

suBPEriod - Byte Pair Encoding at speed

Jarrold Connolly

September 21, 2026

Abstract

suBPEriod is a C++ Byte Pair Encoding encoder I wrote so I could work on speed without giving up accuracy. Unit tests require the same token ids as tiktoken, including Unicode, and a fuzzer checks that odd inputs do not hang or crash. I kept a change only when a second run and the profiles agreed. On a 100 megabyte English sample it used about half of tiktoken's time, and Chinese stayed near parity.

Introduction

Why build another Byte Pair Encoding tokenizer when tiktoken already exists?

I wanted an encoder written from scratch so I could focus on performance, and I chose C++ for portability and speed. tiktoken was a good reference, because it is a well known implementation and modern OpenAI models use it. I wanted to own the whole pipeline so I could tune every stage, and so I could research and test every data structure and algorithm in it.

suBPEriod is that encoder. It loads o200k_base, the well known vocabulary used by GPT-4o and the o-series models, then encodes text into token ids and decodes those ids back into text, with no training loop involved. I follow the tiktoken contract, so the same text has to come out as the same token ids. I develop the C++ on Linux, and that is the machine I measure on.

Before any of the speed work, I treated precision as equal to performance, so all of the unit tests run against both implementations and require exact token output. The interesting part showed up only after that match held and the encoder was actually fast. I did not find one tokenizer with one bottleneck. I found two regimes, and the time does not sit in the same place in both.

This is the baseline I can reason about, and the rest of the article is how I checked that the encoder was correct, how a fuzzer covers inputs the unit tests miss, how I kept the measurements honest, and which changes the profiles supported.

Correctness Before Speed

The pre-tokenizer is where a BPE encoder usually goes wrong. o200k splits text into chunks before any merge happens, and I wrote that scanner myself so I could see it and change it. A scanner can look fine on English and still split other languages differently from tiktoken, and I would not trust a speed number from a scanner that does not match.

The check is exact token output, so the unit tests run the same input through subPeriod and tiktoken and diff the token ids, and a separate set of cases covers Chinese, Japanese, Korean, Arabic, Greek, Cyrillic, Hebrew, Thai, Hindi, emoji, and whitespace that is easy to get wrong. The vocabulary is byte-level, so every byte value has a token, and a bad split shows up as the wrong ids. There is no unknown-token id, so a bad split cannot pass as an unknown byte.

The merge itself is the usual greedy loop, which always merges the adjacent pair with the best rank and repeats until no legal merge is left. That part is well documented, so my job was to feed it the same pieces tiktoken would have fed it. I got into the habit of not keeping an English-only win until the Unicode cases matched, because a speedup on a scanner that is wrong is a measurement of a different program.

A fuzzer for inputs I did not write

Those unit tests are strings I chose, so they will not catch a hang on a byte sequence I never wrote down. The fuzzer builds those inputs by mixing random bytes, plain ASCII, valid UTF-8, UTF-8 with bits flipped so the bytes are no longer valid, and short slices of real text that it then changes by inserting, deleting, or replacing bytes. A mixed run picks among those, and it can run the pre-tokenizer, the encoder, or both.

Each input runs on its own with a limit of 250 milliseconds, so a hang becomes a saved case instead of a run that never finishes. I can replay that case from the saved bytes, or from the seed that produced them.

The fuzzer does not check token ids. The unit tests do that, including the Unicode cases, and the fuzzer only checks that strange inputs do not hang or crash.

Measuring Speed

Once those diffs were clean, I measured single-thread encode throughput. Both sides use the same vocabulary, already loaded, and I time the encode itself. tiktoken's timed call is its Rust encode path, because I wanted the comparison to be against the library people actually run, on the same terms.

The first profiles spent a real share of their samples writing token ids to the terminal, so I added a silent mode and kept that output out of the timed run. I read the corpus in large fixed chunks, about 8 MiB, so memory use stays flat as the file grows, and I copy the corpus onto a memory filesystem so disk time is not part of the result. I take a warm run first, which loads the vocabulary into the file cache, and tiktoken gets the same warm-up, with its vocabulary loaded before I start timing.

