2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) | 979-8-3315-5103-2/26/$31.00 ©2026 IEEE | DOI: 10.1109/ISPASS69572.2026.00053

HeadSkip: Characterizing and Accelerating 'Small' Language Models on Edge CPUs

Ian Sartor*, Karthik Ganesan*, Anh Nguyen†, Andreas Moshovos*‡
*The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON
†Fujitsu Limited, Toronto, ON
‡Vector Institute for Artificial Intelligence, Toronto, ON
Email: {ian.sartor, karthik.ganesan, andreas.moshovos}@utoronto.ca}, anh.nguyen@fujitsu.com

***Abstract*— Recent years have seen a surge in the popularity of large language models. Such models are typically run in datacenters using high-end GPUs. However, due to latency and privacy concerns, companies have proposed 'small' language models (SLMs), designed to run locally on edge devices. We characterize two notable SLMs (i.e., Gemma 3 and Qwen3) on an edge CPU and find that the final layer accounts for up to 31.5% of the total SLM inference runtime.**

This final layer maps the output of the last decoder block to a specific token in the model's vocabulary. Therefore, this layer's runtime scales with the vocabulary size, which can be very large even for SLMs (e.g., 262k tokens for Gemma 3). Building on this observation, we present `HeadSkip`, two complimentary techniques to speed-up the final layer. First, we execute the final layer in chunks in order from most frequently to least frequently seen tokens. As we run, we check the logit output for tokens in a chunk and if any exceed a pre-set threshold, we output the token with the largest logit and skip executing the rest of the layer, obtaining a significant speedup.

We perform this re-ordering statically based on the frequency of words in English. However, this does not capture words which are generally rare but may appear frequently in some contexts. For such words, a static order may still execute most of the layer until we check this 'infrequent' token, diminishing our speedup gains. To avoid this, `HeadSkip` also caches tokens which appear in the current context. This helps us achieve even greater speed-ups (up to 45%) on modern SLMs, when evaluated on two edge CPUs.

***Index Terms*—Language models, edge computing, neural network**

I. Introduction

Large language models (LLMs) are increasingly intertwined in our daily lives. While systems such as ChatGPT receive the majority of use, there is a rise in the use of such models on edge devices such as smartphones. Driven by latency and privacy concerns, companies like Google and Apple have deployed smaller versions of LLMs on mobile devices [1], [9]. To fit within the limited resource constraints of mobile devices, such "small" language models (SLMs) use far fewer parameters than LLMs running in large datacenters. These changes mean that the performance bottlenecks and design trade-offs of SLMs can differ substantially from those of LLMs. To identify potential opportunities for optimizing SLMs, we characterize two popular SLMs on a representative edge CPU.

Our **first contribution** is characterizing these modern SLMs across a range of prompt sizes, and showing that – uniquely for these small models – the final layer (i.e., the 'LM-head') accounts for a significant proportion (up to 31.5%) of the overall runtime. This last layer computes the probabilities for every token in the vocabulary to be the next token output by the model. Naturally, during each invocation of the last layer, most tokens have negligible probability of being output. However, computing such "useless" outputs accounts for the significant runtime of the LM-head, since the vocabulary size can be very large even for SLMs (e.g., 262k tokens for Gemma 3 [9]). Motivated by this observation, we ask: "*can we avoid most of the LM-head cost spend on computing low-probability tokens?*"

Our **second contribution** is `HeadSkip`, which comprises two techniques to accelerate the LM-head. We investigate a staged evaluation strategy: first, we score only a small subset of candidate output tokens, and if one is deemed "good enough", we skip computing scores for the remaining tokens. Doing so requires achieving two goals: 1) computing the most likely tokens first and 2) determining if a token is good enough to be output.

`HeadSkip` achieves the first goal by *statically* reordering the order in which the LM-head computes output tokens. To obtain the new order of tokens, we build on the well-known skew in natural-language word frequencies: a small fraction of words occur far more often than the rest [13]. Text generated by language models also exhibit this distribution [19].

In the baseline, the LM-head always runs to completion, producing logits for all tokens regardless of their likelihood. We instead evaluate tokens from most to least likely, generated by analyzing word frequency in a representative corpus (i.e., Simple English Wikipedia [28]). After computing each token's logit, we compare it to a pre-determined threshold; if the logit exceeds the threshold, we output that token and skip the remaining logit calculations. This allows us to terminate the LM-head layer early and introduces a trade-off between speed-up and fidelity (i.e., how often the chosen output differs from the baseline).

The second technique in `HeadSkip` exploits an opportunity missed by the first: static token reordering works well in

979-8-3315-5103-2/26/$31.00 ©2026 IEEE
DOI 10.1109/ISPASS69572.2026.00053

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

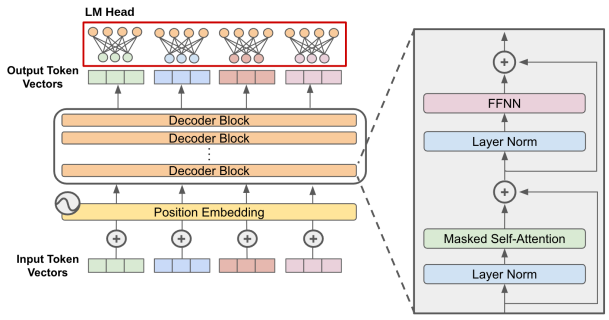


Fig. 1: Architecture diagram for a modern language model, showing many outputs across time [29]

most cases, but it cannot account for tokens that are frequent only in a particular context. For example, "cache" may appear often in a computer architecture paper but rarely elsewhere. To capture these "contextually frequent" tokens, we add a small cache to the LM-head: tokens seen during the current context are inserted into the cache. We then check this cache along with the first chunk before computing the rest of the tokens in the statically determined order. Doing so yields speedups from both globally frequent as well as contextually frequent tokens.

In summary, we make the following contributions:

1) We perform a characterization of modern 'small' language models (SLMs) on an edge system.
2) We observe that the LM-head layer accounts for a significant proportion (up to 31.5%) of the overall execution time of SLMs.
3) We propose `HeadSkip`, two techniques to speed-up the LM-head execution. `HeadSkip` yields up to a 45% speed-up using one configuration on two frequently used SLMs when run using llama.cpp.

II. Background

In this section, we provide some background on the architecture of widely-used large language models (Sec. II-A). We then highlight the key differences with "small" language models, which is the basis of our work (Sec. II-B

A. Language models

Figure 1 shows the major steps in executing a language model. For a given 'query', which is a set of words which are input to the model, the network takes all the words and converts them to tokens based on the pre-trained model 'vocabulary'. The vocabulary is a set of all the tokens the model was trained on along with multi-dimensional vectors representing each token. The tokens are then input to a series of repeated 'decoder' blocks.

Decoder blocks: The bulk of language models consists of several decoder blocks [26], [27]. The number of decoder blocks varies based on the size of the language model. Each decoder block consists of the Multi-Head Attention (MHA) mechanism and a Feed-Forward Network (FFN). The attention mechanism is what allows the language model to 'attend' to different tokens from all the tokens it has seen from the beginning (referred to as the 'context'). This is done using three vectors:

1) **Query (Q)**: the "question" or the current token the model is processing.
2) **Key (K)**: the "index" for all the other tokens seen so far in the current context.
3) **Value (V)**: the actual content associated with each token.

