Phonex-C2: A Research-Grade Pure C Transformer Engine

From-Scratch GPT-Style Language Modeling with Zero Dependencies

Aditya Mishra

Abstract

This paper presents Phonex-C2, a research-grade implementation of a GPT-style transformer written entirely in pure C (C99) with zero external dependencies. Phonex-C2 implements the complete decoder-only transformer architecture, including multi-head causal self-attention, rotary positional embeddings (RoPE), pre-layer normalization, GELU activations, AdamW optimization, and autoregressive text generation. The system emphasizes educational clarity, architectural minimalism, and hardware-aware performance, employing AVX2 SIMD vectorization, OpenMP parallelism, and a custom arena allocator for memory efficiency. With approximately 500,000 parameters and byte-level tokenization, Phonex-C2 demonstrates end-to-end language modeling capability while remaining fully inspectable and modifiable at the source level. This work serves as both a pedagogical reference and a foundation for low-level experimentation in transformer systems.

1 Introduction

Modern transformer implementations are typically embedded within large software stacks that obscure architectural fundamentals. Popular frameworks such as PyTorch, TensorFlow, and JAX provide high-level abstractions that enable rapid prototyping but create significant barriers to understanding the underlying computational mechanisms. Phonex-C2 addresses this gap by providing a single-file, from-scratch transformer engine that exposes every component (from tensor memory layout to optimizer state updates) without relying on frameworks, runtimes, or external libraries.

The motivation for this work stems from several observations in the machine learning community. First, there exists a growing disconnect between theoretical understanding of transformers and their practical implementation details. Second, debugging and optimization of transformer models often requires understanding low-level computational patterns that are hidden beneath layers of abstraction. Third, educational resources for transformers typically focus on mathematical formulations or high-level API usage, leaving a gap in understanding the translation from mathematics to executable code.

The project follows a staged evolution:

- **Phonex-C0** validated transformer mathematics with a minimal proof-of-concept implementation focusing on correctness of attention mechanisms and feed-forward networks.
- **Phonex-C1** introduced persistence capabilities, training stability improvements, and gradient computation for all components.
- **Phonex-C2** delivers a complete, GPT-style language model with practical generation capability, hardware optimizations, and production-ready memory management.

1.1 Contributions

The primary contributions of this work include:

1. A complete, self-contained implementation of a decoder-only transformer in pure C with zero dependencies.
2. Detailed mathematical exposition of all transformer components with corresponding implementation strategies.
3. A custom memory management system optimized for transformer training workloads.
4. Hardware-aware optimizations using AVX2 SIMD instructions and OpenMP parallelization.
5. Empirical analysis of performance characteristics and memory usage patterns.

2 Background and Related Work

2.1 Transformer Architecture

The transformer architecture, introduced by Vaswani et al., revolutionized sequence modeling by replacing recurrent mechanisms with self-attention. The core innovation lies in the scaled dot-product attention mechanism, which allows each position in a sequence to attend to all other positions.

The original transformer employed an encoder-decoder structure for sequence-to-sequence tasks. However, decoder-only variants (popularized by the GPT series) have demonstrated remarkable success in language modeling by focusing on autoregressive generation.

2.2 Existing Implementations

Several notable transformer implementations exist across the spectrum from educational to production-grade:

High-level frameworks: PyTorch and TensorFlow provide transformer modules that prioritize ease of use and rapid experimentation. These implementations leverage automatic differentiation and GPU acceleration but obscure low-level details.

Optimized libraries: Projects like Megatron-LM and DeepSpeed focus on distributed training and inference optimization for large-scale models, introducing advanced techniques like model parallelism and gradient checkpointing.

Educational implementations: Resources like "The Annotated Transformer" provide well-documented code but still rely on framework abstractions.

Phonex-C2 occupies a unique position by providing complete transparency at the lowest level while maintaining architectural fidelity to modern transformer designs.

