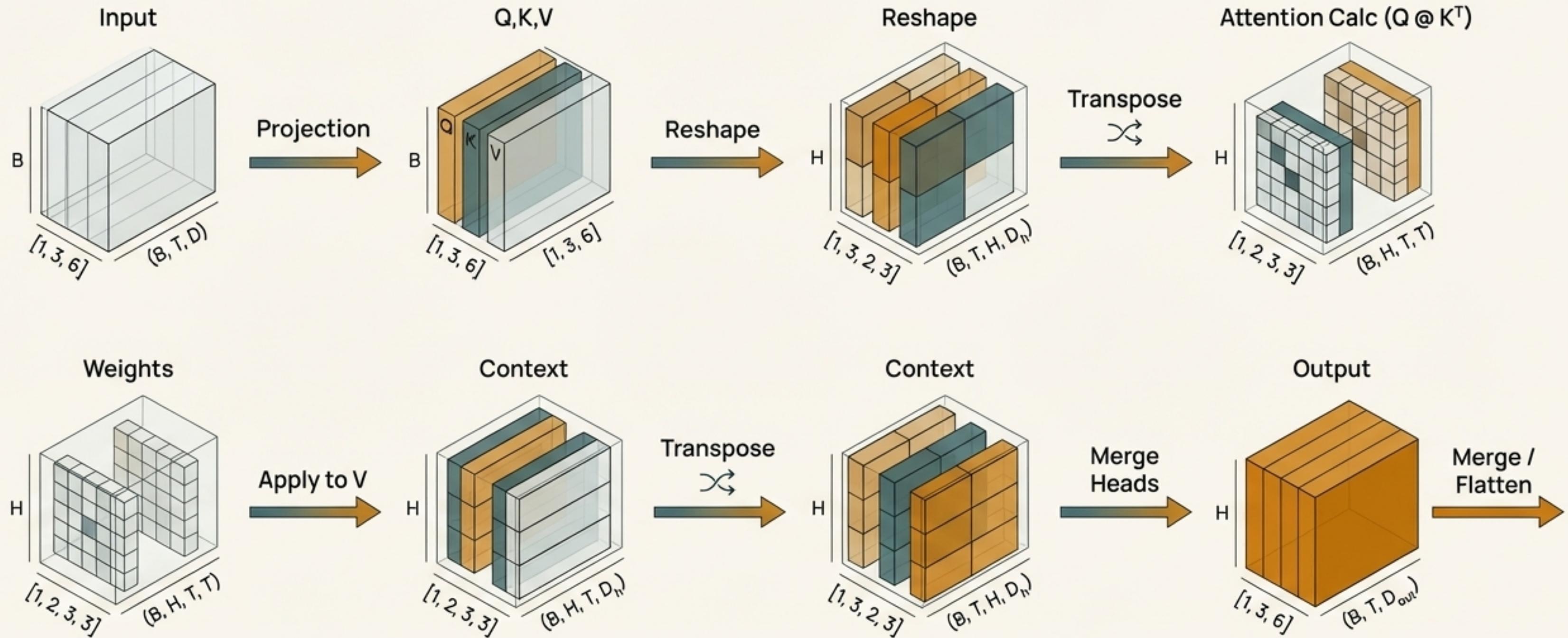
Merge / Flatten

After



```
context_vec = context_vec.contiguous().view(B, num_tokens, self.d_out)  
# Shape changes from [1, 3, 2, 3] to [1, 3, 6]
```

The Full Journey: A Tensor's Transformation



Why This Matters: The Scale of Modern LLMs

This efficient, parallelizable implementation of multi-head attention is not just a theoretical improvement; it is the core engine that makes training and running massive models practical.

GPT-2 (Smallest)

12

Attention Heads

768

Embedding Size (`D_out`)

GPT-2 (Largest)

25

Attention Heads

1600

Embedding Size (`D_out`)

GPT-3 (Largest)

96

Attention Heads

12288

Embedding Size (`D_out`)

The Power of Dimensionality

Mastering multi-head attention is an exercise in mastering dimensions. By strategically reshaping and transposing tensors, we convert a computationally expensive series of operations into an elegant, highly parallelizable architecture.

This fluency with linear algebra and tensor manipulation is at the heart of building state-of-the-art language models.

