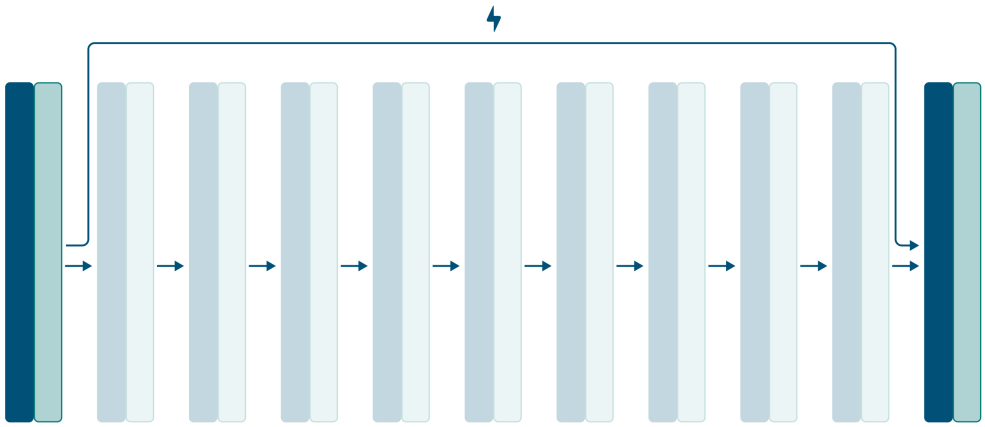
CHALMERS
UNIVERSITY OF TECHNOLOGY



Skip, Search, Speculate

Turning a Frozen LLM into Its Own Lightweight Drafter for Lossless Inference Speedup

Master's thesis in Computer science and engineering

Hugo Olsson

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Master's thesis 2026

Skip, Search, Speculate

Turning a Frozen LLM into Its Own Lightweight Drafter for Lossless Inference Speedup

Hugo Olsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Skip, Search, Speculate
Turning a Frozen LLM into Its Own Lightweight Drafter for Lossless Inference Speedup
Hugo Olsson

© Hugo Olsson, 2026.

Supervisor: Matti Karpa, Data Science och AI
Examiner: Devdatt Dubhashi, Data Science och AI

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2026

Skip, Search, Speculate
Turning a Frozen LLM into Its Own Lightweight Drafter for Lossless Inference Speedup
Hugo Olsson
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis has investigated the possibility to speedup inference by producing a self-speculative setup where the original model is verifier and drafter. The model is frozen so the original quality is preserved and no retraining required. The frozen model switches between being verifier and drafter. In the drafter mode, a gap of almost all layers are skipped and an approximated nearest neighbors (ANN) LM-head is used. To prevent completely degraded performance from skipping many layers, this thesis proposes Hidden Vector Casting (HVC) which casts the hidden vector from the geometric representation of the exit-layer to the geometric representation of the entrance-layer.

The thesis has produced an open source implementation to train the HVC, build the ANN head and run self-speculative inference with KV-cache handling. It also includes experiments to evaluate what layer-skipping ablations that are the best to skip as many layers as possible while degrading the generation performance as little as possible.

The thesis found that skipping an contiguous internal gap of layers is in general advantageous compared to early-exit, late-start of an periodic variant of skipped layers. The selected layer-skipping ablations to produce inference systems were (1,1) and (2,2) which means that all layers except the first and the last are skipped or all layers skipped except the two first and two last respectively.

The found results are average speedups of 1.21× to 1.63× while having on average 0.9% higher peak memory usage compared to normal inference and producing identical generation to normal inference up to what the selected floating point precision allows for. The speedup depends on the LM model and the type of prompt used. Larger models seem to have higher potential for speedup while smaller models seem to have less overhead to save computation and thus smaller potential for speedup. Since the mechanism fundamentally is to cheaply speculate ahead of the full model, the results also indicate that more concrete prompts get larger speedups while more open-ended prompts produces lower speedups in general.

The measured total times to train the HVC and to build the ANN LM-head are 16 to 38 minutes depending on the model. The HVC was trained on a NVIDIA RTX PRO 6000 and the ANN LM-head was built on an Apple MacBook Pro M5 24GB.

The thesis uses the models Llama 3.2 1B Instruct, Llama 3.2 3B Instruct, Llama 3.1 8B Instruct, Mistral 7B Instruct 0.3v, and Qwen3 4B. The thesis only investigated smaller models in the range 1–10B parameters due to limitation in computational access. The concept should however work for larger models as well.

Keywords: large language models, self-speculative decoding, inference acceleration, layer skipping, hidden vector casting, approximate nearest-neighbor LM-heads, lossless generation.

Acknowledgements

I would like to thank my supervisor Matti Karppa and examiner Devdatt Dubhashi for guidance in producing this masters thesis. I would also like to thank the department of physics and department of computer science for 5 great years that have allowed me to learn engineering from core math and first principles to concrete implementation.

Hugo Olsson, Gothenburg, June 2026

List of Acronyms

Below is the list of acronyms and abbreviations that have been used throughout this thesis, listed in alphabetical order:

ANN	Approximate nearest neighbors
ANNH	Approximate Nearest Neighbors Head
CE	Cross entropy
CUDA	Compute Unified Device Architecture
GPU	Graphics processing unit
HVC	Hidden vector casting
KL	Kullback-Leibler divergence
KV-cache	Key-value cache
LM	Language model
LLM	Large language model
LM-head	Language model head
VRAM	Video random-access memory

Table of Contents

List of Acronyms ix
List of Figures xii
List of Tables xv
List of Algorithms xvii
1 Introduction 1
1.1 Background and theory 1
1.2 Research Questions 8
1.3 Scope and Limitations 9
1.3.1 Scope 9
1.3.2 Limitations 9
2 Methodology 11
2.1 Environment 11
2.2 Datasets 12
2.3 Body Approximation 12
2.3.1 Finding best layer-skipping ablations 13
2.3.2 HVC setup 14
2.3.3 Training skipping layers 15
2.4 ANNH implementation 16
2.4.1 ANNH cluster builder 17
2.5 Inference 18
2.5.1 Self-speculative decoding 18
2.5.1.1 KV-cache 20
2.6 Benchmarking 20
2.6.1 Prompt sets 20
2.6.2 Benchmark phases 22
2.6.3 Timing measurements 22
2.6.4 Plot metrics 22
3 Results 25
3.1 Skip ablations 25
3.2 ANNH cluster building and evaluation 30
3.2.1 Llama 3.2 1B Instruct 30
3.2.1.1 5344 clusters 30
3.2.1.2 2672 clusters 31
3.2.1.3 8016 clusters 32
3.2.1.4 16032 clusters 33
3.2.2 Llama 3.2 3B Instruct 34
3.2.2.1 8016 clusters 34
3.2.2.2 16032 clusters 35
3.3 Training HVC-bridge 36
3.3.1 Training (1,1) gap 36
3.3.2 Training (2,2) gap 37

3.4 Measuring speedups and memory usage ... 38
3.4.1 Gap (1,1), block size 2, bfloat16 ... 38
3.4.1.1 Concrete prompt set ... 39
3.4.1.2 Python-diverse prompt set ... 44
3.4.2 Benchmark summary ... 46
3.4.3 float32 debug test ... 47
4 Discussion ... 49
4.1 Interpreting the results ... 49
4.1.1 Layer-skipping ablations ... 49
4.1.2 HVC bridge training ... 49
4.1.3 ANNH cluster building ... 50
4.1.4 ANNH accuracy ... 50
4.1.5 ANNH with self-speculation ... 51
4.1.6 Exact match and numerical precision ... 51
4.1.7 Why not test with fewer skipped layers ... 52
4.1.8 Self-speculation speedups and memory usage ... 53
4.1.9 How long is the total training time? ... 54
4.1.10 Does this approach make sense? ... 54
4.2 Answering the research questions ... 55
4.2.1 Which layer-skipping strategy minimizes damage to generation quality per layer skipped, and does this pattern hold across model families? ... 55
4.2.2 Can a lightweight HVC bridge recover the generation quality lost from skipping layers well enough to produce an effective drafter? ... 55
4.2.3 What is the minimum acceptance rate required for the proposed self-speculative setup to outperform normal inference, given empirically observed verifier and drafter costs? ... 55
4.2.4 To what extent can inference for LLMs be sped up by using a setup where the draft model is made computationally cheaper in both body and head by using the techniques: skipping layers + HVC + ANNH + self-speculative decoding? ... 56
4.3 Hypothesis about easy and hard tokens ... 57
4.4 Future work ... 58
4.4.1 Adaptive block sizes ... 58
4.4.2 Testing on larger models ... 58
4.4.3 Training strategy ... 59
4.5 Related work ... 60
4.5.1 Efficient LM heads ... 60
4.5.2 Speculative decoding ... 60
4.5.3 Layer skipping and self-speculative decoding ... 61
4.5.4 Intermediate representations and Tuned Lens ... 62
4.5.5 Position of this thesis ... 62
5 Conclusion ... 63
5.1 Summary ... 63
5.2 Contributions ... 64
Bibliography ... 65

