Evaluating Tokenizers for Memory, Compute and Cache Efficiency: YouTokenToMe v/s TikToken

Divy Patel, Smit Shah, Sujay Chandra Shekara Sharma, Venkata Abhijeeth Balabhadruni
(dspatel6, spshah25, schandrashe5, balabhadruni)@wisc.edu
Group - 10

1 Introduction

Natural language processing (NLP) is an interdisciplinary subfield of computer science and linguistics. The primary goal of NLP is to give computers the ability to interpret, manipulate, and generate human language. The field of NLP enjoys a long history dating back to the famous paper by Alan Turing [17] which proposes the task of understanding and processing human text as a barometer of intelligence for a computer. However, the field has gained renewed focus and importance in recent years with the development of models based on neural networks that were able to outperform statistical methods that were considered to be state-of-the-art [1]. As a result, significant strides have been made in NLP, and NLP techniques are now used for a variety of problems like text classification and extraction, named entity recognition, sentiment analysis, and more recently, building AI chatbots [6].

One fundamental step in most NLP tasks is that of tokenization which involves breaking down raw text into smaller atomic units called tokens [20]. These tokens could be words, subwords, or even characters depending on the tokenizer being employed and the NLP task. Tokenization forms the foundation of many NLP tasks by transforming raw textual data into a structured format that can be easily processed by machine learning models. Since this forms one of the first steps in most NLP tasks, any errors in tokenization can lead to incorrect interpretations of the text and further errors in the downstream NLP tasks [5]. Given the importance of tokenization to the overall success of an NLP task and the large number of tokenizers available, it becomes vital to evaluate tokenizers on a variety of metrics and choose the tokenizer that is best suited to a specific task [7].

In this project, we explore the popular open-source tokenizer YouTokenToMe [19] and evaluate it using various profiling tools to get more insights about its performance bottlenecks. We also aim to identify potential areas for improvement by analyzing its memory utilization at each level of the caching hierarchy. Finally, we compare YouTokenToMe to OpenAI's TikToken [8] which acts as the benchmark for our experiments.

In the past, researchers have compared multilingual models with monolingual models by testing them on tasks designed for one language, using different tokenizers. They found that using monolingual tokenizers with multilingual models for these tasks can improve downstream performance of the pipeline. They use measures like F1 score and accuracy, to evaluate the performance of various downstream models in context with monolingual tokenizers utilized upstream.

In addition to that, there has been some work evaluating the compression factor of various tokenizers. They show how tokenizers with higher compression rates can increase the context lengths of the models and hence, can increase the amount of input text that can be ingested as part of inference. Also, they show how the compression rate of tokenizers affects the downstream memory usage and compute usage.

Current work focuses on evaluating the compression factor and accuracy, F1 score of tokenizers. None of the recent work evaluates the memory and cache efficiency of tokenizers. If we wish to verify the system level performance and identify the bottlenecks there is no existing work for the same. We plan to see the impact of two tokenizers (YouTokenToMe and Tiktoken) on system performance, specifically the following:

1. How does an increase in word count have an impact on CPU cache performance?
2. How does an increase in word count have an impact on the latency of the tokenizer?
3. How does CPU time change over different tokenization algorithms?
4. How does CPU utilization change over different tokenization algorithms?
5. How do the above changes vary on different languages like Hindi and Russian?

We expect to see what function in the tokenizer algorithm is a bottleneck causing an increase in cache and CPU latency. The rest of this report is structured as follows:

- Section 2 provides background on the tokenization algorithm used by both TikToken and YouTokenToMe.
- Section 3 describes the methodology followed in order to obtain metrics related to the compute, memory, and cache efficiency for these two tokenizers.
- Section 4 presents the results of our evaluation experiments and an analysis of these results.
- Section 5 covers some of the related work concerning the evaluation of tokenizers.
- Section 6 details some of the challenges faced by us while working on this project.
- Section 7 highlights some of the avenues for future work related to Tokenizer performance.
- Section 8 summarizes some of the key takeaways from the project.
- Section 9 goes over the contributions of each team member.

2 Background

Natural Language Processing (NLP) tasks involve the processing and understanding of human language by computers. Tokenization, a fundamental step in many NLP tasks, entails breaking down raw text into smaller atomic units called tokens, which could be words, subwords, or characters. This process facilitates further analysis and manipulation of text by machine learning models.

Various tokenization algorithms and tools are available and the choice of tokenizer can significantly impact the performance of downstream NLP tasks. Understanding the impact of tokenization algorithms on system-level performance is essential for optimizing NLP pipelines, especially in resource-constrained environments. By identifying performance bottlenecks and inefficiencies in tokenization algorithms, researchers and practitioners can develop strategies to enhance overall system performance and scalability. One very popular and frequently used tokenization algorithm is Byte Pair Encoding (BPE) [13].