I run at least twice. A swing of 10 to 15 percent between runs is normal on this machine, so I write down the range, because one fast number from a noisy machine is not a result.

The files I measure

text8 is the easy file, the usual ASCII Wikipedia extract of about 96 MiB, and it is what I run first.

The set I trust is six Wikipedia slices, one each for English, Japanese, Chinese, Arabic, Russian, and Greek. Each file is cut to the same length, 100 MiB, exactly 104,857,600 bytes. I stream them from a Wikipedia snapshot and stop at that size, so I am not downloading a whole language dump just to compare encoders. The same length means a slower file is slower text, not a bigger file.

English and Chinese do not put the time in the same place, so a change that only wins on text8, or on the English slice, is a result about ASCII. I also keep 1 GiB copies of the same languages, and I use those after a 100 MiB win, to see whether the result still holds when the file is about ten times as long.

The figure and the table below are one recorded run from 27 May 2026, and a ratio under 1 means suBPEriod used less wall time than tiktoken. Tokens per second are there for intuition, but the ratio is the comparison I keep, because absolute seconds drift from day to day even when the code does not.

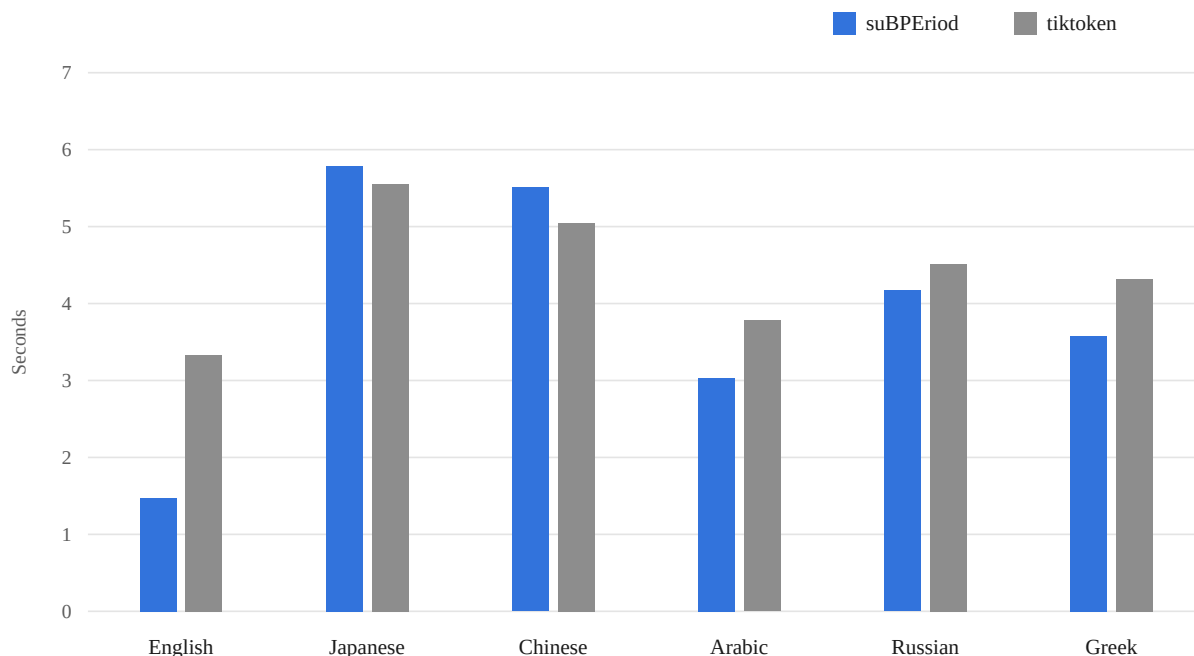


Figure 1: Wall time on six 100 megabyte samples, suBPEriod beside tiktoken

Corpus	suBPEriod	tiktoken	Ratio	Tokens/sec
English	1.47 s	3.33 s	0.44x	15.8M
Japanese	5.79 s	5.55 s	1.04x	5.4M
Chinese	5.51 s	5.05 s	1.09x	6.0M
Arabic	3.03 s	3.78 s	0.80x	6.5M
Russian	4.17 s	4.51 s	0.92x	4.5M
Greek	3.58 s	4.32 s	0.83x	6.6M