The output of the final decoder block is then passed to a final fully connected layer (called the 'LM Head'), which maps it back to a token in the vocabulary.

Phased execution: Language models execute in two distinct phases: pre-fill and decode. During the pre-fill stage, the model processes all the tokens in the query in parallel. The tokens processed during the pre-fill stage are then used to generate the answer during the decode phase. However, the decode phase executes serially, as each token generated depends on all the tokens before it in the current context.

KV cache: A key computational challenge with language models is that the current token depends on all the tokens which have been processed so far. Thus, computing token 1000 requires performing the KQV operations with a 1000 dimension matrix, which can be very slow. However, this is typically a concern for 'large' language models, which increasingly support larger and larger context lengths. For example, models such as Claude Sonnet 4, GPT-4.1, and Llama 4 Maverick all support a context length of 1 million tokens [17]; this is not the case for SLMs as we now show.

B. How small language models are different

Demand for private, low-latency, and reliable AI has driven deployment of language models directly on edge devices such as mobile phones [14], [31]. However, limited compute and memory constrain these models to roughly 1–4B parameters. We focus on SLMs in this range.

For large language models, the weights and the KV Cache have traditionally been the key performance bottlenecks [11]. However, this does not exactly hold for modern SLMs, due to two key architectural strategies:

Attention Optimizations: Many contemporary SLMs incorporate techniques that limit KV-cache growth, reduce the computation it induces, or both [26]. For example, Gemma 3 uses Sliding Window Attention (SWA) [21], which restricts attention to a fixed window of recent tokens. Other approaches, such as Mamba [10], replace attention with structured state-space models, reducing the cost to scale linearly with context length.

Fewer Decoder Blocks: SLMs utilize significantly fewer Decoder blocks than their larger counterparts (e.g., Gemma 2 2B has 26 blocks, while the 9B version has 42) [22], [32]. Thus, smaller models have a higher percentage of their

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

overall weights in the LM head. As we show in Sec. V, these strategies reduce the number of calculations needed for attention, thereby reducing its impact on runtime.

III. Related Work

In this section, we detail prior art that most closely matches our work. We start with work on lexical shortlisting (Sec. III-A) and prior approaches for early stopping (Sec. III-B).

A. Lexical Shortlisting

Lexical shortlisting means restricting the vocabulary of language models to a small subset. Many existing lexical shortlisting approaches are based on static, predetermined vocabularies [25], [33] while others supplement their statically selected vocabulary with tokens seen in prompts [4], [35]. Vocabulary selection techniques can be broken down into two main categories. The first is unicode based filtering, where a target language is specified and tokens from all other languages are removed [4]. For example, removing Cyrillic characters for an English language task. While this removes many unused tokens, the resulting vocabulary still includes many tokens that are used very infrequently, as language often follows a Zipfian distribution [13].

Another technique is corpus based filtering [4], [25], [33], [35]. Corpus based filtering typically shrinks the vocabulary by removing tokens not found in some representative corpus. Corpus based filtering is sometimes augmented with additional information such as token frequency, removing tokens that are not found in many outputs [35], or removing tokens that do not appear many times over the entirety of the calibration text [25]. This approach is typically tuned for a specific domain or task; tuning it for a broad range of tasks increases the size of the subset vocabulary, thereby diminishing the performance benefit.

Both approaches are susceptible to two main issues: 1) they fail to fully leverage the inherent frequency patterns in language, being unable to stop early in the case a very common token is good enough, and 2) they risk selecting an incorrect token if the correct token lies outside the statically restricted vocabulary. By contrast, our method keeps the entire vocabulary and merely *re-orders* the computations based on token frequency in a calibration set and augments our ordering by adding a cache of recently seen tokens to check immediately.

However, merely computing tokens in a different order does not yield any performance gain. We must stop computing the remaining tokens once we identify a token to output. This is explored in prior work on early stopping, which we describe next.

B. Early stopping

Early stopping (or early exiting) accelerates inference by returning the model's output before completing all remaining computation, thereby reducing latency.

Existing early-stopping methods operate at the layer level, adding exit points after decoder blocks (e.g., CALM [18] and SkipDecode [6]). These approaches use specialized prediction heads to estimate the model's output confidence after specific decoder blocks. They then decide whether the current 'early' token prediction is sufficient; if so, they skip the remaining decoder blocks and directly generate the model's output token. In general, layer-level early exiting predates language models and relies on two mechanisms: 1) using a separate classifier trained to predict when an decoder block's output is "good enough" [23], and 2) threshold-based rules that generate an output once a confidence criterion is met. CALM [18], explores this, as well as specialized exit classifiers.

Our work adopts a confidence-based threshold strategy but applies it at a finer granularity, targeting the LM-head instead of decoder blocks.

IV. Methodology

We now describe our methodology for characterizing the runtime of small language models. We evaluate our models using the widely used open-source `llama.cpp` framework [8], and profile inference runtime with the built-in `llama-bench` utility.

Models: We evaluate three representative language models. The first two are state-of-the-art SLMs that are similar in scale to proprietary on-device models available from Google [20] and Apple [2]. Both SLMs use orthogonal techniques to limit computation, to fit the tight constraints of local inference. **Gemma 3 1B** uses Sliding Window Attention (SWA), to limit the computation required for the attention mechanism [21] **Qwen3 1.7B** uses fewer decoder blocks than larger members of its family but does not include explicit attention optimizations such as SWA [32]. We also include Llama 2 [24] a widely used but older model, to highlight differences between modern SLMs in addition to Gemma 3 4B and Qwen3 4B to show that even at slightly larger model sizes the characterization remains relevant.

We also consider three quantization levels representing different tradeoffs between model size and accuracy loss. Quantization enables lower energy costs, faster performance and enables more capable models to run on more constrained devices. As we benchmark our models using `llama.cpp`, we use the quantization schemes available through the gguf format [7]. For 4-bit quantization we use the `Q4_K_M` setting and for 8-bit quantization we use the `Q8_0` setting. However, for all models, `llama.cpp` keeps the LM-head and the embedding table in higher precision, as prior work shows that these are more sensitive to quantization [34]. For Gemma 3 1B, the `Q4_K_M` setting uses `Q8_0` for the LM-head, and for all other models it uses the gguf `Q6_K` quantization.

Hardware and software setup: We evaluate `HeadSkip` on two hardware platforms: an x86 system, similar to high-end edge devices and a Raspberry Pi 5 as a low-end edge device.

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Summary of Tasks for Model Evaluation

Task	Dataset	Criteria
Summarization	XSum [12]	BertScore [37]
Info. Extraction	SQuAD [16]	F1 score
Math Reasoning	GSM8K [5]	Accuracy

For our high-end system, we use an Intel i9-13900HX CPU, with 32GB of DDR5 RAM. We focus on a CPU since literature shows that in mobile spaces GPUs do not necessarily have a performance advantage during inference [30], [36]. We select this CPU as its single-core architecture and advanced vector processing extensions (e.g., AVX2) are similar to the high-performance cores found in modern, high-end mobile and edge systems. Furthermore, running on an x86 system allows us to run widely used tools such as `llama.cpp` and lets us use low-level profiling tools such as Intel VTune. For software, we run on Linux Mint version 22.

