wordchipper: Fast BPE Tokenization with Substitutable Internals

Crutcher Dunnavant crutcher@zspacelabs.ai
Dilshod Tadjibaev dilshod@gmail.com

1 Abstract

Tokenization throughput bounds every LLM preprocessing pipeline — batch inference, dataset preparation, and context-window accounting all pass through the tokenizer first. A family of existing tokenizers (OpenAI's `tiktoken`[7], HuggingFace's `tokenizers`[8], GitHub's `bpe-openai`[13]) have explored performance tuning over various pre-tokenization lexer implementations and BPE merge algorithms. We present `wordchipper`[0], a Rust-based byte-pair encoder tokenizer for the OpenAI GPT-2 tokenizer family. Through a combination of strict allocation discipline, factoring along the implementation lines of the pre-tokenization and BPE merge algorithm choices, thread-local resources, and extensive metrics; we were able to achieve throughput speedups relative to `tiktoken-rs`[12] in rust on a 64 core machine of 4.3-5.7x (4 to 64 cores) for general regex BPE vocabularies, and 6.9x-9.2x when using custom DFA lexers for specific OpenAI vocabularies. Under python wrappers, we see a range of 2x-4x (4 to 64 cores) speedups over `tiktoken`[7]. The substitutable design yields a benchmark cross-product that reveals workload-dependent encoder selection and corpus-modulated performance inversion between algorithm families.

Contents

2 Introduction

Tokenization throughput bounds every preprocessing pipeline that feeds an LLM: batch inference,

dataset preparation, and context-window accounting all pass through the tokenizer first. The existing OpenAI-compatible tokenizers (`tiktoken`[7], HuggingFace `tokenizers`[8]) are Python-first and expose no substitutable internals.

`wordchipper`[0] is a Rust tokenizer for the OpenAI byte-level BPE family. Its contributions:

- *Substitutable two-phase architecture.* Spanning (pre-tokenization) and span encoding (BPE merge) are independent trait-backed axes, yielding a benchmark cross-product that isolates each algorithmic choice.
- *Three-tier lexer.* Runtime regex (`fancy-regex`[11], `regex-automata`[10]) through compile-time `logos`[9] DFAs, with a shared post-processing pass that corrects DFA longest-match vs. regex first-match divergences.
- *Zero-allocation fast path.* Retained buffers, pre-allocated outputs, and lock-free thread-local pooling eliminate heap traffic after setup.
- *2–9x end-to-end throughput* over `tiktoken` (Python to Rust), with logos DFA spanning at 14–21x over `fancy-regex`.

3 Background

3.1 Subword Tokenization

Text tokenization for neural language models evolved from word-level and character-level schemes toward *subword* representations that balance vocabulary coverage against sequence length. The first practical subword algorithm for NLP was introduced by Sennrich, Haddow, and Birch[1], who adapted the Byte Pair Encoding (BPE) compression heuristic of Gage[2] to neural machine translation. BPE training is iterative: given a corpus of pre-segmented words and their frequencies, it repeatedly merges the most frequent adjacent symbol pair, adding the merged symbol to the vocabulary until a target vocabulary size is reached[15]. The result is a vocabulary of subword units; common words appear as single tokens while rare words decompose into reusable pieces. This construction gives BPE a natural decomposition property: any string over the byte alphabet is encodable without an out-of-vocabulary fallback.

WordPiece[3], used in **BERT**[4], uses the same merge-based structure but substitutes a likelihood-maximization objective for the frequency heuristic. **SentencePiece**[5] reframes vocabulary construction as a unigram language-model pruning problem and additionally treats the raw byte stream as input, replacing spaces with a sentinel character ▁ to enable language-agnostic segmentation without a whitespace pre-tokenizer. These two alternatives diverge from BPE in ways that make their pre-tokenization incompatible with the OpenAI regex pipeline; wordchipper targets the BPE family exclusively.

3.2 Pre-Tokenization Byte-Level BPE

GPT-2[6] introduced two modifications to plain BPE that define the target class for wordchipper.

First, tokenization is preceded by a *pre-tokenization* step that splits input text into non-overlapping spans using a hand-crafted regular expression. This regex enforces linguistic constraints (contractions, punctuation adjacency, whitespace handling), before BPE merge rules are applied within each span.

Secondly, the vocabulary is defined over UTF-8 bytes rather than Unicode characters, guaranteeing a lossless encoding for any input without special unknown-token handling.

Span boundaries are never crossed by merges, so the BPE sub-problem is embarrassingly parallel across spans once the regex split is complete.

Successive OpenAI vocabulary releases (`r50k_base`, `p50k_base/edit`, `cl100k_base`, `o200k_base`) have retained this two-phase structure while extending the regex patterns to improve multilingual and code coverage[7].

The `cl100k_base` and `o200k_base` patterns rely upon extended regex negative-lookahead patterns; which complicate implementations of DFA longest-match semantics.

3.3 Related Work

OpenAI's `tiktoken`[7] is the reference implementation for OpenAI-style pre-tokenization BPE tokenizers. `tiktoken` is a public Python library with a Rust implementation core. `tiktoken` has a significant amount of performance tuning for batch parallel throughput. `tiktoken` makes use of thread-local lexer clones to avoid contention, but its encoder working memory uses of many small heap allocations.

`tiktoken-rs`[12] is an evergreen vendored fork of `tiktoken`, which adds support for Rust-specific crate features and dependency management, dropping the

Python components. For purposes of benchmarking, we use `tiktoken-rs` for Rust, and `tiktoken` for Python.

HuggingFace's `tokenizers`[8] is a broader library supporting **BPE**, **WordPiece**, and **SentencePiece**, also with a Rust core. `tokenizers` has high-quality native support for Rust cargo dependants. It covers the wider ecosystem of model families and is the standard choice for training pipelines, but its design prioritizes generality and research flexibility, rather than throughput.

GitHub's `bpe-openai`[13] was built as a demo of a high-performance OpenAI-style BPE (with direct support for `cl100k` and `o200k`). `bpe-openai` focuses on a novel BPE merge algorithm, and leveraging the `regex-automata`[10] crate.

4 Architecture

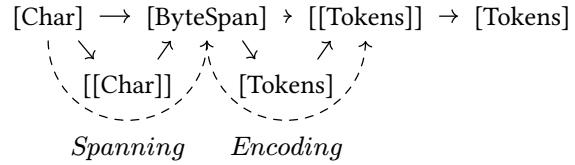


Diagram 1: Tokenizer Spanning / Encoding Flow

In `wordchipper`, a **spanner** partitions raw input text into a sequence of labeled spans, each carrying a byte slice and a classification: a **Special** token, a **Word**, or a **Gap**. The **spanner**'s sole responsibility is this segmentation; it has no knowledge of the token vocabulary or merge rules. A **span encoder** accepts a single word span and a vocabulary, and is responsible for resolving it to a sequence of one or more tokens via BPE merge. The **span encoder** owns whatever working memory its algorithm requires — scratch buffers, priority queues, linked list nodes — retaining that memory across calls to avoid repeated allocation on the fast path.

The central goal of `wordchipper` is to serve as its own experimental apparatus. Rather than committing to a single pre-tokenization strategy or BPE merge algorithm, the library defines stable interface boundaries at algorithmic decision points, permitting implementations to be substituted independently. This yields a benchmark space that is the cross-product of span encoder implementations, spanner backends, vocabulary configurations, and thread counts.

A heavy focus is also placed on memory allocation discipline, ensuring that the algorithm's working memory is efficiently managed and reused across calls. Memory discipline is enforced by hooking the allocator under benchmarking. Allocation counts are observable via `divan::AllocProfiler`[16], which instruments the global allocator, making heap traffic on the fast path directly measurable alongside throughput.

4.1 Encoder Setup

Procedure 1: EncodeTokens

```
1 procedure EncodeTokens(vocab, text)
2     let s ← GetSpanner()
3     let e ← GetEncoder()
4     let capacity ← ExpectedCapacity(text)
5     let tokens ← Allocate(capacity)
6     EncodeAppend(vocab, s, e, text, tokens)
7     return tokens)
8 end
```

`EncodeTokens` is the outer stage for encoding a text segment. It is responsible for acquiring spanner and span encoder resources, allocating a token buffer, and then delegating the actual encoding to `EncodeAppendTokens`.

Spanner lexer internals (such as regex buffers) and span encoder working memory are common sources of thread contention. To avoid contention, `EncodeTokens` acquires pre-allocated thread-local spanner and span encoder resources.

`EncodeTokens` then pre-allocates a token buffer using a statistical capacity estimate from the size of the input text, before calling `EncodeAppendTokens`. If the estimate holds, no buffer reallocations occur during encoding; if the estimate is too high, we allocate memory that is not used, if the estimate is too low, an allocation spill and copy will occur. The estimate is based upon population measures of observed byte-to-token rates.

The encoding fast path is designed to produce zero heap allocations after initial setup. Read-only data (the vocabulary, merge tables, and BPE Aho-Corasick automaton) is shared via `Arc` with no per-thread duplication.

4.2 Encode Append

Procedure 2: EncodeAppendTokens

```
1  procedure EncodeAppendTokens(
2      vocab, s : Spanner, e : SpanEncoder,
3      text, tokens (in/out))
4  for each (span, label) in s.Span(text):
5      if label == Special :
6          let token ← vocab.LookupSpecial(span)
7          assert token != nil
8          tokens.Append(token)
9      if label == Word :
10         let token ← vocab.LookupToken(span)
11         if token != nil :
12             tokens.Append(token)
13         else :
14             encoder.EncodeAppendSpan(
15                 vocab, span, tokens)
16 end
```

`EncodeAppendTokens` is the inner stage for encoding a text segment. It is responsible for iterating over pre-tokenization text spans produced by the spanner, resolving each span to a sequence of tokens via a span encoder, and appending the resulting tokens to the token buffer.

Within `EncodeAppendTokens`, the spanner drives iteration, classifying each span by label. Special tokens are resolved directly against the special vocabulary; single-token words are resolved by a direct vocabulary lookup. Only spans that cannot be resolved by either fast path are forwarded to `EncodeAppendSpan`, where the BPE merge algorithm executes. This design confines algorithmic variation to a single call site, with `AcquireSpanner` and `AcquireEncoder` as the two independent axes of policy substitution.

5 Lexer Implementations

A spanner's segmentation is driven by a **lexer** — an engine that applies the vocabulary's pre-tokenization pattern to raw input bytes. Because pre-tokenization is I/O-bound, scanning every byte of input, lexer throughput is a primary determinant of end-to-end performance. The spanner interface permits substitution of the underlying lexer implementation independently of the span encoder and vocabulary configuration, permitting controlled measurement of lexer throughput in isolation.

5.1 fancy-regex Lexer

The first lexer implementation in `wordchipper` was built upon the `fancy-regex`[11] crate, as used by `tiktoken`[7]. This regex library provides support for extended regex pattern negative lookaheads, which are used to implement the `(?!\S)` whitespace fixup pattern.

5.2 regex-automata Lexer

The `regex-automata`[10] crate compiles regular expressions into a DFA at runtime, avoiding the backtracking interpreter used by `fancy-regex`. It does not support lookaheads, so the `\s+(?!\S)` branches in the OpenAI patterns must be transformed before compilation. For each known pattern, wordchipper maintains a parallel version with the lookahead branches collapsed to `\s+` and applies a lightweight post-processing pass that truncates multi-character whitespace matches by one trailing character, reproducing the lookahead semantics.

This approach was motivated by `bpe-openai`[13], which demonstrated that `regex-automata` provides substantial speedups over `fancy-regex` for these patterns. In wordchipper, the `regex-automata` backend serves as a middle tier: 4-8x faster than `fancy-regex`, available for all patterns without per-pattern development, but slower than the compile-time logos DFAs. Search state is held in an external `Cache` object; `bpe-openai` shares a single cache behind a mutex, which introduces contention visible at 8+ threads. wordchipper instead distributes per-slot caches across its `PoolToy` thread pool, eliminating cache contention entirely.

`PoolToy` is a fixed slot locking cache, scaled for the local machine parallelism, with dispatch based upon a hash of the thread ID of the caller. Other tokenizers use similar mechanisms. This approach introduces some possibility of contention between threads, in the situation where hashed thread ids collide; but the complexity of explicit thread async dispatch (and tuning) has been deferred to future optimization.

5.3 logos DFA Lexers

The `logos`[9] crate compiles regex fragments into a DFA at build time via a derive macro, eliminating runtime compilation and backtracking. For `cl100k` and `o200k`, this yields 14-21x throughput over `fancy-regex` (250–300 MB/s single-threaded).

Two semantic gaps separate logos from the OpenAI patterns. First, logos has no lookahead support, so the `\s+(?!\S)` branches cannot be expressed directly. Second, logos uses longest-match semantics rather than regex first-match, which produces different span boundaries when Unicode category classes overlap across token branches (e.g. `\p{M}` appearing in both word and punctuation patterns for `o200k`). A shared post-processing state machine corrects both: it buffers whitespace tokens and, based on the classification of the following token, splits the last character into the next span (simulating the lookahead) or absorbs a preceding ASCII space into punctuation. Each logos token variant maps to a role enum that drives these rewrite rules, keeping the vocabulary-specific definitions (token enum, role mapping) separate from the shared correction logic. Three vocabulary families are implemented: `r50k_base`, `cl100k_base`, and `o200k_base`.

Correctness is validated by combinatorial equivalence testing. The Unicode predicates in the OpenAI patterns partition the codepoint space into 22 equivalence cells. The test harness generates all k-tuples $(k = 1..6)$ of 29 representative codepoints (594 million inputs per lexer) and asserts byte-identical span boundaries against the regex reference.

6 Span Encoder Implementations

6.1 BufferSweep Encoder

The `BufferSweep` encoder[Listing 1] is the reference implementation against which all other span encoders are benchmarked. Its working memory is a single retained token buffer, cleared but not deallocated between calls.

On each invocation, the working buffer is cleared, and then populated using `AppendByteTokens`. The encoder then iteratively scans the working buffer for the lowest-ranked adjacent pair, replaces the pair with the merged token in-place, and removes the right element. This continues until no mergeable pair remains, at which point the working buffer is appended to the output.

The algorithm is $O(n^2)$ in span length: each merge requires a full scan to locate the minimum pair, and up to n merges may be performed. This cost is acceptable in practice because BPE spans are short by construction. The implementation's value is in its simplicity — the behavior is unambiguous, making it the reference against which correctness of other encoders is verified.

6.2 TailSweep Encoder

The `TailSweep` encoder[Listing 2] applies the same $O(n^2)$ sweep algorithm as `BufferSweep`, but eliminates the separate working buffer by operating directly on the tail of the output buffer. The encoder carries no per-instance state.

On each invocation, the length of the tokens buffer is stored, and then the tail of the tokens buffer is populated using `AppendByteTokens`. The encoder then iteratively scans the tail of the buffer for the lowest-ranked adjacent pair, replaces the pair with the merged token in-place, and removes the right element. This continues until no mergeable pair remains, at which point the working buffer is appended to the output.

The tradeoff against `BufferSweep` is allocation pattern versus cache behavior. `TailSweep` avoids the separate working buffer and the final copy, but performs merges directly into the shared output buffer, which grows monotonically across spans. Whether this improves or degrades cache utilization relative to `BufferSweep`'s isolated working buffer is workload-dependent and is one of the questions the benchmark cross-product is designed to answer.

6.3 MergeHeap Encoder

The `MergeHeap` encoder[Listing 3] retains the $O(n^2)$ sweep structure of `TailSweep` but eliminates repeated pair vocabulary lookups by maintaining a parallel `p_ranks` buffer alongside the token buffer. This buffer caches the merge rank for every adjacent token pair, kept invariant across merges by recomputing only the two entries neighboring each merge site.

On each invocation, the length of the tokens buffer is stored, the tail of the tokens buffer is populated using `AppendByteTokens`, and `p_ranks` is populated with one optional rank entry per adjacent pair. Each merge iteration scans `p_ranks` for the minimum non-sentinel entry, performs the merge in-place on the token buffer, removes the corresponding `p_ranks` entry, and recomputes at most two neighboring rank entries.

The retained `p_ranks` buffer is the distinguishing feature of this encoder. Scans operate over a compact vector of rank values rather than requiring pair lookups on each pass, improving cache utilization relative to `BufferSweep` under concurrent workloads where multiple encoder instances are active simultaneously.

6.4 PriorityMerge Encoder

The `PriorityMerge` encoder[Listing 4] replaces the linear scan per merge round (as in Sennrich et al. [1]) with a standard binary min-heap that yields the lowest-ranked pair directly. The linked list gives $O(1)$ removal and neighbor access; the heap gives $O(\log n)$ extraction. Stale entries — invalidated by prior merges — are detected in $O(1)$ by comparing stored token identities against current node state, a standard lazy-deletion technique. Since each merge reduces the node count by one and enqueues at most two new pairs, total heap operations are linear in span length, giving $O(n \log n)$ per span. HuggingFace `tokenizers`[8] uses a structurally similar heap-and-linked-list approach.

6.5 BpeBacktrack Encoder

The `BpeBacktrack` encoder[Listing 5] works top-down rather than bottom-up: instead of starting from bytes and merging upward, it greedily matches the longest token at each position via the Aho-Corasick automaton[18] and backtracks when boundary validation fails. The key invariant is that a bitfield tracks candidate positions, and positions are only ever cleared, never re-set. This monotonicity bounds total backtracking to at most n steps across the entire span — each byte position is visited at most once going forward and at most once during backtracking. The prefix-chain walk at each position and the ValidPair decomposition check are bounded by vocabulary depth, independent of input length. Total work is $O(n)$ amortized in span length. The algorithm and its correctness argument derive from van Antwerpen and Neubeck[14].

7 Language Bindings

7.1 Python Bindings

The Python binding exposes `wordchipper` via `PyO3`[19], a Rust crate that generates a native CPython extension module. The extension is built and packaged using `maturin`[20], which handles ABI compatibility and PyPI publication. The binding is intentionally thin: the core `Tokenizer` and `TokenizerOptions` types are re-exported directly from the Rust library, with no reimplementation of encoding logic in Python. The GIL is released during encoding via `py.detach()`, permitting concurrent calls from Python threads without serialization through the interpreter lock.

7.2 WebAssembly Bindings

The WebAssembly binding is compiled via `wasm-bindgen` and `wasm-pack`. Because the core library is unconditionally `#![no_std]`, the same encoding and decoding code paths that run natively also compile to WASM with no conditional compilation or reimplementation. The binding crate depends on `wordchipper` with `default-features = false`, which disables `std`, `rayon`, and filesystem I/O. The absence of filesystem access requires vocabulary loading to be separated from parsing: a TypeScript wrapper fetches `.tiktoken` data over the network and passes raw bytes to a Rust-side parser that builds the vocabulary entirely from `&[u8]`, avoiding any platform-specific I/O.

8 Methodology

8.1 Machines

For benchmarking, we used 2 machines/os combinations:

The `amd3990X` benchmarks are for an Ubuntu 24.04 / AMD 3990X Threadripper 64-core/128-thread machine with 256G of system memory. This machine has 4 numa nodes, which we do not model or explore in our benchmarking, as core affinity is outside the scope of the `rayon` user APIs.