BPE is an algorithm for encoding strings of text into subword-based tokens for use in downstream modeling. The basic idea of BPE is to iteratively merge the most frequent pair of consecutive characters in a text corpus into a new token until a predefined vocabulary size is reached.

The BPE algorithm consists of the following steps:

1. Initialize the vocabulary with all the characters in the text corpus.
2. Calculate the frequency of each character in the text corpus.
3. Repeat the following steps until the desired vocabulary size is reached:
   (a) Find the most frequent pair of consecutive characters in the text corpus.
   (b) Merge the pair to create a new subword unit.
   (c) Update the frequency counts of all the characters that contain the merged pair.
   (d) Add the new subword unit to the vocabulary.
4. Represent the text corpus using the subword units in the vocabulary.

In this project, we take a look at the compute, memory and cache efficiency of tokenizers such as YouTokenToMe and TikToken which are based on the Byte Pair Encoding algorithm. These have gained popularity for their effectiveness in processing textual data in recent years.

TikToken is a fast open-source BPE-based tokenizer developed by OpenAI which is now being used by OpenAI's Large Language Models (LLMs).

YouTokenToMe is an open-source unsupervised text tokenizer focused on computational efficiency. It currently implements fast BPE in C++ and leverages additional features like multi-threading and BPE-dropout [11] in order to be more performant.

3 Design

We prompt the user to input the text file they wish to tokenize along with the minimum and maximum sentence lengths they desire for statistical profiling. We then iterate through the range from the *minLength* to the *maxLength* in multiples of two.

For example, if *minLength* is 100 and *maxLength* is 500, then we will consider sentences having 100 words, 200 words, and 400 words. For each sentence we used *perfstat* to calculate statistics like total time taken for tokenization, L1/L3 cache hits and misses, overall cache hits and misses, and number of instructions (FLOPS) for the whole tokenization.

Now, for every sentence length, there can be varying word lengths, which might add variance. For example,

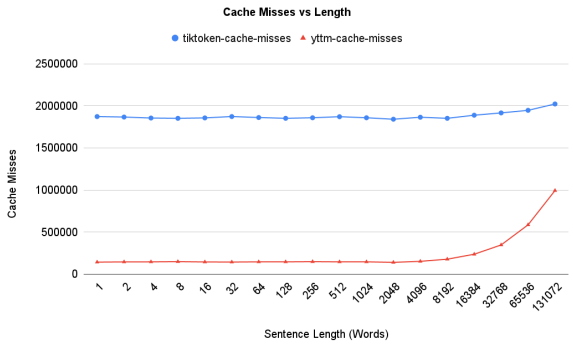

Figure 1: Cache Misses vs Sentence Length (in words)

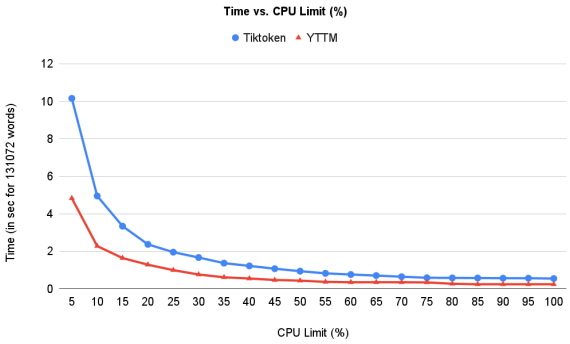

Figure 2: Time vs CPU Limit (%)

consider two sentences with 250 words, one can have words of smaller length and one can have words of longer length. To remove this variance, we profile 50 different sentences for each sentence length and then average it out.

The 50 sentences are generated in the following way: If we want a sentence with x words, the first x words of the file will be the first sentence, then the x+1..2x words in the file will be the second sentence and so on. Here we can reach to a point where number of words we want are more than the number of words available from that point to end of file. In that case, we take a cyclic approach and consider the remaining words from the start of the file.

To obtain all the aforementioned statistics, we executed the Linux command *perf-stat* within a Python3 script, utilizing the *subprocess* module for this purpose.

We also tried to limit the CPU available for tokenization using an enhanced version of the tool *cpulimit* by *Opsengine* [9]. We have profiled with 25%, 50%, 100% CPU limits while varying sentence lengths from 1 to 525k.

While TikToken is a pre-trained model, YouTokenToMe required training as it wasn't pre-trained. We trained the TikToken model and utilized the Flores dataset [4] for profiling statistics during the process.