3 Model Architecture

3.1 Overview

Phonex-C2 implements a decoder-only transformer with pre-normalization, following the GPT architectural paradigm. The model specifications are:

- **Vocabulary size (V):** 256 (byte-level tokens)

- **Context length (T):** 32 tokens
- **Number of layers (L):** 4 transformer blocks
- **Embedding dimension ($d_{\textbf{model}}$):** 128
- **Number of attention heads (h):** 4
- **Head dimension (d_k):** $d_{\text{model}}/h = 32$
- **FFN hidden dimension ($d_{\textbf{ff}}$):** 512
- **Total parameters:** approximately 500,000

The model processes input sequences through multiple transformer blocks, each applying self-attention and feed-forward transformations with residual connections and layer normalization.

3.2 Input Embedding

Given an input sequence of tokens $\mathbf{x} = (x_1, x_2, \ldots, x_T)$ where $x_i \in \{0, 1, \ldots, V-1\}$, the embedding layer maps each token to a dense vector representation:

$$\mathbf{E} = \text{Embed}(\mathbf{x}) \in \mathbb{R}^{T \times d_{\text{model}}} \tag{1}$$

where the embedding matrix $\mathbf{W}_{\text{emb}} \in \mathbb{R}^{V \times d_{\text{model}}}$ contains learnable parameters. Each row $\mathbf{W}_{\text{emb}}[i]$ represents the embedding vector for token i.

3.3 Transformer Block Architecture

Each transformer block implements the following computation sequence:

$$\begin{align}
\mathbf{H}_1 &= \text{LayerNorm}(\mathbf{X}) \tag{2}\\
\mathbf{A} &= \text{MultiHeadAttention}(\mathbf{H}_1) \tag{3}\\
\mathbf{X}_1 &= \mathbf{X} + \mathbf{A} \tag{4}\\
\mathbf{H}_2 &= \text{LayerNorm}(\mathbf{X}_1) \tag{5}\\
\mathbf{F} &= \text{FFN}(\mathbf{H}_2) \tag{6}\\
\mathbf{X}_{\text{out}} &= \mathbf{X}_1 + \mathbf{F} \tag{7}
\end{align}$$

where $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{model}}}$ is the input to the block, and $\mathbf{X}_{\text{out}}$ is the output. This pre-normalization structure has been shown to improve training stability compared to post-normalization variants.

3.4 Layer Normalization

Layer normalization is applied independently to each position in the sequence:

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{8}$$

where μ and σ^2 are the mean and variance computed across the feature dimension:

$$\mu = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i \tag{9}$$

$$\sigma^2 = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} (x_i - \mu)^2 \tag{10}$$

The learnable parameters $\gamma, \beta \in \mathbb{R}^{d_{\text{model}}}$ allow the model to adapt the normalization as needed. The constant $\epsilon = 10^{-5}$ ensures numerical stability.

3.5 Output Layer

After passing through all transformer blocks, the final hidden states are normalized and projected to vocabulary logits:

$$\mathbf{H}_{\text{final}} = \text{LayerNorm}(\mathbf{X}_L) \tag{11}$$

$$\mathbf{L} = \mathbf{H}_{\text{final}}\mathbf{W}_{\text{out}} \in \mathbb{R}^{T \times V} \tag{12}$$

where $\mathbf{W}_{\text{out}} \in \mathbb{R}^{d_{\text{model}} \times V}$ is the output projection matrix. The logits $\mathbf{L}$ are used to compute the probability distribution over the vocabulary for next-token prediction.

4 Multi-Head Causal Self-Attention

4.1 Scaled Dot-Product Attention

The fundamental attention mechanism computes a weighted sum of value vectors based on the compatibility between query and key vectors. For a single attention head, given queries $\mathbf{Q} \in \mathbb{R}^{T \times d_k}$, keys $\mathbf{K} \in \mathbb{R}^{T \times d_k}$, and values $\mathbf{V} \in \mathbb{R}^{T \times d_k}$:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} \odot \mathbf{M}\right)\mathbf{V} \tag{13}$$

where $\mathbf{M} \in \mathbb{R}^{T \times T}$ is the causal mask ensuring that position i can only attend to positions $j \leq i$:

$$M_{ij} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases} \tag{14}$$

The scaling factor $1/\sqrt{d_k}$ prevents the dot products from growing too large in magnitude, which would push the softmax function into regions with extremely small gradients.

