FuzzyTok: A Continuous and Dynamic Autoregressive Tokenizer via GPU-Accelerated Causal Dilated Convolutions

FuzzyTok Research Team

Abstract—Traditional Large Language Models (LLMs) rely heavily on discrete subword tokenization algorithms, such as Byte Pair Encoding (BPE), WordPiece, or SentencePiece. These methods introduce significant drawbacks, including static and rigid vocabularies, high sensitivity to spelling mistakes, sub-optimal representation of rare words (out-of-vocabulary issues), and complex parameter scaling overheads. This paper presents *FuzzyTok*, a continuous and dynamic tokenizer that entirely bypasses discrete vocabulary tables. FuzzyTok reads raw UTF-8 bytes directly and maps them to continuous, context-aware latent embeddings using an stack of 1D Causal Dilated Convolutions. To achieve production-grade performance, a highly optimized GPU kernel is implemented in OpenAI Triton featuring full channel tiling, vectorization, and hardware-native Tensor Core arithmetic. Our empirical evaluation on an NVIDIA Tesla T4 GPU demonstrates that FuzzyTok achieves a 4.46$\times$ training speedup over naive Triton kernels, yields up to a 38.1% reduction in video memory (VRAM) usage compared to PyTorch Conv1D baselines (and up to 95.7% compared to causal Transformers), and reduces autoregressive cross-entropy loss to 0.4110 (yielding a perplexity of 1.51 versus BPE Conv1D's 8.38, LSTM's 9.32, and Transformer's 11.64). Furthermore, strict causality is mathematically verified, proving that no future information leaks during representation learning.

Index Terms—Dynamic Tokenization, Causal Convolutions, Triton, GPU Kernels, Tensor Cores, Deep Learning Systems.

1 Introduction

Modern NLP systems are built on top of discrete tokenization frameworks that segment continuous textual strings into integers from a pre-computed static vocabulary. This approach has led to major architectural challenges. BPE-based models partition words in a way that often discards morphological structure (e.g., separating "pre-processing" into arbitrary sub-tokens like "pre", "pro", and "cessing"). This fragmentation compromises the semantic alignment of morphologically related words and creates vulnerability to noise, typos, and adversarial inputs.

Additionally, embedding tables scale linearly with the vocabulary size V, taking up a significant portion of parameter budgets in smaller models. Conversely, resolving these challenges at the raw byte level using recurrent neural networks (RNNs) or standard Transformers is computationally expensive due to the quadratic sequence length expansion.

To overcome these challenges, this paper introduces **FuzzyTok**, a continuous, dynamic, and fully autoregressive character-level tokenization framework. FuzzyTok processes raw byte sequences directly through causal convolutions with exponentially growing dilations. It projects these characters into continuous latent vectors $z_t \in \mathbb{R}^D$ which act as dynamic embeddings. This process eliminates vocabulary tables and handles out-of-vocabulary (OOV) inputs natively.

The primary contributions of this work are:

- A detailed mathematical framework of FuzzyTok, proving its strict autoregressive causality.
- The design of a custom low-level GPU kernel in OpenAI Triton that implements 2D Channel Tiling, reducing memory bandwidth bottlenecks by a factor of 32$\times$ and enabling native Tensor Core arithmetic.
- A comprehensive benchmark on an NVIDIA Tesla T4 GPU, highlighting significant improvements in loss convergence, training speed, and memory usage.

—

Manuscript received June 23, 2026. Corresponding author email: support@fuzzytok.ai

2 Mathematical Framework

Let $X = (x_1, x_2, \ldots, x_T)$ be a sequence of raw UTF-8 characters (bytes) representing the input text, where $x_t \in \{0, 1, \ldots, V_{char} - 1\}$ and $V_{char} = 256$.

2.1 Continuous Latent Mapping

Instead of mapping x_t to a static index in a large vocabulary of size $V_{subword} \approx 32000$, a continuous character mapping is applied. First, each byte is projected into a dense character representation:

$$E = \text{LayerNorm}(\text{Embedding}(X)) \in \mathbb{R}^{B \times C_{in} \times T} \quad (1)$$

where B is the batch size, C_{in} is the channel dimension, and T is the temporal sequence length.

2.2 Causal Dilated Convolutional Stack

To capture context-aware multi-scale morphological relationships without leaking future information, the character representation is passed through a stack of Causal Dilated Convolutions.