List of Figures

Figure 1 Illustration of the idea to score clusters, then only select among token unembedding vectors inside the top-k clusters. In the illustration top-$k = 3$. The green vector is the query hidden vector produced by the body. In this case, the query vector is in the intersection of all returned clusters, so the top matching vector can be in any of them. The figure uses clustering in 3D presented as a 2D profile image. 3
Figure 2 Illustration of how KV-cache helps to not recalculate the entire prefix when generating next token. 6
Figure 3 Structure of skipping layers. The low opacity layers in the middle are skipped. . . 12
Figure 4 The input to the HVC bridge. It gets a stacked vector of the final hidden vector from token position $t - 1$ and the hidden vector from the last layer before the gap at position t. 14
Figure 5 How a training window is structured. It is a window of context_len tokens divided into num_draft_sections sections. The vertical splitting lines indicate starting points where the student continues from how the teacher's KV-cache was at that position. At each position it is getting the teachers $t - 1$ final hidden token, this is the shifted pink array, and the t position last layer hidden vector before the gap. 15
Figure 6 KL divergence per skipped layer for skip ablations of meta-llama/Llama-3.2-1B-Instruct. 25
Figure 7 Top-1 agreement with the full model for skip ablations of meta-llama/Llama-3.2-1B-Instruct. 26
Figure 8 KL divergence per skipped layer for skip ablations of meta-llama/Llama-3.2-3B-Instruct. 26
Figure 9 Top-1 agreement with the full model for skip ablations of meta-llama/Llama-3.2-3B-Instruct. 27
Figure 10 KL divergence per skipped layer for skip ablations of mistralai/Mistral-7B-Instruct-v0.3. 27
Figure 11 Top-1 agreement with the full model for skip ablations of mistralai/Mistral-7B-Instruct-v0.3. 28
Figure 12 KL divergence per skipped layer for skip ablations of meta-llama/Llama-3.1-8B-Instruct. 28
Figure 13 Top-1 agreement with the full model for skip ablations of meta-llama/Llama-3.1-8B-Instruct. 28
Figure 14 KL divergence per skipped layer for skip ablations of Qwen/Qwen3-4B-Instruct-2507. 29
Figure 15 Top-1 agreement with the full model for skip ablations of Qwen/Qwen3-4B-Instruct-2507. 29

Figure 16 Mean assigned cosine similarity during ANNH clustering of meta-llama/Llama-3.2-1B-Instruct LM-head vectors over clustering iterations. 30
Figure 17 Top-1 agreement between drafter and verifier during training for the $(1, 1)$ gap setting. Curves show the first 10 points raw, followed by a 5-point centered moving average. The right panel reports the final average over the last 20 datapoints. . . 36
Figure 18 KL divergence from verifier to drafter during training for the $(1, 1)$ gap setting. Curves show the first 10 points raw, followed by a 5-point centered moving average. The right panel reports the final average over the last 20 datapoints. 37
Figure 19 Top-1 agreement between drafter and verifier during training for the $(2, 2)$ gap setting. Curves show the first 10 points raw, followed by a 5-point centered moving average. The right panel reports the final average over the last 20 datapoints. . . 37
Figure 20 KL divergence from verifier to drafter during training for the $(2, 2)$ gap setting. Curves show the first 10 points raw, followed by a 5-point centered moving average. The right panel reports the final average over the last 20 datapoints. 38
Figure 21 Per-token self-speculation speedup for Llama 3.1 8B Instruct on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt. 39
Figure 22 Per-token self-speculation speedup for Llama 3.2 3B Instruct on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt. 40
Figure 23 Per-token self-speculation speedup for Llama 3.2 1B Instruct on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt. 41
Figure 24 Per-token self-speculation speedup for Mistral 7B Instruct 0.3v on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt. 42
Figure 25 Per-token self-speculation speedup for Qwen3 4B Instruct on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt. 43
Figure 26 Per-token self-speculation speedup for Llama 3.1 8B Instruct on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2. 44
Figure 27 Per-token self-speculation speedup for Llama 3.2 3B Instruct on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2. 44
Figure 28 Per-token self-speculation speedup for Llama 3.2 1B Instruct on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2. 45
Figure 29 Per-token self-speculation speedup for Mistral 7B Instruct v0.3 on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2. 45

Figure 30 Per-token self-speculation speedup for Qwen3 4B Instruct on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2. 46
Figure 31 Per-token self-speculation speedup for Llama 3.2 1B Instruct on the concrete prompt set. This is a debug run to investigate generation correctness. The datatype for the model is float32. 47
Figure 32 Observed self-speculation speedup with ANNH compared to model size on the concrete prompt set. The fitted line suggests a positive correlation between model size and speedup in these measurements, but the sample is small and mixes model families, so it should be interpreted an illustraion of what this project has seen rather than any conclusive result. 58

List of Tables

Table 1 The different layer skip ablations tested in the project. 13
Table 2 Summary of HVC training metrics 16
Table 3 Prompt sets used in the main self-speculation benchmark. 20
Table 4 Setup and runtime fields shown in the benchmark plot footers. 23
Table 5 Measured result and profile fields shown in the benchmark plot footers. 23
Table 6 Cluster quality metrics for the 5344-cluster ANNH index built from Llama 3.2 1B Instruct LM-head vectors. 30
Table 7 Match-rate sweep over top_k_clusters. 31
Table 8 Cluster quality metrics for the 2672-cluster ANNH index built from Llama 3.2 1B Instruct LM-head vectors. 31
Table 9 Match-rate sweep over top_k_clusters. 32
Table 10 Cluster quality metrics for the 8016-cluster ANNH index built from Llama 3.2 1B Instruct LM-head vectors. 32
Table 11 Match-rate sweep over top_k_clusters. 32
Table 12 Cluster quality metrics for the 16032-cluster ANNH index built from Llama 3.2 1B Instruct LM-head vectors. 33
Table 13 Match-rate sweep over top_k_clusters. 33
Table 14 Cluster quality metrics for the 8016-cluster ANNH index built from Llama 3.2 3B Instruct LM-head vectors. 34
Table 15 Match-rate sweep over top_k_clusters. 34
Table 16 Cluster quality metrics for the 16032-cluster ANNH index built from Llama 3.2 3B Instruct LM-head vectors. 35
Table 17 Match-rate sweep over top_k_clusters. 35
Table 18 Self-speculation benchmark results. Each cell reports the total per-generated-token speedup for the skipped-layers + ANNH variant, followed by total draft-token acceptance rate. Bold values mark the better block size for each prompt set and model, when only one block size was tested for that prompt set, that value is bold by default. All combinations in this table are runs from the main benchmark matrix. 46
Table 19 Acceptance rate required to reach a theoretical self-speculation speedup of 1.4 × with block size $\gamma = 1$ and verifier cost $v = 1.05$. 50
Table 20 Acceptance rate needed for a larger (N, N) drafter with ANNH to match the 1.47 × theoretical speedup of a (1,1) + ANNH drafter with 65% acceptance rate, using block size $\gamma = 1$, verifier cost $v = 1.05$, and a 32-layer model. 53
Table 21 Total setup time for the HVC bridge and ANNH index used in the self-speculation speedup benchmarks. 54

Table 22 Minimum draft-token acceptance rate needed to outperform normal inference, using measured verifier and drafter costs from the concrete prompt set with gap $(1, 1)$, block size 2, and ANNH enabled. The measured acceptance rates and speedups are from the same benchmark runs. 56

List of Algorithms

Algorithm 1 Algorithm to build FlashHead-like ANNH cluster of token vectors. 17
Algorithm 2 Algorithm for approximate greedy best matching token lookup using built ANNH cluster. 18
Algorithm 3 The procedure for self-specualtive decoding and KV-cache management. 19

1

Introduction

Large language models (LLMs) have become popular since the release of ChatGPT [1] and are used weekly by hundreds of millions of people for personal and professional tasks [2]. LLMs are expensive to run. They require a lot of memory and compute [3,4]. It is of interest to make them efficient so that the best possible model can run on available hardware and energy resources. LLMs use an architecture called the *transformer* which can be understood as having a body and a head. A recent paper called FlashHead [5] proposed a solution to speed up the head using approximate nearest neighbors (ANN). This ANN head solution seems promising but a key insight is that the head is usually 1–20% of the total compute to produce the next token. This means that even if the head becomes *zero* in computational cost, the total speedup cannot exceed 1.01×–1.25× since the compute of the body is still there and becomes the bottleneck. This thesis takes an Amdahl's law framing and produces an inference setup where both the head and the body are approximated by skipping layers and using an ANN head respectively. Together, they compose a formula to enable a frozen LLM[1] to have a cheap drafter mode for itself to perform self-speculation. This enables speedups where the Amdahl's ceiling is substantially increased, while also guaranteeing an unchanged model and lossless generation compared to normal inference.

