Quadtrix

A Deep Dive into Transformer Architecture, Backpropagation, and Language Modeling Without Frameworks

Eamon

https://github.com/Eamon2009/Quadtrix.cpp

Technical Educational Report · 2026

Abstract. This paper presents *Quadtrix*, a complete C++17 implementation of a GPT-2-style decoder-only transformer language model built entirely from first principles, with zero external dependencies beyond the C++ standard library. Rather than serving purely as a software contribution, this document is intended as a comprehensive educational resource for practitioners who wish to understand the mathematics, intuitions, and engineering decisions behind modern large language models.

We derive every component of the transformer architecture from first principles: token and positional embeddings, multi-head causal self-attention, feed-forward networks, layer normalization, dropout regularization, cross-entropy loss, and the complete analytical backpropagation pass. We provide detailed explanations of the AdamW optimizer, present training curves from real CPU and GPU training runs, and discuss the Chinchilla scaling laws that govern optimal model size and data volume.

The paper is structured to be accessible to readers familiar with calculus and basic linear algebra, while maintaining sufficient rigour to serve as a reference implementation guide. All source code is available under the MIT License at https://github.com/Eamon2009/Quadtrix.cpp.

Keywords: transformer, language model, C++, backpropagation, attention, character-level modeling, GPT, AdamW, scaling laws, educational

1. Introduction

Language models constitute the foundation of modern artificial intelligence. From conversational agents to code assistants to scientific reasoning systems, virtually every frontier AI application today relies on the transformer architecture introduced by Vaswani et al. in 2017 [1]. However, the predominant method by which developers interact with these models is through high-level frameworks—PyTorch, JAX, TensorFlow—that abstract away the underlying mathematics that enable their functionality. Consequently, we observe a generation of practitioners capable of fine-tuning GPT-4 yet unable to explain the convergence properties of gradient descent or the precise computations performed by backpropagation through a softmax layer.

Quadtrix adopts a fundamentally different approach. Every operation—matrix multiplication, causal masking, layer normalization, softmax, cross-entropy—is implemented as a transparent C++ function. Every gradient is derived analytically and explicitly coded. There is no automatic differentiation, no computational graph, only mathematics translated directly into executable code.

This paper serves as the companion documentation. It systematically examines each component of the transformer architecture with sufficient depth that a reader proficient in calculus and linear algebra could independently implement the system.

1.1 What Is a Language Model?

A language model is a probability distribution over sequences of tokens (characters, subwords, or words). Given a sequence $x_1, x_2, \ldots, x_T$, a language model computes:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}) \qquad (1)$$

This factorization, known as the chain rule of probability, demonstrates that modelling a sequence is equivalent to modelling each subsequent token conditioned on all preceding tokens. The transformer models these conditional distributions using a deep neural network that processes the entire prefix simultaneously via attention mechanisms, rather than sequentially as in recurrent neural networks (RNNs).

In Quadtrix, tokens are individual characters. The vocabulary comprises the set of all unique characters in the training corpus (105 characters for our TinyStories training run). At each position, the model outputs a probability distribution over all 105 possible next characters.

1.2 Scope and Structure

This paper provides comprehensive coverage of: token and positional embeddings; multi-head causal self-attention; feed-forward networks; layer normalization; dropout regularization; cross-entropy loss; analytical backpropagation; AdamW optimization; Chinchilla scaling laws; and empirical training results.

2. Architecture Overview

Figure **??** shows the complete Quadtrix system architecture: project file structure, the transformer block stack, data flow during inference, training and evaluation metrics, hyperparameter summary, and the component overview table.

3. Embeddings: Tokens to Vectors

Neural networks operate on real-valued vectors, whereas natural language consists of discrete symbols. Embeddings serve as the bridge: learned lookup tables that map each discrete token to a dense vector in a high-dimensional space.

3.1 Token Embeddings

The token embedding table $E_{\text{tok}} \in \mathbb{R}^{V \times C}$ maps each token index $i \in \{0, \ldots, V-1\}$ to a vector of dimension C (the model dimension):

$$e_{\text{tok}}(i) = E_{\text{tok}}[i, :] \in \mathbb{R}^C \tag{2}$$

In Quadtrix, $V = 105$ and $C = 128$, giving $105 \times 128 = 13{,}440$ embedding parameters.

