NIKTOK: FASTER TOKENIZERS NEED SMARTER NETWORK HARDWARE, NOT CPUS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tokenization running on the host CPU competes for the same cycles as agentic workflows with applications increasingly processing longer input contexts while relying on smaller models and tokenization delays making up a larger share of end to end inference. This overhead comes primarily from the host's network stack and OS, which handles every request before it reaches the tokenizer. To eliminate this overhead and free host CPU resources, we introduce NikTok, a system that runs tokenization entirely on network hardware. NikTok accelerates the pipeline, splitting each request between many parallel hardware execution units, overlapping computation and network transfer, with tokenized output directly being written to GPU memory without invoking the host CPU or the OS. Our evaluation using real network hardware across four widely used tokenizers (Qwen3-Embedding-0.6B, Harrier-270M, EmbeddingGemma-300M, and mmBERT-Embed) demonstrates that NikTok cuts average inference latency by up to 9.8× and increases throughput by up to 883%, while freeing up 83.5% of host CPU capacity.

1 INTRODUCTION

Tokenization is becoming an increasingly visible bottleneck in latency-sensitive language model serving. Agentic workflows and semantic routing increasingly rely on small, specialized models (Ong et al., 2025; Belcak et al., 2025), whose shorter execution times make input preprocessing a larger fraction of request latency. Long-context workloads further expose this cost: prefix caching reduces redundant GPU prefill computation but does not, by itself, eliminate tokenization of incoming text (Zhu et al., 2026; Crusoe AI Research, 2026). Meanwhile, host-side tokenization competes for CPU resources with agent tools, retrieval, and orchestration (Raj et al., 2025; Chung et al., 2026).

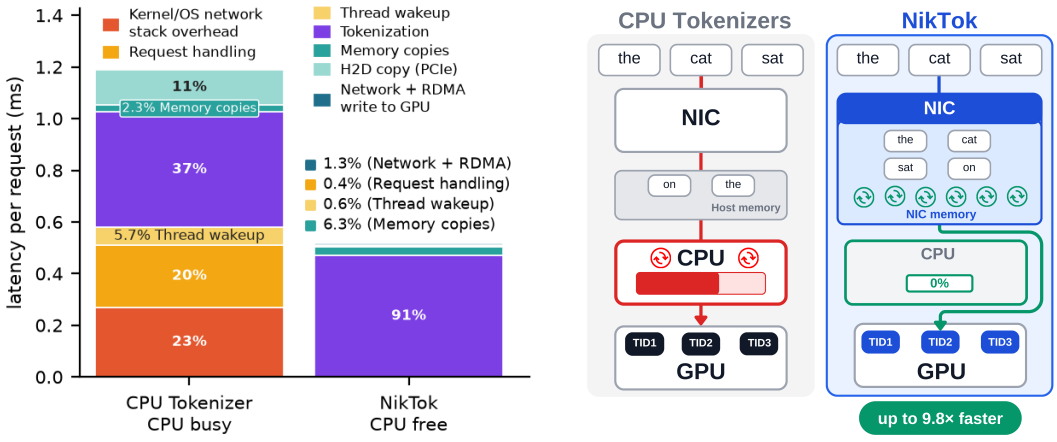


Figure 1: Overview of NikTok's efficiency vs. CPU tokenizers (left) and high-level architecture of NikTok avoiding CPU host entirely for more efficient tokenizer (right)

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The bottleneck extends beyond the tokenization algorithm itself. In a conventional serving pipeline, incoming text passes through host network-stack processing, request dispatch, and memory buffers before tokenization; the resulting token IDs must then be transferred to GPU memory. These surrounding costs contribute substantially to request latency (Section 1). CPU optimizations reduce processing overhead but continue to consume host cores (Choi et al., 2026; Shao et al., 2026; Crusoe AI Research, 2026), while GPU tokenizers use accelerator resources that could otherwise serve model execution (You, 2025; Kadamba & Jaisankar, 2026). Accelerating tokenization alone does not eliminate the overhead of its placement in the serving pipeline.

We argue that tokenization should instead be integrated into the network-to-GPU data path. Network hardware already sits on the request-ingress path, providing an opportunity to tokenize text before host-side processing. Executing tokenization there offers three complementary benefits: independent requests can be processed in parallel without consuming host CPU or GPU compute; token IDs can be delivered directly to GPU memory without intermediate host staging; and asynchronous delivery can overlap with the processing of subsequent requests. The advantage is therefore architectural: eliminating host-side processing and data movement can outweigh the lower per-core compute performance of network processors.

We present NikTok, an in-network tokenization framework that combines network-resident computation, using a programmable network hardware (a.k.a SmartNIC), with direct-to-GPU delivery using remote direct memory access (RDMA). At initialization, NikTok compiles tokenizer vocabularies, merge rules, and Unicode character properties into lookup tables. At runtime, persistent workers perform pretokenization and Byte Pair Encoding (BPE) using fixed-width integer lookups and preallocated worker-local state, avoiding dynamic allocation. Workers process independent requests in parallel and issue asynchronous RDMA writes to GPU memory, overlapping tokenization with output transfer across requests (Section 1). Together, these mechanisms eliminate using host CPU and memory from the tokenization data path.

Contributions. Our main contributions are:

- **Network-to-GPU tokenization architecture.** NikTok presents a generalizable architecture that integrate network-resident tokenization, persistent parallel workers, and asynchronous RDMA delivery into a host-bypass preprocessing pipeline. NikTok architecture can easily be adopted on different types and versions of network hardware.
- **Table-driven tokenization engine.** We reformulate pretokenization and BPE using precompiled lookup tables and bounded per-worker buffers, retaining reference merge ordering without runtime memory allocation.
- **Hardware evaluation.** We implement NikTok on NVIDIA BlueField-3 and evaluate four embedding-model tokenizers: Qwen3-Embedding-0.6B, Harrier-270M, EmbeddingGemma-300M, and mmBERT-Embed. We validate reference token-ID equivalence on the evaluated inputs. Compared with the host baseline, NikTok reduces average tokenization latency by up to $9.8\times$, increases throughput by up to 883%, and frees up to 83.5% of host CPU capacity.

2 Related Work & Preliminaries

SmartNIC and DPU Offloading. Several recent works investigate offloading components of the LLM inference pipeline to SmartNICs and Data Processing Units (DPUs) to mitigate host CPU and OS bottlenecks. Blink (Siavashi et al., 2026) offloads network transport, request dispatching, and tokenization/detokenization to a SmartNIC, while delegating scheduling to persistent GPU kernels. However, Blink executes its tokenizer on the SmartNIC's embedded ARM Cortex-A78 cores, which rely on conventional software execution, dynamic memory, and OS-managed threads that remain off the critical ASIC packet datapath. An early poster proposal, NICTokenizer (Galvankar et al., 2025), highlighted the conceptual benefits of offloading tokenization to programmable network hardware. As empirically characterized by Chen et al. (2024), a SmartNIC's on-path Data Path Accelerator (DPA) features wimpy single-thread compute, strict scratchpad SRAM limits, and no support for standard runtime allocations (`malloc`). In contrast to prior works that either remain confined to off-path ARM cores or leave datapath execution unaddressed, NikTok is the first to execute complete BPE tokenization directly on the DPA's parallel execution units (EUs), processing incoming packet buffers wire-speed and streaming tokens straight to GPU memory via RDMA.

Host-Level CPU Optimization and Kernel Bypass. Other works optimize host-side software to reduce tokenization overheads. For instance, PinTok (Choi et al., 2026) optimizes CPU tokenization via dedicated core pinning, hugepage pre-allocation, and DPDK-based network stack bypass to mitigate OS scheduling jitter and memory latency. While these software-centric techniques lower context-switch overhead and memory copying delays, they fundamentally rely on reserving scarce host CPU cores and memory bandwidth exclusively for preprocessing. In modern agentic AI workloads, host CPU capacity is already heavily saturated: tool execution and multi-step orchestration can consume up to 90.6% of end-to-end pipeline latency on the host (Raj et al., 2025; Zhu et al., 2026). Reserving dedicated host CPU cores solely for tokenization escalates this compute contention, ultimately stalling accelerator kernel launches and degrading multi-GPU serving throughput (Chung et al., 2026). NikTok eliminates this trade-off by shifting tokenization entirely out of host CPU space and into the network datapath, freeing valuable host CPU cycles for agentic workflows.

GPU-Accelerated Tokenization. Another line of research investigates offloading tokenization to GPUs (You, 2025; Kadamba & Jaisankar, 2026). However, the sequential branching logic and iterative merge operations inherent to BPE map poorly to the GPU's wide SIMT architecture. To extract parallelism, prior GPU solutions such as BlockBPE discard regex pre-tokenization, which compromises token-ID exactness and risks downstream model degradation, while simultaneously consuming valuable GPU streaming multiprocessors (SMs) needed for model forward passes. Crucially, GPU offloading fails to eliminate host-side overheads: incoming raw text must still traverse the network interface, be buffered in host DRAM, and be dispatched across the PCIe bus, leaving the host CPU and system bus on the critical path. Taking advantage of its strategic in-network placement, NikTok resolves this inefficiency by processing text directly within the SmartNIC datapath, eliminating intermediate host hops and guaranteeing exact, reference-compliant tokenization without expending accelerator compute.

Algorithmic Tokenizer Acceleration. Standard subword tokenization algorithms such as BPE (Sennrich et al., 2015), WordPiece (Schuster & Nakajima, 2012), and SentencePiece (Kudo & Richardson, 2018) rely on inherently sequential merge heuristics or longest-prefix matching. To alleviate these serial bottlenecks, recent efforts focus on parallelizing the tokenizer on the host. For example, LoPT (Shao et al., 2026) enables lossless chunk-level parallelization across multithreaded host CPUs via dynamic boundary alignment. However, while these algorithmic advancements reduce raw execution time, they remain fundamentally tied to host CPU cores and memory buses, continuing to contend for compute resources with concurrent agentic workflows. Rather than competing with these algorithmic techniques, NikTok provides an in-network architecture that executes tokenization on the network datapath before payload data reaches the host.

3 NikTok Overview

3.1 Architecture

This section describes the implementation details of NikTok, focusing on these key techniques. Figure 2 and 3 illustrate how NikTok maps tokenization onto the SmartNIC's DPA compute and memory hierarchy. NikTok differentiates itself from existing tokenization systems through three core co-design techniques: **1. On-path DPA offloading**, **2. Persistent DPA workers**, and **3. Overlapped Compute and RDMA**.