4.2 Multi-Head Mechanism

Multi-head attention allows the model to jointly attend to information from different representation subspaces. The input $\mathbf{X}$ is linearly projected into h sets of queries, keys, and values:

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q \in \mathbb{R}^{T \times d_k} \tag{15}$$

$$\mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K \in \mathbb{R}^{T \times d_k} \tag{16}$$

$$\mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V \in \mathbb{R}^{T \times d_k} \tag{17}$$

where $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ for $i = 1, \ldots, h$.

Each head computes attention independently:

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) \tag{18}$$

The outputs from all heads are concatenated and linearly transformed:

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{19}$$

where $\mathbf{W}^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ is the output projection matrix.

4.3 Rotary Position Embeddings (RoPE)

Rather than adding positional information to the input embeddings, Phonex-C2 employs Rotary Position Embeddings (RoPE), which encode positional information through rotation matrices applied to the query and key vectors.

For a position m and dimension pair $(2i, 2i+1)$, the rotation is defined as:

$$\mathbf{R}_m^{(i)} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \tag{20}$$

where the frequency θ_i is computed as:

$$\theta_i = 10000^{-2i/d_k} \tag{21}$$

The queries and keys are rotated before computing attention scores:

$$\tilde{\mathbf{Q}}_m = \mathbf{R}_m \mathbf{Q}_m \tag{22}$$

$$\tilde{\mathbf{K}}_m = \mathbf{R}_m \mathbf{K}_m \tag{23}$$

This formulation has the elegant property that the inner product $\tilde{\mathbf{Q}}_m^T \tilde{\mathbf{K}}_n$ depends only on the relative position $m - n$, enabling the model to naturally generalize to sequences longer than those seen during training.

4.4 Attention Score Computation

The complete attention computation for a single head proceeds as follows:

5 Feed-Forward Network

5.1 Architecture

The feed-forward network (FFN) in each transformer block consists of two linear transformations with a non-linear activation function:

$$\text{FFN}(\mathbf{x}) = \text{GELU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \tag{24}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$, and $\mathbf{b}_1 \in \mathbb{R}^{d_{\text{ff}}}$, $\mathbf{b}_2 \in \mathbb{R}^{d_{\text{model}}}$ are learnable parameters.

Algorithm 1 Causal Self-Attention with RoPE

```
**Input:** $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{model}}}$
**Output:** $\mathbf{Y} \in \mathbb{R}^{T \times d_k}$

$\mathbf{Q} \leftarrow \mathbf{X}\mathbf{W}^Q$
$\mathbf{K} \leftarrow \mathbf{X}\mathbf{W}^K$
$\mathbf{V} \leftarrow \mathbf{X}\mathbf{W}^V$

for $m = 1$ to $T$ do
    Apply RoPE rotation to $\mathbf{Q}_m$ and $\mathbf{K}_m$
end for

$\mathbf{S} \leftarrow \frac{1}{\sqrt{d_k}}\mathbf{Q}\mathbf{K}^T$
Apply causal mask: $\mathbf{S}_{ij} \leftarrow -\infty$ for $j > i$
$\mathbf{P} \leftarrow \text{softmax}(\mathbf{S})$ (row-wise)
$\mathbf{Y} \leftarrow \mathbf{P}\mathbf{V}$
return $\mathbf{Y}$
```

5.2 GELU Activation

The Gaussian Error Linear Unit (GELU) activation function is defined as:

$$\text{GELU}(x) = x \cdot \Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \tag{25}$$

where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution. In practice, the following approximation is commonly used:

$$\text{GELU}(x) \approx 0.5x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right]\right) \tag{26}$$

This approximation provides a good balance between computational efficiency and accuracy. The GELU activation has been shown to outperform ReLU in transformer architectures, particularly for language modeling tasks.