For a single convolution layer with weight parameter $W \in \mathbb{R}^{C_{out} \times C_{in} \times K}$ (where K is the kernel size) and bias $b \in \mathbb{R}^{C_{out}}$, the output at output channel c_o and timestep t is defined as:

$$y_{c_o,t} = b_{c_o} + \sum_{c_i=0}^{C_{in}-1} \sum_{k=0}^{K-1} W_{c_o,c_i,k} \cdot E_{c_i,t-k \cdot d} \tag{2}$$

where d is the dilation factor. To ensure strict causality, a temporal mask is applied:

$$E_{c_i,t-k \cdot d} = 0 \quad \text{if} \quad t - k \cdot d < 0 \tag{3}$$

By stacking multiple Residual Causal Blocks with exponentially increasing dilations (e.g., $d \in \{1, 2, 4\}$), the effective receptive field (RF) of the tokenizer scales as:

$$RF = 1 + \sum_{l=1}^{L} (K-1) \cdot d_l \tag{4}$$

For $K = 3$ and two blocks with $d \in \{1, 2, 4\}$, the receptive field covers $RF = 43$ characters, allowing the tokenizer to model prefix-suffix relationships and word stems dynamically.

—

3 GPU Kernel Optimization via Triton

Naive GPU implementations of causal dilated convolutions suffer from severe memory bandwidth limitations. Since the temporal and channel dimensions are large, reloading the input tensor from Global Memory (VRAM) to the shared memory (SRAM) for every channel iteration creates a memory bottleneck.

3.1 2D Channel Tiling

To solve this, a custom GPU kernel is designed using OpenAI Triton. The kernel partitions both the input channels C_{in} and output channels C_{out} into 2D blocks of size $BLOCK\_C\_IN \times BLOCK\_C\_OUT$. The program grid is defined as:

$$\text{Grid} = (B, \lceil C_{out}/BLOCK\_C\_OUT \rceil, \lceil T/BLOCK\_T \rceil) \tag{5}$$

By loading a 2D tile of inputs $X_{tile} \in \mathbb{R}^{BLOCK\_C\_IN \times BLOCK\_T}$ and a weight tile $W_{tile} \in \mathbb{R}^{BLOCK\_C\_OUT \times BLOCK\_C\_IN}$ into SRAM, the global memory load overhead is reduced by a factor of $BLOCK\_C\_OUT$ (a 32× reduction for a tile size of 32).

3.2 Tensor Core Arithmetic Integration

Once tiles are residing in the register file and L1 cache, the convolution calculation for each kernel step k is executed as a high-speed matrix multiply-accumulate (MMA) operation using the hardware's native Tensor Cores:

$$\text{acc} \leftarrow \text{acc} + \text{tl.dot}(W_{tile}, X_{tile}) \tag{6}$$

where $\text{acc} \in \mathbb{R}^{BLOCK\_C\_OUT \times BLOCK\_T}$.

3.3 Strict Causal Pointer Clamping

To prevent GPU thread divergence and memory segmentation faults caused by out-of-bounds negative index calculation ($t - k \cdot d < 0$), a hardware-friendly clamping mechanism is implemented using Triton's conditional selection:

$$\text{causal\_mask} = t_{\text{offsets}} \geq k \cdot d \tag{7}$$

$$\text{clamped\_t} = \text{tl.where}(\text{causal\_mask}, t_{\text{offsets}} - k \cdot d, 0) \tag{8}$$

This clamping ensures that the GPU never issues memory requests to negative addresses, and the out-of-bounds positions are zero-masked out at the register level during the load step.

—

4 Experimental Evaluation

FuzzyTok was evaluated against a standard BPE-based Transformer embedding baseline. Both models were trained on the same corpus to predict the next character autoregressively. The experiments were conducted on an NVIDIA Tesla T4 GPU (16GB VRAM, Turing architecture) running CUDA 13.0 and PyTorch 2.4.

4.1 Ablation of Channel Tiling Speedup

Table 1 shows the throughput comparison between the non-tiled naive Triton kernel (subject to memory register spilling due to large tiles) and our optimized, register-safe channel-tiled Triton kernel.

TABLE 1
Throughput Optimization (Tokens/Second)

Config ($B \times T$)	Naive Triton (tok/s)	Tiled Triton (tok/s)	Speedup
1×128	36,491	56,186	1.54×
8×256	63,026	135,239	2.15×
16×512	63,319	142,790	2.25×
32×1024	43,969	195,930	4.46×

The optimized kernel achieves up to **4.46× throughput** in batch settings ($B = 32, T = 1024$), effectively utilizing the Turing GPU's Tensor Cores by keeping register usage within hardware bounds.

4.2 Convergence and Memory Footprint

The comparative training run for 40 epochs yields the metrics summarized in Table 2.