On-path DPA offloading To eliminate host-side CPU and memory-bus contention with concurrent host processes, NikTok offloads the compute-heavy BPE algorithm directly to the SmartNIC's ASIC-based DPA execution units (EUs). By dedicating a set of isolated, hardware-managed EUs on the network device, NikTok processes incoming text in-stream. Each EU is allocated its own private, statically bounded scratchpad SRAM to perform mutable BPE merge operations. Meanwhile, the tokenizer's vocabulary, merge rules, and BPE caches are stored in a shared, read-only memory region. Because all mutable merge states are strictly confined to an individual EU's private scratchpad, concurrent workers never require synchronization or locks. Consequently, tokenization throughput scales roughly proportionally with the number of allocated execution units up to the bandwidth limits of the shared memory subsystem, as evaluated in Section 4.

Persistent DPA Workers Rather than launching DPA workers for each packet event, NikTok keeps a worker resident on each execution unit. Workers are initialized once, bound to their receive queues, and remain active for the lifetime of the system. On packet arrival, a worker tokenizes the prompt using its private scratchpad SRAM, initiates an asynchronous RDMA write to GPU memory, and returns to a low-overhead wait state. Keeping workers resident eliminates per-request worker launch and scheduling overhead from the critical path.

Overlapped Compute and RDMA After tokenization, a worker posts the resulting token IDs to GPU memory using an asynchronous RDMA write rather than synchronously waiting for the transfer to complete. The RDMA operation is tracked independently through its completion event, allowing the worker to repost its receive buffer and process subsequent requests while earlier outputs are in flight. This pipelines tokenization with network transfer, hiding RDMA overhead across concurrent requests. Consequently, under sufficient load, steady-state throughput is governed primarily by the slower of tokenization and RDMA transfer rather than their sum. We evaluate the resulting network overhead in Section 4.

Parallelism BPE's merge chain is sequential within each pretokenized span, as each merge can change the candidate pairs considered by subsequent merges. NikTok does not parallelize this merge chain. Instead, it exploits request-level parallelism by assigning independent requests to different DPA workers. Each worker maintains private merge state, allowing requests to be processed independently without synchronization.

3.2 DPA-Compatible Tokenization

NikTok adapts the BPE procedure used by the HuggingFace tokenizer to the DPA's constrained execution environment, where dynamic memory allocation is unavailable. Algorithm 1 shows the one-time host initialization, Algorithm 2 describes the persistent per-EU worker loop, and Algorithm 3 details the BPE merge procedure executed by each worker.

At initialization, the host converts the tokenizer vocabulary and merge rules into fixed-width integer representations and copies the resulting lookup tables to DPA-accessible memory. Incoming bytes are first mapped to their corresponding 32-bit token IDs using a 256-entry lookup table.

For each BPE merge rule, NikTok represents a pair of adjacent 32-bit token IDs (u, v) as a 64-bit key, $(u \ll 32) \mid v$. The key is used to query a statically allocated hash table containing the merge rank and resulting 32-bit token ID. After a merge, the resulting token ID replaces the pair and participates in subsequent merges. This representation provides expected constant-time merge-rule lookup using fixed-width keys while avoiding dynamic allocation during DPA execution.

Because the DPA does not provide the regular-expression support used by conventional pretokenizers, NikTok implements a table-driven pretokenizer using pre-generated Unicode tables. Rather than copying substrings, it records $(\mathrm{offset}, \mathrm{length})$ spans into the receive buffer. Each worker maintains its BPE symbol state and 4-ary min-heap in statically allocated scratch space. The heap orders candidate merges by rank and position; after each merge, the worker updates only the affected neighboring candidates and skips entries invalidated by earlier merges. This design preserves the reference BPE merge procedure while executing without dynamic memory allocation on the DPA.

Algorithm 1 NikTok Initialization

$(vocab, merges, pretok_rules) \leftarrow$ **Parse**$(tokenizer.json)$
$(vocab_table, merge_table) \leftarrow$ **CompileForDPA**$(vocab, merges)$
CopyToDPAMemory$(vocab_table, merge_table, pretok_rules)$
InitializeReceiveQueues$()$
LaunchPersistentWorkers$()$ ▷ one worker per assigned EU

3.3 Design Analysis

When does DPA offloading help? A DPA execution unit is weaker than a host CPU core, so NikTok does not rely on faster per-request computation. Instead, its advantage comes from removing

Algorithm 2 Persistent DPA Worker

BindToReceiveQueue()
while true **do**
  WaitForReceiveEvent()
  while MessagePending() **do**
    $msg \leftarrow$ **NextPendingMessage**()
    $ids \leftarrow$ **NikTokTokenize**$(msg, byte_to_token, merge_table, pretok_rules)$
    if $ids \neq \emptyset$ **then**
      PostAsyncRDMAWrite(ids)
    end if
    RepostReceiveBuffer(msg)
  end while
end while

Algorithm 3 BPE Merge on a DPA Worker

Require: $input_ids, merge_table$
$symbols \leftarrow$ **BuildFixedSymbolList**$(input_ids)$ ▷ preallocated per-worker storage
$H \leftarrow$ **Empty4AryMinHeap**() ▷ fixed-capacity priority queue
PushInitialPairs$(symbols, merge_table, H)$
while $H \neq \emptyset$ **do**
  $(rank, pos, id) \leftarrow$ **PopMin**(H)
  if IsOutdated$(symbols, pos, rank, id)$ **then**
    continue ▷ candidate invalidated by an earlier merge
  end if
  MergeSymbols$(symbols, pos, id)$
  PushNeighborPairs$(symbols, pos, merge_table, H)$
end while
return ReadTokenIDs$(symbols)$

host-side processing from the critical path. Let $T_{\text{host}}(n)$ and $T_{\text{dpa}}(n)$ denote the tokenization time for an input of size n, and let $H_{\text{host}}(n)$ and $H_{\text{dpa}}(n)$ denote the remaining data-movement and system overheads for the two placements. Their end-to-end tokenization latencies are

$$L_{\text{host}}(n) = T_{\text{host}}(n) + H_{\text{host}}(n), \qquad L_{\text{dpa}}(n) = T_{\text{dpa}}(n) + H_{\text{dpa}}(n). \tag{1}$$

Thus, DPA offloading improves latency whenever

$$\boxed{T_{\text{dpa}}(n) - T_{\text{host}}(n) < H_{\text{host}}(n) - H_{\text{dpa}}(n)} \tag{2}$$

That is, the DPA need not tokenize faster than the host: any additional compute cost can be offset by eliminating host network-stack processing, intermediate memory movement, and the subsequent host-to-GPU transfer. In NikTok, the remaining output transfer is a direct RDMA write from the SmartNIC to GPU memory.

Request-level parallelism. BPE remains sequential within a request, so NikTok scales by processing independent requests across multiple DPA execution units. Let $N(n)$ denote the number of tokens produced for an input of size n. The ideal compute-limited token throughput of E workers is

$$R_{\text{compute}}(n, E) = \frac{E\, N(n)}{T_{\text{dpa}}(n)}. \tag{3}$$

provided sufficient concurrent requests are available. In practice, aggregate throughput is bounded by both compute and shared system resources:

$$R(n, E) \leq \min\left(R_{\text{compute}}(n, E), R_{\text{mem}}, R_{\text{rdma}}\right), \tag{4}$$

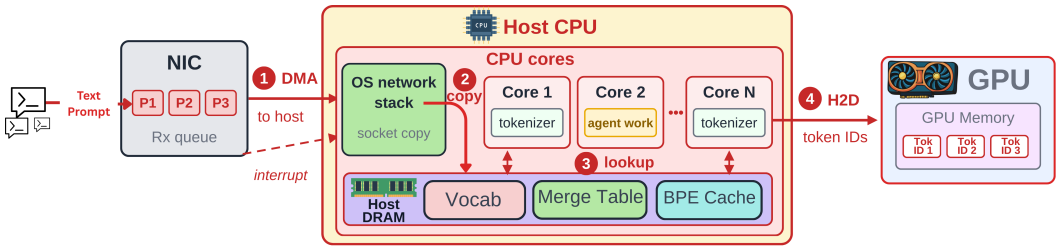


Figure 2: The conventional host-based tokenization incurs multiple redundant data copies and host OS/CPU overheads

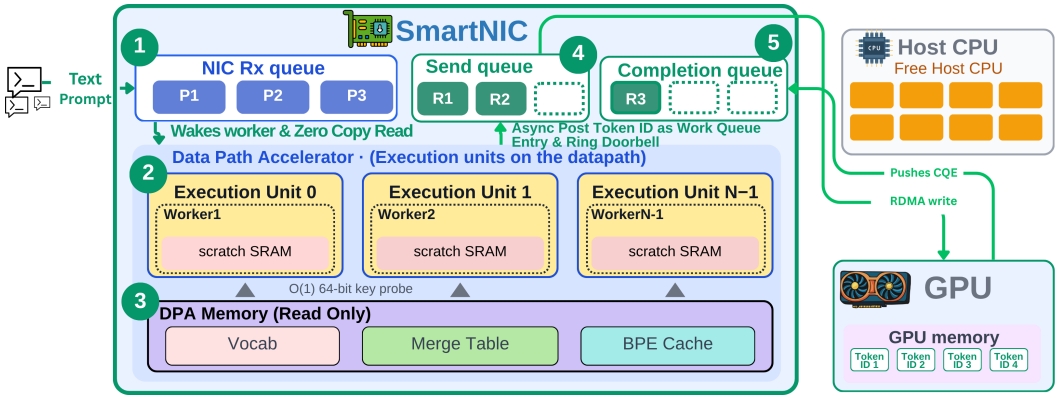


Figure 3: NikTok in-network tokenization fully bypasses and reclaims the host CPU and memory.

where R_{mem} and R_{rdma} capture the limits of the shared memory subsystem and RDMA path, respectively. We evaluate this scaling behavior by varying the number of active DPA workers in Section 4.