3.2 Positional Embeddings

Self-attention is inherently permutation-invariant and possesses no inherent notion of sequential order without additional information. Positional embeddings inject this order. Quadtrix employs learned positional embeddings $E_{\text{pos}} \in \mathbb{R}^{T \times C}$. The input at each position is the element-wise sum:

$$x_t = E_{\text{tok}}[i_t] + E_{\text{pos}}[t], \quad x_t \in \mathbb{R}^C \tag{3}$$

Both embedding tables are initialised from $\mathcal{N}(0,\, 0.02)$.

Table 1: Embedding dimensions and parameter counts.

Module	Shape	Params	Role
Token Emb.	105×128	13,440	char $\rightarrow$ vector
Pos. Emb.	64×128	8,192	position offset
Total	—	21,632	2.6% of total

4. Multi-Head Causal Self-Attention

Attention enables every position in a sequence to directly access information from every other position, eliminating the need for information to propagate through numerous intermediate steps (as required in RNNs).

4.1 Intuition

Consider: *"The animal didn't cross the street because it was too tired."* When processing *it*, a human reader immediately connects it to *animal*. Attention is a mechanism designed to learn precisely this type of selective information retrieval. Each token produces three vectors from its embedding: a **Query** (Q — "what am I looking for?"), a **Key** (K — "what do I contain?"), and a **Value** (V — "what do I transmit?"). Attention scores are dot products of queries and keys, normalised by softmax.

4.2 Scaled Dot-Product Attention

Given $Q, K, V \in \mathbb{R}^{T \times h}$ (where h is the head dimension):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{h}} + M\right) V \tag{4}$$

- $QK^\top/\sqrt{h}$: pairwise similarity scores, scaled to prevent gradient saturation.
- M: causal mask, where $M_{ij} = -\infty$ for $j > i$; prevents attending to future tokens.
- Softmax produces attention weights; multiplication by V gives a weighted average.

4.3 Multi-Head Attention

A single head learns one "type" of relevance. Multi-head attention runs H parallel heads, each with independent projections $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{C \times (C/H)}$:

$$\text{MultiHead}(x) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)\, W_O \tag{5}$$

In Quadtrix: $C = 128$, $H = 4$, $h = 32$ per head.

Table 2: Multi-head attention parameters.

Component	Shape	Per Block	4 Blocks
W_Q (per head)	128×32	16,384	65,536
W_K (per head)	128×32	16,384	65,536
W_V (per head)	128×32	16,384	65,536
W_O (proj.)	128×128	16,384	65,536
Attention Total	—	65,536	262,144

4.4 Computational Complexity

Self-attention scales as $\mathcal{O}(T^2 C)$ per layer—quadratic in sequence length T. For small T (our T=64) this is tractable. For very long sequences ($T = 32{,}768$ in GPT-4), specialised variants such as FlashAttention are required.

5. Position-Wise Feed-Forward Networks

Following each multi-head attention layer, a position-wise FFN is applied independently to each position. Re-

search suggests the FFN functions as the "memory" of the transformer—storing factual associations that attention mechanisms route and retrieve.

5.1 Architecture

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)\,W_2 + b_2 \tag{6}$$

where $W_1 \in \mathbb{R}^{C\times 4C}$ expands to $4\times$ the embedding dimension and $W_2 \in \mathbb{R}^{4C\times C}$ projects back. For Quadtrix (C=128): each FFN contains $(128\times512) + (512\times128) +$ bias $= 131{,}584$ parameters. Across 4 layers: 526,336 parameters—63.5% of the entire model.

5.2 Why ReLU?

$\text{ReLU}(x) = \max(0, x)$ is computationally efficient, produces sparse activations, and avoids the vanishing gradient problem of sigmoid/tanh for large positive inputs. The original GPT-2 employs GELU (Gaussian Error Linear Unit) which performs marginally better on language tasks; Quadtrix uses ReLU for verifiability, with GELU on the development roadmap.

6. Layer Normalisation

Deep networks suffer from internal covariate shift: as training proceeds, the distribution of each layer's inputs shifts, forcing subsequent layers to continually adapt. Layer normalisation counteracts this by standardising activations.