6 Training Methodology

6.1 Loss Function

For language modeling, the training objective is to minimize the cross-entropy loss between predicted and actual next tokens. Given a sequence $\mathbf{x} = (x_1, \ldots, x_T)$ and the model's logit predictions $\mathbf{L} \in \mathbb{R}^{T \times V}$, the loss is:

$$\mathcal{L} = -\frac{1}{T-1}\sum_{t=1}^{T-1} \log p(x_{t+1}|x_1, \ldots, x_t) \tag{27}$$

where the conditional probability is computed via softmax:

$$p(x_{t+1} = v|x_1, \ldots, x_t) = \frac{\exp(\mathbf{L}_{t,v})}{\sum_{v'=1}^{V} \exp(\mathbf{L}_{t,v'})} \tag{28}$$

6.2 AdamW Optimizer

Phonex-C2 employs the AdamW optimizer, which decouples weight decay from the gradient-based update. For each parameter θ, the update rule is:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \tag{29}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \tag{30}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \tag{31}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t} \tag{32}$$

$$\theta_t = \theta_{t-1} - \alpha_t \left(\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} + \lambda \theta_{t-1} \right) \tag{33}$$

where:

- $\mathbf{g}_t$ is the gradient at step t
- $\mathbf{m}_t$ and $\mathbf{v}_t$ are the first and second moment estimates
- $\beta_1 = 0.9$ and $\beta_2 = 0.999$ are the exponential decay rates
- α_t is the learning rate at step t
- $\lambda = 0.01$ is the weight decay coefficient
- $\epsilon = 10^{-8}$ is a small constant for numerical stability

6.3 Gradient Clipping

To prevent exploding gradients, global gradient norm clipping is applied:

$$\mathbf{g}_t \leftarrow \mathbf{g}_t \cdot \min\left(1, \frac{\tau}{\|\mathbf{g}_t\|_2}\right) \tag{34}$$

where $\tau = 1.0$ is the clipping threshold and $\|\mathbf{g}_t\|_2$ is the ℓ_2 norm of all gradients concatenated.

6.4 Learning Rate Schedule

The learning rate follows a warmup-cosine decay schedule:

$$\alpha_t = \begin{cases} \alpha_{\max} \cdot \frac{t}{T_{\text{warmup}}} & \text{if } t \leq T_{\text{warmup}} \\ \alpha_{\min} + \frac{1}{2}(\alpha_{\max} - \alpha_{\min}) \left(1 + \cos\left(\pi \cdot \frac{t - T_{\text{warmup}}}{T_{\max} - T_{\text{warmup}}}\right)\right) & \text{if } t > T_{\text{warmup}} \end{cases} \tag{35}$$

where T_{warmup} is the number of warmup steps, $T_{\max}$ is the total number of training steps, $\alpha_{\max}$ is the maximum learning rate, and $\alpha_{\min}$ is the minimum learning rate.

7 Backpropagation

7.1 Gradient Flow

The backward pass computes gradients with respect to all learnable parameters by applying the chain rule through the computational graph. Starting from the loss gradient, gradients flow backward through:

1. Output projection layer
2. Final layer normalization
3. Each transformer block (in reverse order)
4. Input embedding layer

7.2 Layer Normalization Gradients

Given the forward pass:

$$\mathbf{y} = \gamma \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{36}$$

The gradients with respect to inputs and parameters are:

$$\frac{\partial \mathcal{L}}{\partial \gamma} = \sum_i \frac{\partial \mathcal{L}}{\partial y_i} \cdot \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{37}$$

$$\frac{\partial \mathcal{L}}{\partial \beta} = \sum_i \frac{\partial \mathcal{L}}{\partial y_i} \tag{38}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \odot \left(\frac{\partial \mathcal{L}}{\partial \mathbf{y}} - \frac{1}{d} \sum_j \frac{\partial \mathcal{L}}{\partial y_j} - \frac{\mathbf{x} - \mu}{\sigma^2 + \epsilon} \cdot \frac{1}{d} \sum_j \frac{\partial \mathcal{L}}{\partial y_j} (x_j - \mu) \right) \tag{39}$$

7.3 Attention Gradients

For the attention mechanism, gradients must flow through the softmax operation and the query-key-value computations. Given:

$$\mathbf{Y} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} \right) \mathbf{V} \tag{40}$$

The gradient with respect to values is straightforward:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{V}} = \mathbf{P}^T \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \tag{41}$$

where $\mathbf{P} = \text{softmax}(\mathbf{Q}\mathbf{K}^T / \sqrt{d_k})$.