Fixed-memory execution. Unlike conventional host implementations, a DPA worker cannot grow its working state dynamically during tokenization. For an input of size n, each worker therefore requires a statically bounded memory budget

$$M_{\text{worker}}(n) = M_{\text{symbols}}(n) + M_{\text{heap}}(n) + M_{\text{output}}(n) \leq M_{\text{budget}}. \tag{5}$$

NikTok satisfies this constraint by storing the symbol state, merge priority queue, and output IDs in preallocated per-worker buffers. Consequently, the maximum supported request size is determined by these fixed capacities rather than by runtime memory allocation.

4 Experiments

We evaluate NikTok as a standalone tokenizer as well as a drop in integration within an end to end model serving pipeline. All experiments were conducted on a dedicated 2 node heterogeneous cluster (testbed details provided in Appendix C).

Baselines. We benchmarked NikTok's latency and throughput against the HuggingFace *Rust* tokenizer to quanitfy its gains due to its popularity and deployment in all major production inference engines across multiple systems, matching the legends in Figure 4: **NikTok on Bluefield-3's Data path Accelrator(DPA)** (190 execution units), **HF-Rust on an x86 host** 20 threads on the x86 host (reserving 4 threads for host OS background tasks) and **Blink on BF3 ARM Cores** Siavashi et al. (2026) (single core and all 16 general purpose Arm cores on BlueField-3 SmartNIC DPU distinct from the datapath accelerator NikTok itself runs on).

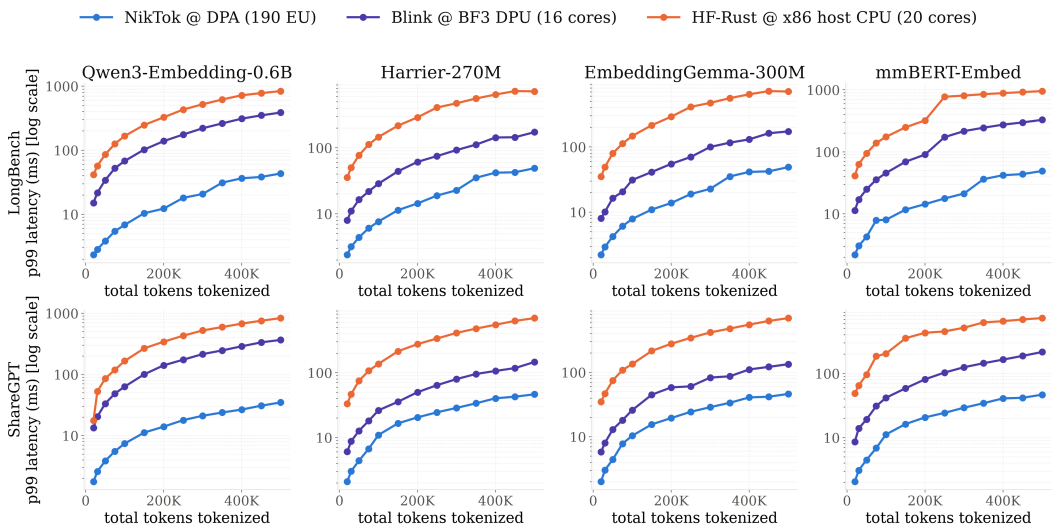


Figure 4: p99 tokenization latency vs. total tokens tokenized (log scale) for four embedding model tokenizers on LongBench (top) and ShareGPT (bottom).

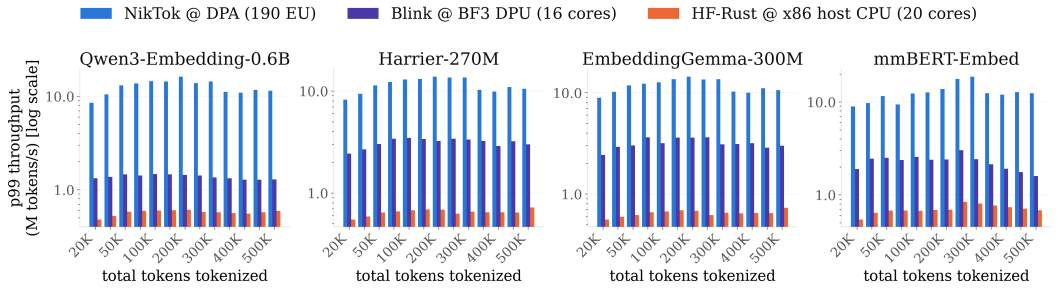


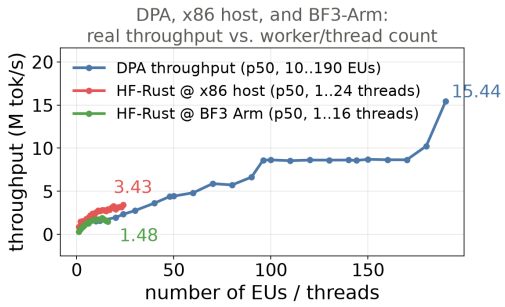
Figure 5: p99 tokenization throughput vs. total tokens tokenized on LongBench (log scale)

Workloads. We evaluate four production embedding model tokenizers: **Qwen3-Embedding-0.6B** Zhang et al. (2025), **Harrier-270M** Microsoft (2026), **EmbeddingGemma-300M** Schechter Vera et al. (2025), and **mmBERT-Embed** LLM Semantic Router (2026). These span range of pretokenizers while representing the small, embedding models that motivate this work (Section 1). We drive each system with two 4 MB corpora from LongBench Bai et al. (2024) and ShareGPT RyokoAI (2023), split by an identical whitespace aware rule into documents from 512 B to 64 KB concurrently sending 190 documents. ShareGPT being real multi turn chat conversations is a good representative of agentic workloads and longbench which includes long form documents is a good representation of RAG as well as long document summarization workloads. Figure 4 reports tail (p99) latency against document size for all four models on both corpora.

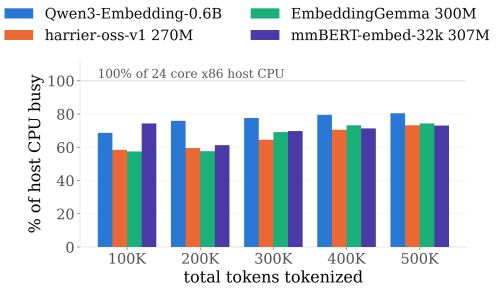
4.1 Tokenizer Benchmark Results

We benchmarked the performance of NikTok against the high performance Huggingface Rust tokenizer across different hardware. All results were evaluated over 20 independent trials. Figure 4 shows the tokenization latency distribution across both the datasets showing the 99th percentile (p99) tail latencies and Figure 5 reports the throughput (M tokens/s) as a function of the input document size (512B-64kB) for longbench.

For completeness all statistical numerical results like min, median latency (p50), 90th percentile (p90) and the throughput behavior for the ShareGPT dataset for all the configurations are provided in Appendix 7.



(a) Throughput as parallel worker count increase for NikTok, x86 host and DPU Arm cores.



(b) x86 Host CPU utilization (20 threads) for increasing token counts.

Figure 6: Ablation Studies

Table 1: End-to-end latency: RDMA vs. TCP

Tokens	RDMA (ms)	TCP (ms)	Slowdown
20K	1.59	47.26	29.7×
50K	3.68	49.63	13.5×
100K	7.18	52.57	7.3×
150K	10.45	54.54	5.2×
250K	16.07	60.99	3.8×
350K	22.49	70.28	3.1×
500K	33.32	76.10	2.3×

Table 2: Tokenization time: DPA vs. host CPU

Tokens	DPA (ms)	Host CPU (ms)	Speedup
20K	0.994	3.78	3.8×
50K	2.689	8.16	3.0×
100K	4.978	16.44	3.3×
150K	7.576	22.95	3.0×
250K	13.015	36.05	2.8×
350K	18.233	45.18	2.5×
500K	26.870	61.58	2.3×

4.2 Ablation Study

RDMA Overhead Table 1 helps isolate the performance gains from choosing RDMA as a transport mechanism for NikTok by replacing it with a host mediated TCP path keeping the NikTok's tokenization fixed. Without RDMA as the transport, latency would increase by 29.7× at 20K tokens and 2.3× at 500K tokens restating RDMA being a crucial design choice in NikTok's design. Table 2 isolates the DPA hardware's performance itself by running the exact same tokenization on a host CPU. The DPA is only 2.3–3.8× faster with the margin shrinking with size, because each execution unit is individually weaker than a host core and the gain comes from parallelism instead of per-core speed. Together the two ablations emphasize the the choices of using the parallel DPA execution units and RDMA for transport.

Scaling across execution units Figure 6a sweeps the number of persistent DPA worker threads from 12 to 190 EUs, 1 to 20 threads on an x86 host CPU system and 1 to 16 threads on the DPU ARM cores as per the availability of physical cores across the system. We note that even after the DPA cores being not as strong as corresponding x86 host cores NikTok outperforms the throughput performance solely by depth of parallelism.

Host CPU Liberation Figure 6b shows the Huggingface Rust Tokenizer's CPU utilization using mpstat. A continuous looped client shows a utilization between 50-80% for varying number of tokens across all the 4 models. NikTok avoids this overhead by offloading tokenization directly to the a SmartNIC freeing host CPU capacity for other tasks.

5 Future Work

NikTok demonstrates the ability of network devices to offload tasks that are typically performed by host CPUs. Building upon this architectural foundation, we identify following key avenues for future exploration:

Hierarchical Multi Device In-Network Pipeline. NikTok's pipeline decomposes naturally across network tiers. While the SmartNIC's DPA is well suited for stateful BPE merges, the preliminary regex based pre-tokenization stage consists primarily of pattern matching. Future work could offload pre-tokenization to upstream programmable switches (using P4 match-action pipelines) operating at line rate. This multi device decomposition would allow the network fabric to partition incoming byte streams into tokenizable spans before packets reach the server's SmartNIC, further amortizing DPA compute.

Tight Coupling with In-Network KV Caching. In modern agentic and long context serving systems, a majority of prompt tokens hit pre-existing prefix caches. Because prefix caches are indexed by token IDs, computing token sequences directly at the network boundary enables the SmartNIC to calculate prefix hashes immediately upon request arrival. We plan to integrate NikTok with in-network prefix cache pools, allowing the DPU to trigger asynchronous GPUDirect RDMA transfers of cached KV blocks to the GPU concurrently with tokenization, completely hiding latency.