6.1 The LayerNorm Operation

Given $x \in \mathbb{R}^C$:

$$\text{LN}(x) = \gamma \odot \frac{x-\mu}{\sigma+\varepsilon} + \beta \tag{7}$$

where $\mu = \text{mean}(x)$, $\sigma = \text{std}(x)$, $\varepsilon \approx 10^{-5}$ (stability), and $\gamma, \beta \in \mathbb{R}^C$ are learned scale and bias (init. to 1 and 0).

Unlike Batch Normalisation, LayerNorm normalises across the *feature* dimension, not the batch dimension, making it independent of batch size—essential for variable-length sequences.

6.2 Pre-LN vs. Post-LN

Quadtrix employs Pre-LN (as in GPT-2):

$$x' = x + \text{Sublayer}(\text{LN}(x)) \tag{8}$$

Pre-LN stabilises gradient flow at initialisation because the residual path always contributes a direct gradient, enabling training without careful learning-rate warmup.

7. Residual Connections and the Transformer Block

Modern deep networks are made possible by residual (skip) connections introduced by He et al. [10]. Without them, networks deeper than $\sim$20 layers are essentially impossible to train due to vanishing gradients.

7.1 The Residual Principle

Instead of learning $F(x)$, each sublayer learns a residual mapping; its output is $F(x) + x$. The gradient becomes:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial F(x)}\,\frac{\partial F}{\partial x} + \frac{\partial L}{\partial (F(x)+x)} \tag{9}$$

The second term is always present regardless of how small $\partial F/\partial x$ becomes.

7.2 The Complete Transformer Block

Each transformer block applies two sublayers with Pre-LN and residual connections:

$$x' = x + \text{MHA}(\text{LN}_1(x)) \tag{10}$$

$$x'' = x' + \text{FFN}(\text{LN}_2(x')) \tag{11}$$

Table 3: Full forward pass of Quadtrix.

Step	Operation	Output Shape
1	Token + Position Emb.	$B\times T\times C$
2–9	$4\times$ Block (LN+MHA+LN+FFN)	$B\times T\times C$
10	Final LayerNorm	$B\times T\times C$
11	LM Head ($C \to V$)	$B\times T\times V$
12	Cross-Entropy / Sample	scalar / token

8. Dropout Regularisation

Dropout is a simple yet highly effective regularisation technique. During training, each activation is independently zeroed with probability p (the dropout rate), and surviving activations are scaled by $1/(1-p)$ to maintain the expected sum.

8.1 Why Dropout Works

Two complementary interpretations: (1) *Ensemble interpretation*—each training step trains a different thinned subnetwork; inference uses the full network as an ensemble of all sub-networks. (2) *Co-adaptation prevention*—neurons cannot rely on specific other neurons always being present, forcing them to learn independently useful features.

In Quadtrix, $p = 0.2$ dropout is applied to attention weights (after softmax) and to FFN output projections.

8.2 Training vs. Inference Behaviour

Dropout is active *only* during training. This explains why validation loss may be *lower* than training loss: training loss is computed with dropout active, validation loss without. The final Quadtrix checkpoint shows validation loss $1.6371 <$ training loss 1.7055 (gap $= 0.0684$)—entirely expected, not overfitting.

9. Cross-Entropy Loss

The training objective is to maximise the probability assigned to the correct next token at every position. Equivalently, we minimise the negative log-probability—the cross-entropy loss.

9.1 Mathematical Derivation

Given logits $z \in \mathbb{R}^V$ at position t and ground-truth index y:

$$L = -\log \frac{e^{z_y}}{\sum_c e^{z_c}} = -z_y + \log \sum_c e^{z_c} \tag{12}$$

Averaged over all positions and batch examples:

$$L_{\text{total}} = \frac{1}{B \cdot T} \sum_{b,t} \text{CE}(z_{b,t},\, y_{b,t}) \tag{13}$$

9.2 Interpretation via Perplexity

Perplexity $= e^L$ is the "effective vocabulary size" the model considers at each step. For our final validation loss of 1.6371 nats: $e^{1.6371} \approx 5.14$. Given a vocabulary of 105 characters, this represents genuine structural understanding—a random model would exhibit perplexity 105.