For the low-end system, we use a Raspberry Pi 5 with 16GB of DDR4 RAM, running Debian v13.2. On all platforms, we use `llama.cpp` commit version 8423d01, compiled using CMake to enable performance enhancements such as AVX2/NEON.

This CMake-based compilation method can lead to minor variations in Table II due to various compiler optimizations that impact floating point rounding. Table II was created using several processors. To ensure accuracy, all processors were compiled using `-march=raptorlake`.

Benchmarks: We select benchmarks representative of common use cases for SLMs run on edge devices. Table I lists the selected tasks as well as the dataset and evaluation criteria used for each.

1) **Summarization** is an important task implemented on edge devices, in local processing scenarios such as running on-device mobile assistants [3]. Recent research demonstrates the viability of using local models for real-time tasks like meeting summarization [31].
2) **Information Extraction and Question Answering** is a proxy for commonly used tasks such as suggesting words for text completion or managing personal email and scheduling.
3) **Mathematical Reasoning** measures the model's ability to answer text-based mathematical questions. While not directly applicable to SLMs, we select this benchmark for two reasons: 1) to increase the scope of our evaluation outside of purely linguistic tasks, and 2) this score is widely reported by prior works [21], [22], [32].

V. Characterization

In this section, we present results from our characterization of SLMs. First, we break down end-to-end inference time across major operation types (e.g., QKV projection, attention, and the LM-head) to identify the dominant costs (Sec. V-A). We then study how the dominant components (i.e., attention and LM-head) scale with increasing prompt length (Sec. V-B).

A. Where Time Goes in SLMs

We start with a per-operation breakdown of inference time for the three models we study. We do this using short prompts which are more common in edge-device scenarios. Figure 2a shows the normalized inference-time breakdown for a 32-token prompt and 32-token generation workload. Each stacked bar in Figure 2a shows the relative time taken for each operation time we measure. For the baseline 16-bit models, the multi-head attention (MHA) and Feed-Forward Network (FFN) projection layers account for the majority of runtime. However, we see a major difference between newer, smaller models (i.e., Gemma 3 and Qwen3) and the older Llama2 7B model. The LM-head accounts for just 2% of the runtime for Llama2 7B, while it accounts for 9% to 31.5% of the runtime in newer models. The newer models use a much larger, multi-lingual vocabulary compared to the smaller, mono-lingual vocabulary used by Llama2 7B. This is one reason for the LM-head taking much more of the total time on newer models.

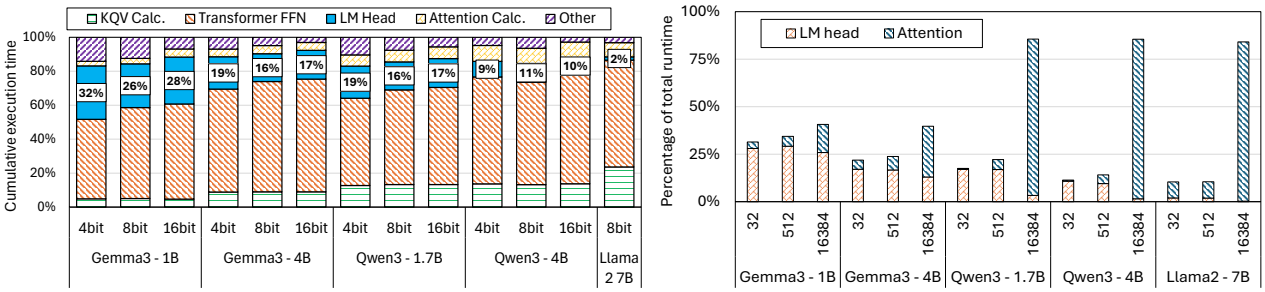
The proportion of the runtime taken by the LM-head is similar for 16-bit and 8-bit quantization, but increases for 4-bit quantization. This is because in the 4-bit case, the LM-head uses 6- or 8-bits, as we described in Sec. IV. This leads to the LM-head taking relatively longer for the 4-bit models. As models are increasingly moving to using even fewer bits, we believe that the LM-head will account for an even longer portion of the runtime in future models. Next, we analyze the impact of increasing the context length on model runtime.

B. Varying Prompt and Generation size

We now show how the time taken by attention and the LM-head varies as we change the number of tokens used. We do this to see if attention still consumes a large proportion of runtime on small models as it does in larger models, particularly given optimizations made in SLMs to speed-up attention. All models are evaluated with `FP16` with the exception of Llama2, which is evaluated in `Q8_0`. Figure 2b shows the scaling of attention across different prompt/generation pairs for different model sizes and architectures. As expected, attention takes significantly longer for long context lengths. This is more noticeable for the older Llama2 7B model and for the Qwen3 models. These models do not use any optimizations to limit attention computation, leading to attention taking up to 84% of the total time in these models.

In contrast, Gemma 3 uses alternating local and global attention layers to reduce attention computations. As a result, the timeshare of attention for Gemma 3 1B only ranges from 4% to 15%. Gemma 3 4B also shows minimal growth, with attention reaching a maximum of 27% of runtime. Building on the benefits seen in Gemma 3, The recently developed Qwen-3-Next model also incorporates attention optimizations designed to reduce the size and computation time of the attention mechanism [15].

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.



(a) (b)

Fig. 2: (a) Runtime breakdown per operation type for three models with different quantization levels, for a prompt length of 32 tokens and (b) Proportion of time taken by the LM-head and attention calculations for different networks, as we vary prompt length from 32 to 16384.

C. Final Layer Trends Across Prompt/Generation Sizes

Unlike attention, the LM-head is statically sized and cost of computing the LM-head remains constant regardless of the context length. Figure 2b also has bars to show how the LM-head scales across different prompt/generation pairs for different models.

As the task size increases, the fixed cost of the LM-head gets amortized over increasing attention computations. Qwen3, which has the largest growth in attention timeshare, sees a measurable drop in LM-head timeshare. Thus, there remains significant opportunity for speed-ups for small context-length tasks.

Conversely, Gemma 3 – which sees little growth in attention timeshare – sees a consistent LM-head timeshare across token lengths. The time taken by the LM-head is dependent primarily on the vocabulary size and the quantization level used. Gemma 3 1B, which has the largest vocabulary, has the largest timeshare at 31.5% for the shortest prompt. Going from 16-bit to 4-bit quantization leads to a timeshare increase (2% for Qwen3, 4% for Gemma 3) for the LM-head. This suggests that as optimizations reduce the total runtime of the main layers and attention, the fixed, high-cost LM-head increasingly dominates the overall latency.

Our analysis shows that the LM-head is an important and growing bottleneck in highly quantized SLMs, particularly at short context lengths. Thus, the LM-head requires targeted optimizations to achieve optimal performance on edge devices, to save both time and battery life.

VI. HeadSkip

We now present `HeadSkip`, a set of two optimizations designed to mitigate the final layer bottleneck identified in Section V. We address the highly-skewed distribution of token probabilities by leveraging *Static Token Re-ordering* with an *Early-Exit Confidence Threshold* (Sec. VI-A), complemented by a small, *Context-Aware Dynamic Cache* (Sec. VI-B).