Generalization to Non BPE Algorithmic Families. While BPE dominates generative models, several prominent encoder architectures and multilingual foundation models rely on WordPiece or Unigram/SentencePiece tokenization. Extending NikTok's table driven compilation to support dynamic programming based Viterbi paths and longest prefix matching within statically bounded DPA scratchpad memory will broaden the applicability of in-network preprocessing across arbitrary foundation models.

Dynamic Multi-Tenant Vocabulary Scheduling. Modern inference clusters routinely host mixture-of-models or semantic routing frameworks where multiple small models serve varying tasks concurrently. Future iterations can introduce dynamic memory banks and execution unit partitioning on the SmartNIC to maintain multiple active model vocabularies simultaneously, elastically reallocating DPA workers based on realtime traffic across different model endpoints.

Distributed NIC to NIC semantic routing Finally, NikTok opens new pathways for highly distributed network coordinated inference architectures particularly for cluster scale Mixture of Experts (MoE) serving. Paired with a lightweight on-NIC classifier would let the data plane steer each request to the most suitable node such as the one hosting the relevant experts in a cluster scale MoE deployment overlapping the hop with tokenization to reduce tail latency.

6 Conclusion

We demonstrate NikTok, a system that offloads BPE entirely off the host CPU onto the network data plane critical for accelerating agentic AI and semantic routing. We provide results from the experiments we conducted by implementing NikTok on an Nvidia Bluefield-3 SmartNIC's Data Path accelerator chip that demonstrate the performance gains generated by the offload. Our ablation study consisting of sweeping available execution units for demonstrating the gain from parallel execution, studying the effect of increasing document size on the time spent by Niktok in the network over RDMA proving that network bandwidth is ample and bottlenecked by compute instead of network and the streaming gains generated by integrating persistent hardware workers and asynchronous network compute overlap. NikTok's architecture shows that preprocessing such as tokenization belongs directly on the network and paves the way for efficient disaggregated serving by eliminating operating system overhead and redundant host to device transfers.

A promising future direction could be to couple NikTok with distributed KV caching architectures to bypass the host CPU and eliminate host side bottlenecks during context ingestion as in upcoming applications like Agentic AI majority of prefill compute is avoided by prefix caching. Furthermore, future works could explore expanding the DPA's role in handling concurrent multi tenant tokenization, dynamically reallocating execution units across multiple model vocabularies. Upcoming future generation of SmartNICs like Nvidia's Bluefield-4 NVIDIA (2026b) could leverage its 4x higher memory bandwidth NVIDIA (2026a) providing an opportunity to further extend NikTok's performance.

AI USE STATEMENT

We used generative AI tools to assist with parts of the implementation including writing and debugging experimental benchmarking code, to edit the paper for readability and to search for related literature. We did not use them to form hypotheses, design experiments, or interpret results. All experimental methodology and conclusions were determined by the authors who verified every AI assisted measurement against real hardware runs before inclusion. Other required disclosure tasks are not applicable. The authors reviewed and tested all AI assisted code, checked all citations against their sources and take full responsibility for the content of this work.

ETHICS STATEMENT

This work improves the efficiency of LLM tokenization. It involves no human subjects, new data collection or model training. We use public tokenizers and two public corpora under their original licenses: LongBench and ShareGPT, which is released under the creative commons license. We use these texts only as tokenizer input and do not analyze or redistribute their content. NikTok produces token IDs identical to the reference tokenizer so it does not change model outputs or safety behavior.

REFERENCES

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 3119–3137, 2024.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai, 2025. URL https://arxiv.org/abs/2506.02153.

Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pp. 1–12. IEEE, 2024.

Sean Choi, Myungheon Chin, and Ernest K Ryu. Pintok: Tokenizers deserve dedicated pinned cpu-compute and memory. 2026.

Euijun Chung, Yuxiao Jia, Aaron Jezghani, and Hyesoon Kim. Characterizing cpu-induced slowdowns in multi-gpu llm inference. *arXiv preprint arXiv:2603.22774*, 2026.

Crusoe AI Research. Reducing ttft by cpumaxxing tokenization. https://www.crusoe.ai/resources/blog/reducing-ttft-by-cpumaxxing-tokenization, 2026.

Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 51–66, 2018.

Shaunak Galvankar, Sudarshan Mehta, and Sean Choi. Nictokenizer: Tokenizer offload to a smartnic. In *Proceedings of the ACM SIGCOMM 2025 Posters and Demos*, ACM SIGCOMM Posters and Demos '25, pp. 79–81, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400720260. doi: 10.1145/3744969.3748428. URL https://doi.org/10.1145/3744969.3748428.

Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles*, pp. 121–136, 2017.

Venu Gopal Kadamba and Kanishkha Jaisankar. Gputok: Gpu accelerated byte level bpe tokenization. *arXiv preprint arXiv:2603.02597*, 2026.

Mikhail Khalilov, Salvatore Di Girolamo, Marcin Chrapek, Rami Nudelman, Gil Bloch, and Torsten Hoefler. Network-offloaded bandwidth-optimal broadcast and allgather for distributed ai. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–17. IEEE, 2024.

Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 756–771, 2021.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing, 2018. URL https://arxiv.org/abs/1808.06226.

ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. {ATP}: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 741–761, 2021.

Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 137–152, 2017.

Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 318–333. 2019.

LLM Semantic Router. mmbert-embed: Multilingual bert-style embedding model with 32k context and 2d matryoshka. Hugging Face model card, 2026. URL https://huggingface.co/llm-semantic-router/mmbert-embed-32k-2d-matryoshka. ModernBERT encoder, 307M params, 1800+ languages (Glot500), 2D Matryoshka (64–768d).

Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 15–28, 2017.

Microsoft. harrier-oss-v1: Multilingual text embedding models. Hugging Face model card, 2026. URL https://huggingface.co/microsoft/harrier-oss-v1-270m. Decoder-only embedding models (270M, 0.6B, 27B) with last-token pooling; MTEB v2 SOTA at release.

NVIDIA. Nvidia bluefield-4 powers new scale-in network infrastructure for agentic ai factories, aug 2026a. URL https://developer.nvidia.com/blog/nvidia-bluefield-4-powers-new-scale-in-network-infrastructure-for-agentic-ai-factories/.

NVIDIA. Nvidia bluefield-4 dpu datasheet, 2026b. URL https://resources.nvidia.com/en-us-accelerated-networking-resource-library/bluefield-4-dpu-datasheet.

NVIDIA Corporation. NVIDIA DOCA documentation: DPA subsystem. https://docs.nvidia.com/doca/sdk/dpa-subsystem/index.html. Accessed July 2026.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data, 2025. URL https://arxiv.org/abs/2406.18665.

Ritik Raj, Hong Wang, and Tushar Krishna. A cpu-centric perspective on agentic ai. *arXiv preprint arXiv:2511.00739*, 2025.

RAPIDS Team. cuDF documentation: Subword tokenizer. https://docs.rapids.ai/api/cudf/stable/. GPU-accelerated WordPiece tokenization. Accessed July 2026.

RyokoAI. Sharegpt52k, 2023. URL https://huggingface.co/datasets/RyokoAI/ShareGPT52K. Collection of approximately 52k–90k voluntarily shared ChatGPT conversations, CC0 license.

Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. Scaling distributed machine learning with {In-Network} aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 785–808, 2021.

Henrique Schechter Vera, Sahil Dua, Biao Zhang, Daniel Salz, Ryan Mullins, Sindhu Raghuram Panyam, Sara Smoot, Iftekhar Naim, Joe Zou, Feiyang Chen, Daniel Cer, Alice Lisak, Min Choi, Lucas Gonzalez, Omar Sanseviero, Glenn Cameron, Ian Ballantyne, Kat Black, Kaifeng Chen, Weiyi Wang, Zhe Li, Gus Martins, Jinhyuk Lee, Mark Sherwood, Juyeong Ji, Renjie Wu, Jingxiao Zheng, Jyotinder Singh, Abheesht Sharma, Divya Sreepat, Aashi Jain, Adham Elarabawy, AJ Co, Andreas Doumanoglou, Babak Samari, Ben Hora, Brian Potetz, Dahun Kim, Enrique Alfonseca, Fedor Moiseev, Feng Han, Frank Palma Gomez, Gustavo Hernández Ábrego, Hesen Zhang, Hui Hui, Jay Han, Karan Gill, Ke Chen, Koert Chen, Madhuri Shanbhogue, Michael Boratko, Paul Suganthan, Sai Meher Karthik Duddu, Sandeep Mariserla, Setareh Ariafar, Shanfeng Zhang, Shijie Zhang, Simon Baumgartner, Sonam Goenka, Steve Qiu, Tanmaya Dabral, Trevor Walker, Vikram Rao, Waleed Khawaja, Wenlei Zhou, Xiaoqi Ren, Ye Xia, Yichang Chen, Yi-Ting Chen, Zhe Dong, Zhongli Ding, Francesco Visin, Gaël Liu, Jiageng Zhang, Kathleen Kenealy, Michelle Casbon, Ravin Kumar, Thomas Mesnard, Zach Gleicher, Cormac Brick, Olivier Lacombe, Adam Roberts, Yunhsuan Sung, Raphael Hoffmann, Tris Warkentin, Armand Joulin, Tom Duerig, and Mojtaba Seyedhosseini. Embeddinggemma: Powerful and lightweight text representations. 2025. URL https://arxiv.org/abs/2509.20354.

Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5149–5152, 2012. doi: 10.1109/ICASSP.2012.6289079.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Wei Shao, Zheng Lingchao, Pengyu Wang, Peizhen Zheng, Li Jun, and Yuwei Fan. Lopt: Lossless parallel tokenization acceleration for long context inference of large language model. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33107–33122, 2026.

Mohammad Siavashi, Mariano Scazzariello, Gerald Q Maguire Jr, Dejan Kostić, and Marco Chiesa. Blink: Cpu-free llm inference by delegating the serving stack to gpu and smartnic. *arXiv preprint arXiv:2604.07609*, 2026.

Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 117–131, 2020.

Amos You. Blockbpe: Parallel bpe tokenization. *arXiv preprint arXiv:2507.11941*, 2025.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint*, 2025.

Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Tracelab: Characterizing coding agent workloads for llm serving. *arXiv preprint arXiv:2606.30560*, 2026.