1.1 Background and theory

The architecture of a transformer model can split into a body and a head. The body is composed of multiple layers, where each layer is one Attention block and one MLP block. The body produces an internal state[2] per token position, that is updated layer by layer. The internal state per token position is represented as a high-dimensional vector called the *hidden vector*. After all layers the hidden vector at the last token position is transformed into the next token the transformer will generate. The transformation from hidden vector to next token is performed in the head. Tokens are the atoms the transformer operates with, it doesn't generate character by character and it also doesn't necessarily generate entire words at a time. Tokens are learned[3] good sub-strings to represent the domain of text the LLM is trained for. In practice, those sub-

[1] A frozen model means that its parameters are unchanged. No retraining or fine-tuning performed.

[2] The internal state can be thought of as the transformer's "understanding" at that token position per layer. So there are internal states for each token position and after each layer.

[3] Finding good tokens is usually an optimization problem performed separately before training of the LLM parameters.

strings are often something like “hello”, “Chal”, “mers”, “apple”, “opti”, “miz”, “ing” etc. All these tokens together make up what is called the transformer’s *vocabulary*.

The head has a large matrix called the *unembedding matrix*. This matrix is like a stack of vectors, one for each token in the vocabulary. The head projects the produced hidden vector for the last position to this unembedding matrix to get a dot product score with each token-unembedding-vector inside it. Tokens that get a high match with the hidden vector are likely to be sampled as the next token to generate. The selection can also be so simple that we just generate the token that has the highest match with the hidden vector – this is called greedy sampling.

As mentioned, the head is usually 1–20% of the total compute to produce a next token, depending on how big the head is compared to the body. Smaller models are closer to the upper bound whereas larger models are closer to the lower bound due to how the head/body ratio typically grows with parameter count.

The paper FlashHead proposes a technique to approximate the head projection with approximate nearest neighbors (ANN) [5]. This allows the transformer to find the next token to generate without calculating a score for all tokens in the vocabulary. However, the key framing in this thesis is that even though you make the head *zero* in computational cost, the total speedup is bounded by the fraction of total compute the head represents. This is the Amdahl’s law perspective that motivates this thesis. So if the head is 20% of the total compute to generate the next token, the total speedup by approximating the head cannot exceed $\frac{1}{0.8} = 1.25$.

The idea is then to improve the speed of both the head and the body to increase the upper bound of possible speedup. The technique investigated to speedup the body is *skipping layers*[4]. The hypothesis is that, during generation, there are easy and hard tokens, and for easy tokens, maybe not all layers of processing will be needed. Some of the layers can then be skipped without hurting the quality of generation significantly. This would inherently reduce the computation needed in the body since fewer layers processed means less matrix-multiplications and calculations to perform.

A key constraint in this project is that these techniques should be applied as drop-in or close to drop-in. This means that skipping layers is evaluated from the perspective of not retraining the model to handle this. In theory, allowing retraining of the model can allow for better behavior, but it also raises the bar for needed training-compute and it loses the guarantee that the model will behave the same as the original. Improving the efficiency of both the head and the body becomes even more important when the model grows in size since the head shrinks in portion of total compute, making the Amdahl ceiling even lower.

When skipping layers, there are primarily three different cases: **early-exit**, **gap-jump** or **late-start**. Early-exit refers to taking the hidden vector after a certain number of layers and then using it directly in the head, essentially exiting the body early. Gap-jump refers to skipping a number of contiguous layers in the middle so the hidden vector is taken after a layer and is given back to a later layer, not directly to the head. Late-start is the opposite of early-exit, it is when the token embedding vector skips a number of initial layers and then is processed by a number of layers before the head. There are theoretically also other variations where you skip every other layer or preserve a custom set of the total layers. These will be focused less on but

[4]The common way to skip layers is by exiting early, then you pull the hidden vector early and skip the last layers. Here *skipping layers* is used more generally since it can mean to skip the first layers, a gap in the middle, every other layer, the last ones etc.

are still included in some measurements. The report uses the notation (N, M) to represent a section of skipped layers. With this notation, (1,1) would mean that all layers except the first and the last are skipped. (2,2) would mean that all but the first two and last two are skipped. The notation for an early-exit is then (N, 0) and (0, N) for late-start.

Hidden vector casting

When skipping layers, it's not obvious that the hidden vector can be efficiently interpreted where it lands. For the easy tokens, even if the hidden vector has semantically converged early, it's not necessarily the case that the same semantics is represented geometrically equal in later layers. This means that the hidden vector could semantically converge quickly for easy tokens, aligning with the hypothesis, while at the same time not producing correct tokens when moved directly to the head or later layers. Essentially that semantic convergence does not mean geometric convergence for the hidden vector. This will likely break the ability to skip layers by just picking the hidden vector and putting it back in a more downstream layer. To address this challenge, this thesis uses a technique it calls hidden vector casting (HVC). The idea is that a small learned transformation can translate geometry from where it is taken to where it is placed, essentially casting it to the correct geometric representation corresponding to its semantic meaning. In practice, this HVC can be implemented in multiple ways but this project's chosen transformation to evaluate is a linear HVC.

ANN LM-Head

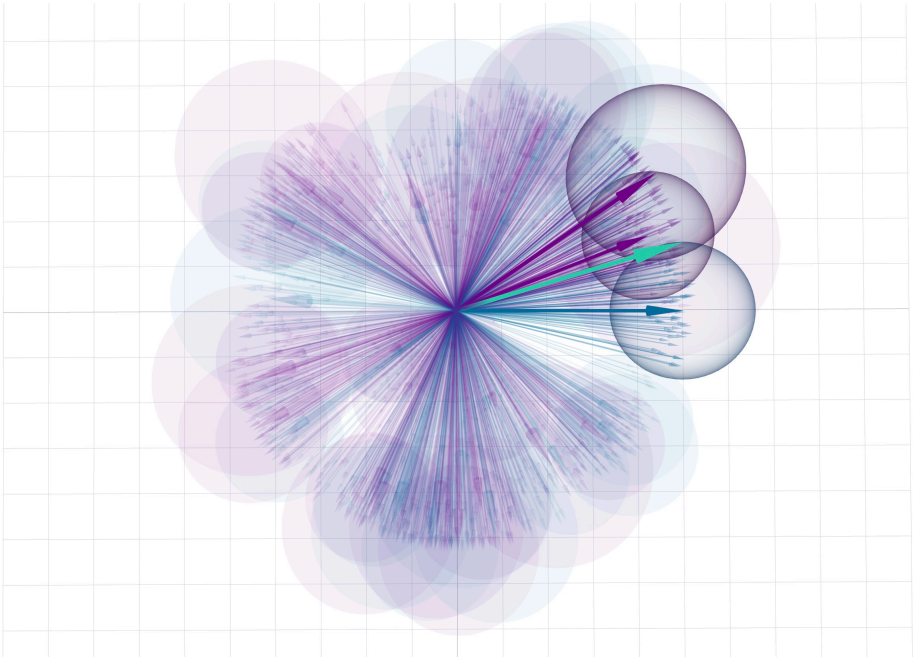


Figure 1: Illustration of the idea to score clusters, then only select among token unembedding vectors inside the top-k clusters. In the illustration top-k = 3. The green vector is the query hidden vector produced by the body. In this case, the query vector is in the intersection of all returned clusters, so the top matching vector can be in any of them. The figure uses clustering in 3D presented as a 2D profile image.

