Aurora AI — Token / Parameter Specifications & Vector Mechanics (Plain-Text Version)

Below is a fully text-safe, technical, researcher-grade breakdown.

---

1. Tokenization Specification

1.1 Text Normalization

Aurora's tokenizer applies the following normalization steps:

1. Unicode normalization: NFC
2. Case-preserving (no forced lowercase)
3. Whitespace handling: collapse multiple spaces, preserve newlines and tabs
4. Control characters removed except newline and tab
5. Full-width → half-width conversion for CJK text

1.2 Subword Model

Aurora uses a BPE or Unigram LM tokenizer.

- Vocabulary size: typically between 64k and 256k
- Joint text + code training
- Optimized for identifiers, URLs, snake_case, camelCase, JSON keys

Pre-tokenization

Splitting happens on:

- Spaces
- Standard punctuation
- Unicode categories Zs and P*
- Punctuation retained as standalone subword candidates

1.3 Vocabulary Layout (Plain Text IDs)

Recommended layout:

- `0: <pad>`
- `1: <bos>`
- `2: <eos>`
- `3: <unk>`
- `4: <mask>`
- `5–31: reserved system/control tokens`
- `32–V-1: learned subword tokens`

Example special token subset: `{0..31}`

1.4 Sequence Length & Packing

- Maximum context length: L_max (example: 16,384 tokens)
- Input tensor shape: `[batch_size, L_max]`
- Attention mask shape: `[batch_size, L_max, L_max]`
- Sequences may be "packed" using <sep> and block-diagonal attention masks

---

2. Parameter Specification

This section uses **plain-text arithmetic instead of formulas**.

2.1 Definitions

Let:

- V = vocabulary size
- d_model = transformer hidden dimension
- n_layer = number of transformer blocks
- n_head = number of attention heads
- d_head = dimension per head
- d_ff = hidden dimension inside the feedforward network

Relationship:

- `d_model = n_head * d_head`

2.2 Parameter Counts (Plain-Text Descriptions)

2.2.1 Embedding Matrix

Shape: `[V, d_model]`
Parameter count: `V * d_model`

2.2.2 Positional or Rotary Embeddings

- RoPE has negligible learned parameters
- Learned positional embeddings (optional) would have shape: `[L_max, d_model]`

2.2.3 Transformer Block (per layer)

Each block includes:

- LayerNorm
- Multi-head self-attention
- MLP / feedforward network

Attention Parameters Per Layer

Each block contains 4 projection matrices:

- `W_Q: [d_model, d_model]`
- `W_K: [d_model, d_model]`
- `W_V: [d_model, d_model]`
- `W_O: [d_model, d_model]`

Total attention parameters per layer:

- `4 * (d_model * d_model)`

MLP / FFN Parameters Per Layer

Standard 2-layer FFN:

- `W1: [d_model, d_ff]`
- `W2: [d_ff, d_model]`
- `Total = 2 * d_model * d_ff`

Gated FFN (e.g., SwiGLU):

- `Total = 4 * d_model * d_ff`

LayerNorm Parameters

Very small:

- `Each LayerNorm has 2 * d_model parameters`
- `Two LayerNorms per block → approx 4 * d_model per block`

Usually considered negligible at this scale.

2.2.4 Output Projection / LM Head

If **tied** with embeddings:

- No additional parameters.

If untied:

- Same shape as embedding matrix: `[V, d_model]`

2.2.5 Total Parameter Count

Approximate:

- `Total ≈ (V * d_model) + n_layer * (4 * d_model^2 + 4 * d_model * d_ff)`

(Assuming gated MLP and tied embeddings.)