7 APPENDIX

7.1 PRELIMINARIES

7.1.1 CAPABILITIES OF PROGRAMMABLE NETWORK DATA PLANES

Network data planes have stopped being fixed function over a decade now since the RMT (Reconfigurable match tables) architecture demonstrated that a switching ASIC can expose programmable parsers and reconfigurable match action tables while forwarding at line rate Bosshart et al. (2013) with P4 making these pipelines programmable in a high level language across Switches, SmartNICs and FPGAs leading to their widespread adoption. Since then the in-network computing domain has demonstrated a single programmable switch serving over 2 billion key value queries per second for over 64k items also improving the backend store's throughput upto an order of magnitude Jin et al. (2017), load balancing 10 million concurrent connections that replaces hundreds of software load balancers Miao et al. (2017) and aggregating distributed gradients for distributed training Sapio et al. (2021), Lao et al. (2021). Even on NICs, production clouds offload the entire virtual networking stack to FPGA SmartNICs Firestone et al. (2018), research systems offload key value stores Li et al. (2017), distributed file systems Kim et al. (2021), server orchestration Tork et al. (2020) as well as runnning distributed applications Liu et al. (2019). Modern SmartNICs such as Nvidia's Bluefield-3 are capable of pushing programmability into the packet datapath itself. It augments its NIC engines and Arm cores with a Data Path Accelerator (DPA), a cluster of 16 RISC-V cores at 1.8 GHz exposing 16 execution engines (EUs) per core behind a 1.5 MB last level cache Khalilov et al. (2024). DPA kernels react to arriving packets through hardware triggered events and access NIC queues and host memory as well as the DPU memory while bypassing both the host CPU and the DPU's Arm subsystem NVIDIA Corporation. Individual DPA cores are weaker as compared to host standards Chen et al. (2024), however tokenization is a table lookup bounded by memory accesses rather than arithmetic where vocabulary and merge tables occupy only megabytes and fit within NIC attached memory. The DPA is architected to hide memory latency behind hundreds of hardware threads which is exactly the type of hardware that is required for offloading a process like tokenization.

7.1.2 WHY IS OFFLOADING TOKENIZATION DIRECTLY TO THE GPU INEFFICIENT?

If tokenization was to be accelerated by the virtue of parallelization the obvious candidate would be the GPU as inference compute runs on it anyways and could offer higher degree of parallelism as opposed to a SmartNIC to free the host CPU cores. Prior work like Nvidia Rapids cuDF RAPIDS Team which accelerates wordPiece tokenization Schuster & Nakajima (2012) for encoder models and BlockBPE You (2025) which ports BPE merges to GPU thread blocks have already explored this. BPE is a greedy algorithm by nature which applies the merges such that every merge depends on the outcome of the earlier merges and the pretokenization regex matching is also a sequential branching logic which is naturally suited to run on a CPU. BlockBPE was only able to implement parallelism in this algorithm by discarding the regex matching stage and accepting a small loss in the generation quality generating tokens IDs that can deviate from the reference tokenizer Kadamba & Jaisankar (2026). Exactness would matter as the model trained on the reference tokenizer segmentation and lossless parallelization efforts consequently fallback to the CPU Shao et al. (2026). As GPU tokenization pays off in high batch offline regimes but not in latency sensitive serving **?** even the economics of offloading tokenization to the GPU are unfavorable. Tokenization is a byte granularity lookup workload with negligible arithmetic intensity, so every streaming multiprocessor it occupies is one taken from model kernels that monetize the accelerator. The streaming responses need to be detokenized one token per request per iteration making the decode side even worse leaving an effective batch size of one that leaves massively parallel hardware idle while still paying kernel launch overhead. As text enters the server through the NIC and is copied in host memory a GPU tokenizer would still requires host side request handling plus an additional transfer before prefill can begin. The host remains on the critical datapath and the offload adds a hop instead of removing one making the SmartNIC which sits upstream and every byte from every prompt already traverses a natural candidate for the offload.

7.2 TESTBED SETUP

The experimental testbed setup used for performing all the experiments in this work consists of a 2 node cluster connected via a direct point to point connection. The first node is an NVIDIA DGX Spark system powered by the GB10 Grace Blackwell Superchip. This system features a hybrid 20 core Arm CPU (comprising 10 Cortex X925 and 10 Cortex A725 cores) an NVIDIA Blackwell architecture GPU capable of up to 1 PetaFLOP of FP4 AI performance, 128 GB of unified LPDDR5x system memory (273 GB/s bandwidth) and 1 TB of NVMe storage. Network connectivity on this node is driven by an integrated NVIDIA ConnectX-7 SmartNIC. The second node is an x86 64-bit host server powered by an AMD Ryzen 9 3900X 12 core 24 thread processor operating at up to 3.8 GHz supplemented by 32 GB of system DDR5 RAM. To facilitate hardware accelerated networking and computational offloading this x86 server is equipped with an NVIDIA BlueField-3 SmartNIC consisting a Data Processing Unit (DPU). The BlueField-3 DPU features a programmable compute subsystem consisting of 16 Arm v8.2+ A78 cores and 32 GB of dedicated onboard DDR5 memory. It also integrates a dedicated Datapath Accelerator (DPA) subsystem an embedded hardware accelerator optimized for ultra low latency, parallel in-network compute and packet processing. The DPA comprises of 16 programmable RISC-V cores operating at 1.8 GHz leveraging hardware multithreading to scale up to 256 concurrent threads (16 execution units (EUs) per core). This accelerator operates with a memory hierarchy consisting of a 16 KB L1 cache per core, a 1.5 MB shared L2 cache and a 3 MB shared L3 cache and maps a 1 GB fast path execution region carved directly from the DPU's onboard DDR5 RAM to bypass host system bottlenecks. The DGX Spark and the x86 host are connected using a 40 Gbps direct connection over a QSFP cable providing a high bandwidth low latency communication path between the ConnectX-7 and the BlueField-3. The NikTokcodebase used in these experiments will be made publicly available to support reproducibility.

7.3 CORRECTNESS

NikTok can only be a drop-in replacement if it produces the exact same token IDs as the reference tokenizer for the same input. Embedding models are trained on one specific tokenizer's segmentation, so even a single differing token ID can feed the model something it never saw in training. Such Token ID mismatch does not crash or generate an error but it just silently degrades output quality making it easy to miss. We validate NikTok against the HuggingFace reference by comparing token ID sequences for the same input. Across all four tokenizers and both corpora, NikTok exactly reproduces the reference token IDs once inputs are in the tokenizer's expected form. Reaching this exactness required correcting defects which a naive conversion of strings to its integer form introduced as NikTok was designed to run on dedicated network hardware. Firstly to get the exact token IDs as the baseline we introduced another parameter, the "position" in the keys for entries in the BPE merge heap. This was implemented specifically as BPE merges byte pairs according to the priority but when there is a tie in priorities for merging it arbitrarily chooses the left node in the merge heap. To replicate this behavior in the integer only DPA our key on the heap had two parameters ¡priority, position¿ where the entry with lower position value could be utilized to break a tie in the merge priority. Secondly our approach introduced hash collisions when both the token IDs were 0, however that could mean a valid token $\rightarrow$ token ID mapping (0 is "!" in the Qwen3 Embedding 0.6B vocabulary). This was resolved by only marking a slot as empty when both the input IDs and the output ID for a merge were 0 since all real merges are guaranteed to have a non zero output ID. Unlike GPU BPE ports that discard pretokenization and accept quality loss You (2025), NikTok preserves exactness while running entirely in network hardware.

7.4 METRICS

We evaluate a tokenizer implementation on a corpus split into W independent documents $d_1, \ldots, d_W$ (the units of parallel work), processed by E workers (host threads, or DPA execution units). Let $T_{\text{tot}} = \sum_j |T(d_j)|$ be the total tokens produced and C the wall-clock time to finish the corpus.

Latency. For each document we time the *receive-and-tokenize* window,

$$\ell(d) \;=\; t_{\text{last token out}}(d) \;-\; t_{\text{first byte in}}(d), \tag{6}$$

identical across systems, and report its median ℓ_{50} and tail ℓ_{99} (Fig. 4).

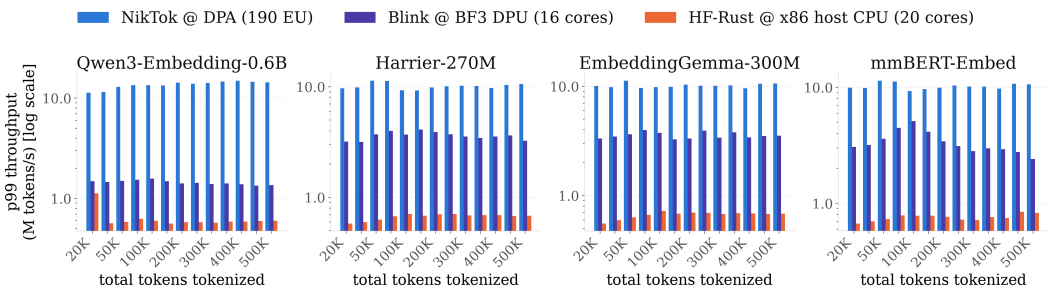


Figure 7: p99 tokenization throughput vs. total tokens tokenized on ShareGPT (log scale)

Throughput and speedup. Throughput is tokens per unit time, and speedup is its ratio to a baseline b on the same corpus:

$$R = \frac{T_{\text{tot}}}{C}, \qquad \text{Speedup}_b = \frac{R_{\text{nik}}}{R_b} = \frac{\ell_{50}^{b}}{\ell_{50}^{\text{nik}}}. \tag{7}$$

Because both systems tokenize the same text, the throughput ratio equals the latency ratio.

Scaling. To isolate how well the design uses more hardware we report per-worker throughput,

$$r(E) = \frac{R(E)}{E}, \tag{8}$$

which is constant in E under ideal linear scaling and decays under contention (Figs 6a).

Provisioning. Parallel speed is capped by how many documents are available to spread across workers. We define the *occupancy* $n = W/E$ (documents per worker) and call a run *fully provisioned* when $n \geq 8$. When work is evenly balanced, throughput follows

$$R = \frac{1}{\tau} \cdot \frac{W}{\lceil W/E \rceil}, \qquad \text{so} \qquad R \leq \frac{E}{\tau}, \tag{9}$$

where τ is the per-token cost: The ceiling E/τ is reached only once $W \gtrsim E$, and below it ($n < 1$) throughput falls with document count rather than tokenizer speed.