The LM-head takes the hidden vector produced by the body and projects this to the unembedding vectors for all tokens in the vocabulary. This produces a dot product score with all tokens, essentially matching-scores. These scores are called logits and can be positive or negative. To then get probabilities for each token, a softmax is performed over the vocabulary

which normalizes the logits into a probability distribution that sums to 1. This is expensive, the hidden vector often has thousands of dimensions and the vocabulary is often more than 100,000, so the matrix to multiply is big. FlashHead proposed an idea to avoid performing the full LM-head unembedding matrix multiplication. The core concept is to organize similar unembedding vectors into clusters where each cluster has a centroid vector that is the average of the included vectors. During inference, the query hidden vector is first used to calculate dot products only with the cluster centroids. Then the top-k best matched clusters are selected, and scores are calculated only with the unembedding vectors from those. For example, if the vocabulary is 150,000, the number of clusters is 10,000 and top-k of 100 is used, then the initial matrix multiplication is with 10,000 vectors instead of 150,000 vectors, and then $100 \times \frac{150000}{10000} = 1500$ unembedding vectors are gathered and scored. So the process of finding the best matching tokens is divided into the two-step process of scoring a small routing-matrix and then scoring only the unembedding vectors that are close to the query hidden vector.

Figure 1 illustrates this concept. The green query hidden vector is scored with the centroids and the top 3 clusters are selected, illustrated with clusters rings of more contrast. That illustration is only a 2D image of clusters in 3D, so for a real vocabulary there will be many more dimensions and thus more space for the unembedding vectors to exist in.

This method is not guaranteed to return the best matching token. Since the clusters are represented by a centroid that is the average of their included unembedding vectors, the routing can score clusters with a well matching centroid higher than the cluster that actually has the true top token.

This project produces an implementation of this idea and refers to it as ANNH which stands for Approximate Nearest Neighbors Head.

Speculative decoding

The proposed methods to speedup the head and body use approximations of the full calculations. This can save compute but will also make generation quality worse, it's just a question about how much worse. When selecting an inference setup, it is usually not satisfying to apply inference speedups without clear information how the generation quality is affected. If the speedup is 30% but the generation quality has dropped with 30% then it's not necessarily a good deal. To keep the final output equivalent to normal decoding, this project uses speculative decoding [6,7]. The idea behind speculative decoding is that it's much faster to verify tokens than it is to produce them, because it can be performed in parallel. The setup is that you have a *drafter* and a *verifier* model. The drafter generates a block of tokens and the verifier verifies if the tokens are the same as the verifier would have generated. If yes, then the tokens are accepted, and if not, then they are rejected from the point where the verifier disagrees. If the drafter is fast and fairly accurate, this setup can be more performant than running the model normally while also not compromising the quality of generation. For this thesis, since the model parameters are unchanged, the verifier and drafter can be the same model, just running with different inference logic during drafting and verification. This is called *self-speculation*. The advantage with this is that only one model needs to be loaded into GPU memory.

The speedup from self-speculative decoding can be theoretically estimated by using the variables γ to denote the block size, d to denote the drafter to normal ratio, v for the verifier to normal ratio, and a to denote the acceptance rate.

The acceptance rate is here defined as

$$a = \frac{\text{total accepted draft tokens}}{\text{total drafted tokens}}.$$

The variables d, v are defined as time to call the drafter or verifier compared to the normal full model. If $d = 0.1$, then the drafter needs 10% to generate a next token compared to what the full normal model would take. If $v = 1.05$, then it takes the verifier $1.05\times$ of the normal next token generation time to verify the entire block with γ tokens. The following estimation is produced by this project but similar estimations might exist elsewhere in the literature.

Let T_{normal} be the absolute time for one normal full model generation step. Let T_{verifier} be the absolute time for the verifier to verify a drafted block of tokens, and let T_{drafter} be the absolute time for one drafter step.

Each speculated block is γ tokens. So the expected number of accepted draft tokens per round is

$$\gamma a.$$

From the way a transformer works, each verifier call produces an extra token after the last token the verifier processed. This is called a bonus token. Therefore, the expected number of output tokens per speculative round is

$$\mathbb{E}[\text{tokens per round}] = 1 + \gamma a.$$

The time cost of one speculative round is one verifier call plus γ drafter steps:

$$\mathbb{E}[\text{time per round}] = T_{\text{verifier}} + \gamma T_{\text{drafter}}.$$

Normal autoregressive decoding would need one normal model call per output token. Therefore, producing the same expected number of tokens normally would take

$$\mathbb{E}[\text{normal time for same tokens}] = (1 + \gamma a) T_{\text{normal}}.$$

The estimated speedup is normal time divided by self-speculative time:

$$S = \frac{(1 + \gamma a) T_{\text{normal}}}{T_{\text{verifier}} + \gamma T_{\text{drafter}}}.$$

Now substitute the relative variables that were defined in the beginning $T_{\text{verifier}} = vT_{\text{normal}}$ and $T_{\text{drafter}} = dT_{\text{normal}}$ gives

$$S = \frac{(1 + \gamma a) T_{\text{normal}}}{vT_{\text{normal}} + \gamma dT_{\text{normal}}}.$$

Factoring out T_{normal} from the denominator gives

$$S = \frac{(1 + \gamma a) T_{\text{normal}}}{(v + \gamma d) T_{\text{normal}}}.$$

The T_{normal} terms cancel, leaving

$$S = \frac{1 + \gamma a}{v + \gamma d}. \tag{1}$$

Using this form, the speedup can be estimated from how much compute the drafter and verifier use relative to normal inference. Since the verifier is the full model processing γ draft tokens in a single parallel forward pass, v should be greater or equal to 1. Furthermore, since verifying γ tokens in parallel should be cheaper than generating them sequentially, v is expected to be

significantly less than the block size γ. This is the core of why speculative decoding can be faster than normal generation.

The acceptance rate definition used in this thesis is not necessarily consistent with all other papers in the space, as the convention varies. Some papers use a per-position conditional rate, here called a_c, defined as the probability that a drafted token is accepted given that all prior tokens in the draft block were accepted. This means that acceptance rates cannot necessarily be compared without having this in mind.

Key-value cache

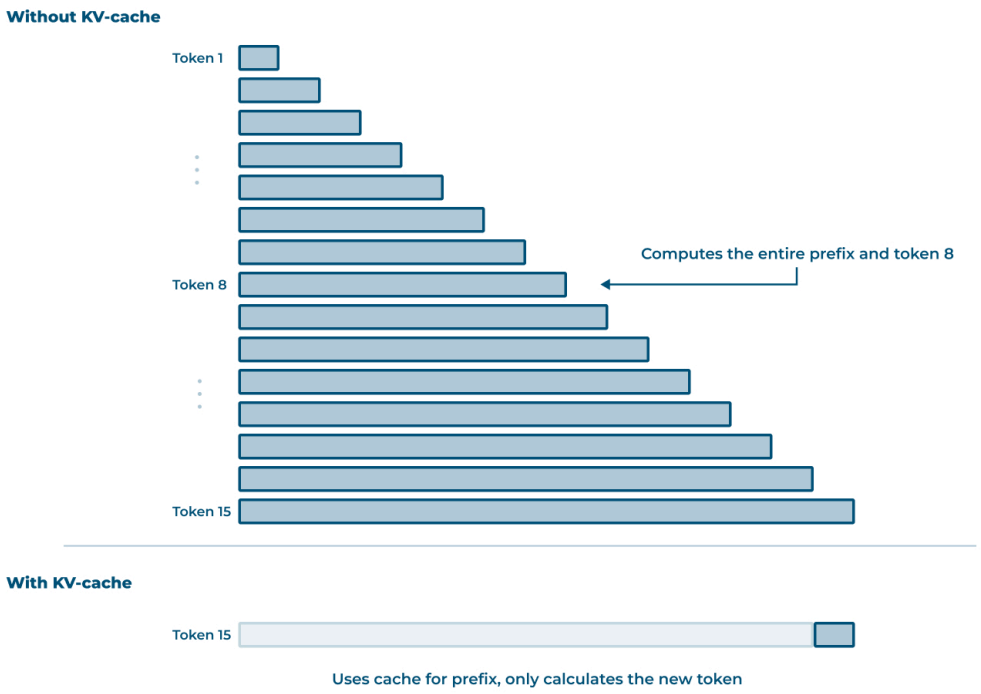


Figure 2: Illustration of how KV-cache helps to not recalculate the entire prefix when generating next token.

During generation, each new token attends to all previously generated tokens. Without caching, this would require recomputing the same attention values for every prior token at every new step, making generation quadratically more expensive as the sequence grows. The key-value cache (KV-cache) avoids this by storing the intermediate attention computations for already processed tokens, so each new step only needs to compute attention for the newest token. This makes generation significantly faster and is standard in practical LLM inference systems.