The gradient with respect to the attention scores (before softmax) requires the Jacobian of the softmax function:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{S}} = \mathbf{P} \odot \left(\frac{\partial \mathcal{L}}{\partial \mathbf{P}} - \text{diag}\left(\frac{\partial \mathcal{L}}{\partial \mathbf{P}} \mathbf{P}^T \right) \right) \tag{42}$$

where $\mathbf{S} = \mathbf{Q}\mathbf{K}^T / \sqrt{d_k}$ and the diagonal operation creates a matrix with the vector values on the diagonal.

7.4 FFN Gradients

For the feed-forward network, gradients flow through the GELU activation. The derivative of GELU is:

$$\text{GELU}'(x) = \Phi(x) + x \cdot \phi(x) \tag{43}$$

where $\phi(x) = \frac{1}{\sqrt{2\pi}} \exp(-x^2/2)$ is the standard normal probability density function.

The gradients for the FFN parameters are computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = \mathbf{h}^T \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \tag{44}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_2} = \sum_i \frac{\partial \mathcal{L}}{\partial y_i} \tag{45}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \mathbf{W}_2^T \tag{46}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \odot \text{GELU}'(\mathbf{z}) \tag{47}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \mathbf{x}^T \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \tag{48}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_1} = \sum_i \frac{\partial \mathcal{L}}{\partial z_i} \tag{49}$$

where $\mathbf{z} = \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1$ and $\mathbf{h} = \text{GELU}(\mathbf{z})$.

8 Memory Management

8.1 Memory Architecture

Phonex-C2 employs a dual-tier memory management strategy designed to optimize both memory usage and computational performance:

Tier 1: Persistent Parameter Storage

All model parameters, gradients, and optimizer state variables are allocated using aligned memory allocations. This includes:

- Weight matrices: $\mathbf{W}_{\text{emb}}, \mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V, \mathbf{W}^O, \mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_{\text{out}}$
- Biases: $\mathbf{b}_1, \mathbf{b}_2$
- LayerNorm parameters: γ, β
- Gradients for all parameters
- Adam optimizer moments: $\mathbf{m}, \mathbf{v}$

Total persistent memory: approximately 8 MB

Tier 2: Ephemeral Activation Storage

A 256 MB arena allocator manages all intermediate activations during forward and backward passes. The arena is reset at the beginning of each training step, providing:

- Zero heap fragmentation
- Deterministic memory usage
- Fast allocation (simple pointer bump)
- Cache-friendly sequential access patterns

8.2 Arena Allocator Design

The arena allocator maintains a large pre-allocated memory buffer and a current offset pointer:

Algorithm 2 Arena Memory Allocation

```
struct Arena {
  char* buffer
  size_t capacity
  size_t offset
}
arena_allocarena, size, align
aligned_offset ← (offset + align - 1) & ∼(align - 1)
if aligned_offset + size > capacity then
  return NULL
end if
ptr ← buffer + aligned_offset
offset ← aligned_offset + size
return ptr
arena_resetarena
offset ← 0
```

This design ensures that all activations are allocated sequentially in memory, improving cache locality and reducing allocation overhead to a single pointer arithmetic operation.

8.3 Memory Layout for SIMD

All parameter matrices are stored in row-major format with 32-byte (AVX2) alignment:

$$\text{Address}(\mathbf{W}[i,j]) = \text{base} + i \cdot \text{stride} + j \cdot \text{sizeof(float)} \tag{50}$$

where stride is rounded up to the nearest multiple of 32 bytes. This alignment ensures that SIMD load/store operations can be performed efficiently without crossing cache line boundaries.