The `m2x12` benchmarks are for an OSX 14.3.1 / M2 12-core machine with 32G of system memory. This machine has a 8 performance / 4 efficiency core split, which we do not model or explore in our benchmarking, as core affinity is outside the scope of the `rayon` user APIs.

8.2 Datasets

Benchmarks use three corpora, chosen to exercise distinct performance regimes of the spanner and span encoder stages.

Single-string corpora. Two static text files ship with the benchmark suite in `dev-crates/wordchipper-bench/benches/data/`. `english.txt` (7 KB) is predominantly ASCII English prose. `multilingual.txt` (9 KB) is a purpose-built stress corpus spanning Latin-extended European languages, CJK (Simplified and Traditional Chinese, Japanese, Korean), Arabic, Hebrew, Thai, emoji sequences (including ZWJ families and flag pairs), structured data fragments (JSON, HTML), English contraction clusters, and mixed-script operator/symbol runs. Each file is repeated 10× at load time, yielding 70 KB and 90 KB working inputs respectively. The English corpus isolates encoder throughput on

inputs where the spanner produces short, vocabulary-friendly spans; the multilingual corpus forces longer multi-byte spans, higher codepoint diversity, and more frequent vocabulary misses, stressing both the regex engine's Unicode handling and the encoder's worst-case merge behavior.

Batch corpus. Parallel encoding benchmarks draw the first 10,240 documents from the FineWeb-Edu dataset (Karpathy's 100B-token shuffled subset[22] hosted on HuggingFace). Each document averages 4777 bytes, at a byte-to-token ratio of 4.8.

All three corpora are shared across the Rust and Python benchmark suites and across all vocabulary configurations (r50k_base, cl100k_base, o200k_base), ensuring that observed throughput differences are attributable to encoder, lexer, or binding overhead rather than to corpus variation.

8.3 Pattern Speed

When evaluating performance, the relative internal costs of BPE tokenizers shifts as functions of the computational complexity of the spanning patterns; as well as the size and complexity of the vocabulary.

We compare lexer and encoder performance against an example simple/fast lexer pattern (the `gpt2/r50k` pattern); and against a much more complex/slow pattern (the `o200k` pattern).

8.4 Rust Benchmarks

Rust benchmarks are implemented using `divan`[16], a statistical microbenchmark harness that reports latency distributions, throughput, and allocation counts per benchmark invocation. Allocation profiling is enabled globally via `divan::AllocProfiler`, making heap traffic on the fast path directly observable alongside timing.

The benchmark suite is structured to reflect the cross-product variation space of the library. The primary axes are: vocabulary configuration (`r50k_base`, `cl100k_base`, `o200k_base`), span encoder implementation (`BufferSweep`, `TailSweep`, `MergeHeap`, `PriorityMerge`, `BpeBacktrack`), spanner backend (Logos DFA, `regex-automata`, `fancy-regex`), and thread count. Spanning throughput and encoding throughput are measured in separate benchmark targets, isolating the two phases. Parallel encoding benchmarks operate over fixed-size batches drawn from the FineWeb-Edu[17] dataset, providing a realistic document-length distribution in place of synthetic inputs.

External comparisons include `tiktoken-rs` and HuggingFace `tokenizers`, benchmarked under identical corpus and thread conditions. `divan` output is machine-readable via a custom JSON pipeline (`bench-json`) that parses `divan`'s human-readable output into structured records, which are then consumed by a `plotters`-based report tool to produce the figures in this paper.

8.5 Python Benchmarks

Python benchmarks are implemented using `pytest-benchmark` and mirror the structure of the Rust benchmark suite, covering single-string and batch encoding across the same vocabulary configurations (`r50k_base`, `cl100k_base`, `o200k_base`) and the same FineWeb-Edu corpus. External comparisons include `tiktoken` and HuggingFace `tokenizers`, benchmarked under identical corpus and invocation conditions. Lexer acceleration is exposed as a benchmark axis via `TokenizerOptions`, permitting direct comparison of Logos DFA and `fancy-regex` backends from the Python layer. Batch encoding benchmarks additionally vary the parallelism strategy, comparing `rayon`-based internal parallelism against Python-level thread pool dispatch.

9 Span Encoder Evaluation

When comparing span encoder algorithms, we normalize the throughput against the fastest span encoder at a given thread count.

Naively, we would expect CPU cache bandwidth effects to have a higher relative impact on span encoder throughput when the lexer consumes a lower portion of the total time. Faster regex patterns / lexer implementations should be more sensitive to cache effects.

9.1 r50k Rust Relative Encoders

Lexer	Encoder	T=4	T=8	T=16	T=32	T=64
logos	buffer_sweep	1.00	0.99	0.96	0.97	0.99
logos	tail_sweep	1.00	1.00	1.00	1.00	1.00
logos	merge_heap	1.00	0.99	0.92	0.94	0.91
logos	priority_merge	0.85	0.85	0.85	0.85	0.84
logos	bpe_backtrack	0.88	0.89	0.92	0.93	0.95
regex-automata	buffer_sweep	1.00	0.99	1.00	0.94	1.00
regex-automata	tail_sweep	1.00	1.00	1.00	0.95	0.99
regex-automata	merge_heap	0.83	0.98	1.00	1.00	0.94
regex-automata	priority_merge	0.90	0.90	0.91	0.93	0.84
regex-automata	bpe_backtrack	0.96	0.95	0.95	0.98	0.95
fast-regex	buffer_sweep	1.00	1.00	1.00	0.99	0.96
fast-regex	tail_sweep	1.00	1.00	1.00	1.00	1.00
fast-regex	merge_heap	1.00	1.00	0.99	0.98	0.94
fast-regex	priority_merge	0.96	0.95	0.97	0.95	0.93
fast-regex	bpe_backtrack	0.97	0.98	0.99	0.98	0.96

Table 1: Relative Mean Throughput
amd3990X, r50k

For the fast pattern `r50k` (Benchmark 1), we see a strong preference for the simple buffer algorithms (`BufferSweep`, `TailSweep`, `MergeHeap`) under the fastest `logos` DFA lexer; more mixed results are seen when using the default `regex-automata` backend; and complex algorithms (`PriorityMerge`, `BpeBacktrack`) do relatively well under the slowest `fancy-regex` lexer.

9.2 o200k Rust Relative Encoders

Lexer	Encoder	T=4	T=8	T=16	T=32	T=64
logos	buffer_sweep	0.99	0.98	0.98	0.97	0.99
logos	tail_sweep	1.00	1.00	1.00	1.00	1.00
logos	merge_heap	1.00	1.00	0.91	0.93	0.92
logos	priority_merge	0.86	0.87	0.87	0.85	0.87
logos	bpe_backtrack	0.82	0.81	0.82	0.83	0.85
regex-automata	buffer_sweep	1.00	0.99	1.00	0.99	0.98
regex-automata	tail_sweep	1.00	1.00	1.00	1.00	1.00
regex-automata	merge_heap	0.99	1.00	1.00	0.98	0.97
regex-automata	priority_merge	0.90	0.90	0.92	0.90	0.84
regex-automata	bpe_backtrack	0.88	0.87	0.88	0.90	0.89
fast-regex	buffer_sweep	0.99	1.00	1.00	0.99	1.00
fast-regex	tail_sweep	1.00	1.00	0.99	1.00	1.00
fast-regex	merge_heap	1.00	1.00	1.00	0.99	1.00
fast-regex	priority_merge	0.98	0.98	0.97	0.97	0.97
fast-regex	bpe_backtrack	0.97	0.96	0.97	0.99	0.98

Table 2: Relative Mean Throughput
amd3990X, o200k

For the slow pattern `o200k` (Benchmark 3), we see a strong preference for simple buffer algorithms (`BufferSweep`, `TailSweep`, `MergeHeap`) under the fastest `logos` DFA lexer; intermediate preferences under `regex-automata`, and a collapse of distinctions under the slowest `fancy-regex` lexer. From this we conclude that lexer bottleneck effects can entirely mask performance differences between span encoders.

9.3 Corpus Encoder Effects

Single-string encoding throughput varies significantly with corpus composition. We compare an English prose corpus (70 KB after 10x repeat) against a diverse multilingual stress corpus (90 KB) spanning CJK, Arabic, Hebrew, Thai, emoji, and mixed-script runs. The D/E ratio (diverse throughput divided by English throughput) quantifies each encoder's sensitivity to multi-byte, vocabulary-sparse input.

External encoders (`tiktoken`, `bpe-openai`, `tokenizers`) show near-unity D/E ratios (0.92–1.03) because their bottleneck lies outside the BPE merge loop: regex overhead, Python interpreter cost, or allocation-heavy codepaths dominate, masking any corpus-dependent merge behavior.

Within `wordchipper`, the effect is pronounced and encoder-dependent. Under the fastest `logos` DFA lexer, the simple sweep algorithms (`buffer_sweep`, `tail_sweep`) retain only 36–38% of their English throughput on diverse `o200k` input (80–84 to 30 MB/s). `merge_heap` fares slightly better at 43%. By contrast, `priority_merge` and `bpe_backtrack` retain 76–78%, degrading from 106–109 to 83 MB/s. The pattern is consistent across all three lexer backends: under `regex-automata`, sweeps drop to 46–48% while `priority_merge` holds at 86%; under `fancy-regex`, the lexer itself dominates (11–18 MB/s) and encoder differences largely collapse.

The mechanism is span-length sensitivity. Diverse text produces longer multi-byte spans with higher codepoint diversity and more frequent vocabulary misses. Sweep-based algorithms scan linearly per merge round, so their cost scales with span length squared in the worst case. `priority_merge` and `bpe_backtrack` use rank-ordered data structures that skip over non-viable pairs, maintaining near-linear behavior regardless of span composition.

The effect scales with vocabulary complexity: under `logos`, `r50k` shows a moderate 0.69 D/E for sweeps, `cl100k` is similar at 0.69–0.70, while `o200k` collapses

to 0.36–0.38. Larger vocabularies produce more merge ranks, amplifying the sweep algorithms' quadratic worst case on unfamiliar input. The three-tier lexer view confirms that corpus sensitivity is an encoder property modulated by lexer speed: faster lexers expose more of the encoder's merge behavior, while slower lexers mask it behind their own overhead.

Encoder	r50k			cl100k			o200k		
	Eng	Div	D/E	Eng	Div	D/E	Eng	Div	D/E
tiktoken	14.2	14.1	0.99	11.7	11.7	1.00	11.7	11.5	0.99
bpe-openai	-	-	-	45.9	42.2	0.92	46.2	44.2	0.96
tokenizers	-	-	-	7.2	7.4	1.03	6.6	7.5	1.14
wc:logos:buffer_sweep	86.0	59.8	0.69	87.4	61.0	0.70	80.3	30.5	0.38
wc:logos:tail_sweep	86.5	59.7	0.69	87.5	60.8	0.69	84.2	30.2	0.36
wc:logos:merge_heap	92.5	69.2	0.75	93.9	62.0	0.66	90.6	38.7	0.43
wc:logos:priority_merge	109.8	91.6	0.83	115.0	90.9	0.79	109.3	83.2	0.76
wc:logos:bpe_backtrack	101.3	75.9	0.75	110.8	83.1	0.75	105.9	83.0	0.78
wc:ra:buffer_sweep	53.4	42.2	0.79	55.5	43.7	0.79	54.5	26.0	0.48
wc:ra:tail_sweep	54.1	42.7	0.79	56.1	44.1	0.79	55.5	25.5	0.46
wc:ra:merge_heap	56.0	46.9	0.84	58.1	44.5	0.77	57.7	31.2	0.54
wc:ra:priority_merge	61.5	56.5	0.92	65.2	57.1	0.88	64.1	54.8	0.86
wc:ra:bpe_backtrack	59.1	49.5	0.84	64.1	54.0	0.84	63.8	54.6	0.86
wc:regex:buffer_sweep	17.4	16.3	0.94	15.9	15.5	0.97	11.1	9.5	0.86
wc:regex:tail_sweep	17.5	16.4	0.94	16.0	15.4	0.96	11.1	9.5	0.85
wc:regex:merge_heap	17.7	17.0	0.96	16.1	15.5	0.96	11.2	10.2	0.91
wc:regex:priority_merge	18.3	18.1	0.99	16.6	16.8	1.01	11.5	11.8	1.03
wc:regex:bpe_backtrack	18.0	17.1	0.95	16.3	16.2	1.00	11.4	11.8	1.03

Table 3: Corpus encoder effects: English vs. diverse/ UTF-8 throughput (MB/s) and retention ratio (D/E) across vocabulary models.

9.4 Default Encoder

On the benchmark dataset, which does contain mixed language usage, `TailSweep` is the fastest algorithm. However, our examination of corpus-dependent effects shows that these algorithms are sensitive to corpus composition and vocabulary complexity.

We select `BpeBacktrack` as the default encoder in `wordchipper`, for its relative performance under pathological conditions. The cost of this selection is a 10% drop in throughput in normal operation, to avoid a 40%-60% drop under pernicious conditions.

10 Lexer Evaluation

The cost complexity of the spanning regular expression of a vocabulary has significant impact on performance behavior. More computationally complex expressions exhibit a larger share of the pipeline time being spent in the spanner, and see greater impact from custom DFA customizations.

The models in question (`gpt2/r50k_base`, `cl100k_base`, `o200k_base`) have regular expressions that, generally, grow in runtime complexity. This is reflected in the relative performance of lexer choices versus reference implementations.

In evaluating `wordchipper` lexers, we will be reporting the `TailSweep` performance against the shuffle dataset.

For `o200k`, `wordchipper::fancy-regex` has performance so comparable to `tiktoken` that the graphs overlap.

As `tiktoken`/`tiktoken-rs` both use thread local `fancy-regex` pools for lexing, and the actual span encoder algorithms are so similar to each other; we speculate that the performance difference lies in incremental heap allocation, and the choice of lookup keyspace in the vocabularies.

For small thread counts (≤ 8), `bpe-openai` outperforms the `fancy-regex` implementation. Investigating this performance led us to observe that `bpe-openai` is `regex-automata` based, but sits upon the builtin `regex-automata` thread local cache pools, which have a duplication limit of 8.

Our `regex-automata` implementation derives from this investigation; adding in the `regex-automata` lexer backend; and building a thread local resource pool which scaled up to machine thread counts. The resulting implementation outperforms `bpe-openai` significantly, without the change in scaling at T=8.

10.1 Rust Errata

We observe anomalously low throughput for `tokenizers` under our harness; we report these numbers for completeness but do not draw conclusions from them.

10.2 r50k Rust Throughput

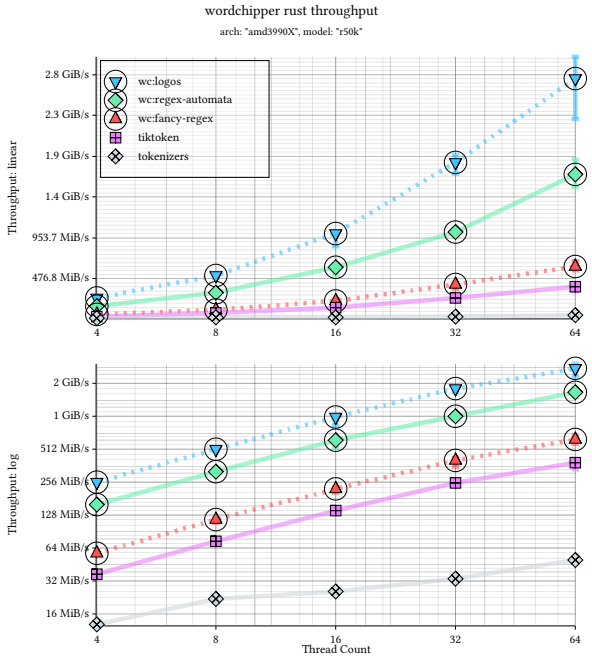
For the fast pattern `r50k` (Benchmark 4), the `logos`, `regex-automata`, and `fancy-regex` lexers all outperform `tiktoken-rs` and `tokenizers`. There is no `r50k` support for `bpe-openai` yet.

The `regex-automata` implementation (our default) consistently performs 4.3x-4.5x the throughput of `tiktoken-rs` on all thread counts.

The `fancy-regex` implementation (our fallback) is closest in performance to `tiktoken-rs`; generally performing 1.55x the throughput of `tiktoken-rs`.

For fast patterns, the `logos` DFA implementation is the fastest, ranging from 6.9-7.2x the throughput of `tiktoken-rs`; with a peak measured speed of 2.7 GiB/s.

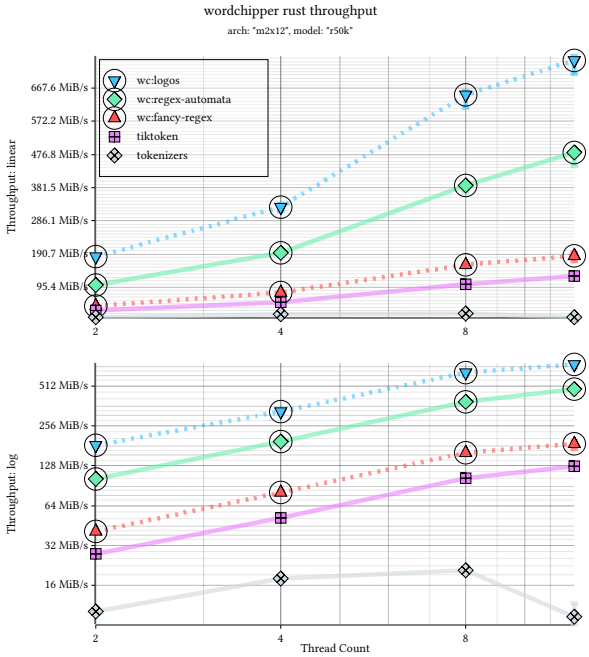
The Apple **M2** outperforms the **AMD3990X** at equivalent thread counts, but the relative performance behavior is roughly equivalent on the two systems.