Tail determinism. We summarize latency predictability by the tail-to-median ratio

$$D = \frac{\ell_{99}}{\ell_{50}}, \tag{10}$$

where $D = 1$ means no tail.

Host-CPU liberation. Finally, over the K host cores with per-core busy fractions $u_k \in [0, 1]$, the mean utilization

$$U = \frac{1}{K} \sum_{k=1}^{K} u_k \tag{11}$$

measures the host cost of tokenization. A CPU tokenizer runs at some $U > 0$; NikTok runs at $U \approx 0$ on the host, freeing $K \cdot U$ cores for serving (Fig 6b).

Table 3: Qwen3-Embedding-0.6B (LongBench): latency (ms) and throughput (M tok/s) vs. total tokens tokenized.

Tokens	Latency (ms)				Thr. (M tok/s) at latency pct.			
	min	p50	p90	p99	min	p50	p90	p99
NikTok @ DPA (190 EU)								
20K	1.88	1.96	2.02	2.35	10.64	10.20	9.90	8.51
30K	2.37	2.61	2.65	2.85	12.66	11.50	11.32	10.53
50K	3.41	3.53	3.57	3.82	14.67	14.17	14.01	13.09
75K	5.00	5.07	5.09	5.46	15.00	14.79	14.74	13.74
100K	6.27	6.31	6.42	6.86	15.95	15.85	15.58	14.58
150K	9.28	9.38	9.45	10.4	16.17	15.99	15.87	14.41
200K	11.7	11.8	12.0	12.3	17.15	17.01	16.71	16.29
250K	16.7	16.7	17.6	18.1	15.00	14.94	14.21	13.82
300K	20.2	20.4	20.6	20.8	14.83	14.74	14.54	14.44
350K	30.3	30.7	30.9	31.2	11.54	11.41	11.31	11.20
400K	35.8	36.1	36.2	36.5	11.17	11.09	11.04	10.95
450K	35.8	37.3	37.7	38.3	12.56	12.07	11.94	11.74
500K	40.8	42.3	42.8	43.5	12.26	11.81	11.69	11.50
Blink @ BF3 DPU (16 cores)								
20K	13.8	14.3	14.6	15.1	1.45	1.40	1.37	1.33
30K	20.0	20.9	21.3	21.8	1.50	1.44	1.41	1.38
50K	32.3	33.4	33.8	34.2	1.55	1.50	1.48	1.46
75K	47.8	49.1	49.8	52.5	1.57	1.53	1.51	1.43
100K	62.6	64.6	65.4	68.0	1.60	1.55	1.53	1.47
150K	93.5	96.6	97.8	102.5	1.60	1.55	1.53	1.46
200K	125.3	130.5	131.7	138.9	1.60	1.53	1.52	1.44
250K	163.7	171.0	172.5	175.9	1.53	1.46	1.45	1.42
300K	210.2	216.9	219.0	220.6	1.43	1.38	1.37	1.36
350K	248.6	257.8	260.3	262.7	1.41	1.36	1.34	1.33
400K	293.5	303.6	306.3	311.2	1.36	1.32	1.31	1.29
450K	329.4	341.0	344.2	350.4	1.37	1.32	1.31	1.28
500K	363.7	377.9	381.3	387.8	1.37	1.32	1.31	1.29
HF-Rust @ host CPU (20 cores)								
20K	9.05	14.4	37.3	41.8	2.21	1.39	0.54	0.48
30K	16.5	22.8	51.3	57.2	1.82	1.32	0.59	0.52
50K	32.1	42.6	81.3	85.9	1.56	1.17	0.62	0.58
75K	50.8	59.4	120.8	126.3	1.48	1.26	0.62	0.59
100K	71.5	160.1	164.2	167.0	1.40	0.62	0.61	0.60
150K	103.5	140.4	245.1	248.5	1.45	1.07	0.61	0.60
200K	144.5	151.8	322.3	326.6	1.38	1.32	0.62	0.61
250K	184.4	215.8	426.0	432.4	1.36	1.16	0.59	0.58
300K	226.8	400.7	515.0	521.9	1.32	0.75	0.58	0.57
350K	282.3	592.2	609.2	620.0	1.24	0.59	0.57	0.56
400K	370.5	694.3	707.2	721.3	1.08	0.58	0.57	0.55
450K	372.2	405.9	769.6	779.9	1.21	1.11	0.58	0.58
500K	422.1	442.8	448.3	838.5	1.18	1.13	1.12	0.60

Table 4: Qwen3-Embedding-0.6B (ShareGPT): latency (ms) and throughput (M tok/s) vs. total tokens tokenized.

Tokens	Latency (ms)				Thr. (M tok/s) at latency pct.			
	min	p50	p90	p99	min	p50	p90	p99
NikTok @ DPA (190 EU)								
20K	1.53	1.64	1.69	1.77	13.10	12.22	11.86	11.32
30K	2.31	2.37	2.43	2.60	12.99	12.67	12.35	11.55
50K	3.55	3.66	3.70	3.86	14.09	13.67	13.52	12.96
75K	5.25	5.32	5.36	5.56	14.29	14.10	14.00	13.50
100K	7.15	7.19	7.23	7.43	13.99	13.91	13.84	13.46
150K	10.4	10.5	10.6	11.2	14.37	14.27	14.19	13.37
200K	13.0	13.1	13.3	14.0	15.41	15.32	15.02	14.26
250K	16.0	16.4	17.3	17.9	15.63	15.24	14.42	13.94
300K	19.9	20.1	20.3	21.1	15.07	14.93	14.79	14.19
350K	22.4	22.6	22.9	23.9	15.59	15.45	15.31	14.66
400K	25.5	25.6	25.9	26.7	15.66	15.61	15.42	14.99
450K	28.8	29.9	30.2	30.9	15.61	15.06	14.90	14.56
500K	33.2	33.6	33.9	34.9	15.05	14.89	14.76	14.34
Blink @ BF3 DPU (16 cores)								
20K	11.9	12.3	12.7	13.4	1.69	1.63	1.57	1.49
30K	18.5	19.1	19.5	20.5	1.62	1.58	1.54	1.46
50K	30.8	31.9	32.4	33.4	1.62	1.57	1.54	1.50
75K	44.7	46.2	47.2	48.8	1.68	1.62	1.59	1.54
100K	59.4	60.9	62.3	63.3	1.68	1.64	1.61	1.58
150K	94.2	96.3	97.8	100.6	1.59	1.56	1.53	1.49
200K	129.8	134.6	136.1	141.0	1.54	1.49	1.47	1.42
250K	164.8	170.5	172.3	174.2	1.52	1.47	1.45	1.43
300K	203.4	209.6	211.4	215.3	1.48	1.43	1.42	1.39
350K	234.3	243.0	244.9	247.2	1.49	1.44	1.43	1.42
400K	267.9	281.8	284.2	288.5	1.49	1.42	1.41	1.39
450K	308.5	319.4	322.3	334.8	1.46	1.41	1.40	1.34
500K	344.6	358.4	360.9	367.0	1.45	1.40	1.39	1.36
HF-Rust @ host CPU (20 cores)								
20K	8.07	12.3	14.4	17.7	2.48	1.63	1.39	1.13
30K	14.5	21.4	48.9	53.0	2.07	1.41	0.61	0.57
50K	32.1	77.8	84.8	85.6	1.56	0.64	0.59	0.58
75K	46.5	81.6	117.9	119.3	1.61	0.92	0.64	0.63
100K	67.3	85.6	163.8	166.1	1.49	1.17	0.61	0.60
150K	102.8	186.2	264.4	267.1	1.46	0.81	0.57	0.56
200K	185.8	327.1	338.8	343.2	1.08	0.61	0.59	0.58
250K	193.8	308.1	421.1	429.6	1.29	0.81	0.59	0.58
300K	243.4	502.7	510.7	524.2	1.23	0.60	0.59	0.57
350K	270.7	572.1	588.2	595.6	1.29	0.61	0.60	0.59
400K	306.9	335.8	663.4	676.8	1.30	1.19	0.60	0.59
450K	352.6	721.1	746.9	755.7	1.28	0.62	0.60	0.60
500K	383.3	410.4	417.7	833.9	1.30	1.22	1.20	0.60

Table 5: Harrier-270M (LongBench): latency (ms) and throughput (M tok/s) vs. total tokens tokenized.

Tokens	Latency (ms)				Thr. (M tok/s) at latency pct.			
	min	p50	p90	p99	min	p50	p90	p99
NikTok @ DPA (190 EU)								
20K	2.06	2.12	2.16	2.41	9.67	9.40	9.22	8.27
30K	2.72	2.93	2.95	3.18	11.01	10.22	10.15	9.42
50K	4.06	4.13	4.23	4.40	12.34	12.13	11.84	11.38
75K	5.74	5.81	5.90	6.08	13.04	12.88	12.68	12.31
100K	7.31	7.41	7.44	7.65	13.65	13.47	13.42	13.05
150K	10.6	10.6	10.7	11.4	14.20	14.08	13.98	13.15
200K	13.3	13.4	13.7	14.4	15.01	14.92	14.58	13.90
250K	17.9	17.9	18.1	18.9	14.34	14.29	14.14	13.56
300K	21.6	21.7	21.9	22.8	14.35	14.29	14.14	13.62
350K	34.2	34.6	34.8	35.2	10.61	10.50	10.44	10.31
400K	41.0	41.1	41.5	42.1	10.16	10.13	10.04	9.90
450K	40.5	41.5	41.8	42.6	11.55	11.28	11.20	10.98
500K	46.2	48.0	48.4	49.1	11.25	10.84	10.75	10.58
Blink @ BF3 DPU (16 cores)								
20K	4.95	5.22	5.60	8.01	3.96	3.75	3.50	2.44
30K	7.21	7.56	8.11	11.0	4.10	3.92	3.65	2.68
50K	11.6	12.0	12.4	16.4	4.31	4.16	4.02	3.03
75K	16.9	17.7	18.2	21.9	4.42	4.22	4.09	3.41
100K	22.2	22.9	23.3	28.7	4.49	4.35	4.27	3.47
150K	32.8	33.7	34.4	44.1	4.56	4.44	4.35	3.39
200K	43.5	51.9	58.0	61.3	4.58	3.85	3.44	3.26
250K	62.1	63.6	64.7	75.0	4.12	4.03	3.96	3.41
300K	80.4	82.5	83.5	92.5	3.85	3.76	3.71	3.35
350K	93.1	94.8	97.1	111.9	3.90	3.83	3.74	3.24
400K	110.4	112.9	131.5	143.5	3.77	3.69	3.17	2.90
450K	123.6	125.1	126.7	144.8	3.79	3.74	3.69	3.23
500K	136.1	138.7	145.3	172.9	3.82	3.75	3.58	3.01
HF-Rust @ host CPU (20 cores)								
20K	7.37	16.0	33.8	35.7	2.66	1.22	0.58	0.55
30K	12.1	17.0	47.9	50.2	2.46	1.75	0.62	0.59
50K	24.8	68.3	76.2	77.1	2.01	0.73	0.65	0.64
75K	37.5	109.7	111.3	112.6	1.99	0.68	0.67	0.66
100K	50.8	57.2	139.1	146.3	1.96	1.74	0.71	0.68
150K	77.3	202.8	214.0	216.9	1.93	0.74	0.70	0.69
200K	110.3	274.9	283.3	289.8	1.81	0.73	0.70	0.69
250K	132.2	176.2	387.4	408.7	1.94	1.45	0.66	0.63
300K	164.2	207.9	344.8	472.8	1.89	1.49	0.90	0.66
350K	208.5	461.2	557.3	559.7	1.74	0.79	0.65	0.65
400K	264.8	626.0	636.5	642.1	1.57	0.66	0.65	0.65
450K	290.8	332.1	721.0	724.4	1.61	1.41	0.65	0.65
500K	277.7	333.3	356.9	717.9	1.87	1.56	1.46	0.72