TABLE 2
Comparative Architectural Metrics

Metric	BPE Conv1D	Transformer	LSTM	Fuzz
Trainable Parameters	1,969,920	1,776,640	1,250,816	
Final Cross-Entropy Loss	2.1257	2.4547	2.2320	
Final Perplexity (PPL)	8.38	11.64	9.32	
VRAM ($B = 16, T = 1024$)	232.5 MB	763.8 MB	220.3 MB	1
VRAM ($B = 32, T = 2048$)	489.0 MB	9,240.2 MB	573.1 MB	3
Radical Cos-Sim Ratio	1.0917	1.0524	**1.1186**	
Causality Verification	Verified	N/A	N/A	**Verified**

FuzzyTok achieved a **80.7% lower cross-entropy loss** (0.4110) and a perplexity of **1.51** (compared to BPE

Conv1D's 8.38, LSTM's 9.32, and Transformer's 11.64). This demonstrates that the continuous mapping captures character dependencies more effectively than discrete tokenization.

Additionally, FuzzyTok saved **38.1% of GPU memory (VRAM)** compared to PyTorch Conv1D during training under $B = 16, T = 1024$. Under context-long setups ($B = 32, T = 2048$), FuzzyTok achieved **95.7% VRAM savings** over PyTorch's causal Transformer baseline (392.7 MB vs 9.24 GB), demonstrating the remarkable efficiency of Triton's memory block fusion.

—

5 Conclusion

This paper presented FuzzyTok, a continuous and dynamic tokenizer that reads bytes directly, bypassing traditional discrete vocabulary tables. By designing a custom, channel-tiled Triton kernel, global memory bandwidth bottlenecks were mitigated, achieving significant training speedups (4.46× over naive Triton kernels) and reducing VRAM usage substantially compared to standard PyTorch baselines.

The model's ability to achieve a 5.17× reduction in final loss while maintaining strict autoregressive causality makes FuzzyTok a viable, high-performance alternative to traditional BPE-based tokenizers for next-generation LLMs.

Appendix: Optimized Triton Forward Kernel

Listing 1 displays the complete, optimized Triton forward kernel code.

```
@triton.jit
def _causal_dilated_conv1d_fwd_kernel(
    input_ptr, weight_ptr, bias_ptr, output_ptr,
    in_channels, out_channels, seq_len, dilation,
    stride_ib, stride_ic, stride_it,
    stride_woc, stride_wic, stride_wk,
    stride_ob, stride_oo, stride_ot,
    HAS_BIAS: tl.constexpr, KERNEL_SIZE: tl.constexpr,
    BLOCK_T: tl.constexpr, BLOCK_C_OUT: tl.constexpr, BLOCK_C_IN: tl.constexpr
):
    pid_b = tl.program_id(0)
    pid_co = tl.program_id(1)
    pid_t = tl.program_id(2)

    co_start = pid_co * BLOCK_C_OUT
    t_start = pid_t * BLOCK_T

    co_offsets = co_start + tl.arange(0, BLOCK_C_OUT)
    t_offsets = t_start + tl.arange(0, BLOCK_T)

    co_mask = co_offsets < out_channels
    t_mask = t_offsets < seq_len
    acc = tl.zeros([BLOCK_C_OUT, BLOCK_T], dtype=tl.float32)
    input_base = input_ptr + pid_b * stride_ib

    for c_start in range(0, in_channels, BLOCK_C_IN):
        c_offsets = c_start + tl.arange(0, BLOCK_C_IN)
        c_mask = c_offsets < in_channels
        w_mask = co_mask[:, None] & c_mask[None, :]

        for k in tl.static_range(KERNEL_SIZE):
            src_t = t_offsets - k * dilation
            causal_mask = src_t >= 0
            valid_mask = c_mask[:, None] & (causal_mask & t_mask)[None, :]
            clamped_t = tl.where(causal_mask, src_t, 0)

            x_addrs = input_base + c_offsets[:, None] * stride_ic + clamped_t[None, :] * stride_it
            x_vals = tl.load(x_addrs, mask=valid_mask, other=0.0)

            w_addrs = weight_ptr + co_offsets[:, None] * stride_woc + c_offsets[None, :] * stride_wic + k * stride_wk
            w_vals = tl.load(w_addrs, mask=w_mask, other=0.0)

            acc += tl.dot(w_vals, x_vals)

    if HAS_BIAS:
        bias_vals = tl.load(bias_ptr + co_offsets, mask=co_mask, other=0.0)
        acc += bias_vals[:, None]

    out_base = output_ptr + pid_b * stride_ob
    out_addrs = out_base + co_offsets[:, None] * stride_oo + t_offsets[None, :] * stride_ot
    tl.store(out_addrs, acc, mask=co_mask[:, None] & t_mask[None, :])
```

Listing 1. Triton Forward Kernel implementation.