Thumbnail 1: Min/Mean/Max Throughput
rust, amd3990X, r50k
See Benchmark 4

Name	T=4	T=8	T=16	T=32	T=64
wc:logos	251.5 MiB/s **257.0 MiB/s** 261.7 MiB/s	491.6 MiB/s **511.6 MiB/s** 517.7 MiB/s	852.7 MiB/s **1000.4 MiB/s** 1022.3 MiB/s	1.7 GiB/s **1.8 GiB/s** 1.9 GiB/s	2.3 GiB/s **2.7 GiB/s** 3.0 GiB/s
wc:regex-automata	153.4 MiB/s **159.3 MiB/s** 162.4 MiB/s	303.4 MiB/s **317.3 MiB/s** 321.4 MiB/s	525.7 MiB/s **615.0 MiB/s** 628.3 MiB/s	977.5 MiB/s **1022.3 MiB/s** 1.0 GiB/s	1.5 GiB/s **1.7 GiB/s** 1.8 GiB/s
wc:fancy-regex	55.4 MiB/s **57.0 MiB/s** 58.4 MiB/s	108.3 MiB/s **115.2 MiB/s** 116.3 MiB/s	204.6 MiB/s **222.1 MiB/s** 227.5 MiB/s	355.7 MiB/s **406.0 MiB/s** 423.6 MiB/s	575.3 MiB/s **623.6 MiB/s** 676.8 MiB/s
tiktoken	35.7 MiB/s **36.7 MiB/s** 37.1 MiB/s	70.5 MiB/s **73.9 MiB/s** 74.7 MiB/s	132.4 MiB/s **139.9 MiB/s** 144.2 MiB/s	233.7 MiB/s **249.6 MiB/s** 260.2 MiB/s	334.4 MiB/s **386.0 MiB/s** 412.2 MiB/s
tokenizers	12.5 MiB/s **12.8 MiB/s** 13.1 MiB/s	21.1 MiB/s **21.6 MiB/s** 21.9 MiB/s	23.8 MiB/s **25.6 MiB/s** 26.1 MiB/s	32.0 MiB/s **33.2 MiB/s** 33.5 MiB/s	47.4 MiB/s **49.7 MiB/s** 50.5 MiB/s

Table 4: Min/Mean/Max Throughput
rust, amd3990X, r50k



Thumbnail 2: Min/Mean/Max Throughput
rust, m2x12, r50k
See Benchmark 4

Name	T=2	T=4	T=8	T=12
wc:logos	179.8 MiB/s **182.7 MiB/s** 185.3 MiB/s	315.5 MiB/s **326.4 MiB/s** 334.8 MiB/s	611.6 MiB/s **648.2 MiB/s** 659.2 MiB/s	707.8 MiB/s **746.7 MiB/s** 759.3 MiB/s
wc:regex-automata	98.5 MiB/s **101.2 MiB/s** 102.1 MiB/s	187.6 MiB/s **194.8 MiB/s** 198.4 MiB/s	371.0 MiB/s **388.0 MiB/s** 392.1 MiB/s	445.0 MiB/s **482.6 MiB/s** 495.3 MiB/s
wc:fancy-regex	40.8 MiB/s **41.1 MiB/s** 41.4 MiB/s	77.8 MiB/s **80.1 MiB/s** 81.2 MiB/s	154.2 MiB/s **160.8 MiB/s** 162.3 MiB/s	171.6 MiB/s **186.7 MiB/s** 195.0 MiB/s
tiktoken	27.2 MiB/s **27.8 MiB/s** 28.1 MiB/s	49.7 MiB/s **51.8 MiB/s** 52.6 MiB/s	98.5 MiB/s **102.9 MiB/s** 104.4 MiB/s	118.2 MiB/s **127.2 MiB/s** 133.4 MiB/s
tokenizers	10.1 MiB/s **10.2 MiB/s** 10.2 MiB/s	17.6 MiB/s **18.1 MiB/s** 19.5 MiB/s	20.6 MiB/s **20.9 MiB/s** 21.1 MiB/s	8.1 MiB/s **9.2 MiB/s** 11.7 MiB/s

Table 5: Min/Mean/Max Throughput
rust, m2x12, r50k

10.3 o200k Rust Throughput

For the slow pattern `o200k` (Benchmark 6), the `logos`, `regex-automata`, and `fancy-regex` lexers all outperform `tiktoken-rs` and `tokenizers`. For small thread counts (≤ 8), `bpe-openai` outperforms the `fancy-regex` implementation.

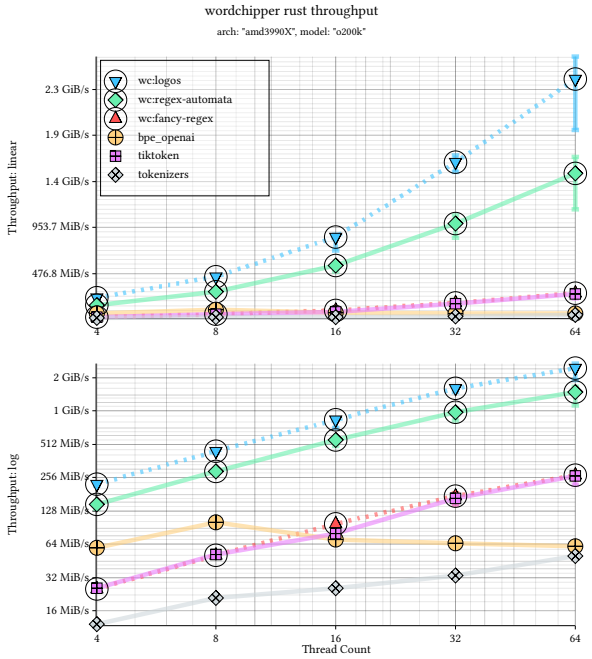
The `regex-automata` implementation (our default) consistently performs 5.7x the throughput of `tiktoken-rs`.

The `fancy-regex` implementation (our fallback) has equivalent performance to `tiktoken-rs`.

The `fancy-regex` implementation (our fallback) eventually outperforms `bpe-openai` at high thread counts, due to the thread local cache pool limitation; but it starts off slower.

For fast patterns, the `logos` DFA implementation is the fastest, ranging from 8.6x (on 4 threads) to 9.2x (on 64 threads) the throughput of `tiktoken-rs`; with a peak measured speed of 2.4GiB/s.

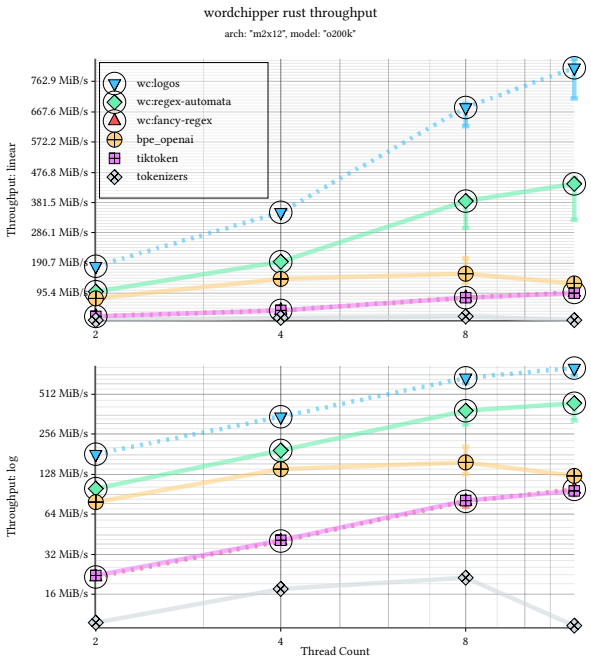
The Apple **M2** outperforms the **AMD3990X** at equivalent thread counts, but the relative performance behavior is roughly equivalent on the two systems.



Thumbnail 3: Min/Mean/Max Throughput
rust, amd3990X, o200k
See Benchmark 6

Name	T=4	T=8	T=16	T=32	T=64
wc:logos	216.8 MiB/s **222.1 MiB/s** 226.8 MiB/s	432.4 MiB/s **443.8 MiB/s** 448.8 MiB/s	718.2 MiB/s **856.6 MiB/s** 879.9 MiB/s	1.5 GiB/s **1.6 GiB/s** 1.7 GiB/s	1.9 GiB/s **2.4 GiB/s** 2.7 GiB/s
wc:regex-automata	134.2 MiB/s **146.4 MiB/s** 149.2 MiB/s	272.1 MiB/s **290.6 MiB/s** 297.0 MiB/s	497.9 MiB/s **556.4 MiB/s** 576.9 MiB/s	844.8 MiB/s **997.5 MiB/s** 1.1 GiB/s	1.1 GiB/s **1.5 GiB/s** 1.6 GiB/s
wc:fancy-regex	24.4 MiB/s **25.3 MiB/s** 25.8 MiB/s	49.2 MiB/s **51.0 MiB/s** 51.5 MiB/s	91.0 MiB/s **97.3 MiB/s** 99.8 MiB/s	147.6 MiB/s **174.8 MiB/s** 185.2 MiB/s	243.4 MiB/s **268.2 MiB/s** 286.2 MiB/s
bpe_openai	57.9 MiB/s **59.5 MiB/s** 61.0 MiB/s	98.3 MiB/s **101.4 MiB/s** 104.5 MiB/s	65.7 MiB/s **70.4 MiB/s** 71.9 MiB/s	64.3 MiB/s **65.6 MiB/s** 66.9 MiB/s	58.2 MiB/s **60.9 MiB/s** 62.4 MiB/s
tiktoken	24.9 MiB/s **25.7 MiB/s** 26.2 MiB/s	48.9 MiB/s **51.3 MiB/s** 52.2 MiB/s	75.6 MiB/s **80.4 MiB/s** 82.7 MiB/s	145.7 MiB/s **165.8 MiB/s** 173.7 MiB/s	220.1 MiB/s **265.2 MiB/s** 281.5 MiB/s
tokenizers	11.8 MiB/s **12.2 MiB/s** 12.4 MiB/s	20.1 MiB/s **21.0 MiB/s** 21.2 MiB/s	25.3 MiB/s **25.7 MiB/s** 26.3 MiB/s	32.2 MiB/s **33.1 MiB/s** 33.5 MiB/s	47.8 MiB/s **50.2 MiB/s** 51.2 MiB/s

Table 6: Min/Mean/Max Throughput
rust, amd3990X, o200k



Thumbnail 4: Min/Mean/Max Throughput
rust, m2x12, o200k
See Benchmark 12

Name	T=2	T=4	T=8	T=12
wc:logos	175.6 MiB/s **180.9 MiB/s** 181.6 MiB/s	342.5 MiB/s **349.4 MiB/s** 358.1 MiB/s	621.1 MiB/s **680.4 MiB/s** 692.7 MiB/s	709.3 MiB/s **807.2 MiB/s** 833.8 MiB/s
wc:regex-automata	88.5 MiB/s **99.5 MiB/s** 103.0 MiB/s	164.1 MiB/s **194.5 MiB/s** 200.3 MiB/s	299.8 MiB/s **385.0 MiB/s** 394.2 MiB/s	326.5 MiB/s **439.4 MiB/s** 460.3 MiB/s
wc:fancy-regex	21.4 MiB/s **21.5 MiB/s** 21.7 MiB/s	39.2 MiB/s **40.3 MiB/s** 41.0 MiB/s	73.0 MiB/s **80.4 MiB/s** 81.5 MiB/s	92.4 MiB/s **98.3 MiB/s** 102.2 MiB/s
bpe_openai	77.4 MiB/s **78.8 MiB/s** 79.6 MiB/s	133.0 MiB/s **139.2 MiB/s** 143.0 MiB/s	127.3 MiB/s **157.3 MiB/s** 205.3 MiB/s	116.3 MiB/s **124.8 MiB/s** 133.2 MiB/s
tiktoken	21.9 MiB/s **22.1 MiB/s** 22.1 MiB/s	39.6 MiB/s **40.9 MiB/s** 41.6 MiB/s	77.9 MiB/s **81.6 MiB/s** 82.6 MiB/s	89.1 MiB/s **95.6 MiB/s** 96.8 MiB/s
tokenizers	9.6 MiB/s **9.8 MiB/s** 10.0 MiB/s	17.1 MiB/s **17.5 MiB/s** 18.3 MiB/s	21.0 MiB/s **21.3 MiB/s** 21.5 MiB/s	9.0 MiB/s **9.3 MiB/s** 9.7 MiB/s

Table 7: Min/Mean/Max Throughput
rust, m2x12, o200k

11 Python Evaluation

For the Python benchmarks, we see a similar trend to the Rust benchmarks; complicated by the python GIL and threadpool interactions.

We benchmark 4 variants of `wordchipper`; the product of `rayon` vrs `threadpool` parallelism with `regex-automata` vrs `logos` lexers. We compare these with the `tiktoken`, `tokenizers`, and `bpe_openai` python modules.

11.1 Python Errata

We report distinct performance differences between `threadpool` behavior for the OSX (the `m2x12` machine) and Linux (the `amd3990X` machine) benchmarks; but we do not draw conclusions from them.

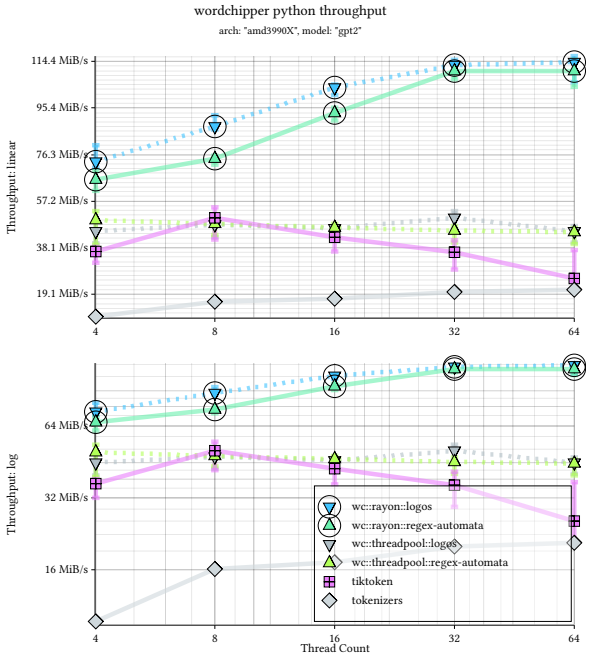
We observe anomalously low throughput for the `bpe_openai` wrapper under our harness; we report these numbers for completeness but do not draw conclusions from them.

11.2 gpt2 Python Throughput

For the fast pattern `gpt2` (Benchmark 7), the `logos`, `regex-automata`, and `fancy-regex` lexers all outperform `tiktoken` and `tokenizers`. There is no `r50k` support for `bpe_openai` yet.

For low thread counts, our `rayon + regex-automata` implementation (the default) outperforms `tiktoken` by 2x. Above 8 threads, the performance of the python `tiktoken` implementation collapses. At 64 threads, the ratio grows to 4.1x; with a peak throughput of `rayon + regex-automata` at 104.3 MiB/s on 64 threads.

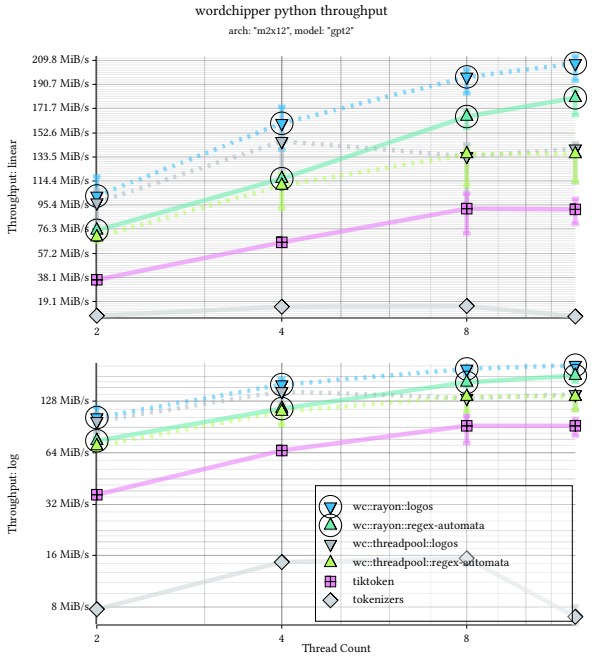
Across thread counts, our `rayon + logos` range between 1.2x to 1.1x the performance of our `rayon + regex-automata` implementation; reaching a peak throughput of 116.7 MiB/s on 64 threads.



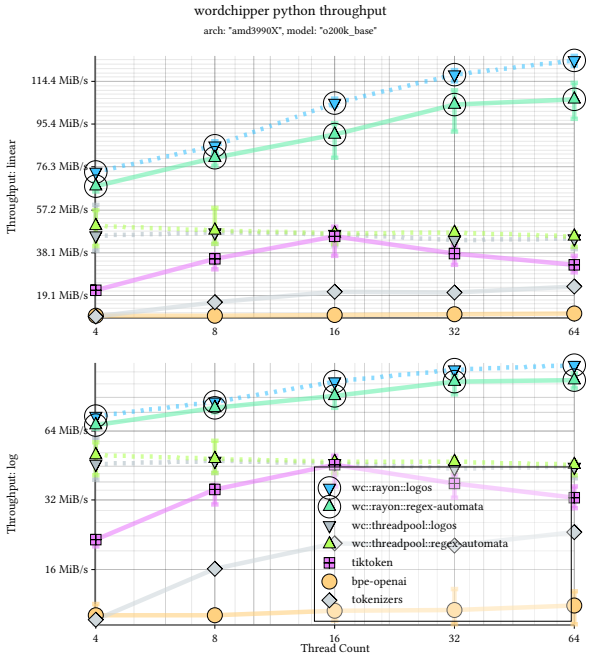
Thumbnail 5: Min/Mean/Max Throughput
python, amd3990X, gpt2
See Benchmark 7

Name	T=4	T=8	T=16	T=32	T=64
wc::rayon::logos	80.1 MiB/s **73.4 MiB/s** 71.0 MiB/s	92.5 MiB/s **87.9 MiB/s** 86.5 MiB/s	105.0 MiB/s **103.7 MiB/s** 100.8 MiB/s	115.0 MiB/s **113.1 MiB/s** 110.1 MiB/s	116.5 MiB/s **114.1 MiB/s** 105.8 MiB/s
wc::rayon::regex-automata	69.8 MiB/s **66.1 MiB/s** 61.3 MiB/s	75.7 MiB/s **74.6 MiB/s** 72.3 MiB/s	95.4 MiB/s **93.4 MiB/s** 90.3 MiB/s	112.9 MiB/s **110.5 MiB/s** 107.1 MiB/s	115.1 MiB/s **110.5 MiB/s** 104.2 MiB/s
wc::threadpool::logos	49.2 MiB/s **44.9 MiB/s** 39.8 MiB/s	49.0 MiB/s **47.6 MiB/s** 42.3 MiB/s	46.7 MiB/s **46.0 MiB/s** 41.6 MiB/s	53.1 MiB/s **50.3 MiB/s** 45.5 MiB/s	47.1 MiB/s **44.8 MiB/s** 40.1 MiB/s
wc::threadpool::regex-automata	53.2 MiB/s **49.5 MiB/s** 39.3 MiB/s	49.0 MiB/s **47.8 MiB/s** 43.3 MiB/s	47.8 MiB/s **46.4 MiB/s** 41.8 MiB/s	47.1 MiB/s **45.3 MiB/s** 40.3 MiB/s	46.1 MiB/s **44.6 MiB/s** 39.6 MiB/s
tiktoken	38.4 MiB/s **36.7 MiB/s** 31.9 MiB/s	54.7 MiB/s **50.2 MiB/s** 41.6 MiB/s	45.3 MiB/s **42.5 MiB/s** 36.4 MiB/s	41.2 MiB/s **36.2 MiB/s** 29.4 MiB/s	37.3 MiB/s **25.5 MiB/s** 20.9 MiB/s
tokenizers	10.1 MiB/s **9.8 MiB/s** 9.4 MiB/s	16.2 MiB/s **16.1 MiB/s** 16.0 MiB/s	17.4 MiB/s **17.2 MiB/s** 17.0 MiB/s	20.5 MiB/s **20.0 MiB/s** 19.3 MiB/s	21.3 MiB/s **20.8 MiB/s** 20.6 MiB/s