A second run the same day moved every ratio. English was 0.40x, Japanese 1.08x, Chinese 1.22x, Arabic 0.76x, Russian 0.89x, and Greek 0.86x. Chinese moved the most, which is why I treat it as near parity, sometimes a little slower. English at roughly half the wall time, and the clearer wins on Arabic and Greek, are the results I trust. text8, measured as a range across runs, sits at 0.48x to 0.51x, about 1.16 to 1.26 seconds against about 2.43 to 2.48 seconds for tiktoken.

Where the Time Goes

The ratios tell me which file got faster, but they do not tell me why. I use a sampling profiler to see which functions take the time, and I use VTune when I need the cache misses and the memory behavior. A hash probe can look cheap in a call graph and still be stalled on a key that does not share a cache line with the

key it just checked, and that is what turned the vocabulary layout from a guess into a measurement. I trust a wall-time ratio more when VTune and the sampling profile agree on the cause.

On text8, and on English Wikipedia at 100 MiB and at 1 GiB, the time sits in hash lookup, key comparison, and the pre-tokenizer scan across a word, and that scan takes a larger share as the merge loop gets cheaper. The merge function is still about half the samples if you count everything it calls, but its own time is smaller, because most of those samples are the lookups under it.

Chinese and Japanese have a different shape, and the 1 GiB files made that obvious. The merge routine is about 35 to 43 percent of its own samples, and about 90 percent once you include what it calls. After I replaced a binary search over Unicode ranges with a direct lookup, the pre-tokenizer fell to a few percent. Hashing is still roughly half the samples if you add the probe, the compare, and the hash mix together, but the shape is still the merge loop, because there are more merges per byte, the pieces are shorter, and there are more of them.

Arabic, Russian, and Greek sit between those two. I still measure English and Chinese before I keep a change, because a win on one and a regression on the other is a failed change, even when the English number alone would look good.

I stopped looking for one change that helps both files. On English I want fewer probes, cheaper compares, and a faster scan, and on Chinese I want a cheaper merge, so I split the work that way on purpose.

Changes That Stayed

A few ideas stayed in the code, each with a before and after on the same files, and the profiles are the evidence.

Cache the pair ranks

The obvious merge looks up the rank of every adjacent pair on every pass, until nothing is left to merge, and that repeats a lot of work because the pair that was second best a moment ago is usually still second best. I precompute the rank of every adjacent pair. After a merge, only the two edges that touch the new piece are out of date, so only those two need a fresh lookup, and the lookup count drops from a full pass per merge to one pass up front, plus two updates each time a merge happens. On text8 that was the step from about 2.5 times tiktoken's wall time down to a little over 2.1 times, and later merge experiments assume those ranks are already integers in a tight array.

Skip the merge when the word is already a token

Before the merge loop I check whether the whole pre-token is already in the vocabulary, and if it is, I emit that id and move on.

A paper on faster superword tokenization describes the same shortcut, and on a vocabulary of this size most pre-tokens are already one token, which I measured on my own runs. In that same stretch of work, text8 dropped from a bit over twice tiktoken's time to about half. Two other changes landed with it, walking only the pieces still active after each merge and keeping short words off the heap. I report them together, because that is how they landed in the recorded runs. If I were starting again, the direct vocabulary check is the one I would try first, since a full merge loop for a word the vocabulary already contains is wasted work, and English has a lot of those words.

Keep short words on the stack

English is mostly short pieces, so allocating several working arrays on the heap for every word, then freeing them, showed up in the samples once the bigger costs were gone. Words that fit in a small fixed buffer stay on the stack, and longer pieces still use the heap.