In language models, the LM-head computes a logit for every token in the vocabulary. This is wasteful, as in natural language word usage is highly skewed [13], and the same holds true for text generated by language models [19]. Thus, despite large vocabularies, conversations typically only use a small subset of the overall vocabulary most of the time. As a result, significant time is wasted calculating output logits for infrequent tokens, which do not contribute to the final output.

A. Static Token Re-ordering

To reduce the runtime of the LM head, we change the order in which tokens are computed. The LM head is implemented as a dense fully-connected layer, where the number of output neurons is the vocabulary size (e.g., $262,144$ tokens for Gemma 3 1B). Typically, these $262,144$ tokens are not calculated in any pre-defined order, but rather in the random order they exist in the model vocabulary.

To determine a more efficient **static** order in which to calculate these tokens, we compute the frequency of tokens in a large, general corpus. We use Simple English Wikipedia [28] for this, although any other large English corpus would also work. We then arrange tokens from the most to the least frequent, and include tokens which mark the end of a sentence. In this way, we increase the probability that we will calculate frequently appearing tokens earlier during execution of the LM-head.

Chunked token execution: To compute logits from most to least frequent, `HeadSkip` runs the LM-head in 'chunks' of C tokens at a time. The steps employed by `HeadSkip` are:

1) Compute the first C tokens' logits in the re-ordered list.
2) Check if the highest-scoring logit (p^{top}) from these C tokens exceeds our pre-defined confidence threshold (τ).
3) If $p^{top} > \tau$, we stop running the layer and output p^{top}.
4) If $p^{top} < \tau$, we proceed to the next chunk and repeat.

Figure 4 also shows these steps as a flowchart.

Early-Exit Confidence Threshold: The next component of `HeadSkip` is to stop computing the LM-head when we have high confidence that we have already computed the logit for the 'correct' token. The earlier we can terminate the LM-head

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

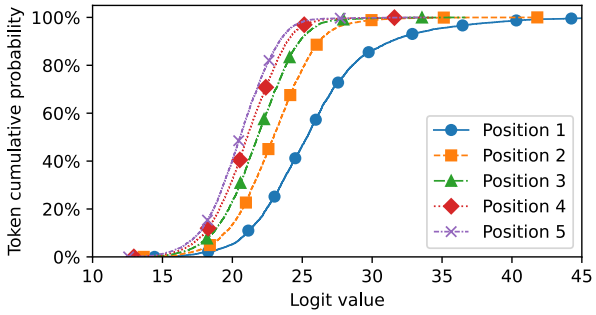


Fig. 3: A cumulative distribution plot of the top-5 logits collected from Qwen3 1.7B. Note the x-axis starts at 10.

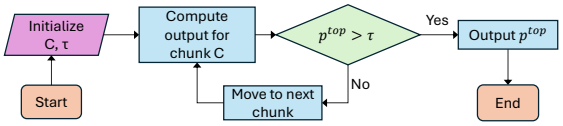


Fig. 4: Flowchart of our technique for executing the final layer in token 'chunks'.

computation, the higher our speed-up. We opt for a simple approach where we calculate a static 'threshold' logit for each model. We then check the logit of all tokens as we compute each chunk; when we get a token whose logit is higher than the threshold, we output the token with the highest logit in its chunk. We obtain this confidence threshold by profiling the logit distributions of the target SLMs.

A natural question is: *Is a single static threshold per model sufficient?* To answer, this we plot the the cumulative distribution of the top 5 logits from 2000 tokens in Figure 3, obtained from the Qwen 3 1.7B model. We see a significant separation in the curves between the highest logit and the rest, giving us confidence that a simple logit threshold can help differentiate the 'correct' token from the rest. Thus, by picking an appropriate threshold, we maximize the chance of the selected token having the highest logit score. To evaluate the impact of threshold selection, we study three threshold values per model in Sec. VII.

B. Dynamic Caching

Thus far, we have discussed how `HeadSkip` statically re-orders tokens. While this approach applies broadly to all models, it is unsuitable for tokens that are 'globally' infrequent, but may appear frequently within a specific context. Using only a statically re-ordered list, a word like "branch", common in some computer architecture papers but not globally, would appear very far down in the list, significantly limiting the benefit of `HeadSkip`. To address this, `HeadSkip` also implements a small, 'context-specific' cache to store tokens which appear frequently in specific contexts.

We add tokens to the cache if they are: 1) seen in the input prompt or 2) generated by the model during the current context. However, to avoid filling the cache with very common words, we omit tokens which are already present in the first static chunk (C_0) of our re-ordered list. The dynamic cache uses a simple Least Recently Used (LRU) replacement policy.

C. Putting it all together

We now describe how `HeadSkip` combines static token re-ordering with an early-exit confidence threshold and context-aware dynamic caching to speed-up SLMs. `HeadSkip` starts by computing the first static chunk C_0 and all tokens inside of the dynamic cache; if the logit of a token exceeds the threshold we output the token with the largest logit calculated in either C_0 or the dynamic cache and cease further computations of the LM-head. If both of these do not have any tokens whose logits exceed the threshold, we continue executing the remaining chunks in our statically re-ordered list. If a chunk contains a token whose logit exceeds the threshold it will stop calculating once that chunk is finished and output the token with the largest logit calculated.

This combined static-dynamic approach ensures maximum token coverage: the static order handles the common words (e.g., "the", "is", "and"), and the dynamic cache handles the specialized, contextually frequent words, enabling significant computation reduction across all token generation steps.

Implementation details: We now briefly mention how we implement `HeadSkip` in `llama.cpp`. To exploit hardware parallelism, `llama.cpp` also processes tokens in a 'batched' manner; the LM-head is computed in batches of 64 tokens at a time. Thus, our 'chunk-based' approach naturally fits with the existing flow of `llama.cpp`. We only make two changes to enable `HeadSkip`. After completing each batch of 64 tokens, we check if any token exceeded the threshold. As our smallest chunk size (i.e., 2048) is much larger than the batch size, our implementation completes many batches per chunk. Once all batches are completed for a chunk, we check if any logit exceeds the threshold. If any logit exceeds the threshold, we output the token for the largest logit computed and move onto processing the next input token. By adapting our technique to the existing approach taken by `llama.cpp`, we minimize the performance impact of `HeadSkip`.

VII. Evaluation

We now present the evaluation results for `HeadSkip`. We begin by presenting the parameters we use in our evaluation (Sec. VII-A). Next, we present output quality results, with and without `HeadSkip` (Sec. VII-B). We then explore the efficacy of `HeadSkip` in generating tokens early (Sec. VII-C). Following this, we describe how `HeadSkip` affects the distribution of generated tokens (Sec. VII-D). Finally, we present the speed-up of `HeadSkip` over the baseline models on two hardware platforms (Sec. VII-E).

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Output quality with scores accompanied by the change from the baseline.