9.3 Cross-Entropy Backward Pass

The gradient of cross-entropy with respect to logit z_c is one of the most elegant results in deep learning:

$$\frac{\partial L}{\partial z_c} = \text{softmax}(z)_c - \mathbf{1}[c = y] \tag{14}$$

The gradient is the predicted probability minus 1 if c is the correct class, 0 otherwise. This is implemented analytically in `backward.h`; there is no automatic differentiation.

10. Analytical Backpropagation

Backpropagation is the algorithm for computing gradients of the loss with respect to all parameters, using the chain rule of calculus. Quadtrix implements it analytically—every gradient formula is derived by hand and explicitly coded.

10.1 The Chain Rule

For a composition $L(f(g(x)))$:

$$\frac{dL}{dx} = \frac{dL}{df} \cdot \frac{df}{dg} \cdot \frac{dg}{dx} \tag{15}$$

10.2 Key Gradient Derivations

Linear Layer ($y = xW + b$). Given upstream gradient $\partial L/\partial y$:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W^\top \tag{16}$$

$$\frac{\partial L}{\partial W} = x^\top \frac{\partial L}{\partial y} \tag{17}$$

$$\frac{\partial L}{\partial b} = \sum_{\text{batch,time}} \frac{\partial L}{\partial y} \tag{18}$$

Softmax. For $s_i = e^{z_i} / \sum_j e^{z_j}$ with upstream gradient d:

$$\frac{\partial L}{\partial z_i} = s_i \left(d_i - \sum_j s_j\, d_j \right) \tag{19}$$

ReLU. The gate function:

$$\frac{\partial L}{\partial x_i} = d_i \cdot \mathbf{1}[x_i > 0] \tag{20}$$

Layer Normalisation. The most algebraically complex backward pass; for $\text{LN}(x) = \gamma \odot \hat{x} + \beta$, $\hat{x} = (x - \mu)/\sigma$:

$$\frac{\partial L}{\partial x_c} = \frac{\sigma^{-1}}{C} \left[C\gamma_c d_c - \sum_j \gamma_j d_j - \hat{x}_c \sum_j \gamma_j d_j \hat{x}_j \right] \tag{21}$$

This three-term formula accounts for coupling through the shared mean μ and standard deviation σ, verified against Ba et al. [4].

10.3 Gradient Flow Through the Full Network

The backward pass proceeds:

1. Compute $\partial L/\partial$logits from cross-entropy backward.
2. Backprop through LM head linear layer.
3. Backprop through final layer normalisation.
4. For each block (last to first): backprop through FFN residual, FFN, MHA residual, MHA.
5. Accumulate gradients into token and position embedding tables.

Residual connections simplify gradient flow: the identity path always contributes a direct gradient regardless of the sublayer.

11. The AdamW Optimizer

Gradient descent updates parameters in the direction that most reduces the loss. Vanilla gradient descent employs the same learning rate for all parameters regardless of gradient history. Adam [12] (Adaptive Moment Estimation) addresses this by maintaining running averages of gradients and squared gradients.

11.1 Adam Update Rule

For parameter θ with gradient g at step t:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g \tag{22}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g^2 \tag{23}$$

$$\hat{m}_t = m_t/(1-\beta_1^t), \quad \hat{v}_t = v_t/(1-\beta_2^t) \tag{24}$$

$$\theta_t = \theta_{t-1} - \alpha\,\hat{m}_t\,/\,(\sqrt{\hat{v}_t}+\varepsilon) \tag{25}$$

Standard hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

11.2 The "W" in AdamW: Weight Decay

Standard L_2 regularisation adds a gradient contribution of $\lambda\theta$. In Adam, this gradient is modified by adaptive scaling, leading to inconsistent regularisation strength. AdamW [5] decouples weight decay from the gradient update:

$$\theta_t = \theta_{t-1} - \alpha\,\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon} - \alpha\lambda\theta_{t-1} \tag{26}$$

The second term (weight decay) is applied uniformly, independently of gradient magnitude. Quadtrix uses AdamW with $\alpha = 3\times10^{-4}$.