Heap traffic in the profiles dropped sharply, and text8 picked up another 5 to 10 percent of wall time and landed in the 0.49x to 0.51x range. This is a heuristic about English text, not a claim about every language. Chinese does not fit in that buffer, which is one more reason a text8 win is not a reason to stop measuring the other files.

A flat property table, then a tighter vocabulary

Binary search over Unicode property ranges showed up on Chinese, so a direct lookup for the Basic Multilingual Plane took that search out of the profiles. Pre-tokenizer time on Chinese Wikipedia, including what it calls, fell to about 3 percent, and wall time improved by roughly 8 percent. ASCII barely changed, because that path is rarely used on plain English.

Then I changed how the vocabulary sits in memory, which is the change the cache view pointed at. Token bytes stay in one contiguous block for the life of the run, and the hash table stores small references into that block, so more keys share a cache line than they did when each token was its own string. Decode does not hash, because the ranks in this file are a dense range and the decoder indexes an array.

On the runs just after that change, Chinese moved to about 1.01x to 1.03x on a 100 MiB slice, and later runs sat between about 1.09x and 1.22x, while text8 stayed at about half of tiktoken's time. Sampling profiles still show key comparison near the top on English. The cache views are why I trust the layout, and why I stopped expecting another string-key change to finish the job.

The text8 history is short. A normal chained hash map of strings started at 4.56 times tiktoken, a flat open-addressing table brought that to 3.34 times, and reserving extra slots after load, so the table sat closer to half full than to full, brought it to 2.50 times. Average probe length dropped, and startup paid for one rehash. The rank cache, the known-word skip, and the stack buffers did the rest. The packed vocabulary is what moved Chinese, because English was already fast by then.

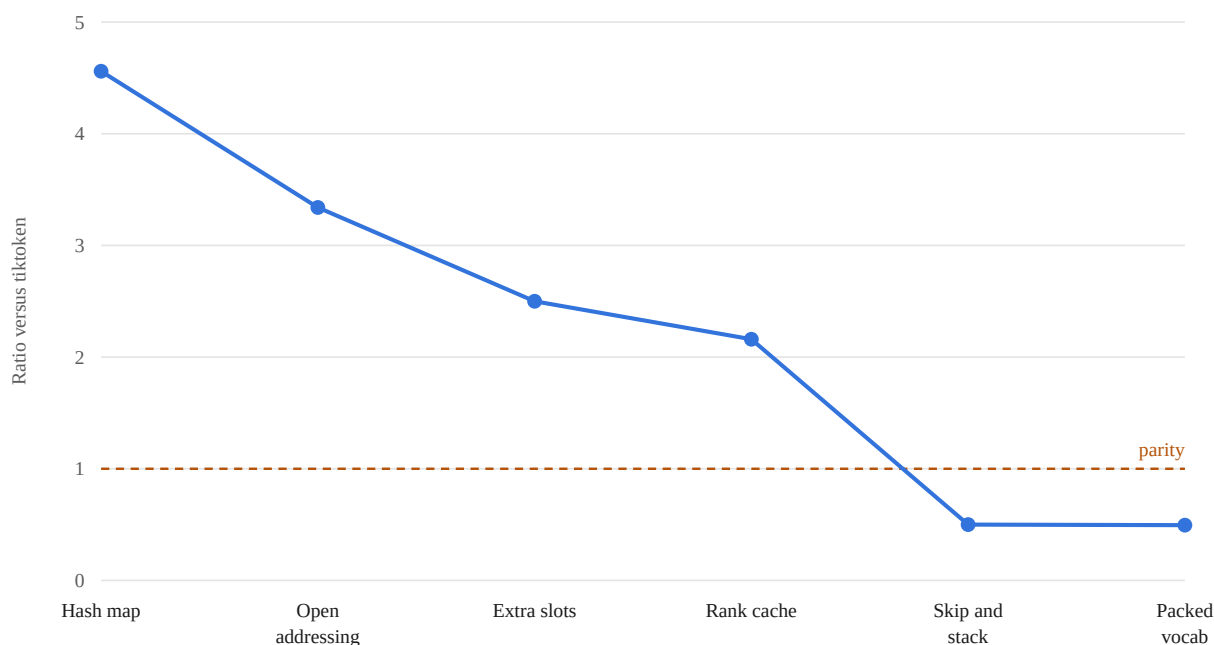


Figure 2: text8 ratio versus tiktoken across those changes

Where a step was a range, the point is the midpoint. Cached pair ranks were 2.13 to 2.19, the known-word skip and stack buffers were 0.49 to 0.51, and the packed vocabulary was 0.48 to 0.51.