C	$\tau = 20/14.5$			$\tau = 22.5/16.5$			$\tau = 25/18.5$		
	DS:0	**DS:1024**	**DS:2048**	**DS:0**	**DS:1024**	**DS:2048**	**DS:0**	**DS:1024**	**DS:2048**
Model: Qwen 3 1.7B									
Math [Baseline 82.34]									
2048	52.31 -36%	67.17 -18%	67.17 -18%	65.96 -20%	71.49 -13%	71.49 -13%	77.63 -6%	79.45 -4%	79.45 -4%
4096	64.14 -22%	74.60 -9%	74.60 -9%	71.19 -14%	73.84 -10%	73.84 -10%	78.85 -4%	79.15 -4%	79.15 -4%
8192	72.71 -12%	78.92 -4%	78.92 -4%	76.80 -7%	79.23 -4%	79.23 -4%	81.50 -1%	80.89 -2%	80.89 -2%
16384	79.53 -3%	80.89 -2%	80.89 -2%	81.05 -2%	81.65 -1%	81.65 -1%	82.80 +1%	82.64 0%	82.64 0%
32768	82.18 0%	81.88 -1%	81.88 -1%	82.26 0%	82.34 0%	82.34 0%	82.94 +1%	83.24 +1%	83.24 +1%
65536	82.71 0%	82.49 0%	82.49 0%	82.18 0%	82.41 0%	82.41 0%	82.71 0%	82.64 0%	82.64 0%
Info. Extraction [Baseline .065]									
2048	.037 -43%	.062 -5%	.062 -5%	.049 -25%	.062 -5%	.062 -5%	.055 -15%	.063 -5%	.063 -3%
4096	.042 -35%	.062 -5%	.062 -5%	.051 -22%	.062 -5%	.062 -5%	.056 -14%	.063 -3%	.063 -3%
8192	.045 -31%	.063 -3%	.063 -3%	.051 -22%	.063 -3%	.063 -3%	.057 -12%	.063 -3%	.063 -3%
16384	.054 -17%	.064 -3%	.064 -3%	.057 -13%	.064 -1%	.064 -1%	.060 -8%	.064 -2%	.064 -2%
32768	.059 -9%	.064 -2%	.064 -2%	.060 -8%	.064 -2%	.064 -2%	.061 -6%	.063 -3%	.063 -3%
65536	.065 0%	.065 0%	.065 0%	.065 0%	.065 0%	.065 0%	.065 0%	.065 0%	.065 0%
Summary [Baseline .419]									
2048	.375 -11%	.400 -5%	.400 -5%	.408 -3%	.410 -2%	.410 -2%	.416 -1%	.418 0%	.418 0%
4096	.395 -6%	.406 -3%	.406 -3%	.409 -2%	.413 -1%	.413 -1%	.417 -1%	.418 0%	.418 0%
8192	.399 -5%	.408 -3%	.408 -3%	.409 -2%	.414 -1%	.414 -1%	.416 -1%	.418 0%	.418 0%
16384	.406 -3%	.410 -2%	.410 -2%	.412 -2%	.415 -1%	.415 -1%	.417 -1%	.418 0%	.418 0%
32768	.413 -1%	.414 -1%	.414 -1%	.417 -1%	.418 0%	.418 0%	.418 0%	.419 0%	.419 0%
65536	.419 0%	.419 0%	.419 0%	.419 0%	.419 0%	.419 0%	.419 0%	.419 0%	.419 0%
Model: Gemma 3 1B									
Math [Baseline 56.03]									
2048	26.46 -53%	28.05 -50%	28.05 -50%	35.71 -36%	37.07 -34%	37.07 -34%	39.12 -30%	43.06 -23%	43.06 -23%
4096	35.71 -36%	40.64 -27%	40.64 -27%	42.61 -24%	44.05 -21%	44.05 -21%	46.93 -16%	49.96 -11%	49.96 -11%
8192	52.31 -7%	54.74 -2%	54.74 -2%	52.01 -7%	53.45 -5%	53.45 -5%	52.92 -6%	55.42 -1%	55.42 -1%
16384	52.01 -7%	53.98 -4%	53.98 -4%	51.48 -8%	53.75 -4%	53.75 -4%	53.83 -4%	55.27 -1%	55.27 -1%
32768	53.98 -4%	56.25 0%	56.25 0%	54.28 -3%	55.72 -1%	55.72 -1%	56.71 +1%	57.24 +2%	57.24 +2%
65536	51.33 -8%	53.90 -4%	53.90 -4%	51.33 -8%	53.53 -5%	53.53 -4%	51.55 -8%	55.04 -2%	55.04 -2%
Info. Extraction [Baseline .207]									
2048	.089 -57%	.100 -52%	.100 -52%	.099 -52%	.184 -11%	.184 -11%	.135 -35%	.239 +16%	.238 +15%
4096	.096 -54%	.100 -52%	.100 -52%	.104 -50%	.184 -11%	.184 -11%	.140 -32%	.239 +16%	.239 +16%
8192	.101 -51%	.100 -52%	.100 -52%	.096 -54%	.164 -21%	.164 -21%	.144 -30%	.229 +11%	.229 +11%
16384	.088 -58%	.103 -50%	.103 -50%	.105 -49%	.161 -22%	.161 -22%	.150 -28%	.228 +10%	.228 +10%
32768	.093 -55%	.105 -49%	.105 -49%	.100 -52%	.162 -22%	.162 -22%	.156 -25%	.228 +10%	.228 +10%
65536	.090 -57%	.216 +4%	.216 +4%	.105 -49%	.218 +5%	.218 +5%	.168 -19%	.220 +6%	.220 +6%
Summary [Baseline .441]									
2048	.309 -30%	.320 -27%	.320 -27%	.352 -20%	.349 -21%	.349 -21%	.417 -5%	.417 -5%	.417 -5%
4096	.315 -29%	.322 -27%	.322 -27%	.358 -19%	.353 -20%	.353 -20%	.421 -5%	.428 -3%	.428 -3%
8192	.363 -18%	.367 -17%	.367 -17%	.382 -13%	.375 -15%	.375 -15%	.425 -4%	.422 -4%	.422 -4%
16384	.375 -15%	.380 -14%	.380 -14%	.392 -11%	.392 -11%	.392 -11%	.428 -3%	.426 -3%	.426 -3%
32768	.379 -14%	.414 -6%	.414 -6%	.417 -5%	.417 -5%	.417 -5%	.427 -3%	.419 -5%	.419 -5%
65536	.440 0%	.439 0%	.439 0%	.440 0%	.440 0%	.440 0%	.439 0%	.439 0%	.439 0%

A. Evaluation parameters

We now detail the different parameters used in our evaluation of `HeadSkip`.

Models: We evaluate `HeadSkip` using the models from Sec. V-A where the LM-head accounted for the highest proportion of overall runtime. Specifically, we use the Gemma 3 1B and Qwen3 1.7B models to best showcase the efficacy of `HeadSkip`. We exclude Llama 2 because of the low potential benefit of `HeadSkip` on this model, as the LM-head makes up only 2% of the runtime.

We run these models using 16-bit quantization (i.e., $FP16$), because output quality can vary based on the quantization scheme applied. That is, based on how the model was quantized to 4- or 8-bits can impact the output quality. To eliminate this as a source of variation in quality results, we limit ourselves to the $FP16$ models only. Our hardware and software are the same as Section IV.

Metrics: We now present the different metrics we use in our evaluation. First, each task uses a task-specific criteria to quantify **output quality**, which we provide in Table I. For Math Reasoning, an answer was considered "correct" if it contained the answer string, for Information Extraction and Summarization the scores are calculated against the entire generated strings. **Cache efficiency** is the percentage of

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

tokens generated by the model, that benefit from `HeadSkip`. **Speed-up** is the increase in tokens/second when `HeadSkip` is applied.