For speculative decoding, if the drafter and verifier are two different models, then they need a KV-cache each. This increases memory usage compared to normal generation which is a drawback. When doing self-speculation, the drafter and verifier can share KV-cache since they are fundamentally the same model just with different inference logic.

Teacher-student training

To train for speculative decoding, the goal is for the drafter to produce the same token the full model would generate. So the task is not to produce the next token that is the best or makes the most sense, because the full model, which will act as verifier, will reject any token that isn't the same as what it would generate[5]. To train for this, a setup called teacher-student training is used. This means that the training signals will be from the deviation in behavior from the full model. In this report, since it is for the application of training for self-speculative decoding, the terms teacher/student and verifier/drafter might be used interchangeably depending what is most natural to the context.

Cross entropy, KL divergence and Top-1

The output of the LM-head is a vector of logits, one value for each token in the vocabulary. After applying softmax, these logits become a probability distribution over the next token. This means that two inference setups can be compared not only by checking whether they choose the same top token, but also by comparing the full probability distributions they assign over the vocabulary.

Cross entropy is typically used when there is a target token, such as when training an LLM to predict the same tokens as an example from a dataset. If the target token is y and the model assigns probability $q(y)$ to it, the cross entropy loss is

$$L_{\mathrm{CE}} = -\log q(y).$$

This loss becomes small when the model gives high probability to the target token. In this thesis, cross entropy is mainly used to train the drafter to put high probability on the token selected by the verifier. This is especially relevant for greedy speculative decoding, where a drafted token is accepted when it matches the verifier's selected token.

KL divergence is used to compare two full probability distributions. Let p be the verifier's next token distribution and q be the drafter's next token distribution. The KL divergence from verifier to drafter is

$$D_{\mathrm{KL}(p \parallel q)} = \sum_i p_i \log \frac{p_i}{q_i}.$$

A low KL divergence means that the drafter assigns probability mass similarly to the verifier, not only to the top token but across the vocabulary. This is useful because two models can have the same top-1 token while still having different probability over other tokens. In this thesis, KL divergence is therefore used as a measure of how closely the drafter imitates the verifier's full next-token behavior.

Top-1 agreement is used to measure whether two distributions make the same top-1 choice. For each token position, the top-1 token is the token with the highest probability. The top-1 agreement between drafter and verifier is then the fraction of positions where they select the same highest probability token:

$$\text{Top-1 agreement} = \frac{\text{matching top-1 tokens}}{\text{evaluated token positions}}.$$

[5]The verifier will reject any token that itself wouldn't generate if greedy decoding is used. For this report, greedy decoding is always used.

Top-1 is not used directly as a training objective in this thesis. It is a measurement of the placed token and does not show how close the rest of the distribution is. For example, a drafter can have the same top-1 token as the verifier while assigning very different probabilities to the other tokens. However, top-1 is a useful and intuitive metric because it measures exactly whether the drafter and verifier would make the same top-1 next token choice. This makes it especially relevant for the greedy self-speculative decoding used in this thesis.

1.2 Research Questions

The following research questions are addressed:

1. Which layer-skipping strategy minimizes damage to generation quality per layer skipped, and does this pattern hold across model families?
2. Can a lightweight HVC bridge recover the generation quality lost from skipping layers well enough to produce an effective drafter?
3. What is the minimum acceptance rate required for the proposed self-speculative setup to outperform normal inference, given empirically observed verifier and drafter costs?
4. To what extent can inference for LLMs be sped up by using a setup where the draft model is made computationally cheaper in both body and head by using the techniques: skipping layers + HVC + ANNH + self-speculative decoding?

Following the Amdahl's law reasoning presented in the background, these enhancements could significantly improve the speed of the draft model. They will make its produced quality strictly worse, but since this setup uses a verifier, the output will be lossless compared to the original model. It is not obvious that this will produce a solution that is better than simply running the model normally. The quality of the drafter could become so weak that there is more harm than good to use this inference setup. In that case, the setup would add complexity without any performance gain. This provides a natural baseline for evaluation.

1.3 Scope and Limitations

1.3.1 Scope

1. To produce an inference implementation for layer skipping with HVC, allowing a model to skip a contiguous block of internal layers.
2. To produce a training setup for the HVC to match inference objectives.
3. To produce an ANNH implementation that uses the method described in the FlashHead paper.
4. To produce a self-speculation setup that uses the original model as verifier and drafter but where skipping layers and ANNH is used in drafter-mode.
5. To produce a KV-cache implementation so that the verifier and drafter share the same KV-cache.
6. To build a testing setup that investigates the best layer-skipping choices.
7. To produce a benchmark that compares real speedup from self-speculation compared to normal generation with detailed diagnostics and profiling.

1.3.2 Limitations

This project has several limitations to keep the workload feasible:

1. Model scale: Only small-scale models are used (about 1B–10B parameters). This limitation makes building and testing quicker since the models need less resources and time to run. It also makes sense because it is where ANNH gives the biggest speedup and thus differentiation for this research.

2. Hardware testing: The project doesn't do comprehensive testing of performance on different chips and possible hardware. The benchmark GPUs are NVIDIA L4 and L40s.

3. Model selection: This project limits itself to a small number of open models from Llama 3.1, Llama 3.2, Qwen 3 and Mistral.

2

Methodology

The overall methodology for the project is to implement the inference setups in PyTorch. Different configurations with different hyper-parameters are measured how well they perform for their targeted task. Metrics like cross-entropy, KL divergence, Top1 match and acceptance rate is used to measure performance. The project is available as an open source repository at SkipSearchSpec[6]. All measurements presented in this thesis uses the same measurement-pipeline so that they always follow a shared structure, includes their git commit/tag and stores the raw data in the repository. By including the exact commit for each result, the reader can always visit the exact state of the project for when a measurement was performed.

The methods to increase the speed of the body and head are largely separable. The job of the body is to deliver well converged hidden vectors and the job of the head is to find matching tokens given a hidden vector. The project therefore has several functions that isolates the task of skipping layers in the body, and then other functions that isolates the task of speeding up the head with ANNH. The best found solutions for each part are used to produce a drafter in a self-speculative setup where the acceptance rate, correctness and speedup are measured.

2.1 Environment

The project is written with Python, PyTorch and Hugging Face. It uses open instruction-tuned models from Meta Llama, Mistral AI and Qwen: meta-llama/Llama-3.2-1B-Instruct, meta-llama/Llama-3.2-3B-Instruct, Qwen/Qwen3-4B-Instruct-2507, meta-llama/Llama-3.1-8B-Instruct, and mistralai/Mistral-7B-Instruct-v0.3. The models are run with completion-style prompt sets through their standard Hugging Face forward functions. To skip layers, targeted transformer layers are overwritten with a hook that turns them into no-operations, passing the hidden vector forward unchanged.

[6]Repository URL: https://github.com/HugoOlsson/SkipSearchSpec.

```python
class NoOpDecoderLayer(nn.Module):
    """
    Cheap replacement for a HF decoder layer.
    """

    def forward(
        self,
        hidden_states: torch.Tensor,
        *args: Any,
        **kwargs: Any,
    ) -> torch.Tensor:

        return hidden_states
```

2.2 Datasets

The datasets used to train the HVC-bridge are 18% HuggingFaceTB/cosmopedia-100k [8], 18% codelion/fineweb-edu-1B [9], 41% MBZUAI/LaMini-instruction [10], 18% flytech/python-codes-25k [11], and 5% roneneldan/TinyStories [12]. All training runs use a single epoch. This is to maximize the amount of examples seen given the allocated compute, but also to get KL/CE/top1 training graphs that don't include progress where the module has seen the data before. The datasets are selected to work with the completion-style training setup and to be a good foundation to handle a spectrum of different prompts.

2.3 Body Approximation

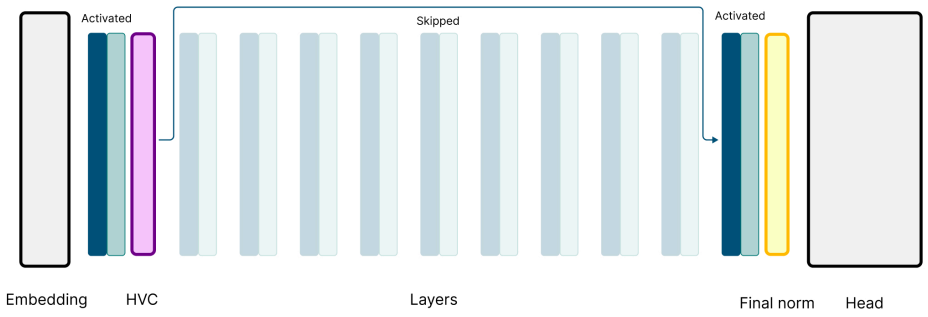


Figure 3: Structure of skipping layers. The low opacity layers in the middle are skipped.