Table 8: Min/Mean/Max Throughput
python, amd3990X, gpt2



Thumbnail 6: Min/Mean/Max Throughput
python, m2x12, gpt2
See Benchmark 13



Thumbnail 7: Min/Mean/Max Throughput
python, amd3990X, o200k
See Benchmark 9

Name	T=2	T=4	T=8	T=12
wc::rayon::logos	117.9 MiB/s **102.6 MiB/s** 94.9 MiB/s	173.4 MiB/s **159.8 MiB/s** 139.2 MiB/s	203.0 MiB/s **197.0 MiB/s** 183.9 MiB/s	213.1 MiB/s **207.4 MiB/s** 193.6 MiB/s
wc::rayon::regex-automata	78.1 MiB/s **75.4 MiB/s** 73.3 MiB/s	119.2 MiB/s **116.2 MiB/s** 109.4 MiB/s	170.0 MiB/s **165.5 MiB/s** 156.3 MiB/s	186.0 MiB/s **180.2 MiB/s** 166.8 MiB/s
wc::threadpool::logos	107.4 MiB/s **97.2 MiB/s** 83.7 MiB/s	150.8 MiB/s **145.9 MiB/s** 124.1 MiB/s	143.5 MiB/s **134.0 MiB/s** 96.6 MiB/s	144.0 MiB/s **139.8 MiB/s** 113.4 MiB/s
wc::threadpool::regex-automata	73.9 MiB/s **69.9 MiB/s** 66.7 MiB/s	120.0 MiB/s **111.0 MiB/s** 92.4 MiB/s	139.2 MiB/s **136.0 MiB/s** 110.9 MiB/s	138.9 MiB/s **135.6 MiB/s** 113.8 MiB/s
tiktoken	37.3 MiB/s **36.2 MiB/s** 34.6 MiB/s	66.6 MiB/s **65.6 MiB/s** 62.7 MiB/s	104.3 MiB/s **92.3 MiB/s** 72.9 MiB/s	100.4 MiB/s **91.7 MiB/s** 80.4 MiB/s
tokenizers	8.2 MiB/s **7.8 MiB/s** 7.5 MiB/s	15.5 MiB/s **14.7 MiB/s** 14.5 MiB/s	15.7 MiB/s **15.4 MiB/s** 15.2 MiB/s	8.1 MiB/s **7.1 MiB/s** 6.3 MiB/s

Table 9: Min/Mean/Max Throughput
python, m2x12, gpt2

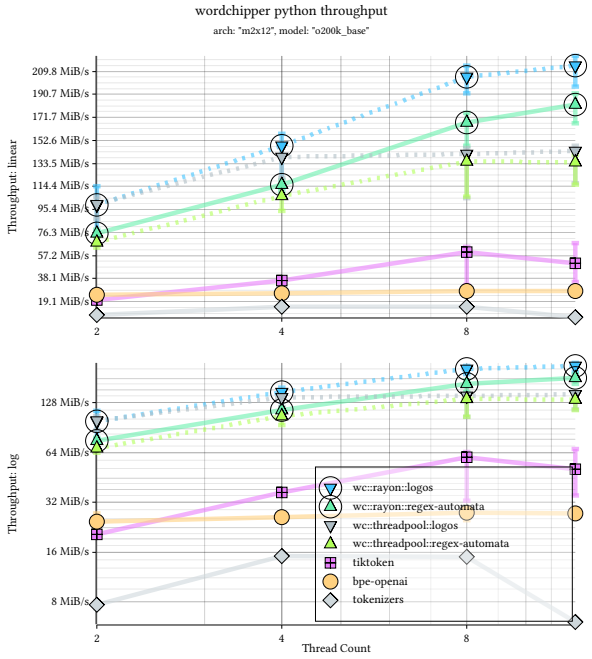
Name	T=4	T=8	T=16	T=32	T=64
wc::rayon::logos	74.6 MiB/s **74.1 MiB/s** 73.4 MiB/s	87.7 MiB/s **85.7 MiB/s** 82.2 MiB/s	106.7 MiB/s **104.9 MiB/s** 103.5 MiB/s	119.3 MiB/s **117.6 MiB/s** 114.4 MiB/s	125.6 MiB/s **123.7 MiB/s** 121.4 MiB/s
wc::rayon::regex-automata	69.0 MiB/s **67.9 MiB/s** 65.3 MiB/s	82.6 MiB/s **80.6 MiB/s** 76.7 MiB/s	95.6 MiB/s **90.7 MiB/s** 80.5 MiB/s	110.6 MiB/s **104.3 MiB/s** 92.1 MiB/s	113.9 MiB/s **106.5 MiB/s** 97.7 MiB/s
wc::threadpool::logos	59.5 MiB/s **45.8 MiB/s** 39.1 MiB/s	48.6 MiB/s **47.2 MiB/s** 42.1 MiB/s	47.5 MiB/s **46.4 MiB/s** 41.4 MiB/s	45.9 MiB/s **43.8 MiB/s** 39.7 MiB/s	46.6 MiB/s **44.6 MiB/s** 39.8 MiB/s
wc::threadpool::regex-automata	57.0 MiB/s **50.2 MiB/s** 41.0 MiB/s	58.0 MiB/s **48.2 MiB/s** 42.1 MiB/s	47.8 MiB/s **46.9 MiB/s** 42.0 MiB/s	48.7 MiB/s **46.9 MiB/s** 42.4 MiB/s	47.1 MiB/s **45.5 MiB/s** 40.9 MiB/s
tiktoken	22.0 MiB/s **21.5 MiB/s** 20.2 MiB/s	37.2 MiB/s **35.5 MiB/s** 30.7 MiB/s	49.9 MiB/s **45.4 MiB/s** 36.8 MiB/s	41.4 MiB/s **37.9 MiB/s** 32.8 MiB/s	36.7 MiB/s **32.7 MiB/s** 29.5 MiB/s
bpe-openai	11.3 MiB/s **10.1 MiB/s** 9.5 MiB/s	10.5 MiB/s **10.1 MiB/s** 9.8 MiB/s	11.3 MiB/s **10.5 MiB/s** 9.6 MiB/s	13.2 MiB/s **10.7 MiB/s** 9.2 MiB/s	12.8 MiB/s **11.1 MiB/s** 9.2 MiB/s
tokenizers	9.9 MiB/s **9.7 MiB/s** 9.2 MiB/s	16.2 MiB/s **16.1 MiB/s** 16.0 MiB/s	21.0 MiB/s **20.8 MiB/s** 20.6 MiB/s	20.7 MiB/s **20.2 MiB/s** 19.7 MiB/s	24.4 MiB/s **23.2 MiB/s** 22.6 MiB/s

Table 10: Min/Mean/Max Throughput
python, amd3990X, o200k

11.3 o200k Python Throughput

The performance of the `o200k` pattern (Benchmark 9) is similar to the `gpt2` pattern, with the `rayon + logos` implementation outperforming the `rayon + regex-automata` implementation.

There are few meaningful differences outside the performance of `tiktoken`.



Thumbnail 8: Min/Mean/Max Throughput
python, m2x12, o200k
See Benchmark 15

Name	T=2	T=4	T=8	T=12
wc::rayon::logos	114.8 MiB/s **99.0 MiB/s** 77.0 MiB/s	158.0 MiB/s **148.1 MiB/s** 113.2 MiB/s	214.5 MiB/s **205.0 MiB/s** 191.3 MiB/s	222.3 MiB/s **214.6 MiB/s** 197.2 MiB/s
wc::rayon::regex-automata	78.9 MiB/s **75.4 MiB/s** 73.2 MiB/s	119.6 MiB/s **116.2 MiB/s** 110.2 MiB/s	175.3 MiB/s **167.2 MiB/s** 148.1 MiB/s	191.9 MiB/s **182.2 MiB/s** 166.7 MiB/s
wc::threadpool::logos	106.3 MiB/s **99.6 MiB/s** 87.4 MiB/s	146.8 MiB/s **138.9 MiB/s** 121.3 MiB/s	146.3 MiB/s **141.0 MiB/s** 105.0 MiB/s	147.9 MiB/s **143.6 MiB/s** 117.3 MiB/s
wc::threadpool::regex-automata	74.0 MiB/s **68.1 MiB/s** 63.9 MiB/s	114.4 MiB/s **106.8 MiB/s** 94.2 MiB/s	140.2 MiB/s **135.1 MiB/s** 105.8 MiB/s	140.5 MiB/s **134.3 MiB/s** 115.7 MiB/s
tiktoken	20.7 MiB/s **20.4 MiB/s** 20.1 MiB/s	37.0 MiB/s **36.7 MiB/s** 36.5 MiB/s	64.5 MiB/s **60.1 MiB/s** 32.8 MiB/s	67.2 MiB/s **50.7 MiB/s** 35.2 MiB/s
bpe-openai	27.4 MiB/s **24.5 MiB/s** 21.7 MiB/s	26.9 MiB/s **26.1 MiB/s** 25.0 MiB/s	28.7 MiB/s **27.7 MiB/s** 25.0 MiB/s	27.7 MiB/s **27.6 MiB/s** 27.3 MiB/s
tokenizers	8.1 MiB/s **7.7 MiB/s** 7.3 MiB/s	15.2 MiB/s **15.1 MiB/s** 14.9 MiB/s	15.4 MiB/s **15.1 MiB/s** 14.8 MiB/s	6.3 MiB/s **6.1 MiB/s** 5.8 MiB/s

Table 11: Min/Mean/Max Throughput
python, m2x12, o200k

12 Discussion

The results suggest that tokenizer performance is driven less by any single algorithmic breakthrough than by how well the implementation matches the workload. In our measurements, the fastest configurations were generally those that minimized heap traffic, reused per-thread working state, and kept the lexer stage efficient enough that encoding remained visible as a meaningful cost. That pattern held across both the Rust and Python evaluations, and across the different vocabulary families we tested.

A notable result is that the simpler span encoders often performed best once the lexer was fast (Benchmark 1, Benchmark 3). On the faster lexer backends, the sweep-based encoders consistently matched or slightly outperformed the more elaborate alternatives, while the `PriorityMerge` and `BpeBacktrack` designs were more competitive when lexer cost dominated. This reinforces the idea that the "best" encoder is workload-dependent: once pre-tokenization becomes cheap, overheads in merge strategy and data structure maintenance matter more. Conversely, when regex work is expensive, encoder differences shrink.

The Python benchmarks tell a similar story, but with extra noise from the interpreter and binding layer. Releasing the GIL and delegating work to Rust helped substantially, yet the Python results still lagged the Rust results by a wide margin in absolute terms. That makes the binding layer an important part of the system, not just a packaging detail. It also means Python-facing throughput depends on how much of the pipeline can stay inside Rust without unnecessary handoff overhead.

One limitation of the study is that the benchmark suite, while broad, is still controlled. The corpora were chosen to exercise representative regimes, but they are not a complete model of real production traffic. In particular, throughput on long-form documents, highly repetitive text, code-heavy corpora, and mixed batch sizes may differ from the patterns observed here. The fine-grained relative comparisons are still useful, but they should not be read as universal rankings for every tokenizer workload.

A second limitation is that some of the encoder implementations are intentionally experimental. Several are designed to explore a specific tradeoff rather than to be the last word in algorithm design, so their performance should be interpreted as evidence about a design space rather than final production guidance. The same is true for the lexer backends: DFA-based approaches are fast, but they require additional transformation and post-processing logic to reproduce the exact behavior of the reference regex patterns.

Finally, the benchmark methodology focuses on throughput and allocation behavior, which are the right first-order signals for this project, but not the whole

picture. We do not deeply analyze latency tail behavior, memory footprint under sustained mixed workloads, or integration costs in larger serving systems. Those would be valuable next steps, especially if the goal shifts from "how fast can this tokenize?" to "how does this behave inside a real pipeline?"

12.1 Future Work

Our parallel implementation sits directly upon simple `rayon` dispatch. There is no isolation between the spanning phase (which is linear) and the encoding phase (which is embarrassingly parallel). Further work in explicit async work modeling should be productive, along several dimensions. Firstly, it may be possible to exploit the relative performance and context switching costs of the lexers and encoders via async work channels to reduce latency. Secondly, explicit thread-local resource management would avoid the long-tail behavior of thread hash collisions between these resources. And thirdly, production bulk processing systems build their batches incrementally through IO and other processes; and providing an async parallel API would permit work to be submitted as the IO/preprocessing completed, and waited on for completion as a batch. Both would require additional metric tracing defined over async channel timing.

Following on the async work, there seems to be value in extracting external tokenizer servers for shared tokenization across multiple nodes in clusters. Distributed tokenization would be a natural extension of the parallel implementation; but introduces several additional layers of timing and throughput metric optimization.

At the lexer-design level; it seems that there are open questions about ideal lexer designs, optimizing for throughput as well as vocabulary utility for neural layers. Further work could be done to simplify the `regex => logos`/DFA conversion process.

13 Conclusion

`wordchipper` shows that OpenAI-style byte-level BPE tokenization still has meaningful headroom for optimization when the implementation is structured around the algorithm's real separation points. By treating pre-tokenization and merge encoding as independent, swappable stages, we were able to compare lexer and encoder choices directly, isolate bottlenecks, and build a tokenizer that is both fast and experimentally useful.

The strongest gains came from three design choices: aggressive reuse of thread-local working memory, zero-allocation fast paths after setup, and compile-time DFA specialization for the lexer stage. No single choice dominates — their relative importance shifts with vocabulary complexity, corpus composition, and concurrency level, which is precisely why the cross-product benchmark methodology matters. That methodology may generalize beyond tokenization to any system whose performance depends on a clean separation between policy, data layout, and execution strategy.

14 References

[0] wordchipper. https://github.com/zspacelabs/wordchipper

[1] R. Sennrich, B. Haddow, and A. Birch, "Neural Machine Translation of Rare Words with Subword Units," in *Proc. ACL*, 2016. doi: 10.48550/arXiv:1508.07909

[2] P. Gage, "A new algorithm for data compression," The C Users Journal archive, vol. 12, no. 2, pp. 23–38, Feb. 1994. doi: 10.5555/177910.177914

[3] M. Schuster and K. Nakajima, "Japanese and Korean voice search," 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Kyoto, Japan, 2012, pp. 5149–5152. doi: 10.1109/ICASSP.2012.6289079

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proc. NAACL*, 2019. doi: 10.48550/arXiv.1810.04805

[5] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing," in *Proc. EMNLP* (System Demonstrations), 2018. doi: 10.48550/arXiv.1808.06226

[6] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," OpenAI Technical Report, 2019. https://openai.com/index/better-language-models/

[7] OpenAI, "tiktoken," https://github.com/openai/tiktoken, 2023.

[8] N. Patry and Y. Belkada, "HuggingFace Tokenizers," https://github.com/huggingface/tokenizers, 2022.

[9] M. Hirsz, "Logos: Create ridiculously fast Lexers in Rust," https://github.com/maciejhirsz/logos, 2023.

[10] A. Gallant, "regex-automata: a low-level regular expression library," https://github.com/rust-lang/regex, 2023.

[11] R. Levien, "fancy-regex: A Rust library for regular expressions using 'fancy' features," https://github.com/fancy-regex/fancy-regex, 2022.

[12] Z. Li, “tiktoken-rs,” https://github.com/zurawiki/tiktoken-rs, 2023.

[13] H. van Antwerpen and A. Neubeck, “bpe / bpe-openai,” GitHub rust-gems, https://github.com/github/rust-gems/tree/main/crates/bpe, 2023. MIT License.

[14] H. van Antwerpen and A. Neubeck, “So many tokens, so little time: Introducing a faster, more flexible byte-pair tokenizer,” The GitHub Blog, 2024. https://github.blog/ai-and-ml/llms/so-many-tokens-so-little-time-introducing-a-faster-more-flexible-byte-pair-tokenizer/

[15] Kozma et al., “Theoretical Analysis of Byte-Pair Encoding.” doi: 10.48550/arXiv:2411.08671

[16] L. Phan, “divan: Fast, zero-config Rust benchmarking,” https://github.com/nvzqz/divan, 2024.

[17] G. Penedo, H. Kydlíček, L. Ben Allal, A. Lozhkov, M. Mitchell, C. Raffel, L. Von Werra, and T. Wolf, “The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale,” in *Advances in Neural Information Processing Systems 37 (NeurIPS)*, Datasets and Benchmarks Track,
2024.

doi: 10.48550/arXiv:2406.17557

[18] A. Gallant, “aho-corasick: Fast multi-pattern string matching,” https://github.com/BurntSushi/aho-corasick, 2023.

[19] PyO3 Project, “PyO3: Rust bindings for the Python interpreter,” https://github.com/PyO3/pyo3, 2024.

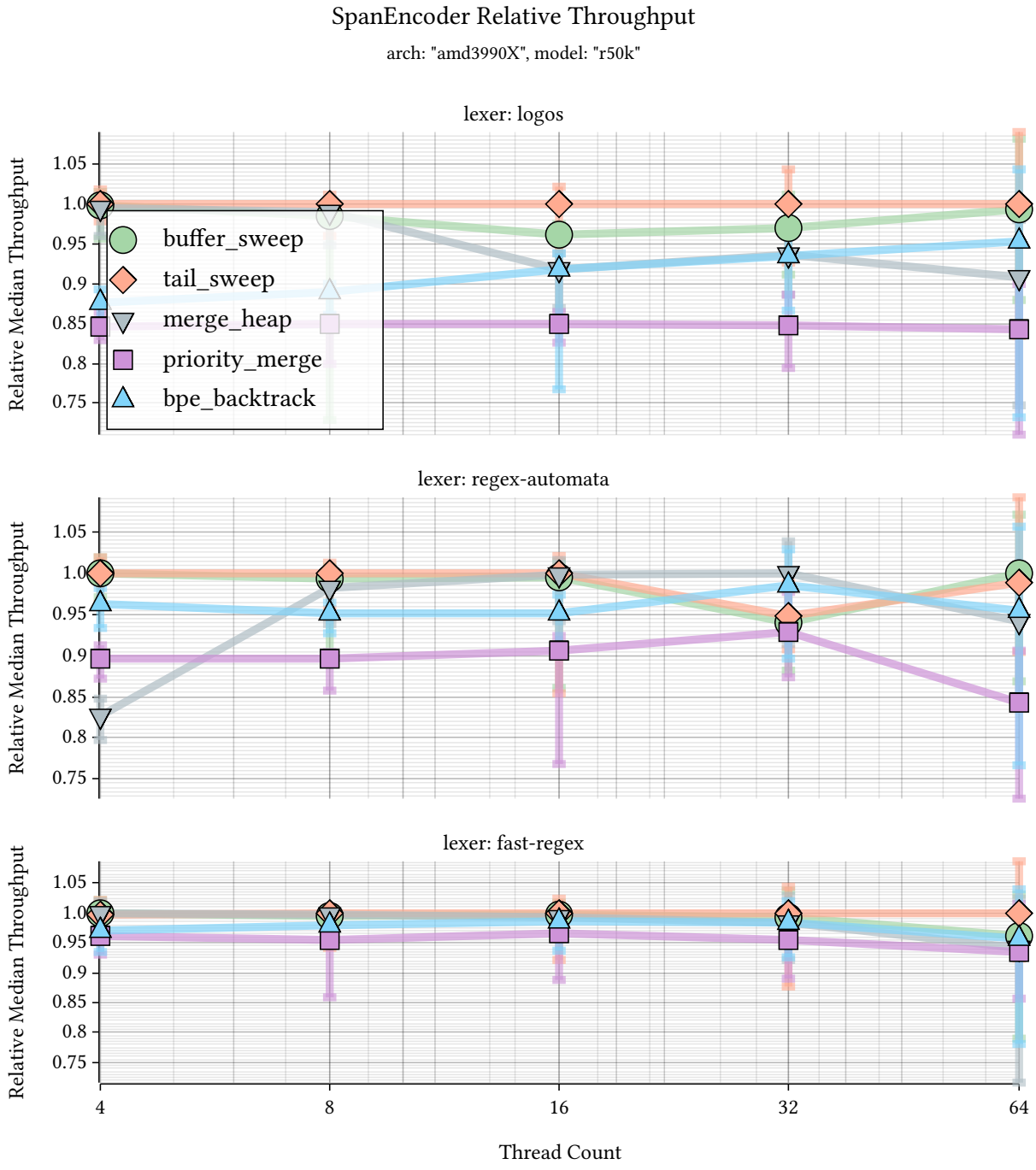
[20] PyO3 Project, “maturin: Build and publish Rust crates as Python packages,” https://github.com/PyO3/maturin, 2024.