9 Hardware Optimizations

9.1 SIMD Vectorization

Phonex-C2 leverages AVX2 instructions for Single Instruction Multiple Data (SIMD) parallelism. The most performance-critical operation, matrix multiplication, is vectorized to process 8 floating-point values simultaneously.

For matrix multiplication $\mathbf{C} = \mathbf{AB}$ where $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$:

The FMA (fused multiply-add) instruction _mm256_fmadd_ps performs $a \cdot b + c$ in a single operation, reducing both instruction count and rounding errors.

Algorithm 3 AVX2 Matrix Multiplication (Simplified)

```
for $i = 0$ to $m - 1$ do
  for $j = 0$ to $n - 1$ step 8 do
    sum ← _mm256_setzero_ps()
    for $k = 0$ to $k - 1$ do
      a_vec ← _mm256_broadcast_ss(&A[i][k])
      b_vec ← _mm256_load_ps(&B[k][j])
      sum ← _mm256_fmadd_ps(a_vec, b_vec, sum)
    end for
    _mm256_store_ps(&C[i][j], sum)
  end for
end for
```

9.2 OpenMP Parallelization

Multi-core parallelism is achieved through OpenMP directives. The primary parallelization targets include:

- Batch processing (when batch size > 1)
- Token-level operations across sequence length
- Attention head computation
- Matrix operations with sufficient work per thread

For example, attention computation across multiple heads is parallelized:

```
#pragma omp parallel for
for (int h = 0; h < num_heads; h++) {
    compute_attention_head(h, ...);
}
```

9.3 Cache Optimization

The implementation employs several cache-aware optimizations:

Memory access patterns: Matrix operations are ordered to maximize sequential access and minimize cache misses. For matrix multiplication, the inner loop iterates over the dimension with stride 1.

Tiling: Large matrix operations are blocked into smaller tiles that fit in L2/L3 cache:

$$\text{Tile size} = \min\left(\sqrt{\frac{L2\_size}{3 \cdot \text{sizeof(float)}}}, 64\right) \tag{51}$$

Prefetching: The compiler's automatic prefetching is supplemented with explicit prefetch hints for critical loops.

9.4 Computational Complexity

The computational complexity for a forward pass through one transformer block is:

- **Attention:** $O(T^2 d_{\text{model}} + T d_{\text{model}}^2)$
- **FFN:** $O(T d_{\text{model}} d_{\text{ff}})$
- **LayerNorm:** $O(T d_{\text{model}})$

For the complete model with L layers:

$$\text{FLOPs}_{\text{forward}} \approx L \cdot (12 T d_{\text{model}}^2 + 2T^2 d_{\text{model}} + 4T d_{\text{model}} d_{\text{ff}}) \tag{52}$$

The backward pass requires approximately twice the FLOPs of the forward pass.

10 Text Generation

10.1 Autoregressive Decoding

Text generation proceeds autoregressively, predicting one token at a time and appending it to the context for subsequent predictions:

Algorithm 4 Autoregressive Text Generation

```
Input: prompt tokens $\mathbf{x}_{\text{init}}$, max length $N$
Output: generated sequence $\mathbf{x}_{\text{gen}}$

$\mathbf{x} \leftarrow \mathbf{x}_{\text{init}}$
for $t = |\mathbf{x}_{\text{init}}|$ to $N - 1$ do
  $\mathbf{L} \leftarrow$ Forward($\mathbf{x}$)
  $x_{t+1} \leftarrow$ Sample($\mathbf{L}_t$)
  $\mathbf{x} \leftarrow$ Append($\mathbf{x}, x_{t+1}$)
  if $x_{t+1}$ is end-of-sequence token then
    break
  end if
end for
return $\mathbf{x}$
```

10.2 Sampling Strategies

Greedy Decoding:

The simplest strategy selects the highest probability token:

$$x_{t+1} = \arg\max_v p(v|x_1, \ldots, x_t) \tag{53}$$

Temperature Sampling:

Temperature scaling modulates the probability distribution before sampling:

$$p_T(v|\mathbf{x}) = \frac{\exp(\mathbf{L}_v/T)}{\sum_{v'} \exp(\mathbf{L}_{v'}/T)} \tag{54}$$

where $T > 0$ is the temperature parameter. Lower temperatures ($T < 1$) make the distribution more peaked (more deterministic), while higher temperatures ($T > 1$) flatten the distribution (more random).