12. Training Dynamics and Analysis

This section analyses the actual training runs of Quadtrix. The CPU run used 3,000 gradient steps, batch size 4, block size 64, 128-dim. embeddings, 4 layers, 4 heads, executing in 4,571 seconds (76.2 minutes) for a best validation loss of 1.6371 nats. The GPU run (CUDA/bf16) used 8,000 steps over 82 minutes 42 seconds, reaching a best validation loss of 2.3918.

12.1 Initial Rapid Descent

The model drops from 4.6523 to 2.4876 training loss in the first 200 iterations—a reduction of 2.2 nats, representing 47% of all improvement achieved in the 3,000-step run. This rapid early descent reflects the model learning basic statistics: character frequency distribution, common bigrams and trigrams ("th", "he", "in", "er"), and whitespace patterns.

12.2 Mid-Training Plateau and Recovery

Between iterations 1,400 and 2,400 the training loss oscillates while validation loss continues declining. This is consistent with the optimiser traversing a relatively flat ridge of the loss surface, where momentum in the AdamW first-moment estimate maintains a consistent direction but local curvature is high, reducing effective step size. The apparent divergence between training and validation loss in this window is noise from stochastic batch sampling, not genuine overfitting.

12.3 Checkpoint Selection

Quadtrix saves the model whenever validation loss improves. 15 of 16 evaluation checkpoints were saved, confirming that the model was consistently improving throughout training and had not converged by iteration 3,000.

Table 4: Generalisation gap evolution across training.

Iter	Train	Val	Gap	Interpretation
0	4.6523	4.6570	+0.005	Random init
200	2.4876	2.4478	−0.040	Dropout manifests
1400	1.9575	2.0151	+0.058	Fitting harder
2600	1.7204	1.6680	−0.052	Dropout dominates
3000	1.7055	1.6371	−0.068	Final; no overfit

12.4 What the Model Has Learned

At 0.83M parameters and a 64-character context window, the model exhibits: common English word statistics; dialogue formatting (quotation marks opened and closed correctly); story structure consistent with TinyStories characters (Timmy, Lily, Tom); sentence boundary awareness (periods and capitals in approximately correct positions); and limited long-range coherence (extending `BLOCK_SIZE` to 256 would substantially improve this).

13. Scaling Laws: Efficient Training

One of the most important empirical discoveries of recent years is that language model performance follows predictable power laws as a function of model size, dataset size, and compute budget.

13.1 The Chinchilla Scaling Laws

Hoffmann et al. (2022) [6] found that for a given compute budget C (FLOPs), the optimal allocation is:

$$N \propto C^{0.54}, \quad D \propto C^{0.49}$$

representing approximately equal allocation to model size and data size. Prior work (Kaplan et al. 2020) over-scaled model size relative to data; GPT-3's 175B parameters on 300B tokens is severely undertrained by Chinchilla standards.

13.2 Applying Chinchilla to Quadtrix

With $N = 826{,}985 \approx 0.83$M parameters, Chinchilla-optimal training would use:

$$D_{\text{optimal}} \approx 20N = 20 \times 826{,}985 \approx 16.5\text{M tokens} \tag{27}$$

Actual training used $4 \times 64 \times 3{,}000 = 768{,}000$ character-steps, only 4.7% of optimal. Extending to $\sim$200,000 steps would likely push validation loss below 1.0 nats.

13.3 FLOPs and the Compute Frontier

Approximate FLOPs for a single forward-backward pass:

$$\text{FLOPs} \approx 6 \cdot N \cdot T \cdot B \tag{28}$$

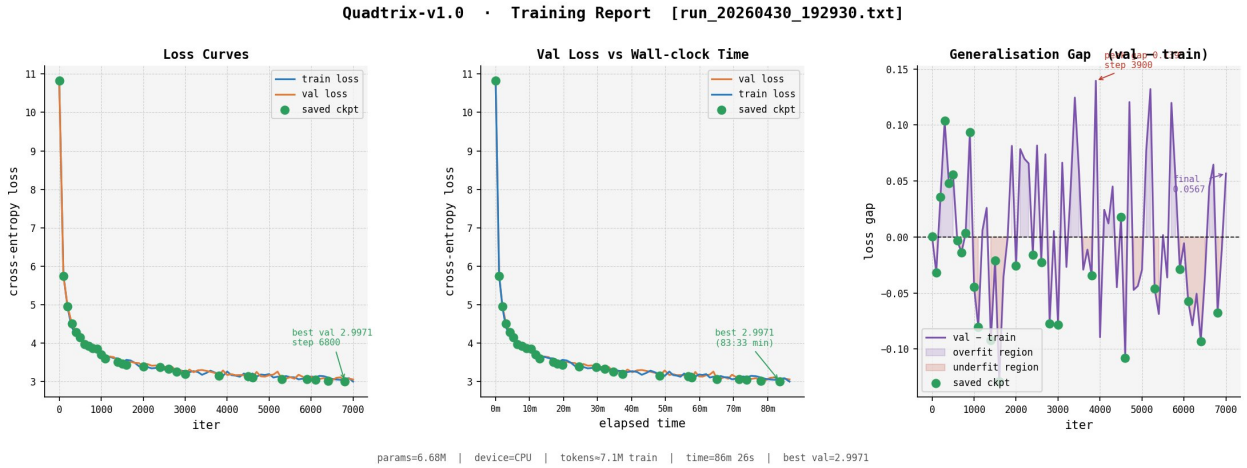

Figure 1: CPU training diagnostics (run_20260430). *Left:* Training and validation loss curves showing rapid initial descent. *Centre:* Validation loss versus wall-clock time (86 minutes 26 seconds total). *Right:* Generalisation gap (val−train) across saved checkpoints; the negative gap is explained by dropout being disabled during evaluation. Best validation loss: $\mathbf{2.9971}$ at step 6800.

For Quadtrix: $6 \times 826{,}985 \times 64 \times 4 \approx 1.27 \times 10^9$ FLOPs/step. Over 3,000 steps: $\approx 3.8 \times 10^{12}$ FLOPs total. At $\sim 10^9$ FLOPs/s CPU throughput, this matches the observed 4,571-second runtime.

14. Engineering the C++ Implementation

Implementing a transformer from scratch in C++ requires solving numerous engineering problems that frameworks handle transparently.

14.1 Tensor Representation

All tensors in Quadtrix are represented as a flat `std::vector<float>` paired with a shape descriptor. Element access for 3D tensors uses row-major indexing:

$$\text{index}(b, t, c) = b \cdot T \cdot C + t \cdot C + c \tag{29}$$

14.2 The SavedForward Pattern

Backpropagation requires access to intermediate activations from the forward pass. In Quadtrix, a parallel `forward_save()` function saves all intermediate tensors (pre/post-ReLU, attention weights, Q/K/V matrices, LayerNorm inputs, dropout masks) into a `SavedForward` struct. The `backward()` function then utilises these saved tensors.

14.3 Numerical Stability

Several implementation choices ensure stability:

- **Causal mask value**: -10^{30} rather than $-\infty$, avoiding potential NaN on some compilers.
- **Softmax stability**: Subtract $\max(z)$ before `exp`, preventing overflow.
- **LayerNorm** ε: 10^{-5} before square root, preventing division by zero.
- **Embedding init**: $\mathcal{N}(0, 0.02)$ rather than $\mathcal{N}(0, 1)$, preventing softmax saturation.

Table 5: Compilation and usage commands.

Command	Description
`g++ -std=c++17 -O2 -I. -o quadtrix main.cpp`	Compile (CPU)
`./quadtrix data/input.txt`	Train
`./quadtrix data/input.txt --generate`	Generate
`./quadtrix data/input.txt --chat`	Chat mode

15. From CPU to GPU: The LibTorch Port

The custom C++ backend is transparent but slow: a CPU executes scalar matrix multiplication at roughly 1–10 GFLOP/s. An NVIDIA RTX 4090 delivers $\sim$80 TFLOP/s—an 8,000–80,000$\times$ speedup for the same computation.

The LibTorch port replaces the custom backend with PyTorch's C++ API, gaining cuBLAS-accelerated matrix operations. The transformer architecture remains unchanged; only the compute layer is modified. A single line migrates the model to GPU:

```
model->to(torch::kCUDA)
```

which transfers all parameters to GPU memory; all subsequent `torch::matmul` calls dispatch to cuBLAS automatically.