[22] A. Karpathy, “fineweb-edu-100b-shuffle,” HuggingFace Datasets, https://huggingface.co/datasets/karpathy/fineweb-edu-100b-shuffle, 2025. Globally shuffled 100B-token subset of FineWeb-Edu[17].

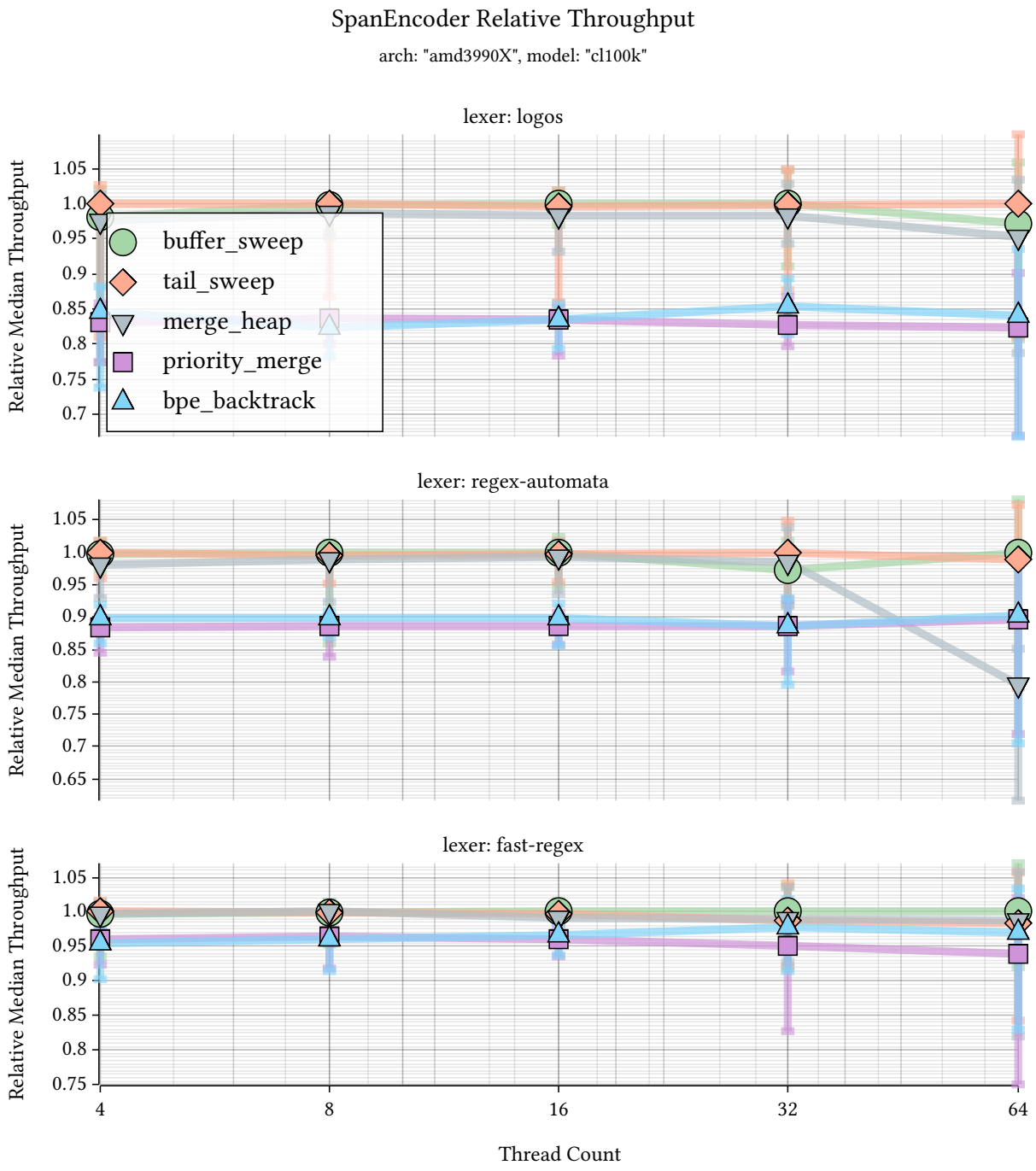
[23] W3C WebAssembly Working Group, “WebAssembly Core Specification, Version 2.0,” W3C, 2024. https://www.w3.org/TR/wasm-core-2/