10.3 Byte-Level Tokenization

Phonex-C2 operates directly on byte-level tokens (vocabulary size 256), eliminating the need for subword tokenization algorithms like BPE or SentencePiece. This approach:

- Supports arbitrary UTF-8 text without preprocessing
- Eliminates out-of-vocabulary issues
- Simplifies the training pipeline
- Increases sequence lengths but reduces vocabulary complexity

Each byte value $b \in \{0, 1, \ldots, 255\}$ is a valid token, and UTF-8 characters are represented as sequences of 1-4 bytes.

11 Experimental Results

11.1 Performance Metrics

Performance measurements were conducted on a system with the following specifications:

- **CPU:** AMD Ryzen 9 8945HS (16) @ 5.26 GHz
- **Memory:** 16GB DDR5
- **Compiler:** GCC 11.2 with -O3 -mavx2 -mfma
- **Operating System:** Linux 5.15

Operation	**Time (ms)**	**Throughput**	**FLOPs**
Forward pass	5.2	6.4 GFLOPs	33.3M
Backward pass	12.1	5.5 GFLOPs	66.6M
Full training step	19.8	5.8 GFLOPs	115.2M
AdamW update	2.5	N/A	5.0M
Total per step	**22.3**	**5.4 GFLOPs**	**120.2M**

Table 1: Per-step performance breakdown for sequence length $T = 32$

The system achieves approximately 45 training steps per second, with memory bandwidth being the primary bottleneck rather than compute throughput.

11.2 Memory Profiling

Memory usage breakdown:

The activation arena is intentionally oversized to accommodate future experimentation with longer sequences and larger batch sizes without recompilation.

Component	**Memory (MB)**
Model parameters	2.1
Gradients	2.1
Adam moments (m, v)	4.2
Persistent total	*8.4*
Activation arena	256.0
Total footprint	**264.4**

Table 2: Memory usage breakdown

11.3 Scaling Analysis

Performance scaling with respect to key hyperparameters:

Sequence length: The attention mechanism's $O(T^2)$ complexity dominates for $T > 64$. For the current implementation:

$$\text{Time}(T) \approx 0.15T + 0.012T^2 \text{ (ms)} \tag{55}$$

Model dimension: Linear scaling is observed for $d_{\text{model}} < 512$ due to efficient cache utilization:

$$\text{Time}(d) \approx 3.2 + 0.125d \text{ (ms)} \tag{56}$$

Number of layers: Nearly perfect linear scaling due to independent layer computation:

$$\text{Time}(L) \approx 2.1 + 4.3L \text{ (ms)} \tag{57}$$

11.4 Training Dynamics

Training on a small corpus of text demonstrates expected learning behavior:

- Loss decreases from approximately 5.5 (random initialization) to approximately 2.1 after 10,000 steps
- Gradient norms stabilize after warmup period
- Model begins generating coherent byte sequences after approximately 3,000 steps

The learning curve follows the expected pattern for small-scale language models, with rapid initial improvement followed by gradual refinement.

12 Design Philosophy and Trade-offs

12.1 Intentional Limitations

Phonex-C2 makes deliberate choices that prioritize clarity and accessibility over raw performance:

Small scale: The 500K parameter model is intentionally tiny by modern standards. This allows the entire system to be comprehended by a single developer and trained on consumer hardware.

Simple attention: The attention implementation uses the standard $O(T^2)$ algorithm rather than approximations like Flash Attention or linear attention mechanisms. This preserves architectural clarity at the cost of scalability.

Single-threaded gradient computation: While forward passes are parallelized, backward passes use simpler threading to maintain code readability.