We also profiled the above mentioned stats for two other languages (Hindi, Russian) in addition to English to see how the tokenizer performs on different languages.

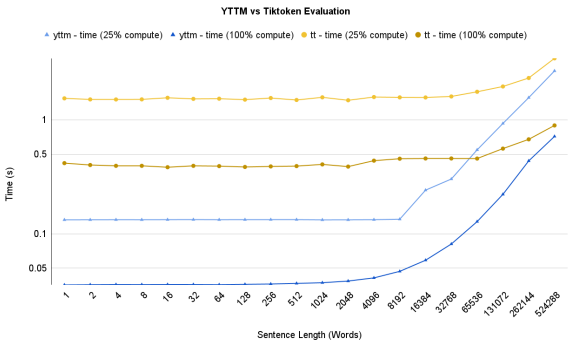

Figure 3: Perform comparison of both tokenizers with 25, 50, 100% CPU Limit

4 Evaluation

Figure 1 tries to capture the effect of varying sentence length on cache misses. It was achieved through the *perf-stat* tool. We observe that on increasing the sentence length, the number of cache misses increases for YouTokenToMe steeply after a threshold.

Next in *Figure 2*, we try to capture the impact of CPU utilization on the tokenization time for a sentence of

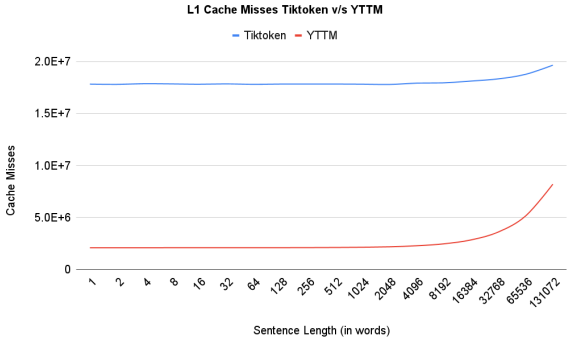

Figure 4: L1 Cache Misses Tiktoken vs YTTM

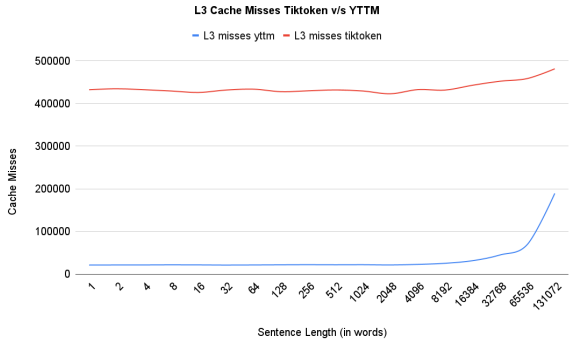

Figure 5: L3 Cache Misses Tiktoken vs YTTM

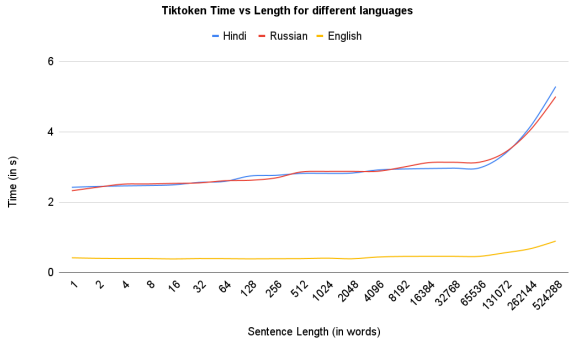

Figure 6: Tiktoken Time vs Length for Hindi, Russian and English languages

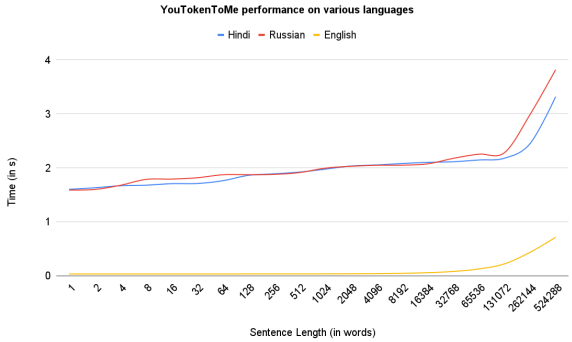

Figure 7: YouTokenToMe Time vs Length for Hindi, Russian and English languages

static length 131072 words. Since we are running the tokenizer on a single CPU, we tried to limit the CPU usage through a package called *cpulimit*. For both YouTokenToMe and Tiktoken, the time taken to tokenize was reduced by increasing the CPU limit gradually.