Deterministic token selection: For all of our measurements, we employ a deterministic token selection strategy, as we always select the token with the highest value each time. We do so to eliminate variations introduced by random token-sampling techniques.

Token generation limit: To limit evaluation time, we limit the maximum number of tokens generated per task: 512 for math, 128 for information extraction tasks, and 256 for summarization. These limits were chosen to ensure reasonable evaluation run-times while allowing sufficient length for the vast majority of task outputs.

Hyperparameters tested: `HeadSkip` introduces a trade-off between computational efficiency and output quality. Early termination risks producing an incorrect token which can propagate errors through subsequent decoding steps. However, terminating earlier increases the speed-up from our approach. This trade-off is tuned based on three hyper-parameters:

1) **Early-Exit Confidence Threshold**: The minimum raw logit value of a predicted token to be accepted as correct and trigger early termination. A lower value indicates a more aggressive optimization; tokens are more easily accepted as correct, but at the risk of producing tokens that do not match the baseline model. A higher value makes our technique more robust about the tokens output, but at the cost of some potential speed-up.
2) **Static Re-ordering Chunk Size**: The number of statically re-ordered tokens computed before the confidence threshold is checked for all of them. Smaller chunks lead to more frequent checks and finer-grained execution savings, increasing the overall speed-up potential. But if chunks are too small we may lower the quality of generated text by selecting bad tokens.
3) **Dynamic Cache Size**: The number of tokens stored in the small, context-aware cache. A smaller cache reduces memory and compute overhead, but can reduce the cache hit rate in certain contexts.

B. Output Quality

We begin by presenting the impact of `HeadSkip` on output quality for all three tasks, as we vary the three hyper-parameters listed above. Table II presents the per-task output quality metric for Qwen3 1.7B and Gemma 3 1B. For each model, we set the threshold, τ to three values, which we empirically select based on the model's logit CDF (such as the one for Qwen 3 1.7B's shown in Figure 3).

Trends Across Models: To begin, we present the trends seen in both models. Both models were able to match the output quality of the baseline model for the highest values we tested for the chunk size and the dynamic cache size (DS). Specifically, the configurations with a dynamic cache present ($DS \neq 0$), or a very large chunk size, matched the quality of the baseline.

In the **Math** task, output quality rivaled the baseline with large chunk sizes, especially in Qwen3, with or without a dynamic cache. Although both models saw some performance degradation at low threshold values, particularly on configurations with low chunk sizes, this was mitigated with the introduction of a dynamic cache.

For **Information Extraction**, both models see an increase in output quality when we use a dynamic cache. We can match the output quality of using a larger chunk size even with a small chunk size, as long as we use a dynamic cache. This is the case even for the small dynamic cache ($DS = 1024$). Using a larger dynamic cache does not greatly benefit tasks, since the cache targets 'contextually' frequent but globally infrequent tokens, which are not very common.

For **Summary**, having a dynamic cache increases output quality for low threshold values. However, we again see little to no increased benefit from a larger cache ($DS = 2048$). As threshold size increases, the cache has negligible effect on output quality; quality increases slightly for Qwen3 while decreasing slightly for Gemma 3. The dynamic cache remains important for speed-up purposes as well as other tasks.

Qwen3 1.7B: The Qwen3 model tested shows output quality close to the baseline on all tasks when `HeadSkip` is applied. It saw a maximum quality degradation of 5% once a dynamic cache was used and chunk size (C) was set to 8192.

Gemma 3 1B: Compared to Qwen3, Gemma 3 was more sensitive to `HeadSkip`. For low threshold evaluations the model struggled in all non-Math tasks, displaying fair output quality degradation from the baseline. In Math it was able to quickly recover even at low thresholds by increasing the chunk size and introducing a dynamic cache. The degradation in other tasks was mitigated through increasing the threshold. For the highest tested threshold case when a dynamic cache was present, the model with `HeadSkip` applied nearly matched or exceeded the baseline in all tasks.

This analysis shows that `HeadSkip` does not negatively impact output quality. A small dynamic cache enables the model to match the performance of a large static chunk size, for a fraction of the runtime. We also observe that a larger dynamic cache does not strongly impact output quality in our tested tasks. For this reason, we adopt the smaller dynamic cache size ($DS = 1024$) for the rest of our evaluation.

C. Cache Efficiency

We now look at the efficiency of the different mechanisms in `HeadSkip` to reduce LM-head calculations. Figures 5 and 6 show the breakdown of where output tokens are being generated from: 1) The dynamic cache (Dynamic Cache Hit), 2) A static chunk before the final chunk (Stopped Before Finishing), or 3) calculating the entire LM-head. For each

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

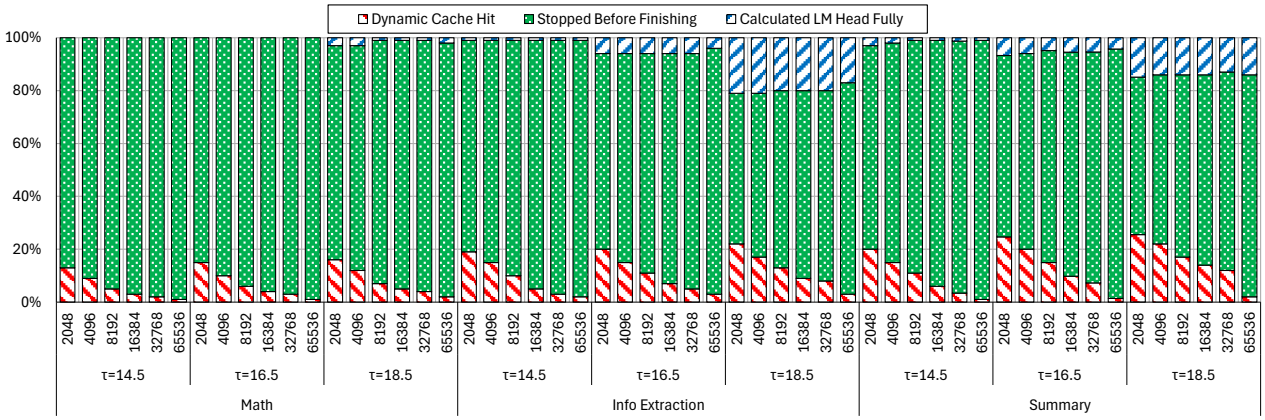


Fig. 5: Cache Efficiency Data for Gemma 3 1B.