Table 6: Harrier-270M (ShareGPT): latency (ms) and throughput (M tok/s) vs. total tokens tokenized.

Tokens	Latency (ms)				Thr. (M tok/s) at latency pct.			
	min	p50	p90	p99	min	p50	p90	p99
NikTok @ DPA (190 EU)								
20K	1.78	1.82	1.88	2.09	11.41	11.16	10.80	9.72
30K	2.71	2.80	2.83	3.03	11.05	10.70	10.58	9.88
50K	4.10	4.17	4.20	4.40	12.22	12.02	11.93	11.39
75K	6.44	6.50	6.53	6.72	11.75	11.64	11.59	11.26
100K	9.92	10.1	10.2	11.0	10.29	10.09	10.01	9.30
150K	15.4	15.4	15.7	16.6	10.01	9.97	9.82	9.28
200K	19.3	19.3	19.6	20.5	10.51	10.47	10.32	9.88
250K	23.3	23.4	23.7	24.5	10.66	10.62	10.49	10.11
300K	27.7	28.0	28.2	28.7	10.56	10.44	10.36	10.21
350K	32.4	32.7	32.9	33.8	10.62	10.51	10.45	10.18
400K	38.6	39.1	39.9	40.4	10.21	10.07	9.89	9.76
450K	40.7	40.8	41.7	42.8	10.92	10.88	10.65	10.39
500K	45.0	45.3	45.8	46.5	10.95	10.86	10.75	10.58
Blink @ BF3 DPU (16 cores)								
20K	4.38	4.74	5.17	6.05	4.40	4.08	3.74	3.19
30K	6.69	7.17	7.60	8.78	4.18	3.89	3.67	3.18
50K	11.2	11.7	12.1	12.7	4.21	4.05	3.90	3.72
75K	15.7	16.4	16.8	18.1	4.59	4.41	4.29	3.98
100K	23.7	24.4	24.8	26.3	4.10	3.99	3.92	3.69
150K	32.6	33.7	34.2	35.7	4.50	4.36	4.30	4.12
200K	44.1	45.4	46.3	50.1	4.42	4.29	4.21	3.89
250K	53.7	55.7	56.8	64.4	4.45	4.29	4.21	3.72
300K	67.1	68.6	69.7	79.9	4.21	4.12	4.06	3.54
350K	77.9	79.8	86.3	95.8	4.25	4.15	3.84	3.46
400K	89.4	91.7	93.2	106.7	4.24	4.14	4.07	3.55
450K	102.8	104.7	106.3	118.0	4.16	4.08	4.02	3.62
500K	114.3	117.1	129.0	146.6	4.16	4.06	3.69	3.24
HF-Rust @ host CPU (20 cores)								
20K	6.67	7.81	31.6	33.4	2.89	2.47	0.61	0.58
30K	9.58	15.9	44.7	46.8	2.92	1.76	0.62	0.60
50K	24.0	67.9	74.3	75.2	1.97	0.70	0.64	0.63
75K	29.4	40.2	102.4	107.2	2.45	1.79	0.70	0.67
100K	41.7	50.5	132.5	137.3	2.33	1.92	0.73	0.71
150K	69.2	118.2	209.5	214.5	2.12	1.24	0.70	0.68
200K	94.1	163.5	267.1	276.6	2.07	1.19	0.73	0.71
250K	118.5	166.8	323.5	337.5	2.02	1.43	0.74	0.71
300K	149.3	170.2	391.7	411.8	1.89	1.66	0.72	0.69
350K	178.3	466.1	476.2	479.8	1.86	0.71	0.70	0.69
400K	198.9	500.1	542.3	550.2	1.91	0.76	0.70	0.69
450K	221.9	536.5	617.6	630.8	1.93	0.80	0.69	0.68
500K	255.8	286.3	655.4	697.0	1.86	1.66	0.73	0.68

Table 7: EmbeddingGemma-300M (LongBench): latency (ms) and throughput (M tok/s) vs. total tokens tokenized.

Tokens	Latency (ms)				Thr. (M tok/s) at latency pct.			
	min	p50	p90	p99	min	p50	p90	p99
NikTok @ DPA (190 EU)								
20K	1.91	1.96	2.03	2.23	10.43	10.16	9.81	8.93
30K	2.71	2.79	2.86	2.93	11.05	10.74	10.47	10.22
50K	4.00	4.03	4.09	4.24	12.52	12.43	12.24	11.81
75K	5.69	5.81	5.88	6.09	13.15	12.88	12.73	12.29
100K	7.28	7.38	7.58	7.88	13.71	13.52	13.17	12.67
150K	10.5	10.6	10.9	11.0	14.25	14.10	13.79	13.65
200K	13.3	13.4	13.5	13.8	15.04	14.88	14.79	14.46
250K	17.9	18.0	18.3	19.0	14.36	14.26	13.99	13.53
300K	21.6	21.8	22.2	22.8	14.35	14.26	13.97	13.60
350K	34.1	34.5	34.7	35.3	10.65	10.54	10.45	10.28
400K	40.6	41.0	41.3	41.5	10.26	10.16	10.09	10.05
450K	40.5	41.6	41.9	42.3	11.56	11.26	11.18	11.08
500K	46.3	47.7	48.1	49.1	11.23	10.90	10.82	10.60
Blink @ BF3 DPU (16 cores)								
20K	4.99	5.26	5.76	8.06	3.92	3.72	3.40	2.43
30K	7.16	7.53	7.91	10.1	4.14	3.93	3.74	2.94
50K	11.5	11.9	12.4	16.4	4.32	4.17	4.00	3.03
75K	16.9	17.6	18.0	20.5	4.40	4.24	4.13	3.63
100K	22.4	23.0	23.4	31.2	4.44	4.32	4.24	3.19
150K	32.8	33.9	35.0	41.3	4.55	4.41	4.27	3.62
200K	43.3	44.6	45.5	55.1	4.60	4.47	4.38	3.62
250K	62.2	63.5	64.4	70.2	4.12	4.04	3.98	3.65
300K	80.5	82.9	88.9	100.2	3.85	3.74	3.49	3.09
350K	93.3	95.2	97.9	116.4	3.89	3.81	3.71	3.12
400K	110.2	112.1	113.4	130.8	3.78	3.71	3.67	3.18
450K	123.4	125.4	147.5	162.4	3.79	3.73	3.17	2.88
500K	137.1	139.2	141.2	173.2	3.79	3.74	3.68	3.00
HF-Rust @ host CPU (20 cores)								
20K	7.67	16.4	33.5	35.0	2.55	1.19	0.58	0.56
30K	10.6	16.6	42.5	49.4	2.81	1.78	0.70	0.60
50K	23.4	66.5	76.3	79.8	2.13	0.75	0.65	0.62
75K	38.7	109.7	111.3	112.6	1.93	0.68	0.67	0.66
100K	52.0	62.5	145.0	147.2	1.91	1.59	0.69	0.68
150K	76.0	90.1	213.4	215.6	1.97	1.66	0.70	0.69
200K	103.5	275.0	284.1	291.6	1.93	0.73	0.70	0.68
250K	134.7	386.1	393.1	412.5	1.90	0.66	0.65	0.62
300K	235.6	450.2	464.2	474.9	1.31	0.69	0.67	0.65
350K	223.4	529.2	552.6	561.1	1.62	0.69	0.66	0.65
400K	269.2	613.6	638.0	642.4	1.55	0.68	0.65	0.65
450K	292.9	338.1	341.1	720.0	1.60	1.38	1.37	0.65
500K	321.7	336.7	357.3	708.9	1.62	1.54	1.45	0.73

Table 8: EmbeddingGemma-300M (ShareGPT): latency (ms) and throughput (M tok/s) vs. total tokens tokenized.