15 Appendix

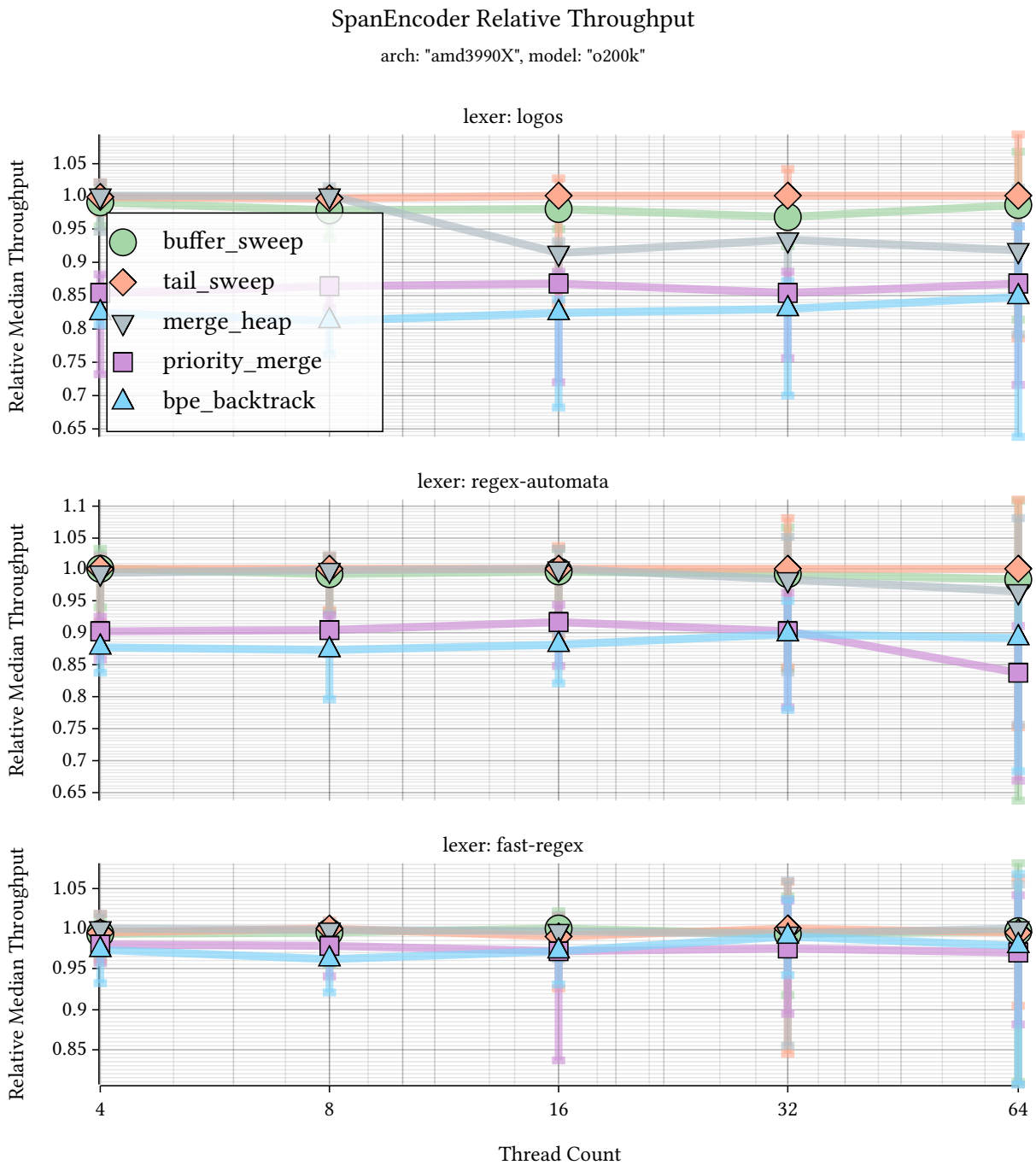
15.1 Benchmarks



Benchmark 1: Encoder Relative Min/Mean/Max Throughput: amd3990X, r50k

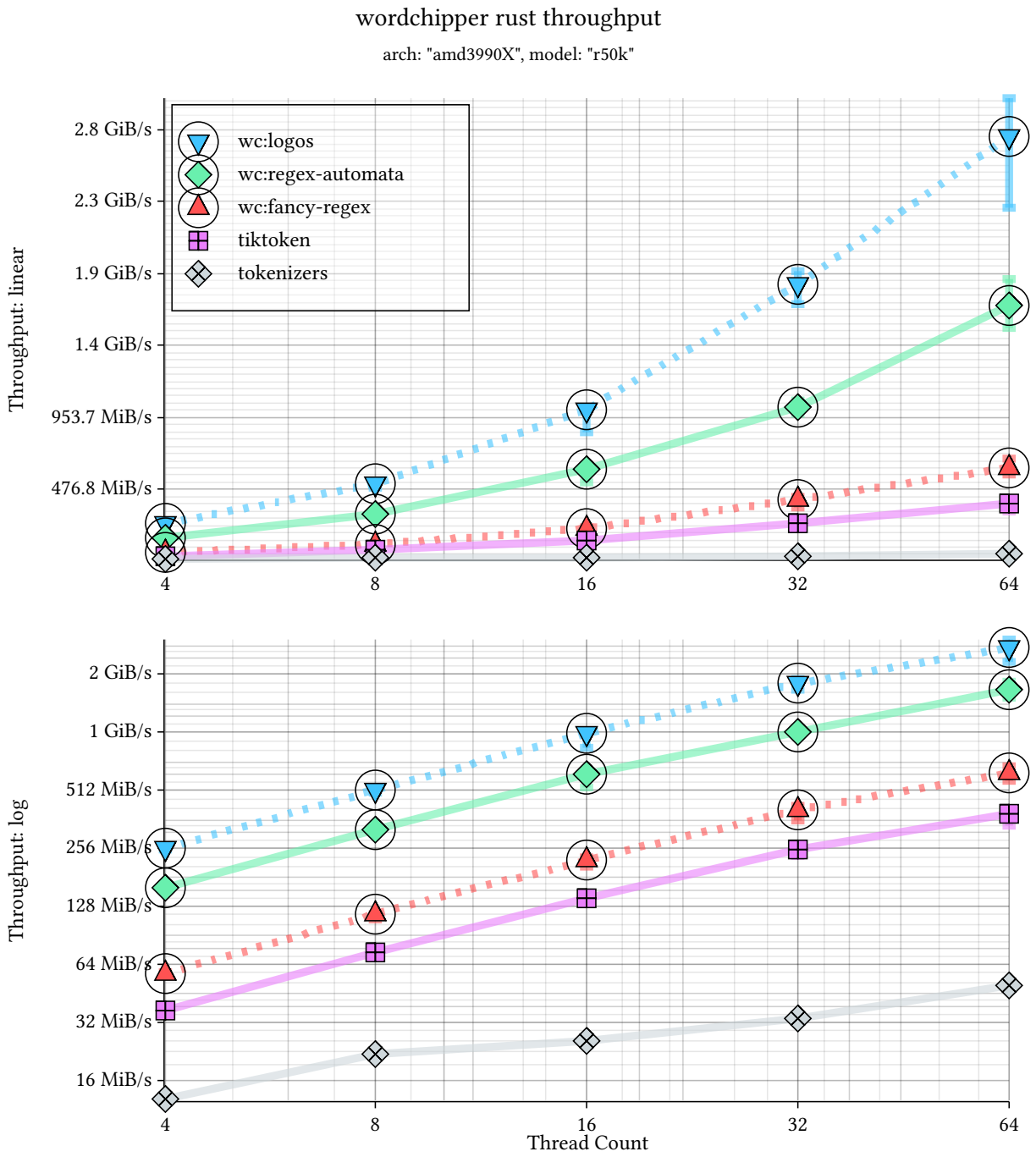


Benchmark 2: Encoder Relative Min/Mean/Max Throughput: amd3990X, o100k

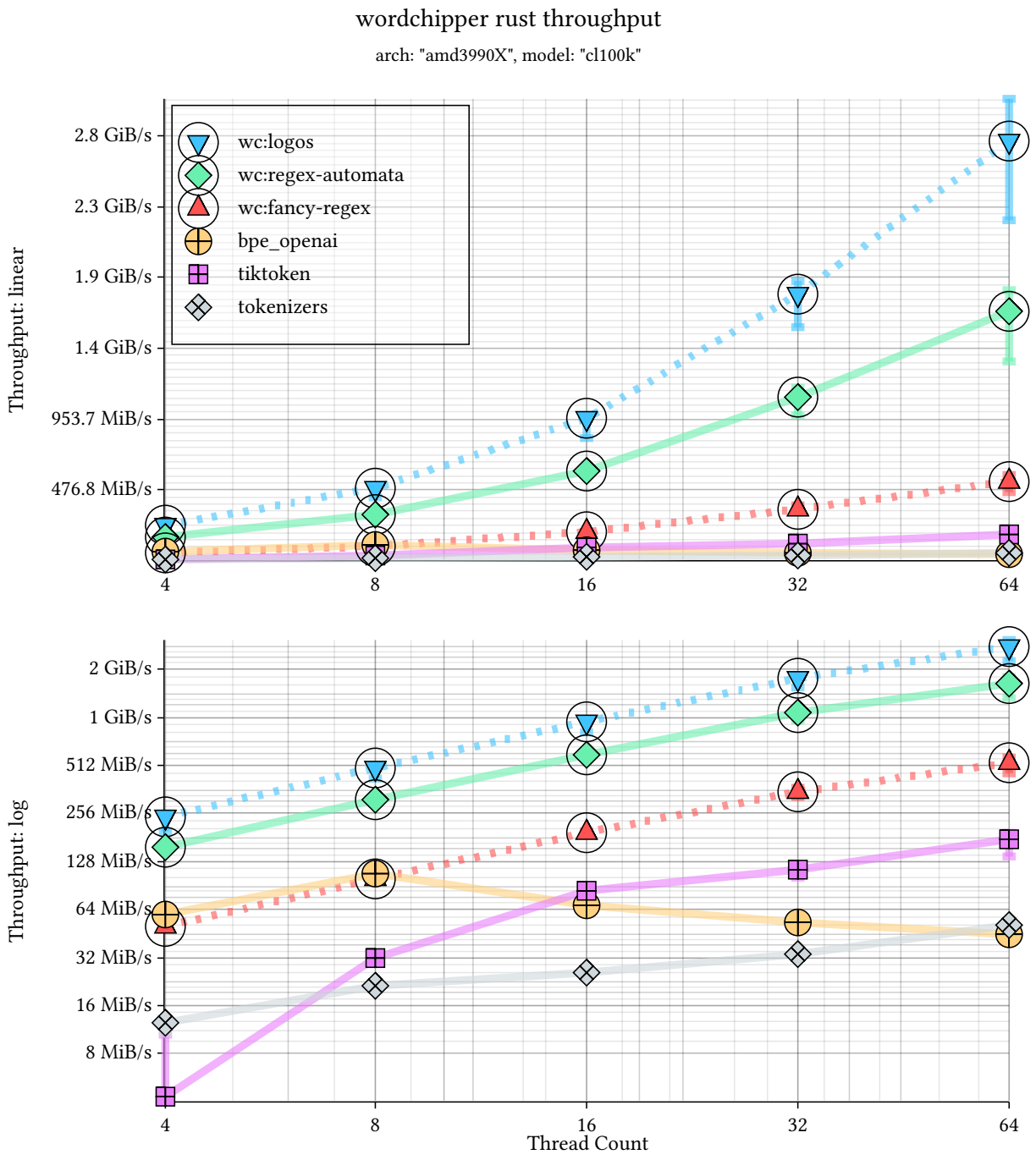


Benchmark 3: Encoder Relative Min/Mean/Max Throughput: amd3990X, o200k

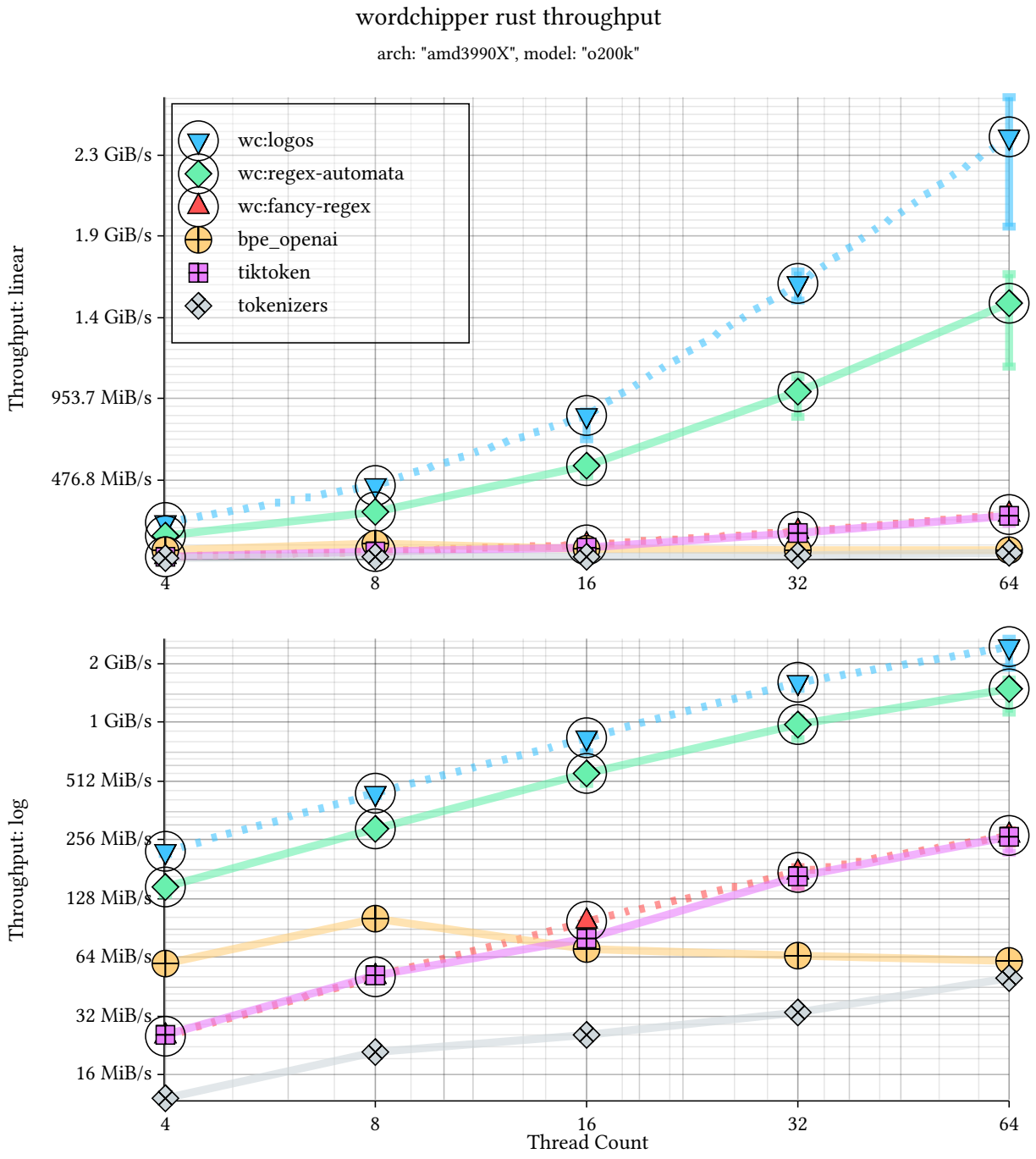
15.1.1 amd3990X



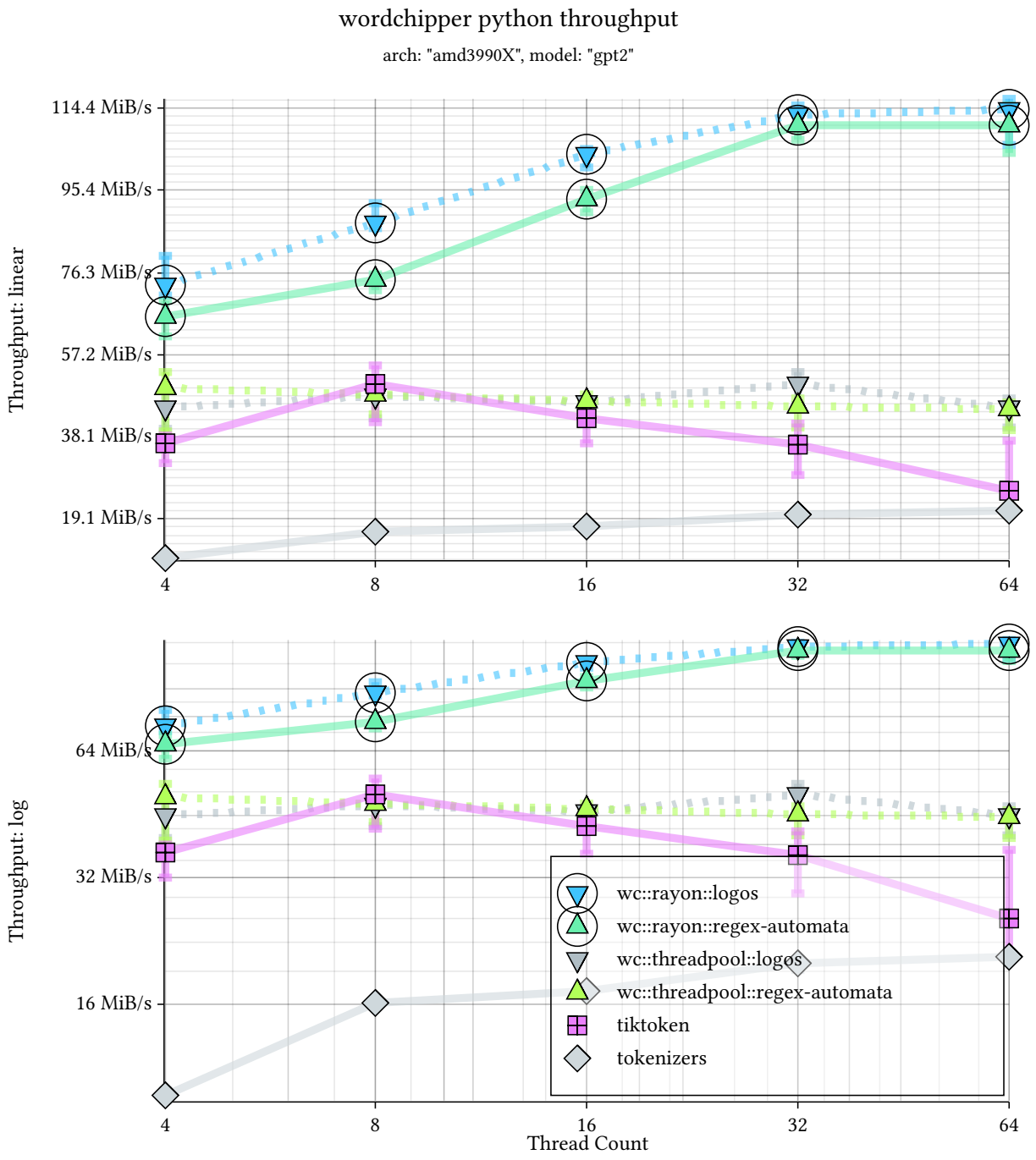
Benchmark 4: Rust Encoder Min/Mean/Max Throughput: amd3990X, r50k



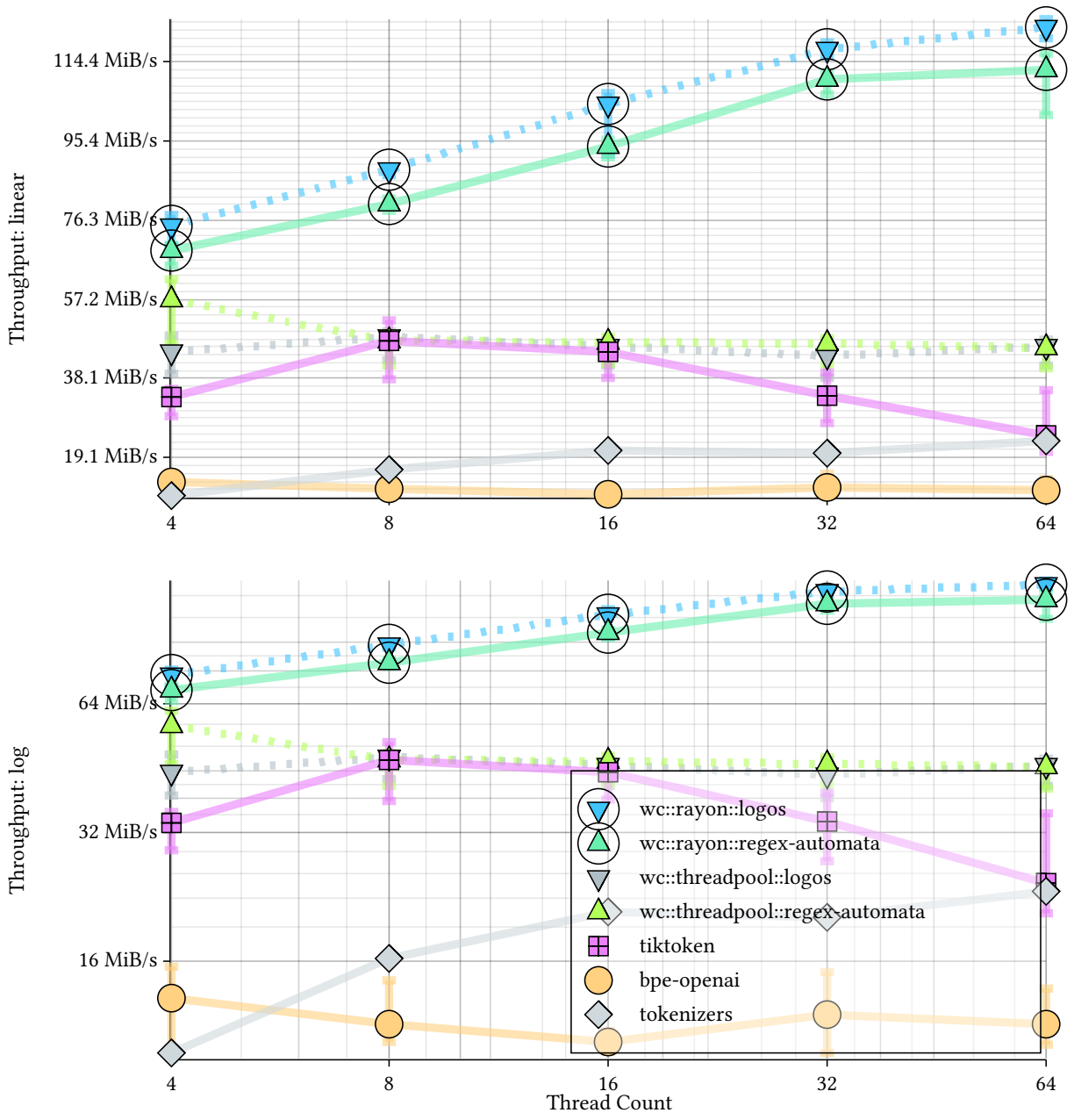
Benchmark 5: Rust Encoder Min/Mean/Max Throughput: amd3990X, cl100k



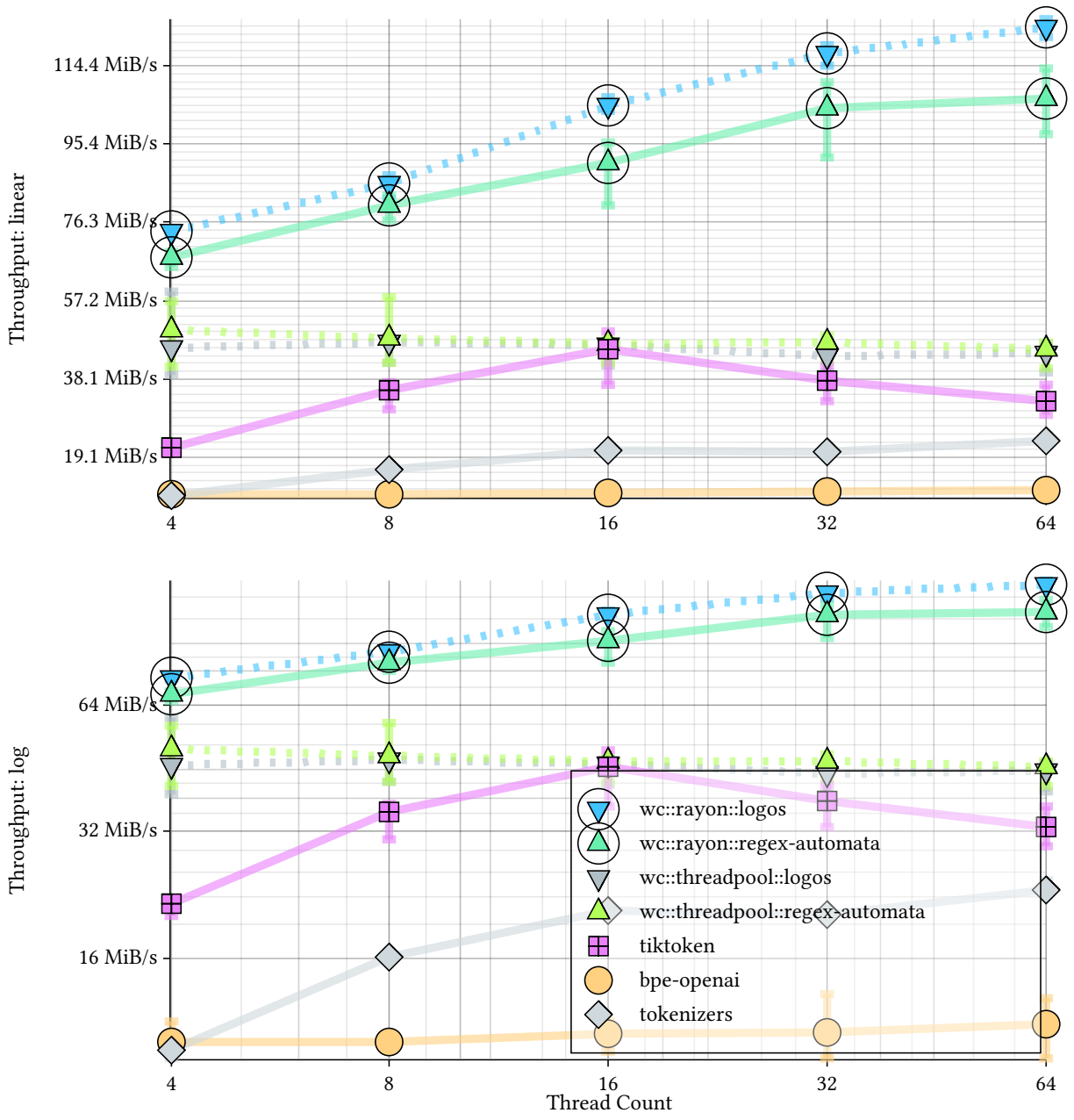
Benchmark 6: Rust Encoder Min/Mean/Max Throughput: amd3990X, o200k