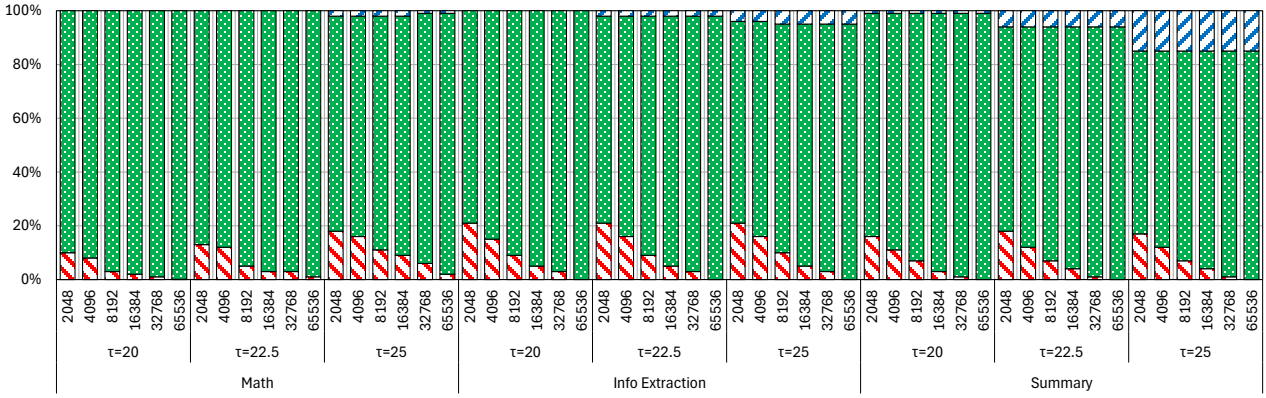


Fig. 6: Cache Efficiency Data for Qwen 3 1.7B.

task, we vary the logit threshold (τ) and the chunk size, matching Table II. We present these results grouped by task.

For all tasks, the vast majority of tokens generated are decided before the entire vocabulary is calculated. This signifies that there should be speed-up for every task. A large portion of these tokens in smaller C cases are from the Dynamic Cache, signifying the maximum possible speed-up. For larger chunk sizes, the Dynamic Cache Hit rate decreases steadily. This is expected since we always check the C most frequent tokens based on the expected static frequency (Section VI). The decrease in dynamic hits as C increases suggests that the tokens that entered the Dynamic Cache at smaller C values are now found within the first chunk, meaning that we are stopping in the first chunk in many cases as well.

As the confidence threshold increases, there is a clear trend in which the proportion of tokens that use the full LM-head increases. This is because a higher threshold makes the Early-Exit condition harder to meet, meaning the LM head executes fully when the token is not present in the cache. For this reason, the Dynamic Cache Hit rate increases slightly at higher thresholds for the same tasks. In such cases, the highest logit token is further down the re-ordered list but is frequent in that specific context. As a result, these tokens are picked again once they enter the dynamic cache, raising the hit rate.

The models differ the most in the Information Extraction task, with the highest threshold we test. While both models exceed the output quality of the baseline model, their cache efficiencies differ significantly. Gemma 3 shows a much higher Calculated LM Head Fully rate of 20% compared to just 5% for Qwen3. We believe this is because Gemma 3 does not produce high enough logits to trigger `HeadSkip` as frequently during Information Extraction due to the preceding context. This may be a case where a single static threshold is sub-optimal.

In all cases, we avoid the full LM-head calculation for the vast majority of generated tokens. Even in our lowest utilized case, Information Extraction on Gemma 3 with the highest tested threshold, we avoid the full calculation about 80% of the time. The results from this section, combined with the output quality analysis, guide the selection of a realistic deployment configuration. For Qwen3, the model shows strong output quality relative to the baseline across

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

a variety of different HeadSkip configurations. For this reason, we will select a configuration that is biased towards increased speed-ups. Gemma 3 shows more sensitivity to different configurations, as it already has greater potential for speed-up than Qwen3. As such, we will select a more conservative configuration to ensure output quality remains high. Experiments in the remaining sections will use $DS = 1024$ for both models and the following model-specific parameters: 1) For Gemma3: $C = 8192$, $\tau = 18.5$ and 2) For Qwen3: $C = 2048$, $\tau = 22.5$.

D. Output quality distribution

As we showed in Sec. VII-B, selecting the right hyper-parameters allows HeadSkip to maintain output quality. However, earlier we only showed a single number averaging the results for each input prompt. Now, we show how the output quality varies for different input prompts, particularly focusing on the bottom 1% and 5% of output quality metric for each task (i.e., the $p99$ and $p95$ scores). We only show these scores for the Information Extraction and Summarization tasks as the Math tasks uses binary scoring which does not produce a distribution of results.

In Information Extraction, both the baseline and optimized models for all configurations have a $p99$ and $p95$ of 0. This shows that HeadSkip does not change the output distribution for this task. However, the summarization task revealed more variance between the models. For Qwen 3, the performance shift was negligible; the $p99$ score for the baseline model was 0.2569 while the HeadSkip optimized model scored 0.2566, a change of $\leq 1\%$. Similarly, the baseline Qwen 3 model had a $p95$ scores of 0.2868, while the HeadSkip optimized model scored 0.2858. Gemma 3 presented a more distinct trend; the baseline scores ($p95$ of 0.1640 and $p99$ of 0) dropped to 0 for both metrics when HeadSkip was used. While this might suggest that HeadSkip slightly worsens the "tail" of the output distribution, the HeadSkip optimized technique significantly closes this gap for $p94$ to 0.1452, Thus, while HeadSkip has the potential to slightly shift the behavior of extreme outliers in certain models, it does not significantly impact the overall output quality in most cases.

E. Speed-up

We now quantify the speed-up obtained by HeadSkip, in tokens per second (Tok/s). We measure speed-up on the two CPUs described in the methodology, using the parameters detailed at the end of Sec. VII-C.

Figure 7 shows the relative speedup of each task for three models: Gemma 3 and Qwen3 (the models studied thus far), and a quantized variant of Gemma 3 (quantized to Q4_K_M). We chose to include the quantized variant of Gemma 3 to investigate how the settings selected for $FP16$ would translate to a quantized environment in terms of speed-up.

HeadSkip provides a speed-up in all scenarios we evaluate. Matching our results in Sec. VII-C, the speed-up varies

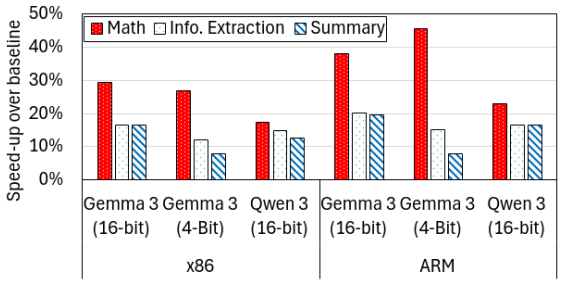


Fig. 7: Speedup compared to full-vocabulary case in selected models and tasks.

across tasks. HeadSkip performs best in the math task, as the correct tokens are typically chosen from a small selection. Conversely, speedup was lowest in the Summarization task because the threshold that balances frequent early termination and high output quality fluctuates with input and context.

We consistently see a greater speed-up on Gemma 3 1B compared to Qwen3 1.7B. As we discussed in Section II Gemma exemplifies trends in recent models that optimized attention, which results in an increased portion of time on the LM-Head. This offers a larger target for optimization, yielding a higher maximum speedup factor. Qwen3 1.7B's proportionally higher runtime spent on attention and larger overall size dampened the final layer speedup effect. The 16 bit floating point version of Gemma 3 had greater relative speedup when compared the 4 bit quantized version. In our characterization, we found that later layers took more of the runtime in these highly quantized models. This analysis shows the benefit of HeadSkip to speed-up SLMs in resource-constrained environments, without hurting output quality.