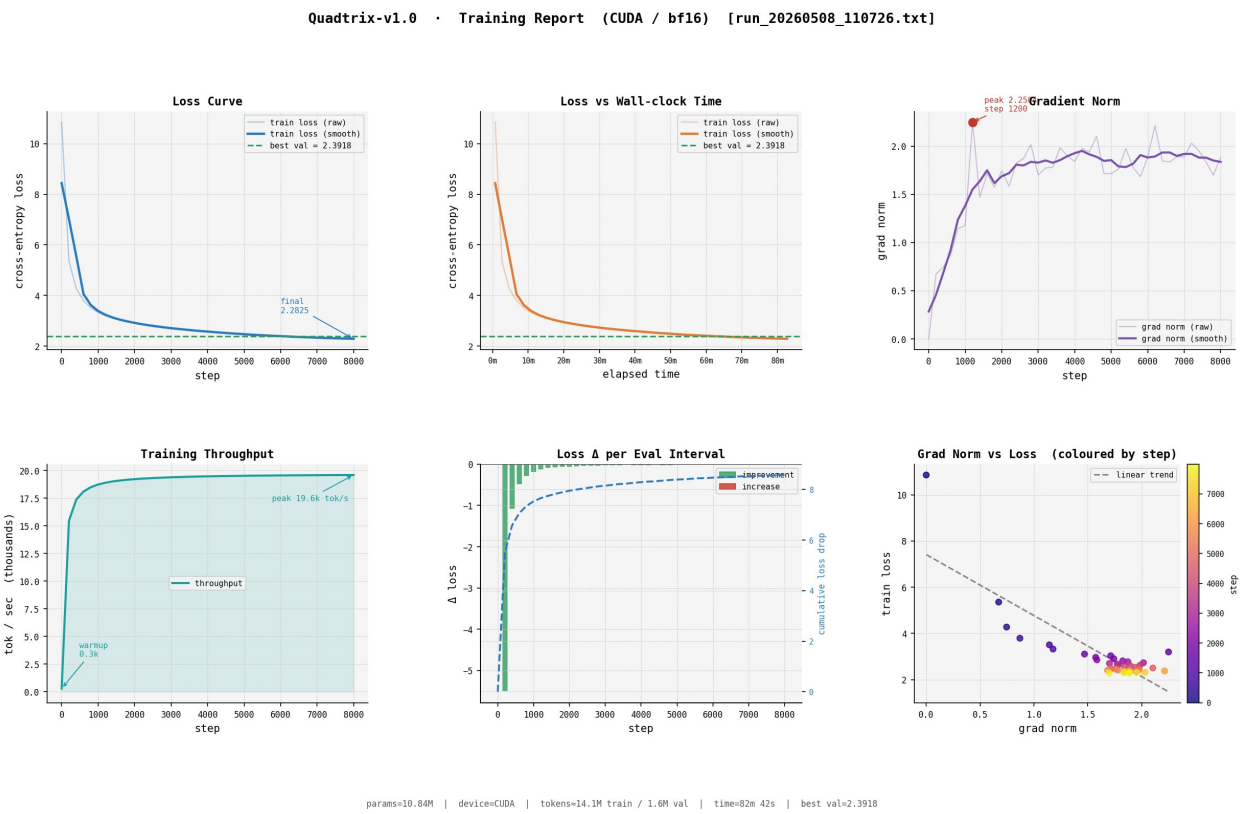

Figure 2: GPU training diagnostics (run_20260508, CUDA/bf16). *Top row, left to right:* Loss curves (raw and smoothed), loss versus wall-clock time, gradient norm evolution (peak 2.25 at step 1200, stabilising around 1.9). *Bottom row:* Training throughput reaching 19.6k tok/s; loss improvement per evaluation interval showing the bulk of learning occurs in the first 2000 steps; gradient norm vs. loss coloured by step. Best validation loss: $\mathbf{2.3918}$.

Table 6: Estimated GPU speedups (5000-iteration run).