Additionally, the effects of CPU limitation on both tokenizers are captured for varying sentence lengths in *Figure 3*. The time taken to tokenize with YouTokenToMe is significantly lesser for smaller sentence lengths and it's only slightly lesser as we approach larger sentence lengths. This discrepancy arises because, in the case of Tiktoken, the time needed for tokenization is predominantly influenced by the encoding load time for shorter sentences.

Figure 4 demonstrates the correlation between L1 Cache Misses and sentence lengths. It's apparent that L1 Cache Misses escalate as the length of the sentence increases. Additionally, there's a notable surge in cache misses between lengths of 32k and 131k in the case of YouTokenToMe. This surge primarily contributes to the observed time increment in both Figure 6 and Figure 7.

Figure 5 illustrates the relationship between the sentence lengths and the fluctuations in Last Level Cache (L3) misses. It's evident that L3 cache misses are significantly lower compared to L1 cache misses for sentences of the same length. This disparity arises because data is first searched in the L1 and L2 caches, and only if it's not found there, is the L3 cache accessed.

Figure 6 illustrates that Tiktoken performs faster when tokenizing English sentences compared to languages like Russian and Hindi. Within the subset of Russian and Hindi, we observe similar performance, with minimal variation.

Figure 7 indicates a similar pattern to TikToken, where YouTokenToMe exhibits faster tokenization for English compared to Russian and Hindi. However, YouTokenToMe demonstrates faster performance across all three languages (English, Russian, Hindi) compared to TikToken.

5 Related Work

5.1 Performance evaluation of Tokenizers

The papers in this subsection evaluate tokenizers either using various accuracy metrics or specific scenarios. [2] examines tokenizer accuracy through precision, recall, and F1 scores. [18] compares tokenizers to determine their effectiveness during the inference phase of machine learning models.

5.2 Evaluation of Tokenizers in case of multilingual models.

The papers in this subsection evaluate tokenizers in the context of multilingual models. [12] investigates how tokenizers affect multilingual language models compared to their monolingual counterparts. [15] examines a multilingual tokenizer employed in a large language model. [10] delves into the significance of tokenizers for multilingual language models.

5.3 Tokenizers Analysis on Specific Foreign Languages.

The papers in this subsection evaluate tokenizers' performance across different languages. [16] analyzes the performance of various tokenizers on the Turkish language. [14] assesses a specific tokenization technique for the Chinese language.

None of the aforementioned papers focuses on analyzing the performance and efficiency of tokenizers within the context of hardware utilization, particularly focusing on cache and memory traffic. This analysis aims to identify areas for improvement in tokenizer implementation, particularly in terms of better resource utilization.

6 Challenges

We faced a couple of challenges related to the setup of the tokenizers. Initially, we were passing sentences directly in a Linux command that executes the tokenizer. The command executed successfully for sentences with word lengths up to 16k. However, when dealing with longer word lengths, an error occurred due to the command exceeding its argument limit. Due to this limitation, we had to come up with an approach where we write the sentences to a file and then read them as input by specifying the file name as an argument to the tokenizer.

Another challenge we faced was that, unlike TikToken, YouTokenToMe did not come with any pre-trained encodings. This resulted in us having to spend time training YouTokenToMe to learn encodings for the desired language.

Once the tokenizer setup was completed, we also ran into some issues during profiling. While trying to limit the compute available to the tokenizer with the help of the *cpulimit* command in Linux, we learned that the tool does not support limiting the compute available to subprocesses of a particular process. As a result, we had to set up and integrate an enhanced version of the *cpulimit* tool by *Opsengine* which had the desired functionality.

Another issue we faced was that the *perf-stat* tool did not support profiling for the L2 cache so we went ahead with profiling only the L1 and L3 cache references and misses.

While trying to limit the memory bandwidth available to the tokenizer using *cgroups*, we ran into a lot of errors during the setup phase itself. Due to a lack of time, we had to park this aspect of our evaluation for future work and instead focus on performing the profiling for different languages like Hindi and Russian.

7 Future work

As mentioned in the challenges section, we tried to do a small Proof-of-Concept for using *cgroups* package for profiling memory. We faced a few issues in the setup and due to a lack of time, we decided to park it for future work. We also wish to evaluate the accuracy of the downstream model associated with a tokenizer to verify if the performance benefits associated with using YouTokenToMe come at the expense of downstream model accuracy. Individual profiling using *Pyflame* to understand the time taken by each component of the tokenizer could also help identify further scope for optimization. There is also scope to extend our evaluation methodology to other tokenizers and languages in the future.