12.2 Extensibility Points

The codebase is designed to support several natural extensions:

- **Larger models:** Scaling to millions or billions of parameters requires only adjusting hyperparameters.
- **Efficient attention:** The attention module can be replaced with Flash Attention or other optimized kernels without affecting other components.
- **Mixed precision:** FP16 or BF16 training can be implemented by modifying the core arithmetic operations.
- **Distributed training:** The clear separation between forward/backward passes facilitates model parallelism implementation.

12.3 Educational Value

The primary value proposition is educational. By studying Phonex-C2, researchers and students can:

1. Understand the exact correspondence between mathematical formulations and executable code
2. Observe the impact of low-level optimizations on training performance
3. Experiment with architectural modifications without framework constraints
4. Debug transformer behavior at the lowest level of abstraction

13 Comparison with Related Systems

13.1 Framework-based Implementations

Compared to PyTorch or TensorFlow implementations:

Advantages:

- Complete transparency of all operations
- No hidden automatic differentiation
- Predictable memory usage
- Direct hardware control

Disadvantages:

- Manual gradient implementation required
- No automatic GPU support
- Limited to CPU execution
- Requires more code for equivalent functionality

13.2 Optimized Inference Engines

Compared to production inference systems (llama.cpp, GGML):

Advantages:

- Full training capability
- Simpler codebase
- Educational clarity

Disadvantages:

- Lower throughput
- No quantization support
- Limited to small models

14 Future Directions

Several enhancements are planned for future versions:

14.1 Technical Enhancements

- **Flash Attention:** Implementing IO-aware attention to reduce memory bandwidth requirements
- **Mixed Precision:** FP16 training with dynamic loss scaling
- **Gradient Checkpointing:** Trading compute for memory to enable larger models
- **KV Caching:** Efficient inference through key-value caching

14.2 Architectural Variants

- **Grouped Query Attention:** Reducing KV cache size for inference
- **Sliding Window Attention:** Supporting longer sequences with local attention
- **Mixture of Experts:** Conditional computation for increased model capacity

14.3 Platform Support

- **ARM NEON:** SIMD support for ARM processors
- **RISC-V Vector:** Support for RISC-V vector extensions
- **GPU Backend:** CUDA or OpenCL implementation for GPU acceleration

15 Conclusion

Phonex-C2 demonstrates that modern transformer language models can be implemented entirely from scratch in pure C without sacrificing architectural fidelity. By exposing every computational and memory decision, it bridges the gap between theoretical understanding and practical implementation.

The system serves multiple purposes: as a pedagogical tool for understanding transformers, as a reference implementation for architectural experiments, and as a foundation for low-level optimization research. While not competitive with production systems in terms of scale or performance, Phonex-C2 occupies a valuable niche in the ecosystem of transformer implementations.

The complete transparency of the implementation enables researchers to understand exactly how mathematical operations translate to executable code, how gradients flow through complex architectures, and how low-level optimizations impact performance. This level of insight is difficult or impossible to achieve with high-level frameworks.

Future work will focus on implementing advanced optimization techniques while maintaining the core philosophy of clarity and accessibility. The goal remains to provide a complete, understandable transformer implementation that serves the educational and research community.

Acknowledgments

The author thanks the open-source machine learning community for numerous discussions and insights that shaped this work. Special thanks to the developers of reference implementations that informed design decisions.

Availability

Source code and documentation are available at: `https://github.com/aam-007/phonex-c2`

The repository includes:

- Complete source code (single C file)
- Build instructions for multiple platforms
- Training scripts and data preparation utilities
- Performance benchmarking tools
- Architectural documentation

References

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.

[2] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI Blog*.

[3] Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Liu, Y. (2021). RoFormer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*.

[4] Hendrycks, D., and Gimpel, K. (2016). Gaussian error linear units (GELUs). *arXiv preprint arXiv:1606.08415*.

[5] Loshchilov, I., and Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

[6] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

[7] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems*, 35.

[8] Shazeer, N. (2020). GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*.