Figure 3 illustrates the architecture when skipping layers. The figure illustrates the case of a gap-jump. If the variant is early-exit then the hidden vector goes directly into the final norm. If the case is late-start, then the hidden vector goes from the embedding to the first layer and then progresses from there. Naive layer skipping is achieved by turning off the HVC.

2.3.1 Finding best layer-skipping ablations

To get best possible drafting performance per layer skipped, it is likely important to skip the right layers for the model. This can be late-start, gap-jump or early-exit and different placements of those. To test what ablations of skipped layers that do the least amount of damage to the generation quality, a setup is used to test the KL, Top1 deviation and CE to the full model for different skip-ablations.

The setup exists in evaluate_layer_skip_ablations.py. It begins with running next-token prediction with the full model over a set of windows from the dataset, it records the logits produced at every position. Different layer-skip-ablations of the model are then run over the same windows and the next-token predictions are compared to the full model. This produces an average KL divergence compared to the full model and an average top1 score for each ablation. This is then presented in a plot that can show the results of KL, KL per removed layer, or top1 for all ablations.

This setup does not use the HVC when skipping layers. Even though the HVC should be relatively lightweight to train, training one for all possible ablations would require a lot of compute. The skip-ablations are therefore measured with the hidden vector going directly from the last layer before the gap to the entry layer. The idea is that this will still show what skip-ablations that are promising starting points to then improve further with the HVC.

A delimitation for this project is that only a single HVC will be used. This then requires there to be a single contiguous gap, not multiple holes of skipped layers. The ablations tested are mostly such with a contiguous gap, but some non-contiguous periodic ablations are also included to see how they perform in this test where the HVC doesn't need to be added. The ablation families are summarized in Table 1.

Table 1: The different layer skip ablations tested in the project.

Layer ablation masks tested.
The model has L layers indexed $0, ..., L-1$.
Each ablation is represented with the set $S \subseteq \{0, ..., L-1\}$.
All layers in S are skipped, and all other layers are kept.

Keep all	Skip no layers: $S = \emptyset$. This is the baseline that is compared to.
Early exit	Skip after the first k layers: $S = \{k, ..., L-1\}$.
Late start	Skip before the last k layers: $S = \{0, ..., L-k-1\}$.
Internal gap	Skip a contiguous block of internal layers: $S = \{s, ..., s+g-1\}$. The skipped gap has start s and length g. The first and last layer are not skipped.
Periodic	Skipping every 2nd or 3rd layer or keeping only every 2nd or 3rd layers. Duplicate masks are removed.

2.3.2 HVC setup

Internally in the code, the HVC is often called *bridge* due to the inherent mechanism of connecting two distant points. The text might refer to it as HVC or bridge.

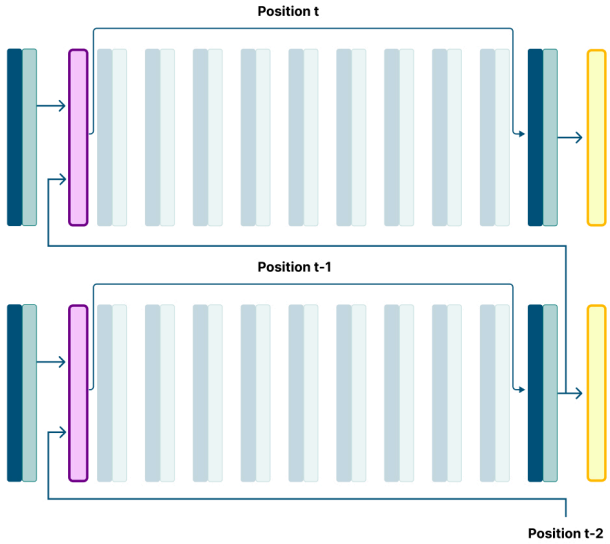


Figure 4: The input to the HVC bridge. It gets a stacked vector of the final hidden vector from token position $t-1$ and the hidden vector from the last layer before the gap at position t.

The bridge is implemented as a linear transformation in PyTorch with residual update and layer normalizations for the two input vectors. As Figure 4 shows, the HVC bridge at position t gets the hidden vector from the last layer before the gap and the final hidden vector from token position $t-1$. In the code prev_reference_hidden is a tensor with previous position hidden vectors. The code to forward the bridge is this:

```
def forward(
    self,
    gap_hidden: torch.Tensor,
    prev_reference_hidden: torch.Tensor,
) -> torch.Tensor:

    bridge_dtype = self.proj.weight.dtype
    x = gap_hidden.to(dtype=bridge_dtype)
    p = prev_reference_hidden.to(dtype=bridge_dtype)

    x_n = self.gap_norm(x)
    p_n = self.prev_norm(p)

    delta = self.proj(torch.cat([x_n, p_n], dim=-1))
    return x + delta
```

2.3.3 Training skipping layers

The file train_skipping_layers.py exposes the function to train the HVC for a certain LM-model and skip ablation. This function uses infrastructure to load datasets, build windows, load bridge module class, setup model and bridge as frozen and non-frozen respectively. It uses the full model in its standard inference mode to act teacher and the inference setup to skip layers as the student. The optimization objectives are to, for a window, minimize the KL divergence and CE compared to the teacher output for the tokens.

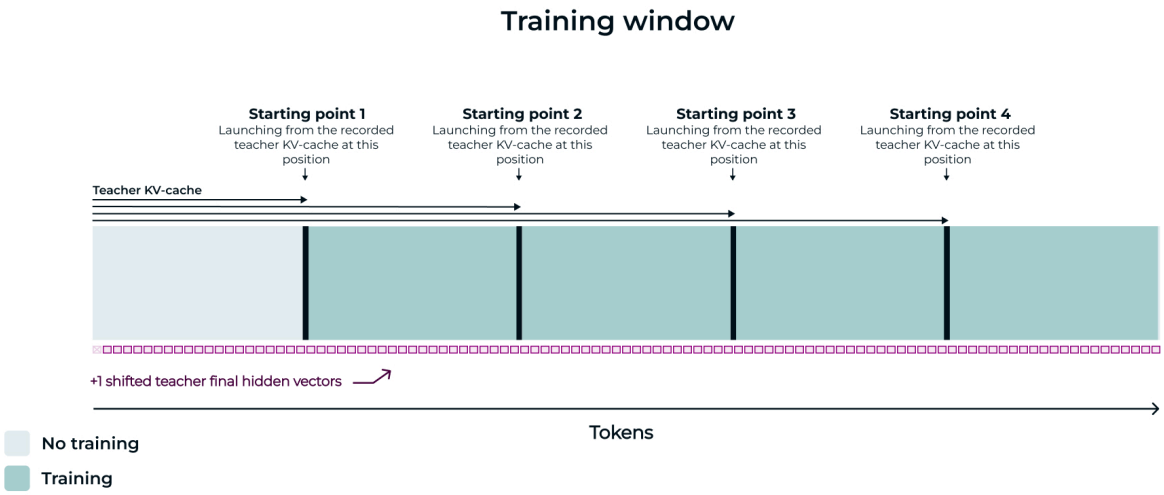


Figure 5: How a training window is structured. It is a window of context_len tokens divided into num_draft_sections sections. The vertical splitting lines indicate starting points where the student continues from how the teacher's KV-cache was at that position. At each position it is getting the teachers $t-1$ final hidden token, this is the shifted pink array, and the t position last layer hidden vector before the gap.

The training aims to produce a good drafter for the full model. When running in self-speculation, the drafter will run from where the verifier last stopped. It will do so by continuing from the KV-cache the verifier produced. The training objective is therefore to "cast" a hidden vector through the gap using the input hidden vectors, but to also do so when starting from the verifier's KV-cache.

To train for this, the teacher runs next-token prediction on the training window and its logits for all positions and the created KV-cache are stored. The window is then conceptually split into multiple sections like Figure 5 shows. The student will do next-token prediction runs on the sections from one starting point to the next. At each boundary, it will start from the KV-cache the teacher has produced at that position. This simulates the objective to start from a verifier prefix and generate from there.