2.3 Example: Aurora-32B Configuration (Plain Text Version)

Example values:

- `V = 128,000`
- `d_model = 6,144`
- `n_layer = 48`
- `n_head = 48`
- `d_head = 128`
- `d_ff = 24,576   (which is 4 * d_model)`
- `L_max = 16,384`
- `MLP = gated (SwiGLU)`

Parameter estimation:

- Embeddings:

  ```
  128,000 * 6,144 ≈ 787 million
  ```
-
- Per-layer attention:

  ```
  4 * (6,144 * 6,144) ≈ 151 million
  ```
-
- Per-layer gated FFN:

  ```
  4 * (6,144 * 24,576) ≈ 604 million
  ```
-
- Total per layer:

  ```
  approx 755 million
  ```
-
- All 48 layers:

  ```
  48 * 755M ≈ 36B parameters
  ```
-

The resulting design is marketed as ~32B due to pruning, weight sharing, and quantization optimizations.

---

3. How Vectors Work in Aurora AI (Plain-Text Version)

This removes all symbolic notation and uses direct prose and pseudo-code.

3.1 Token → Embedding Vectors

Given input token IDs:

- `Input shape: [batch_size, sequence_length]`

Step 1: Look up embeddings:

- `H[batch, position] = embedding_matrix[token_id]`
- `Output shape: [batch_size, sequence_length, d_model]`

Step 2: Apply RoPE inside the attention mechanism, not on embeddings directly.

3.2 Self-Attention (Plain Text Explanation)

For each transformer layer:

1. Project embeddings into Query, Key, Value matrices:

- `Q = H * W_Q`
- `K = H * W_K`
- `V = H * W_V`

Each is reshaped to:

- `[sequence_length, n_head, d_head]`

2. Apply RoPE rotations to Q and K.
3. Compute attention scores:

- `score[t][u][h] = dot(Q[t][h], K[u][h]) / sqrt(d_head)`

Causal rule:

- Positions after t are masked (ignored).

4. Convert scores to attention weights using softmax.
5. Compute attention output:

- `O[t][h] = sum_over_u (attention_weight[t][u][h] * V[u][h])`

6. Concatenate all heads and project through W_O.
7. Add residual connection to form intermediate output.

3.3 MLP / FFN (Plain Text)

Each token's vector goes through:

1. LayerNorm
2. Two linear layers (or three for gated MLP)
3. Activation (Swish for SwiGLU)
4. Residual connection

Equivalent pseudo-code:

- `Z = LayerNorm(H)`

- `U = Z * W1_u`
- `V = Z * W1_v`
- `Gated = Swish(U) elementwise-multiplied by V`
- `M = Gated * W2`
- `H_out = H + M`

3.4 Final Logits (Plain Text)

1. Apply final LayerNorm.
2. Multiply by the transpose of the embedding matrix.
3. Apply softmax to get probabilities for next token.

---

4. External Vector Interfaces (Retrieval, Embeddings, RAG)

4.1 Token Embeddings

Use hidden states from last layer:

- `shape: [batch, sequence_length, d_model]`

4.2 Sequence Embeddings

Common pooling options:

- Take the hidden state at position of <bos> token
- Mean pooling: average all token vectors

- Weighted pooling

Example mean pooling:

- `embedding = average(H[last_layer][tokens])`

4.3 Embedding Projection Head

Optional:

- `projected_embedding = normalize( embedding * W_emb )`

Helps for RAG vector stores.

4.4 Similarity Measurement

Cosine similarity:

- `cosine = dot(a, b) / (norm(a) * norm(b))`

---

5. Vector Steering / Directional Arithmetic

Attributes can be represented as direction vectors by averaging embeddings of examples.

Example:

- `mean_A = average(embeddings from set A)`
- `mean_B = average(embeddings from set B)`
- `direction = mean_A - mean_B`
- `new_vector = z + lambda * direction`

Used experimentally to explore geometry of representation space.

---

6. Numeric Precision

Training:

- `Weights: bf16 or fp16`
- `Activations: bf16, fp16, or fp8`
- `Accumulators: fp32`

Serving:

- `Weights: int8, nf4, or fp16`

`Embeddings returned in fp32 or fp16`

-