What I Reverted

The changes above are the easy ones to write about, but the ones I reverted are where I learned how this program actually behaves. Each one looked reasonable before I measured it, and each one was slower.

Two tables for short keys

Most tokens in o200k are only a few bytes, so comparing those keys as integers should be cheaper than comparing them as strings, and for a while I was sure that was the next step. The profiles agreed with that narrow claim, and equality did get cheaper.

Splitting the vocabulary into two tables did not help. One table held the short keys and the other held everything else, so every lookup paid for a choice of table and then a probe in that table, and the probe loop existed twice. That showed up as instruction-cache pressure, because the two bucket arrays competed for the same cache. The integer compare was faster, but the wall time was still worse, from about 0.50x back toward 0.60x on text8, so I reverted the split.

Comparing short keys as integers is still a sound idea, but if I inline them later, they belong in the same table as everything else.

A faster hash that clustered

I swapped in a multiply-shift hash that is cheap per call and common for integer keys, and on this vocabulary the probe chains got longer. The function that steps to the next slot took more of the profile, and text8 went from about 0.53x to about 0.73x.

I put the stronger mix back. It costs more per hash, but it spreads roughly 200 thousand strings well enough that those extra probes do not show up. Faster hashing was the wrong target, and a more even spread was the right one. A weak mix is fine for a small set of integers, but it was a bad choice for this table.

A heap, too early

A heap of candidate merges should cost less than scanning the live pairs to find the best rank, at least on paper, so I tried it early, before pair ranks were cached. Every pop paid for a hash lookup, and the words were short. The linear scan stayed in cache, and the heap did not, because each step followed a pointer. The ratio got worse, back toward 4 times slower than tiktoken, so I reverted it, and that was the right call for that version of the code.

I expect to try it again for Chinese. The merge loop is now about 35 to 43 percent of its own time at 1 GiB, and the ranks are already integers, so a pop can be a compare, which is a different cost than the one I measured the first time. That result belongs to that version of the code, because the data around it has changed.

This is the clearest failure I have. An algorithm can be faster in a textbook and slower in a profile, until the data around it changes.

What These Numbers Cover

I do not want the claim to be bigger than the runs behind it. The comparison is single-thread encode throughput against tiktoken on the same vocabulary, and a batch encoder that spreads independent chunks across cores would be a separate measurement, with its own table.

The machine is one WSL2 host. I publish ratios because the seconds move from day to day, and the code behind these runs has not changed since they were collected. If I re-run on a different machine, the table should be updated with it.

Two experiments are next, and neither is a result yet. One is a wider scan of the ASCII classification path, because that scan takes a larger share of the profile as everything else gets cheaper, and the other is a heap inside the merge loop, now that the ranks are cached, aimed at Chinese and Japanese. suBPEriod is the baseline I can explain.

Conclusion

The method was to match tiktoken, then change only what the measurements supported, and that took more care than any single optimization. On English, the useful changes were fewer probes, cheaper compares, and a scan that stays off the heap, while on Chinese the merge is still the expensive part. The failed experiments belong here too. A split table, a cheaper hash, and an early heap all looked reasonable, and the profiles showed they were slower.

If you are tuning a tokenizer, or anything else where the hot path is a hash probe around a small merge, measure a file where you already look fast and a file where you do not. Read the samples until you can name the functions, then look at the cache lines and the memory bandwidth, because that is how you tell whether a faster function is actually waiting on data.

The ratio I trust is the one that still holds on both files, in the sampling profile and in VTune, and on a second run.