Benchmark 7: Python Encoder Min/Mean/Max Throughput: amd3990X, gpt2

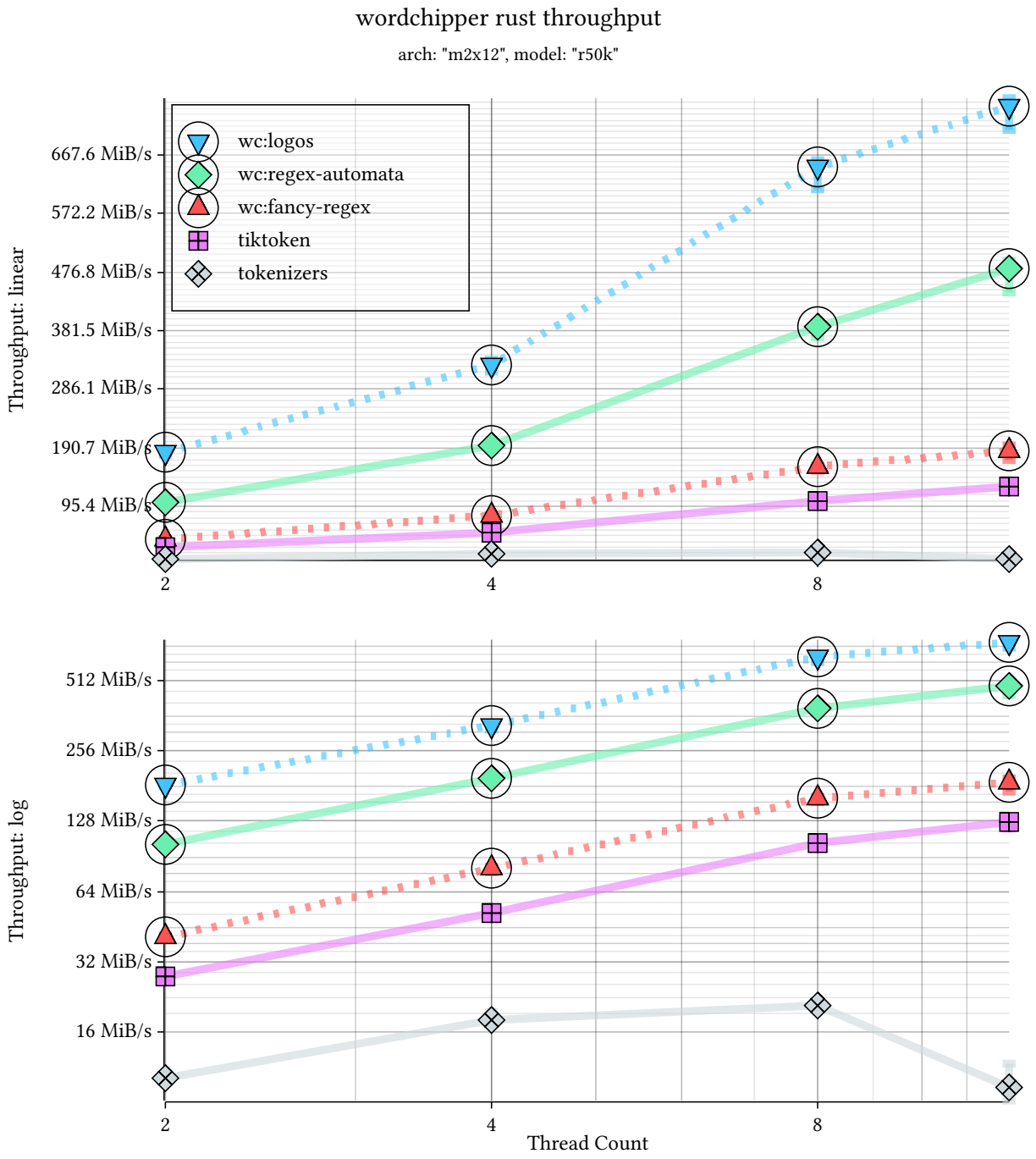


Benchmark 8: Python Encoder Min/Mean/Max Throughput: amd3990X, cl100k

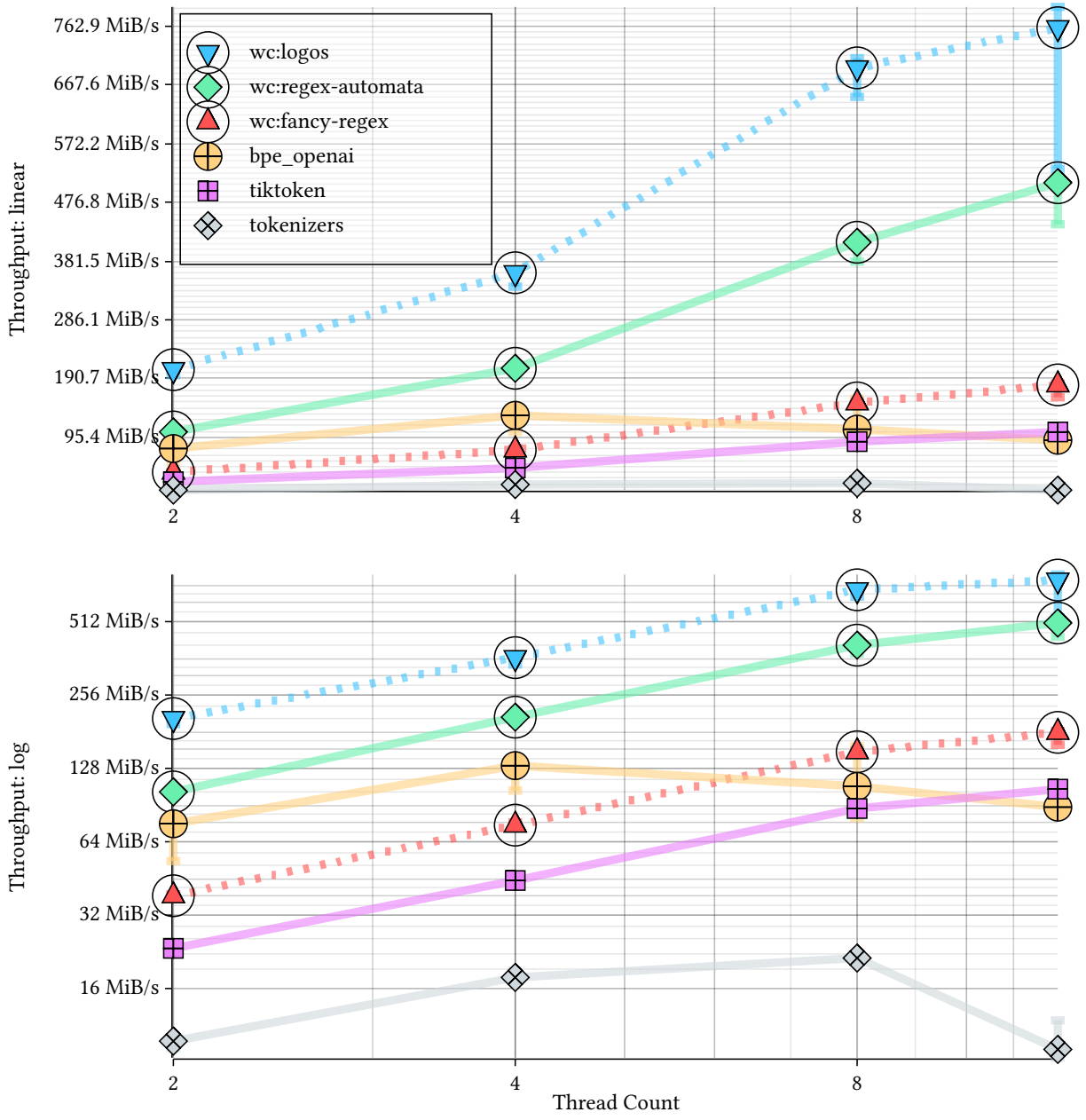


Benchmark 9: Python Encoder Min/Mean/Max Throughput: amd3990X, o200k

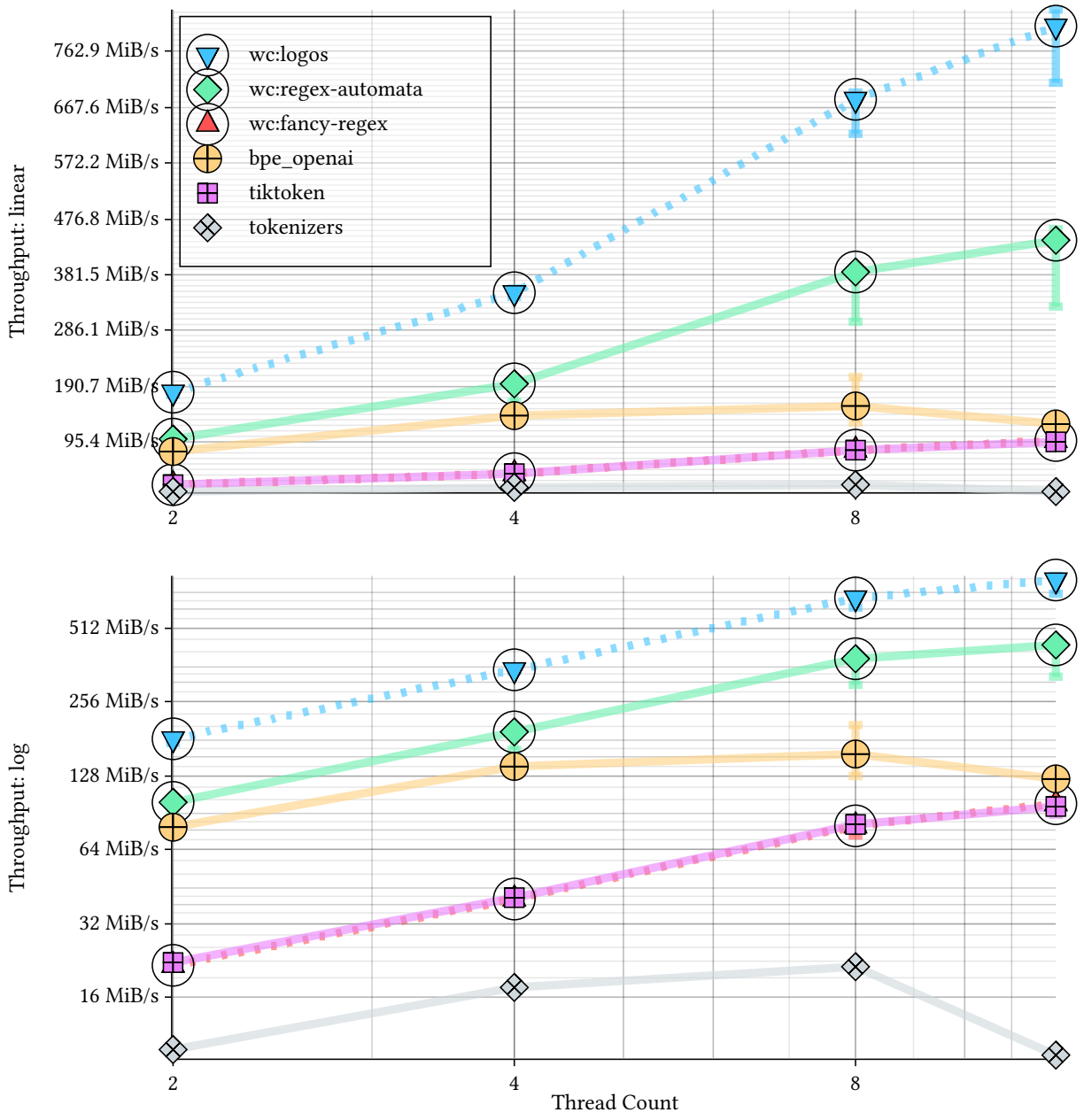
15.1.2 m2x12



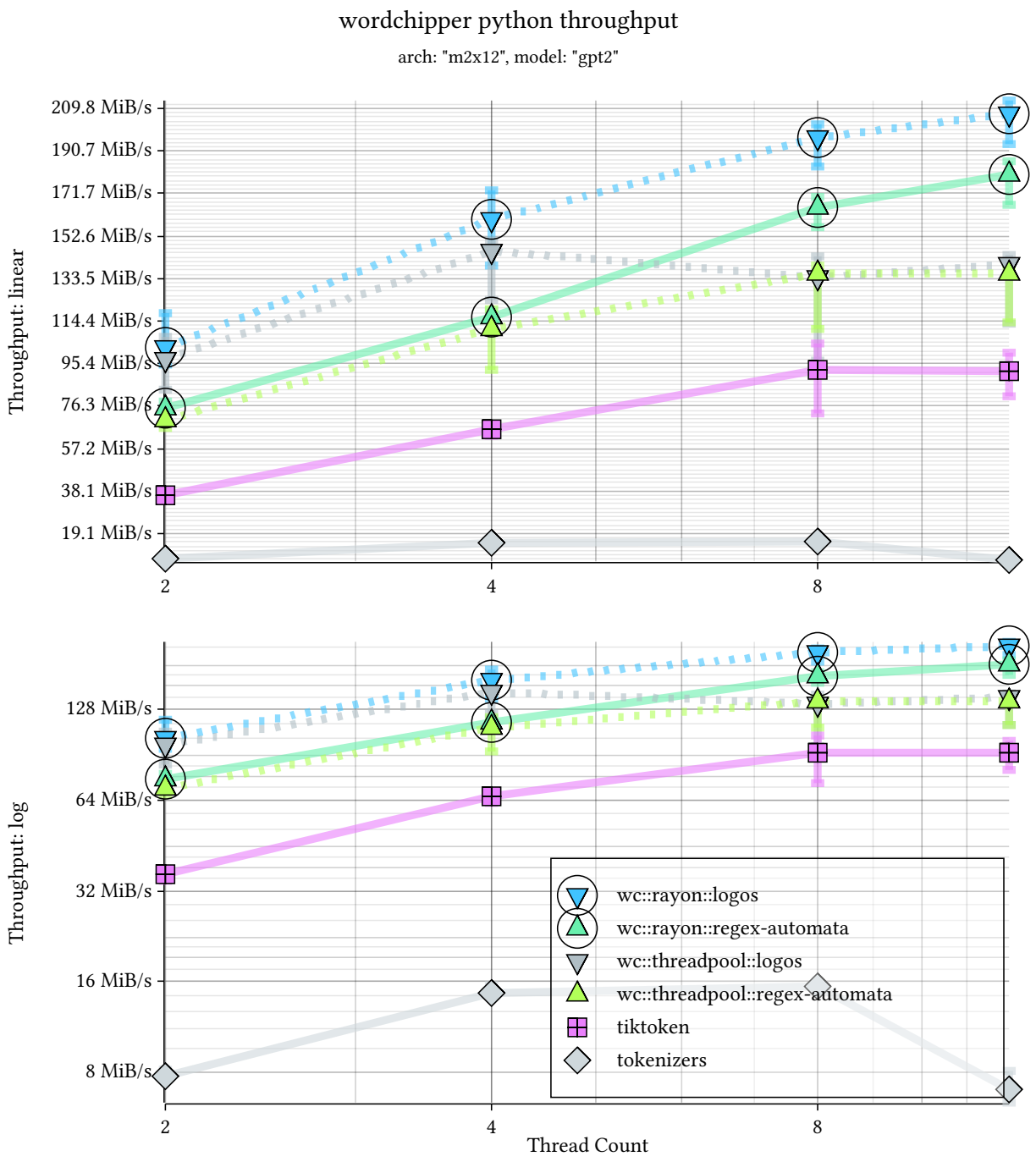
Benchmark 10: Rust Encoder Min/Mean/Max Throughput: m2x12, r50k



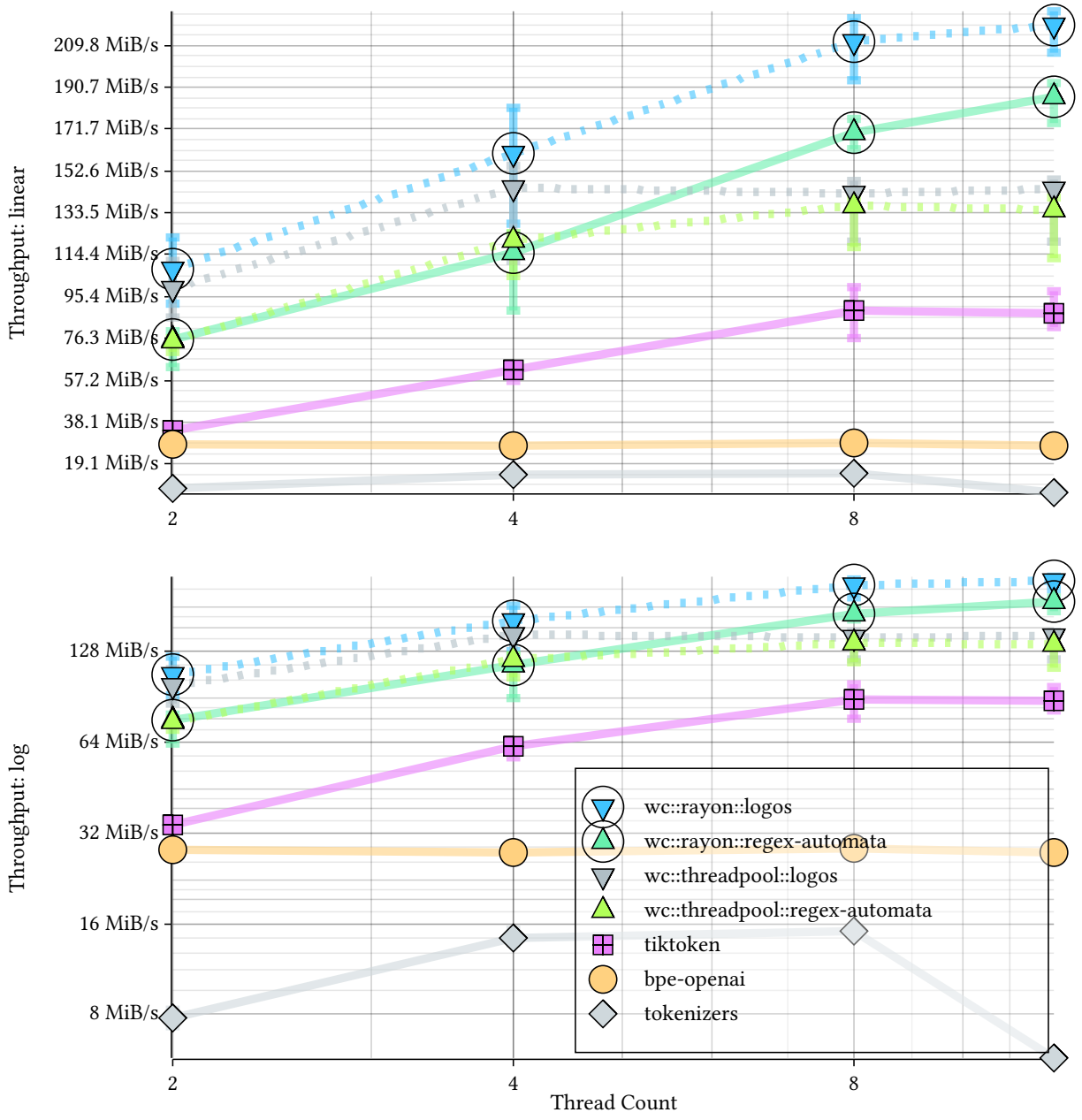
Benchmark 11: Rust Encoder Min/Mean/Max Throughput: m2x12, cl100k



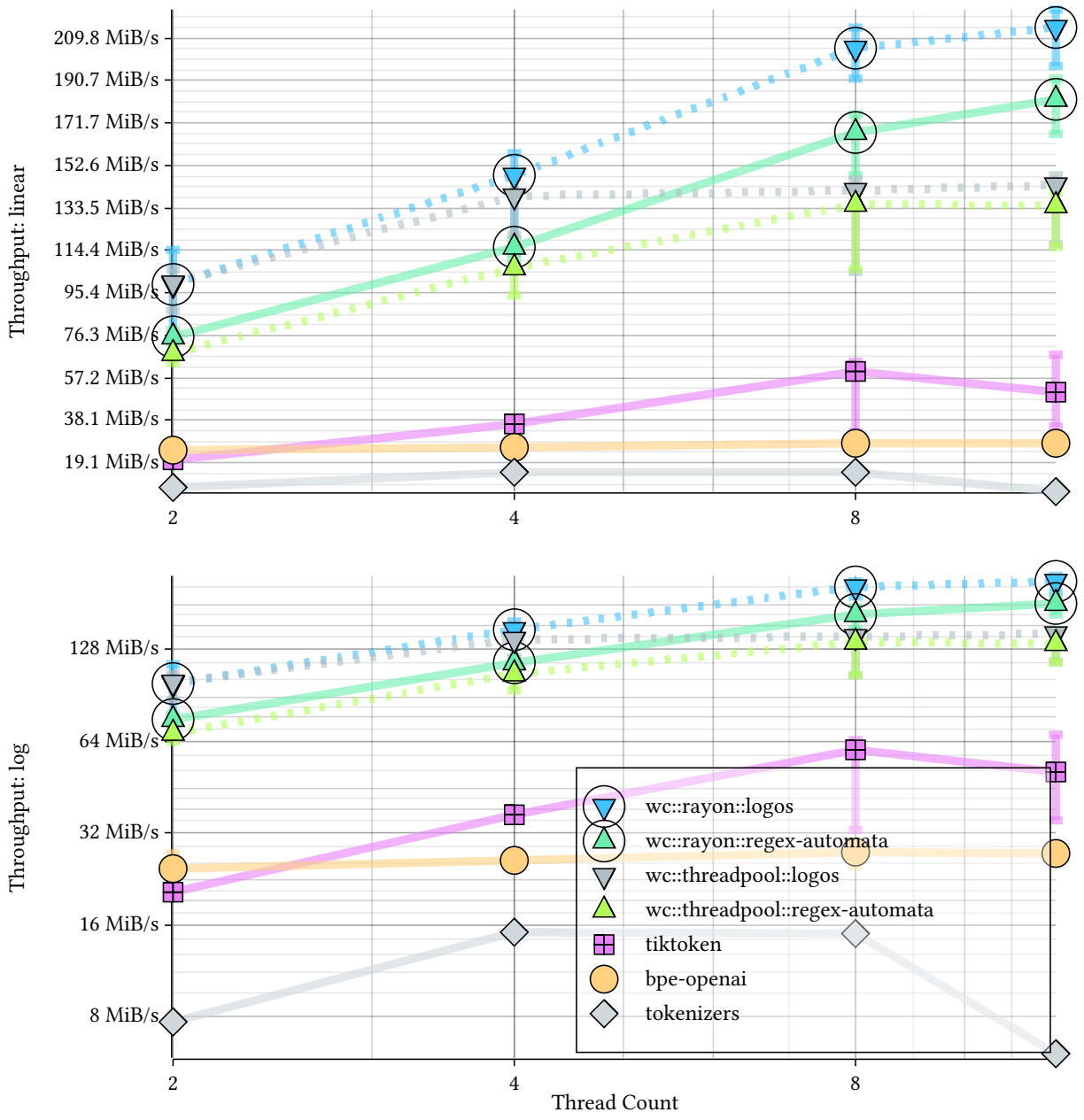
Benchmark 12: Rust Encoder Min/Mean/Max Throughput: m2x12, o200k



Benchmark 13: Python Encoder Min/Mean/Max Throughput: m2x12, gpt2



Benchmark 14: Python Encoder Min/Mean/Max Throughput: m2x12, cl100k



Benchmark 15: Python Encoder Min/Mean/Max Throughput: m2x12, o200k

15.2 Listings

15.2.1 BufferSweep Listing

```rust
#[derive(Default, Debug, Clone)]
pub struct BufferSweepSpanEncoder<T: TokenType> {
    // The working token buffer.
    working: Vec<T>,
}

impl<T: TokenType> SpanEncoder<T> for BufferSweepSpanEncoder<T> {
    fn encode_append_compound_span(
        &mut self,
        vocab: &UnifiedTokenVocab<T>,
        span: &[u8],
        tokens: &mut Vec<T>,
    ) {
        // Fill the working vec with the direct byte token translations.
        self.working.clear();
        vocab
            .byte_vocab()
            .append_tokens(span, self.working.as_mut());

        // Incrementally shrink the working memory
        // Until we can no longer find pairs to merge.
        while self.working.len() > 1 {
            // Find the lowest ranked merge available.
            if let Some((token, idx)) = self
                .working
                .windows(2)
                .enumerate()
                .filter_map(|(idx, w)| vocab.lookup_pair(&(w[0], w[1])).map(|token| (token, idx)))
                .min()
            {
                // buf[idx..=idx+1] (a, b) -> buf[idx] t
                self.working[idx] = token;
                self.working.remove(idx + 1);
            } else {
                // No more merges possible
                break;
            }
        }

        tokens.extend_from_slice(self.working.as_slice());
    }
}
```