If the intended block size for the self-speculation is 1–5, then dividing the window into sections of that size would make sense. However, that would be a lot of compute to produce so many versions of the teacher KV-cache history and to run so many small drafter trainings. Therefore, a number of sections that balances compute and realism will have to be selected. The parameter to set the number of sections is num_draft_sections. In Figure 5, num_draft_sections = 5.

The loss is calculated on all sections except the first one. Let S be those token positions, and let $p_{t,v}$ and $q_{t,v}$ be the probability assigned to vocabulary token v by the teacher and drafter respectively at position t. The KL part compares the full distributions:

$$L_{\mathrm{KL}} = \frac{1}{|S|} \sum_{t \in S} \sum_{v} p_{t,v} \log \frac{p_{t,v}}{q_{t,v}}.$$

The CE part only uses the teacher's top-1 vocabulary token as the target:

$$v_t^* = \arg\max_v p_{t,v}, \quad L_{\text{top-1 CE}} = -\frac{1}{|S|} \sum_{t \in S} \log q_{t,v_t^*}.$$

Both losses have a weight of 1.0 which gives a total loss:

$$L = L_{\mathrm{KL}} + L_{\text{top-1 CE}}.$$

Every HVC-training produces a run.json file that includes training and loss values for every Nth step. These will be used to then plot training convergence and to do analysis. Examples of such values are:

Table 2: Summary of HVC training metrics

Metric	Description / Logic
kl_verifier_to_drafter	The KL divergence between the verifier's next token distribution and the drafter's next token distribution. Lower values mean that the drafter assigns probability mass more similarly to the verifier over the vocabulary.
top1_drafter_matches_verifier	The fraction of token positions where the drafter and verifier have the same top-1 token. This is especially important for greedy self-speculation, since a drafted token is accepted when it matches the verifier's selected token.
loss_ce_drafter_on_verifier_top1	The CE loss of the drafter using the verifier's top-1 token as the target. This trains the drafter to put high probability on the token that the verifier would select at the same position.
loss_bridge_reentry_mse	The MSE loss between the hidden vector produced by the HVC bridge and the verifier's target hidden vector at the re-entry point after the skipped gap. Lower values mean that the bridge better casts the skipped hidden vector into the geometry expected by the later layers.

2.4 ANNH implementation

This project produces an implementation of FlashHead and measures accuracy, clustering times and speedups.

To evaluate ANNH as a replacement for a dense head, three parts are produced:

- A program to build an ANNH cluster and save that in an index structure that is performant to use.
- An accuracy-evaluator that can load a saved ANNH and measure how often it finds the correct token.
- An adapter to run a model with the ANNH.

2.4.1 ANNH cluster builder

Clusters of token unembedding vectors are built using spherical k-means with a strict capacity constraint. After each iteration of k-means, vectors in overloaded clusters are greedily moved to clusters that still are a good match until capacity constraint is satisfied for all clusters.

The number of clusters is fixed by the num_clusters argument. Before clustering, the builder computes strictly equal cluster capacities whose sum is exactly the vocabulary size. Thus, only values of num_clusters that evenly divide the vocabulary size are accepted.

Algorithm 1 is used to produce the clustering.

Algorithm: Strict Equal-Capacity LM-Head Clustering

Input: LM-head vector table $W \in \mathbb{R}^{V \times H}$, number of clusters C,
number of iterations N, normalization flag, random seed S
Output: cluster-to-token table G, centroids μ

Let V be the vocabulary size and H the hidden size
Require $C \geq 1$, $C \leq V$, and $V \bmod C = 0$
Set the strict cluster capacity $q \leftarrow \frac{V}{C}$
if normalization is enabled **then** set $z_t \leftarrow \frac{W_t}{\|W_t\|_2}$ for every token t
else set $z_t \leftarrow W_t$
Randomly sample C token indices using the seed S
Initialize centroids $\mu_1, ..., \mu_C$ from the vectors of the sampled token indices
if normalization is enabled **then** normalize all centroids
for iteration $r = 1, ..., N$ **do**
    Compute similarity scores $s_{t,c} \leftarrow z_t \cdot \mu_c$ for every token t and cluster c
    Assign each token to its nearest centroid: $a_t \leftarrow \text{argmax}_c\, s_{t,c}$
    Compute current cluster sizes $n_c \leftarrow |\{t : a_t = c\}|$
    Set eviction list $E \leftarrow \emptyset$
    for each overloaded cluster c with $n_c > q$ **do**
        Compute regret $\rho_t \leftarrow s_{t,c} - \max_{c' \neq c} s_{t,c'}$ for each token t assigned to c
        Add the $n_c - q$ lowest-regret tokens from c to E
    end for
    Sort E by increasing regret
    for each token $t \in E$ **do**
        Move t to the non-full cluster c with best score $s_{t,c}$ and update sizes
    end for
    Recompute each centroid as the mean of its assigned token vectors
    if normalization is enabled **then** normalize all centroids
    Compute loss $L \leftarrow 1 - \frac{1}{V} \sum_t \left(z_t \cdot \mu_{\text{assigned for t}}\right)$ and report similarity statistics
end for
Run one final strict equal-capacity assignment using the learned centroids
Build G by sorting token ids by assigned cluster and reshaping to shape $C \times q$
return token map a, cluster table G, centroids μ, and cluster sizes

Algorithm 1: Algorithm to build FlashHead-like ANNH cluster of token vectors.

Algorithm 1 builds an output of this Python structure:

```
class BuiltANNHClusters:
    cluster_to_token_ids: Tensor      # [num_clusters, cluster_size], no padding
    centroids: Tensor                 # [num_clusters, hidden_size]
```

To use the built ANNH clusters during inference, a ANNHModule is constructed from the stored BuiltANNHClusters and the model's existing LM-head unembedding tensors. The module stores the transposed centroids and the cluster-to-token table, but it does not copy or pre-cluster the full LM-head weight table. Instead, the original LM-head weight table is borrowed from the model and candidate token rows are gathered from it during lookup.

The function find_token(...) takes a query final hidden vector and searches only the tokens contained in the top-scoring clusters. Its algorithm is:

Algorithm: ANNH Greedy Token Lookup

Input: query final hidden vector $h_q \in \mathbb{R}^H$, transposed centroids $\mu^T \in \mathbb{R}^{H \times C}$, cluster-to-token table $G \in \mathbb{N}^{C \times q}$, LM-head vector table $W \in \mathbb{R}^{V \times H}$, optional LM-head bias $b \in \mathbb{R}^V$, number of clusters to search K_c

where: C is the number of clusters, q is the cluster size, $V = Cq$ is the vocabulary size, and H is the hidden size

Output: selected token id y

Set $K \leftarrow \min(K_c, C)$
Compute cluster scores $r \leftarrow h_q \cdot \mu^T \in \mathbb{R}^C$
Select the K clusters with largest scores; call their indices $P \in \mathbb{N}^K$
Gather candidate token ids from the selected clusters: $T \leftarrow \text{flatten}(G[P]) \in \mathbb{N}^{Kq}$
Gather the corresponding LM-head rows from the original table: $W_T \leftarrow W[T] \in \mathbb{R}^{Kq \times H}$
Compute candidate token scores $\ell \leftarrow W_T h_q \in \mathbb{R}^{Kq}$
if bias b exists **then** add $b[T] \in \mathbb{R}^{Kq}$ to ℓ
Let $m \leftarrow \max_i \ell_i$ be the highest candidate score
return the smallest token id T_i such that $\ell_i = m$

Algorithm 2: Algorithm for approximate greedy best matching token lookup using built ANNH cluster.

2.5 Inference

The best skip-ablations are used for the body and the ANNH implementation is used for the head. Using those together composes a drafter model for a self-speculative decoding setup.

2.5.1 Self-speculative decoding

Self-speculative decoding is used to measure whether an approximated model act as a useful drafter while the original full model acts as the verifier. In this setup, both the drafter and verifier come from the same base model. The verifier uses the original model, while the drafter uses the same model with selected layers skipped and optionally ANNH.

The implementation uses greedy decoding. This means that both the drafter and verifier always select the token with highest probability. A drafted token is accepted if it is exactly equal to the verifier's greedy token at the same position.