8 Conclusion

The majority of the current work related to evaluating tokenizers focuses on aspects like compression factor, accuracy, and F1 scores of tokenizers. None of the recent work evaluates the compute, memory, and cache efficiency of tokenizers. In this project, we develop a methodology to evaluate tokenizers in terms of their compute, memory, and cache efficiency and use this to evaluate two popular tokenizers, TikToken and YouTokenToMe. This evaluation gives us insights into the strengths, limitations, and performance bottlenecks of these tokenizers. This methodology can also be extended and applied to other tokenizers enabling users to determine the most suitable tokenizer based on their application requirements and resource constraints. Additional implementation details and plots can be found at [3]

9 Contributions

We are a team of 4 graduate students pursuing Masters at University of Wisconsin Madison. We would like to thank our professor Shivaram and project advisor/TA Saurabh who guided us throughout the project to analyze things in depth. This paper is a contribution and a joint

work by Abhijeeth, Divy, Smit and Sujay(email aliases available at the top for contact). Abhijeeth was responsible for implementing the code and visualizations, Divy helped us with CloudLab infrastructure setup to run the code, Smit assisted in developing the core algorithm for sentence generation and profiling, Sujay's skills in integrating packages like *cpulimit* and *perf-stat* made it easier for us to navigate things at the OS layer. Collaborating with a peer group to ensure that the right metrics are being evaluated, visiting office hours for clarification of doubts, thinking critically to ensure the robustness of the solution was done collectively. **No parts of the paper should be reproduced unless for educational use without the consent of the authors mentioned above.**

References

[1] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, mar 2003.

[2] S. Choo and W. Kim. A study on the evaluation of tokenizer performance in natural language processing. *Applied Artificial Intelligence*, 37(1):2175112, 2023.

[3] divy9881. Big-data-assignments. https://github.com/divy9881/Big-Data-Assignments/tree/master/Project, 2024. Accessed: Jan 1, 2024.

[4] facebookresearch. Flores. https://github.com/facebookresearch/flores, 2019. Accessed: March 8, 2024.

[5] T. Horsmann and T. Zesch. LTL-UDE @ EmpiriST 2015: Tokenization and PoS tagging of social media text. In P. Cook, S. Evert, R. Schäfer, and E. Stemle, editors, *Proceedings of the 10th Web as Corpus Workshop*, pages 120–126, Berlin, Aug. 2016. Association for Computational Linguistics.

[6] D. Khurana, A. Koli, K. Khatter, and S. Singh. Natural language processing: state of the art, current trends and challenges. *Multimedia Tools and Applications*, 82(3):3713–3744, Jan 2023.

[7] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot, and S. Tan. Between words and characters: A brief history of open-vocabulary modeling and tokenization in NLP. *CoRR*, abs/2112.10508, 2021.

[8] OpenAI. Tiktoken. https://github.com/openai/tiktoken, 2023. Accessed: March 8, 2024.

[9] opsengine. cpulimit. https://github.com/opsengine/cpulimit, 2008. Accessed: March 8, 2024.

[10] A. Petrov, E. L. Malfa, P. H. S. Torr, and A. Bibi. Language model tokenizers introduce unfairness between languages, 2023.

[11] I. Provilkov, D. Emelianenko, and E. Voita. Bpe-dropout: Simple and effective subword regularization, 2020.

[12] P. Rust, J. Pfeiffer, I. Vulić, S. Ruder, and I. Gurevych. How good is your tokenizer? on the monolingual performance of multilingual language models, 2021.

[13] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In K. Erk and N. A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, Aug. 2016. Association for Computational Linguistics.

[14] C. Si, Z. Zhang, Y. Chen, F. Qi, X. Wang, Z. Liu, Y. Wang, Q. Liu, and M. Sun. Sub-Character Tokenization for Chinese Pretrained Language Models. *Transactions of the Association for Computational Linguistics*, 11:469–487, 05 2023.

[15] F. Stollenwerk. Training and evaluation of a multilingual tokenizer for gpt-sw3, 2023.

[16] C. Toraman, E. H. Yilmaz, F. Şahinuç, and O. Ozcelik. Impact of tokenization on language models: An analysis for turkish. *ACM Trans. Asian Low-Resour. Lang. Inf. Process.*, 22(4), mar 2023.

[17] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[18] O. Uzan, C. W. Schmidt, C. Tanner, and Y. Pinter. Greed is all you need: An evaluation of tokenizer inference methods, 2024.

[19] VK.com. Youtokentome. https://github.com/vkcom/YouTokenToMe, 2019. Accessed: March 8, 2024.

[20] J. Webster and C. Kit. Tokenization as the initial phase in nlp. pages 1106–1110, 01 1992.