Tokens	Latency (ms)				Thr. (M tok/s) at latency pct.			
	min	p50	p90	p99	min	p50	p90	p99
NikTok @ DPA (190 EU)								
20K	1.76	1.80	1.87	2.01	11.54	11.28	10.86	10.10
30K	2.70	2.79	2.83	3.04	11.09	10.73	10.58	9.85
50K	4.05	4.12	4.20	4.43	12.37	12.16	11.93	11.31
75K	6.38	6.44	6.53	7.82	11.86	11.75	11.59	9.68
100K	9.84	9.89	9.99	10.4	10.38	10.32	10.22	9.84
150K	15.3	15.4	15.4	15.6	10.05	10.02	9.97	9.88
200K	19.1	19.2	19.4	19.6	10.59	10.52	10.45	10.36
250K	23.1	23.3	23.4	24.4	10.76	10.66	10.60	10.15
300K	27.6	27.7	28.0	28.9	10.59	10.56	10.46	10.14
350K	32.5	32.6	33.2	33.7	10.59	10.55	10.36	10.21
400K	38.9	39.2	40.3	40.9	10.15	10.06	9.78	9.63
450K	40.4	40.8	41.1	41.9	11.00	10.90	10.80	10.60
500K	45.0	45.2	45.6	46.3	10.92	10.90	10.78	10.63
Blink @ BF3 DPU (16 cores)								
20K	4.38	4.80	5.18	5.81	4.41	4.02	3.73	3.32
30K	6.79	7.17	7.42	8.03	4.11	3.90	3.76	3.48
50K	11.2	11.6	12.1	13.0	4.23	4.06	3.92	3.64
75K	15.8	16.3	16.8	18.1	4.57	4.42	4.31	3.98
100K	23.8	24.4	24.9	25.9	4.09	3.98	3.91	3.75
150K	32.3	34.0	38.3	45.0	4.54	4.33	3.84	3.26
200K	43.8	45.5	49.3	58.6	4.45	4.28	3.96	3.33
250K	53.7	55.5	57.0	60.7	4.46	4.31	4.20	3.94
300K	66.8	68.6	69.9	83.2	4.24	4.12	4.04	3.40
350K	77.5	79.4	80.6	86.9	4.27	4.17	4.10	3.81
400K	89.8	92.0	102.8	111.2	4.22	4.12	3.69	3.41
450K	102.7	104.7	106.7	121.8	4.16	4.08	4.01	3.51
500K	114.5	116.6	118.3	134.3	4.15	4.08	4.02	3.54
HF-Rust @ host CPU (20 cores)								
20K	6.98	30.8	32.7	34.9	2.77	0.63	0.59	0.55
30K	9.56	40.8	45.7	46.8	2.92	0.69	0.61	0.60
50K	18.6	26.5	66.7	75.0	2.54	1.79	0.71	0.63
75K	30.8	39.1	99.4	108.7	2.34	1.84	0.73	0.66
100K	40.4	50.2	52.2	134.8	2.41	1.94	1.86	0.72
150K	69.3	79.7	207.9	215.6	2.12	1.84	0.71	0.68
200K	94.0	162.2	274.3	278.7	2.08	1.20	0.71	0.70
250K	118.2	315.4	340.0	344.3	2.03	0.76	0.70	0.69
300K	150.4	301.5	412.7	417.4	1.88	0.94	0.69	0.68
350K	176.7	382.9	474.5	479.2	1.87	0.86	0.70	0.69
400K	200.3	518.7	548.8	552.6	1.89	0.73	0.69	0.69
450K	227.6	257.3	619.4	629.6	1.88	1.66	0.69	0.68
500K	258.8	288.7	686.5	697.1	1.84	1.65	0.69	0.68

Table 9: mmBERT-Embed (LongBench): latency (ms) and throughput (M tok/s) vs. total tokens tokenized.

Tokens	Latency (ms)				Thr. (M tok/s) at latency pct.			
	min	p50	p90	p99	min	p50	p90	p99
NikTok @ DPA (190 EU)								
20K	1.93	2.01	2.14	2.21	10.24	9.83	9.23	8.94
30K	2.75	2.82	2.91	3.06	10.81	10.54	10.21	9.71
50K	4.03	4.06	4.13	4.28	12.34	12.25	12.04	11.62
75K	5.77	5.84	6.97	7.92	12.89	12.74	10.67	9.39
100K	7.23	7.34	7.57	8.05	13.73	13.53	13.12	12.33
150K	10.7	10.8	10.9	11.8	13.92	13.78	13.67	12.67
200K	13.5	13.6	13.8	14.4	14.74	14.64	14.37	13.81
250K	17.0	17.1	17.3	17.7	18.51	18.41	18.17	17.71
300K	20.1	20.3	20.9	21.3	19.95	19.72	19.19	18.79
350K	34.4	35.5	35.9	36.4	13.19	12.78	12.62	12.47
400K	40.7	40.9	41.2	42.1	12.44	12.38	12.27	12.01
450K	41.1	42.4	42.6	43.8	13.60	13.21	13.13	12.76
500K	46.3	48.0	48.3	49.2	13.24	12.78	12.70	12.47
Blink @ BF3 DPU (16 cores)								
20K	9.53	10.3	10.7	11.4	2.28	2.12	2.03	1.91
30K	14.7	15.6	16.2	17.1	2.85	2.69	2.60	2.46
50K	21.7	23.0	24.1	25.0	2.89	2.73	2.61	2.51
75K	30.5	32.3	33.0	35.7	2.77	2.62	2.56	2.37
100K	42.3	44.1	44.9	45.7	2.78	2.66	2.62	2.57
150K	62.8	64.3	65.2	69.4	2.64	2.57	2.54	2.39
200K	82.9	85.3	86.4	90.6	2.63	2.55	2.52	2.40
250K	155.5	165.0	169.1	172.8	3.37	3.18	3.10	3.03
300K	195.8	203.6	212.0	216.3	2.68	2.57	2.47	2.42
350K	217.2	232.0	239.1	245.5	2.41	2.26	2.19	2.14
400K	254.4	267.7	271.7	274.6	2.06	1.96	1.93	1.91
450K	278.8	288.4	293.6	298.6	1.88	1.82	1.79	1.76
500K	304.7	315.5	322.2	328.6	1.72	1.66	1.63	1.60
HF-Rust @ host CPU (20 cores)								
20K	8.43	12.4	31.6	41.1	2.64	1.80	0.71	0.54
30K	14.1	22.4	54.4	63.4	2.89	1.82	0.75	0.64
50K	30.4	36.0	88.7	94.4	2.11	1.78	0.73	0.68
75K	46.8	53.5	131.6	139.8	2.03	1.78	0.72	0.68
100K	63.0	70.6	164.8	174.8	1.87	1.67	0.72	0.68
150K	95.5	100.8	242.8	249.0	1.80	1.71	0.71	0.69
200K	123.7	129.7	316.3	321.6	1.80	1.72	0.70	0.69
250K	229.5	384.7	757.2	774.8	2.86	1.70	0.87	0.85
300K	265.7	420.2	772.9	808.6	2.47	1.56	0.85	0.81
350K	306.6	468.3	786.0	851.4	2.14	1.40	0.83	0.77
400K	351.9	502.4	641.2	886.2	1.86	1.30	1.02	0.74
450K	375.5	536.3	783.4	919.3	1.75	1.22	0.84	0.71
500K	407.9	550.1	847.8	955.1	1.61	1.19	0.77	0.69

Table 10: mmBERT-Embed (ShareGPT): latency (ms) and throughput (M tok/s) vs. total tokens tokenized.

Tokens	Latency (ms)				Thr. (M tok/s) at latency pct.			
	min	p50	p90	p99	min	p50	p90	p99
NikTok @ DPA (190 EU)								
20K	1.79	1.85	1.89	2.06	11.47	11.09	10.86	9.96
30K	2.75	2.84	2.87	3.09	11.08	10.73	10.62	9.87
50K	4.15	4.23	4.26	4.50	12.43	12.19	12.11	11.46
75K	6.56	6.63	6.68	6.88	11.78	11.65	11.57	11.23
100K	9.95	10.1	10.2	11.2	10.48	10.36	10.23	9.29
150K	15.4	15.6	15.6	16.3	10.22	10.15	10.12	9.69
200K	19.4	19.4	19.6	20.7	10.66	10.63	10.54	9.97
250K	23.1	23.3	23.6	24.4	10.98	10.88	10.75	10.42
300K	27.8	27.9	28.3	29.3	10.72	10.66	10.53	10.17
350K	32.6	32.8	33.5	34.5	10.73	10.66	10.45	10.15
400K	39.1	39.6	40.6	41.0	10.21	10.10	9.86	9.76
450K	40.8	41.2	41.6	41.8	11.03	10.91	10.81	10.76
500K	45.1	45.3	46.3	46.8	11.04	10.99	10.75	10.63
Blink @ BF3 DPU (16 cores)								
20K	7.54	7.89	8.37	8.61	3.52	3.36	3.17	3.08
30K	11.8	12.4	12.9	13.9	3.78	3.57	3.45	3.19
50K	17.6	18.3	18.8	19.2	3.94	3.80	3.69	3.60
75K	27.5	28.6	29.4	31.1	5.08	4.89	4.76	4.50
100K	38.8	40.3	41.1	42.0	5.52	5.33	5.22	5.11
150K	54.6	56.5	57.4	58.9	4.48	4.33	4.26	4.15
200K	76.1	78.8	80.0	81.3	3.66	3.53	3.48	3.42
250K	93.0	97.5	99.4	103.6	3.47	3.31	3.25	3.12
300K	116.4	119.9	121.3	125.5	3.05	2.96	2.93	2.83
350K	134.9	139.0	140.7	144.9	3.21	3.12	3.08	2.99
400K	153.0	160.0	161.7	164.9	3.16	3.03	3.00	2.94
450K	175.8	182.7	184.7	188.4	2.98	2.87	2.84	2.78
500K	200.8	210.7	214.0	216.6	2.61	2.49	2.45	2.42
HF-Rust @ host CPU (20 cores)								
20K	10.1	12.9	43.3	48.9	3.24	2.54	0.76	0.67
30K	13.9	21.9	61.7	64.8	3.28	2.07	0.74	0.70
50K	26.6	32.4	89.2	96.6	2.67	2.19	0.80	0.73
75K	51.9	62.8	178.7	186.5	2.83	2.34	0.82	0.79
100K	59.6	68.8	196.2	204.4	2.69	2.33	0.82	0.78
150K	102.3	119.4	340.3	355.2	2.72	2.33	0.82	0.78
200K	130.9	155.1	421.0	432.5	2.53	2.14	0.79	0.77
250K	154.8	172.2	434.3	453.2	2.13	1.91	0.76	0.73
300K	193.2	203.7	500.2	518.6	1.94	1.84	0.75	0.72
350K	228.4	239.3	610.1	626.9	2.09	2.00	0.78	0.76
400K	236.4	260.1	540.0	655.7	2.08	1.89	0.91	0.75
450K	281.8	304.6	680.9	698.0	2.10	1.94	0.87	0.85
500K	315.5	352.7	722.7	732.1	1.92	1.72	0.84	0.83