15.2.2 TailSweep Listing

```rust
#[derive(Default, Debug, Clone)]
pub struct TailSweepSpanEncoder<T: TokenType> {
    marker: core::marker::PhantomData<T>,
}

impl<T: TokenType> SpanEncoder<T> for TailSweepSpanEncoder<T> {
    fn encode_append_compound_span(
        &mut self,
        vocab: &UnifiedTokenVocab<T>,
        span: &[u8],
        tokens: &mut Vec<T>,
    ) {
        // Reuse the output buffer as our working memory.
        // Append the byte-tokens to the buffer.
        let start = tokens.len();
        vocab.byte_vocab().append_tokens(span, tokens);

        // Incrementally shrink the working memory (the new buffer end)
        // Until we can no longer find pairs to merge.
        let stop = start + 2;
        while tokens.len() >= stop {
            // Find the lowest ranked merge available.
            if let Some((token, idx)) = tokens[start..]
                .windows(2)
                .enumerate()
                .filter_map(|(idx, w)| vocab.lookup_pair(&(w[0], w[1])).map(|token| (token, idx)))
                .min()
            {
                // Adjust the window index.
                let idx = start + idx;

                // buf[idx..=idx+1] (a, b) -> buf[idx] t
                tokens[idx] = token;
                tokens.remove(idx + 1);
            } else {
                // No more merges possible
                break;
            }
        }
    }
}
```

15.2.3 MergeHeap Listing

```rust
#[derive(Default, Debug, Clone)]
pub struct MergeHeapEncoder<T: TokenType> {
    pair_ranks: Vec<T>,
}

impl<T: TokenType> SpanEncoder<T> for MergeHeapEncoder<T> {
    fn encode_append_compound_span(
        &mut self,
        vocab: &UnifiedTokenVocab<T>,
        span: &[u8],
        tokens: &mut Vec<T>,
    ) {
        // We reuse the output buffer as our working memory.
        // - `start` is the first index of the working memory buffer.
        let start = tokens.len();

        // Define CURRENT as `tokens[start..]`.
        // - CURRENT[i] := tokens[start + i]
        vocab.byte_vocab().append_tokens(span, tokens);

        let pr_for_tokens = {
            |tok: &[T], a: usize, b: usize| {
                vocab
                    .lookup_pair(&(tok[start + a], tok[start + b]))
                    .unwrap_or(T::max_value())
            }
        };

        // We keep the following property:
        // - pair_ranks[i] = pairs.get(&(CURRENT[i], CURRENT[i + 1]))
        // - pair_ranks.len() = CURRENT.len() - 1 = end - start - 1
        self.pair_ranks.clear();
        self.pair_ranks
            .extend((0..(tokens.len() - start - 1)).map(|i| pr_for_tokens(tokens, i, i + 1)));

        while let Some((new_token, i)) = self
            .pair_ranks
            .iter()
            .enumerate()
            .filter_map(|(i, &new_token)| {
                if new_token != T::max_value() {
                    Some((new_token, i))
                } else {
                    None
                }
            })
            .min()
        {
            // At this point, i selects CURRENT[i], PAIR_RANKS[i] such that:
            // - PAIR_RANKS[i] != max_value
            // - PAIR_RANKS[i] is smallest

            // Set CURRENT[i] to the new target rank.
            tokens[start + i] = new_token;

            if i > 0 {
                // If there is a preceding token, recompute PAIR_RANKS[i-1].
                self.pair_ranks[i - 1] = pr_for_tokens(tokens, i - 1, i);
            }

            if i + 2 < tokens.len() - start {
                // If this pair rank exists,
                // it will become PAIR_RANKS[i] following the remove below.
                self.pair_ranks[i + 1] = pr_for_tokens(tokens, i, i + 2);
            }

            // Drop PAIR_RANKS[i] and CURRENT[i+1].
            self.pair_ranks.remove(i);
            tokens.remove(start + i + 1);
        }
    }
}
```

15.2.4 PriorityMerge Listing

```rust
const NONE: u32 = u32::MAX;

struct Node<T> {
    token: T,
    prev: u32,
    next: u32,
}

/// Heap entry representing a potential merge.
/// Ordered by (rank, `left_idx`) so the lowest-rank, leftmost pair is popped
/// first. `left_tok` and `right_tok` are stored for O(1) stale-entry detection.
#[derive(Eq)]
struct MergeEntry<T: Ord> {
    rank: T,
    left_idx: u32,
    left_tok: T,
    right_tok: T,
}

impl<T: Ord> PartialEq for MergeEntry<T> {
    fn eq(
        &self,
        other: &Self,
    ) -> bool {
        self.rank == other.rank && self.left_idx == other.left_idx
    }
}

impl<T: Ord> Ord for MergeEntry<T> {
    fn cmp(
        &self,
        other: &Self,
    ) -> core::cmp::Ordering {
        self.rank
            .cmp(&other.rank)
            .then(self.left_idx.cmp(&other.left_idx))
    }
}

impl<T: Ord> PartialOrd for MergeEntry<T> {
    fn partial_cmp(
        &self,
        other: &Self,
    ) -> Option<core::cmp::Ordering> {
        Some(self.cmp(other))
    }
}

/// A [`SpanEncoder`] using a binary min-heap with a doubly-linked list.
///
/// Processes BPE merges in O(n log n) time per span, compared to the
/// O(n^2) linear-scan approach used by other encoders.
pub struct PriorityMergeSpanEncoder<T: TokenType> {
    nodes: Vec<Node<T>>,
    heap: BinaryHeap<Reverse<MergeEntry<T>>>,
}

impl<T: TokenType> Default for PriorityMergeSpanEncoder<T> {
    fn default() -> Self {
        Self {
            nodes: Vec::new(),
            heap: BinaryHeap::new(),
        }
    }
}

impl<T: TokenType> core::fmt::Debug for PriorityMergeSpanEncoder<T> {
    fn fmt(
        &self,
        f: &mut core::fmt::Formatter<'_>,
    ) -> core::fmt::Result {
        f.debug_struct("PriorityMergeSpanEncoder").finish()
    }
}

impl<T: TokenType> Clone for PriorityMergeSpanEncoder<T> {
    fn clone(&self) -> Self {
        Self::default()
    }
}

impl<T: TokenType> SpanEncoder<T> for PriorityMergeSpanEncoder<T> {
    fn encode_append_compound_span(
        &mut self,
        vocab: &UnifiedTokenVocab<T>,
        span: &[u8],
        tokens: &mut Vec<T>,
    ) {
        let n = span.len();
        let byte_vocab = vocab.byte_vocab();

        if n < 2 {
            for &byte in span {
                tokens.push(byte_vocab.get_token(byte));
            }
            return;
        }

        // Build doubly-linked list of byte tokens.
        self.nodes.clear();
        self.nodes.reserve(n);
        for (i, &byte) in span.iter().enumerate() {
            self.nodes.push(Node {
                token: byte_vocab.get_token(byte),
                prev: if i == 0 { NONE } else { (i - 1) as u32 },
                next: if i + 1 < n { (i + 1) as u32 } else { NONE },
            });
        }

        // Seed the heap with all initially-mergeable adjacent pairs.
        self.heap.clear();
        for i in 0..(n - 1) {
            let left_tok = self.nodes[i].token;
            let right_tok = self.nodes[i + 1].token;
            if let Some(rank) = vocab.lookup_pair(&(left_tok, right_tok)) {
                self.heap.push(Reverse(MergeEntry {
                    rank,
                    left_idx: i as u32,
                    left_tok,
                    right_tok,
                }));
            }
        }

        // Process merges in priority order (lowest rank first).
        while let Some(Reverse(entry)) = self.heap.pop() {
            let li = entry.left_idx as usize;

            // Validate: left node still active with expected right neighbor.
            let ri_u32 = self.nodes[li].next;
            if ri_u32 == NONE {
                continue;
            }
            let ri = ri_u32 as usize;

            // Bidirectional adjacency + token freshness.
            if self.nodes[ri].prev != entry.left_idx
                || self.nodes[li].token != entry.left_tok
                || self.nodes[ri].token != entry.right_tok
            {
                continue;
            }

            // Merge: left absorbs right.
            let new_token = entry.rank;
            self.nodes[li].token = new_token;
            let right_next = self.nodes[ri].next;
            self.nodes[li].next = right_next;
            if right_next != NONE {
                self.nodes[right_next as usize].prev = entry.left_idx;
            }

            // Enqueue new neighbor pairs.
            let left_prev = self.nodes[li].prev;
            if left_prev != NONE {
                let prev_tok = self.nodes[left_prev as usize].token;
                if let Some(rank) = vocab.lookup_pair(&(prev_tok, new_token)) {
                    self.heap.push(Reverse(MergeEntry {
                        rank,
                        left_idx: left_prev,
                        left_tok: prev_tok,
                        right_tok: new_token,
                    }));
                }
            }
            if right_next != NONE {
                let next_tok = self.nodes[right_next as usize].token;
                if let Some(rank) = vocab.lookup_pair(&(new_token, next_tok)) {
                    self.heap.push(Reverse(MergeEntry {
                        rank,
                        left_idx: entry.left_idx,
                        left_tok: new_token,
                        right_tok: next_tok,
                    }));
                }
            }
        }

        // Collect final tokens by walking the linked list.
        let mut idx = 0u32;
        while idx != NONE {
            tokens.push(self.nodes[idx as usize].token);
            idx = self.nodes[idx as usize].next;
        }
    }
}
```

15.2.5 BpeBacktrack Listing

```rust
/// Precomputed BPE vocabulary data for the backtracking encoder.
///
/// Built once from a [`UnifiedTokenVocab`] and shared via [`Arc`].
pub struct BpeVocab<T> {
    /// `{ (T, T) -> T }` merge table (cloned from vocab).
    pair_lookup: WCHashMap<Pair<T>, T>,
    /// Indexed by `T.to_usize()`. Inverse of `pair_lookup`.
    /// Byte-level tokens map to `(self, self)`.
    split_table: Vec<Pair<T>>,
    /// For each token, the next-longest prefix token (or `T::max_value()`
    /// sentinel).
    next_prefix: Vec<T>,
    /// Aho-Corasick automaton over all token byte sequences.
    ac: AhoCorasick,
    /// Maps AC pattern index to token ID.
    ac_tokens: Vec<T>,
    /// Token byte lengths, indexed by `T.to_usize()`.
    token_lens: Vec<usize>,
}

impl<T: TokenType> BpeVocab<T> {
    /// Build from a [`UnifiedTokenVocab`].
    pub fn from_vocab(vocab: &UnifiedTokenVocab<T>) -> Self {
        // Collect all (bytes, token) pairs, sorted by token ID (= rank).
        let mut span_pairs: Vec<(Vec<u8>, T)> = vocab.span_pairs().collect();
        span_pairs.sort_by_key(|(_, t)| *t);

        let max_token = span_pairs
            .last()
            .map(|(_, t)| t.to_usize().unwrap())
            .unwrap_or(0);
        let table_size = max_token + 1;

        // Build token_lens.
        let mut token_lens = vec![0usize; table_size];
        for (bytes, tok) in &span_pairs {
            token_lens[tok.to_usize().unwrap()] = bytes.len();
        }

        // Build AC automaton (leftmost-longest).
        let patterns: Vec<&[u8]> = span_pairs.iter().map(|(b, _)|
b.as_slice()).collect();
        let ac_tokens: Vec<T> = span_pairs.iter().map(|(_, t)| *t).collect();
        let ac = AhoCorasick::builder()
            .match_kind(MatchKind::LeftmostLongest)
            .build(&patterns)
            .expect("failed to build AhoCorasick automaton");

        // Build next_prefix: for each token, the longest prefix token shorter by 1+
        // bytes.
        let mut next_prefix = vec![T::max_value(); table_size];
        for (bytes, tok) in &span_pairs {
            if bytes.len() > 1
                && let Some(mat) = ac.find(&bytes[..bytes.len() - 1])
            {
                next_prefix[tok.to_usize().unwrap()] =
ac_tokens[mat.pattern().as_usize()];
            }
        }

        // Build pair_lookup and split_table incrementally in rank order.
        // For each token, walk the next_prefix chain to find the canonical
        // split (prefix, suffix) where both halves are lower-ranked.
        let mut pair_lookup: WCHashMap<Pair<T>, T> = WCHashMap::default();
        let mut split_table: Vec<Pair<T>> = vec![(T::zero(), T::zero()); table_size];

        for &(ref bytes, tok) in &span_pairs {
            let id = tok.to_usize().unwrap();
            let mut prefix_tok = next_prefix[id];
            let mut found = false;

            while prefix_tok != T::max_value() {
                let prefix_len = token_lens[prefix_tok.to_usize().unwrap()];
                let suffix_bytes = &bytes[prefix_len..];
                // Look up suffix as a token via AC.
                if let Some(mat) = ac.find(suffix_bytes) {
                    let suffix_tok = ac_tokens[mat.pattern().as_usize()];
                    // Verify the AC match covers exactly the suffix bytes.
                    if mat.start() == 0
                        && mat.end() == suffix_bytes.len()
                        && prefix_tok < tok
                        && suffix_tok < tok
                        && is_valid_token_pair(&pair_lookup, &split_table, prefix_tok,
suffix_tok)
                    {
                        pair_lookup.insert((prefix_tok, suffix_tok), tok);
                        split_table[id] = (prefix_tok, suffix_tok);
                        found = true;
                        break;
                    }
                }
                // Try a shorter prefix.
                prefix_tok = next_prefix[prefix_tok.to_usize().unwrap()];
            }
            if !found {
                // Leaf token (byte-level or no valid split found).
                split_table[id] = (tok, tok);
            }
        }

        Self {
            pair_lookup,
            split_table,
            next_prefix,
            ac,
            ac_tokens,
            token_lens,
        }
    }

    /// Find the longest matching token starting at the beginning of `text`.
    #[inline]
    fn next_match(
        &self,
        text: &[u8],
    ) -> Option<T> {
        self.ac
            .find(text)
            .map(|m| self.ac_tokens[m.pattern().as_usize()])
    }

    /// Get the next-shorter prefix token, or `None` if at a leaf.
    #[inline]
    fn next_prefix_of(
        &self,
        token: T,
    ) -> Option<T> {
        let p = self.next_prefix[token.to_usize().unwrap()];
        if p == T::max_value() { None } else { Some(p) }
    }

    /// Get the byte length of a token.
    #[inline]
    fn token_len(
        &self,
        token: T,
    ) -> usize {
        self.token_lens[token.to_usize().unwrap()]
    }

    /// Check whether two adjacent tokens form a valid BPE split boundary.
    #[inline]
    fn is_valid_token_pair(
        &self,
        t1: T,
        t2: T,
    ) -> bool {
        is_valid_token_pair(&self.pair_lookup, &self.split_table, t1, t2)
    }
}

/// Check whether two adjacent tokens form a valid BPE boundary.
///
/// Recursively undoes BPE merges to verify that no merge rule would have
/// combined bytes across the split point at a lower rank.
fn is_valid_token_pair<T: TokenType>(
    pair_lookup: &WCHashMap<Pair<T>, T>,
    split_table: &[Pair<T>],
    mut token1: T,
    mut token2: T,
) -> bool {
    let mut limit = T::max_value();
    loop {
        if let Some(&combined) = pair_lookup.get(&(token1, token2))
            && combined < limit
        {
            return false;
        }
        if token1 > token2 {
            limit = token1;
            token1 = split_table[token1.to_usize().unwrap()].1;
            if token1 == limit {
                limit = token2 + T::one();
                token2 = split_table[token2.to_usize().unwrap()].0;
                if token2 + T::one() == limit {
                    return true;
                }
            }
        } else {
            limit = token2 + T::one();
            token2 = split_table[token2.to_usize().unwrap()].0;
            if token2 + T::one() == limit {
                limit = token1;
                token1 = split_table[token1.to_usize().unwrap()].1;
                if token1 == limit {
                    return true;
```

```
                }
            }
        }
    }
}

/// Bitfield with all bits initially set to 1.
struct BitField {
    words: Vec<u64>,
}

impl BitField {
    fn new(bits: usize) -> Self {
        Self {
            words: vec![u64::MAX; bits.div_ceil(64)],
        }
    }

    #[inline]
    fn is_set(
        &self,
        bit: usize,
    ) -> bool {
        let (word, bit) = (bit / 64, bit % 64);
        self.words[word] & (1 << bit) != 0
    }

    #[inline]
    fn clear(
        &mut self,
        bit: usize,
    ) {
        let (word, bit) = (bit / 64, bit % 64);
        self.words[word] &= !(1 << bit);
    }

    fn reset(
        &mut self,
        bits: usize,
    ) {
        let needed = bits.div_ceil(64);
        self.words.clear();
        self.words.resize(needed, u64::MAX);
    }
}

/// A [`SpanEncoder`] using BPE backtracking with Aho-Corasick matching.
pub struct BpeBacktrackSpanEncoder<T> {
    vocab: Arc<BpeVocab<T>>,
    bitfield: BitField,
}

impl<T: TokenType> BpeBacktrackSpanEncoder<T> {
    /// Create a new encoder from a shared [`BpeVocab`].
    pub fn new(vocab: Arc<BpeVocab<T>>) -> Self {
        Self {
            vocab,
            bitfield: BitField::new(0),
        }
    }
}

impl<T: TokenType> core::fmt::Debug for BpeBacktrackSpanEncoder<T> {
    fn fmt(
        &self,
        f: &mut core::fmt::Formatter<'_>,
    ) -> core::fmt::Result {
        f.debug_struct("BpeBacktrackSpanEncoder").finish()
    }
}

impl<T: TokenType> SpanEncoder<T> for BpeBacktrackSpanEncoder<T> {
    fn encode_append_compound_span(
        &mut self,
        _vocab: &UnifiedTokenVocab<T>,
        span: &[u8],
        tokens: &mut Vec<T>,
    ) {
        if span.is_empty() {
            return;
        }

        let bpe = &*self.vocab;
        self.bitfield.reset(span.len() + 1);

        let mut pos = 0usize;
        let mut next_token = bpe.next_match(span);

        // buf holds tokens produced for *this span* so we can backtrack.
        let base = tokens.len();

        while let Some(mut token) = next_token {
            let last = if tokens.len() > base {
                Some(tokens[tokens.len() - 1])
            } else {
                None
            };

            loop {
                let end_pos = pos + bpe.token_len(token);
                if self.bitfield.is_set(end_pos)
                    && last
                        .map(|lt| bpe.is_valid_token_pair(lt, token))
                        .unwrap_or(true)
                {
                    // Accept this token.
                    tokens.push(token);
                    pos = end_pos;
                    next_token = bpe.next_match(&span[end_pos..]);
                    break;
                } else if let Some(shorter) = bpe.next_prefix_of(token) {
                    // Try a shorter prefix.
                    token = shorter;
                } else {
                    // Backtrack: mark position as visited, pop previous token.
                    self.bitfield.clear(pos);
                    if let Some(prev) = last {
                        tokens.pop();
                        pos -= bpe.token_len(prev);
                        next_token = Some(prev);
                    } else {
                        next_token = None;
                    }
                    break;
                }
            }
        }
    }
}
```