VIII. Conclusion

The efficient deployment of Small Language Models (SLMs) on resource-constrained edge devices reveals new computational bottlenecks. We show that, due to recent attention optimizations and small model sizes, the traditional focus on the K/V Cache and attention is misplaced in modern SLMs. Instead, our characterization revealed that the fixed, vocabulary-dependent cost of the LM-head is an emerging bottleneck. To address this, we propose HeadSkip, which combines Static Token Re-ordering, Early Exiting, and a Context-Aware Dynamic Cache. HeadSkip achieves a speed-up of up to 45%, when tested on two hardware platforms.

Acknowledgements

We thank the reviewers for their helpful suggestions, which has helped us greatly improve the final version of this paper. We would also like to thank the members of the AENEO group for their feedback during the submission process. This work was funded by an NSERC discovery grant as well as generous funding from Fujitsu limited.

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Apple. (2023) Apple intelligence. [Online]. Available: https://www.apple.com/ca/apple-intelligence/

[2] ——. (2024) Introducing apple's on-device and server foundation models. [Online]. Available: https://machinelearning.apple.com/research/introducing-apple-foundation-models

[3] ——. (2025) Apple intelligence gets even more powerful with new capabilities across apple devices. [Online]. Available: https://www.apple.com/ca/newsroom/2025/06/apple-intelligence-gets-even-more-powerful-with-new-capabilities-across-apple-devices/

[4] N. Bogoychev, P. Chen, B. Haddow, and A. Birch, "The ups and downs of large language model inference with vocabulary trimming by language heuristics," in *Proceedings of the Fifth Workshop on Insights from Negative Results in NLP*, 2024, pp. 148–153.

[5] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, "Training verifiers to solve math word problems," 2021. [Online]. Available: https://arxiv.org/abs/2110.14168

[6] L. Del Corro, A. Del Giorno, S. Agarwal, B. Yu, A. Awadallah, and S. Mukherjee, "Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference," *arXiv preprint arXiv:2307.02628*, 2023.

[7] G. Gerganov and ggml-org Developers, "Gguf," GitHub repository, 2023, [Online; Accessed: Dec. 11, 2025]. [Online]. Available: https://github.com/ggml-org/ggml/blob/master/docs/gguf.md

[8] ——, "llama.cpp: Llm inference in c/c++," GitHub repository, 2023, [Online; Accessed: Dec. 11, 2025]. [Online]. Available: https://github.com/ggml-org/llama.cpp

[9] Google. (2025) Introducing gemini: our largest and most capable ai model. [Online]. Available: https://blog.google/technology/ai/google-gemini-ai/

[10] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," in *First conference on language modeling*, 2024.

[11] C. Hooper, S. Kim, H. Mohammadzadeh, M. W. Mahoney, Y. S. Shao, K. Keutzer, and A. Gholami, "Kvquant: Towards 10 million context length llm inference with kv cache quantization," *Advances in Neural Information Processing Systems*, vol. 37, pp. 1270–1303, 2024.

[12] S. Narayan, S. B. Cohen, and M. Lapata, "Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 1797–1807. [Online]. Available: https://aclanthology.org/D18-1206/

[13] S. T. Piantadosi, "Zipf's word frequency law in natural language: A critical review and future directions," *Psychonomic bulletin & review*, vol. 21, no. 5, pp. 1112–1130, 2014.

[14] G. Qu, Q. Chen, W. Wei, Z. Lin, X. Chen, and K. Huang, "Mobile edge intelligence for large language models: A contemporary survey," *IEEE Communications Surveys & Tutorials*, 2025.

[15] QwenTeam. (2025) Qwen3-next: Towards ultimate training inference efficiency. [Online]. Available: https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list

[16] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," 2016. [Online]. Available: https://arxiv.org/abs/1606.05250

[17] C. scape. (2025) Llms with largest context windows. [Online]. Available: https://codingscape.com/blog/llms-with-largest-context-windows

[18] T. Schuster, A. Fisch, J. Gupta, M. Dehghani, D. Bahri, V. Tran, Y. Tay, and D. Metzler, "Confident adaptive language modeling," *Advances in Neural Information Processing Systems*, vol. 35, pp. 17 456–17 472, 2022.

[19] A. Sujith, "Zipf's law and its application in large language models: A comprehensive analysis," in *Proceedings of Eighth International Conference on Information System Design and Intelligent Applications*. Springer, 2025, pp. 45–55.

[20] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[21] G. Team, A. Kamath, J. Ferret, S. Pathak, N. Vieillard, R. Merhej, S. Perrin, T. Matejovicova, A. Ramé, M. Rivière *et al.*, "Gemma 3 technical report," *arXiv preprint arXiv:2503.19786*, 2025.

[22] G. Team, M. Riviere, S. Pathak, P. G. Sessa, C. Hardin, S. Bhupatiraju, L. Hussenot, T. Mesnard, B. Shahriari, A. Ramé *et al.*, "Gemma 2: Improving open language models at a practical size," *arXiv preprint arXiv:2408.00118*, 2024.

[23] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, 2016, pp. 2464–2469.

[24] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[25] A. Ushio, Y. Zhou, and J. Camacho-Collados, "An efficient multilingual language model compression through vocabulary trimming," *arXiv preprint arXiv:2305.15020*, 2023.

[26] C. Van Nguyen, X. Shen, R. Aponte, Y. Xia, S. Basu, Z. Hu, J. Chen, M. Parmar, S. Kunapuli, J. Barrow *et al.*, "A survey of small language models," *arXiv preprint arXiv:2410.20011*, 2024.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[28] Wikipedia. (2025) Simple english wikipedia. [Online]. Available: simple.wikipedia.org

[29] C. Wolfe. (2024) Decoder-only transformers: The workhorse of generative llms. [Online]. Available: https://cameronrwolfe.substack.com/p/decoder-only-transformers-the-workhorse

[30] J. Xiao, Q. Huang, X. Chen, and C. Tian, "Understanding large language models in your pockets: Performance study on cots mobile devices," *arXiv preprint arXiv:2410.03613*, 2024.

[31] J. Xu, Z. Li, W. Chen, Q. Wang, X. Gao, Q. Cai, and Z. Ling, "On-device language models: A comprehensive review," *arXiv preprint arXiv:2409.00088*, 2024.

[32] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[33] Z. Yang, Y. Cui, and Z. Chen, "Textpruner: A model pruning toolkit for pre-trained language models," *arXiv preprint arXiv:2203.15996*, 2022.

[34] F. Zhang, Y. Liu, W. Li, J. Lv, X. Wang, and Q. Bai, "Towards superior quantization accuracy: A layer-sensitive approach," *arXiv preprint arXiv:2503.06518*, 2025.

[35] H. Zhang, Y. Zhou, T. Fang, Z. Yuan, G. Dai, and Y. Wang, "Vocabtailor: Dynamic vocabulary selection for downstream tasks in small language models," *arXiv preprint arXiv:2508.15229*, 2025.

[36] H. Zhang and J. Huang, "Challenging gpu dominance: When cpus outperform for on-device llm inference," 2025. [Online]. Available: https://arxiv.org/abs/2505.06461

[37] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," 2020. [Online]. Available: https://arxiv.org/abs/1904.09675

Authorized licensed use limited to: The University of Toronto. Downloaded on November 23,2026 at 16:20:15 UTC from IEEE Xplore. Restrictions apply.