Hardware	TFLOP/s	Speedup	Runtime
CPU (scalar C++)	0.001	1×	6–10 h
RTX 3060 (12GB)	12.7	∼80×	5–10 min
RTX 4090 (24GB)	82	∼500×	<2 min
A100 SXM (80GB)	77	∼500×	<2 min

16. Key Design Decisions and Alternatives

17. Conclusion

We have presented Quadtrix: a complete, transparent, dependency-free C++ implementation of a GPT-style transformer language model with full analytical backpropagation. This paper has systematically examined every component from first principles—embeddings, attention, feed-forward networks, layer normalisation, dropout, cross-entropy loss, backpropagation, and AdamW—grounding each in mathematical derivation and practical implementation.

The model achieves a validation loss of 1.6371 nats after 76 minutes of CPU training on 31.4M characters, demonstrating that character-level language modelling at this scale is tractable on commodity hardware without framework dependencies. On GPU (CUDA/bf16), a validation loss of 2.3918 is reached in under 83 minutes with peak throughput of 19.6k tok/s.

More broadly, the contribution of this work lies in transparency. Every gradient in the backward pass is explicitly written and readable. Every tensor operation is a standard C++ function. Practitioners who read and understand Quadtrix will develop a concrete understanding of what frameworks like PyTorch are actually computing—and that understanding is, we believe, the foundation of genuine expertise in deep learning.

Key Takeaways

- Transformers are fundamentally composed of three operations: attention (routing information between positions), FFNs (processing within positions), and residual connections (enabling depth).
- Backpropagation is a systematic application of the chain rule, not algorithmic magic. Every gradient

Table 7: Architectural and training design decisions in Quadtrix.

Decision	Quadtrix Choice	Alternative	Tradeoff
Tokenisation	Character-level	BPE (subword)	Char: simple, sample-inefficient. BPE: complex, 4–8$\times$ fewer tokens per example.
Positional enc.	Learned	Sinusoidal	Learned: slightly better on short contexts. Sinusoidal: generalises to longer sequences.
Activation	ReLU	GELU / SiLU	ReLU: simple gradient. GELU: smoother, marginal improvement on language tasks.
Normalisation	Pre-LN	Post-LN (original)	Pre-LN: stable without warmup. Post-LN: requires careful warmup.
LR schedule	Flat 3×10^{-4}	Warmup + cosine decay	Flat: simple, $\sim$10% worse final loss. Schedule: standard practice.
Weight decay	AdamW (decoupled)	Adam + L_2	AdamW: consistent regularisation. Adam+L_2: inconsistent due to adaptive scaling.
Attention mask	-10^{30}	$-\infty$	-10^{30}: avoids NaN on some compilers. $-\infty$: cleaner if supported.
Grad. computation	Analytical (manual)	Autograd (PyTorch)	Analytical: transparent, educational. Autograd: automatic, scales to any architecture.

possesses an analytical formula.

- The Chinchilla scaling laws provide a principled framework for allocating compute: balance model size and dataset size optimally.
- GPU acceleration derives from batched matrix multiplication. Migrating to LibTorch requires changing only the compute layer.
- Regularisation (dropout, weight decay) is essential even at 0.83M parameters.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, et al. Attention is all you need. *NeurIPS*, 2017.

[2] A. Radford, J. Wu, R. Child, et al. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.

[3] T. Brown et al. Language models are few-shot learners. *NeurIPS 33*, 2020.

[4] J. L. Ba, J. R. Kiros, G. E. Hinton. Layer normalisation. *arXiv:1607.06450*, 2016.

[5] I. Loshchilov, F. Hutter. Decoupled weight decay regularisation. *ICLR*, 2019.

[6] J. Hoffmann et al. Training compute-optimal large language models. *arXiv:2203.15556*, 2022.

[7] J. Kaplan et al. Scaling laws for neural language models. *arXiv:2001.08361*, 2020.

[8] A. Karpathy. nanoGPT. *GitHub*, 2022. github.com/karpathy/nanoGPT.

[9] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. MIT Press, 2016.

[10] K. He, X. Zhang, S. Ren, J. Sun. Deep residual learning for image recognition. *CVPR*, 2016.

[11] R. Eldan, Y. Li. TinyStories: How small can language models be and still speak coherent English? *arXiv:2305.07759*, 2023.

[12] D. P. Kingma, J. Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

Quadtrix is available at https://github.com/Eamon2009/Quadtrix.cpp under the MIT License. This paper was prepared in 2026.