Here is a pseudocode of how the self-speculative decoding works:

Algorithm: Self-Speculative Decoding with an Approximated Drafter

Input: prompt x, maximum generation length T, draft block size K,
full model M, drafter $\widetilde{M}$ formed by skipping a layers and optinally using ANNH, and inserting bridge B
Output: generated sequence y

$y \leftarrow x$
Run verifier M on y
Store verifier KV cache C
Store verifier reference hidden states H
Append the initial verifier bonus token to y
while number of generated tokens $< T$ **do**
    $d \leftarrow$ empty list
    Save verifier cache state $C_0 \leftarrow C$
    for $i = 1, ..., K$ **do**
        Construct previous-reference hidden states from H
        Run drafter $\widetilde{M}$ on the current suffix using C; set $d_i \leftarrow \text{argmax}\, \widetilde{M}(C, H)$
        Append d_i to d
        Temporarily extend C with the drafter cache for the next draft token
    end for
    Restore $C \leftarrow C_0$
    $y_{\text{candidate}} \leftarrow \text{concatenate}(y, d)$
    Run verifier M on the unverified suffix of $y_{\text{candidate}}$ using cache C
    Let $v_1, ..., v_K$ be the verifier greedy predictions aligned with $d_1, ..., d_K$
    $m \leftarrow$ length of the longest prefix such that $d_j = v_j$ for all $j \leq m$
    Append accepted draft tokens $d_1, ..., d_m$ to y
    if $m < K$ **then**
        Append verifier correction token v_{m+1} to y
        Crop verifier KV cache and reference hidden states to the accepted prefix
    else
        Append verifier bonus token predicted after the full draft block to y
        Keep the full verifier KV cache and reference hidden states
    end if
end while
return y

Algorithm 3: The procedure for self-specualtive decoding and KV-cache management.

As shown in Figure 4, the HVC bridge takes the hidden vector from the previous layer but also a hidden vector from the previous position $t - 1$. During speculation when the draft block has

a size of more than 1, the hidden vector at draft step 1 is from the verifier, but from step 2 and forward, it is from the drafter itself.

The real implementation records the generated text, the output token ids, the number of verifier calls, the number of drafted tokens, and the number of accepted draft tokens.

It can optionally also store a token-level trace JSON file for visualization. Each token is marked as either a prompt token, a drafted token or a verifier-produced bonus/correction token. Drafted tokens are additionally marked as accepted or rejected. This makes it possible to inspect where the drafter follows the verifier and where the verifier has to correct the generation.

2.5.1.1 KV-cache

A single KV-cache C is used by both the verifier and the drafter, see Algorithm 3. The drafter will start where the verifier left off and manipulate C in place. Just before a draft block is started, the length of C is stored as C_0. After the drafter has processed a draft block, C will be cropped back to C_0 so that then when the verifier runs, it will start from the prefix of C that is guaranteed to be correct. Then when it verifies the proposed draft block, it will update C with the KV-cache up until the mismatch if there is any or for the entire draft block if it is accepted.

This approach gives that only a single KV-cache needs to be stored instead of having one for the verifier and one for the drafter. The KV-cache is mutated in place with cropping. This avoids having memory spikes that would be created if the rollback position C_0 was a copy. The KV-cache memory pressure is therefore not higher than running the model normally.

2.6 Benchmarking

The benchmark measures total speedup compared to the normal generation baseline and captures internal details. The self-speculative generation is run in these two variants:

1. *Skipped layers*: the drafter uses the HVC-bridge and skipped-layer body, but still uses the dense LM-head.
2. *Skipped layers + ANNH*: the same drafter body is used, but token selection in the drafter uses the ANNH head instead of the dense LM-head.

2.6.1 Prompt sets

The benchmark uses three completion-style prompt sets.

Here are examples from each prompt set:

Table 3: Prompt sets used in the main self-speculation benchmark.

Prompt set	Prompts	Purpose	Example task
concrete-completion-style	120	Concrete tasks with answers that should be clear to produce.	Compute a price, extract an email address, sort values, produce a JSON row etc.
open-ended	100	Natural-language continuations with many valid next-token paths.	Give advice, write an email, tell a short story etc.
python-diverse-completion-style	100	Python code-completion prompts with varied programming tasks.	Validate brackets, parse semantic versions, fix a bug, write tests etc.

All prompts in the three prompt sets are generated by ChatGPT 5.5 Thinking. The main idea with the different prompt sets is to measure if the acceptance rate is higher when the prompt is less open-ended and more concrete. If the prompt is open-ended, there are many possible good continuations from the prompt, and thus possibly less probability that the drafter and the verifier predict the same next tokens.

The reason to generate benchmark prompts instead of using an existing prompt-set is to have full control of the concrete to open-ended dimension, but also that available benchmark prompt-sets might be too difficult for the relatively small models used. If a very complex prompt is used, then the generated answer from a small model like Llama 3.2 1B Instruct might be wrong or primarily a guess. In that situation, the task of the drafter then becomes to predict a guess or an incorrect generation. That would make the evaluation of the drafter performance less clear. By having prompts that are on a controlled difficulty level, the small models can have a chance to generate tokens that aren't pure guesses.

Literal examples from the three prompt sets are shown in Listing 1. The concrete set covers bounded tasks with short, checkable outputs, while the open-ended and Python-diverse sets include prompts where the continuation is naturally longer.

Prompt set	**Example**	**Prompt**
concrete-completion-style	Example 1	Task: Compute the arrival time. A train leaves at 14:35. The trip takes 2 hours and 48 minutes. Return only the arrival time in 24-hour HH:MM format. Answer:
	Example 2	Task: Convert the title to a URL slug. Title: Winter Market Schedule Update Use lowercase words separated by hyphens. Return only the slug. Output:
open-ended	Example 1	What should a new manager do during the first month with a team they just inherited?
	Example 2	Discuss some tradeoffs of using AI tools for writing, studying, and workplace communication.
python-diverse-completion-style	Example 1	# Group a dependency graph into topological layers # deps maps each task to the set of tasks it depends on. # Each returned layer contains tasks whose dependencies are already done. # Sort task names inside each layer. # Raise ValueError if the graph contains a cycle. def topological_layers(deps: dict[str, set[str]]) -> list[list[str]]:
	Example 2	# Summarize invoice rows with discounts and tax # Each row has quantity, unit_price, and discount_percent. # Apply discount per row before tax. # Return the final total rounded to 2 decimals. def invoice_total(rows: list[dict[str, float]], tax_rate: float) -> float:

Listing 1: Example prompts from the three prompt sets used in the main self-speculation benchmark.

2.6.2 Benchmark phases

Each benchmark variant has three phases:

1. *Warmup phase*: An initial 5 prompts are processed to warmup the inference to exclude any errors in the measurements because of possible cold start effects. These are not included in the benchmark values.
2. *Profile phase*: After running the warmup, 15 prompts are run where internal measurements are turned on. This will measure times for the drafter and its internal parts, the verifier, and for the normal model as comparison. This will affect the total inference time due to GPU syncs and possible overhead, so the total times are not used to calculate inference speedup in the benchmark.
3. *Benchmark phase*: Benchmarking all prompts in the prompt set, from the first to the last in order. Internal measurements are turned off to get unaffected performance. Total times are measured and used to calculate and report speedups.

These three phases are run first with the drafter only using skipped layers, and then again with the drafter using skipped layers and ANNH. So Warmup → Profile → Benchmark, then enabling ANNH, and doing Warmup → Profile → Benchmark again.

2.6.3 Timing measurements

All reported speedups use the benchmarking phase. The speedup timing starts after the prompt has been tokenized and ends right before decoding the output token ids to text. Normal generation is measured around a greedy `model.generate(...)` with KV-cache enabled.

The self-speculative timing includes everything that the setup performs. This includes initial verifier pass of the prompt, first verifier produced bonus token, all drafter blocks and their internal calls, all following verifier calls, token acceptance logic, KV-cache handling, and the systems Python logic executions such as mounting and demounting the draft inference components to the model.

Peak PyTorch CUDA memory usage is measured for both drafter versions, skipping layers and skipping layers + ANNH, and normal inference. The CUDA peak allocation statistics are reset before each prompt and the average of the per prompt peak memory is reported for the three versions. This memory usage includes the loaded model, loaded attached components such as HVC-bridge, needed PyTorch inference components, and the KV-cache.

2.6.4 Plot metrics

The benchmark plots include both setup information and measured quantities. The setup and runtime fields are defined in Table 4, while the measured result fields are defined in Table 5.

Table 4: Setup and runtime fields shown in the benchmark plot footers.

Plot field	Format	Meaning
Prompt set	name	The prompt set used for the benchmark run.
Gap	(a, b)	The layer-skipping shape, where a is the number of kept layers before the skipped gap and b is the number of kept layers after it.
Block size	integer	The number of draft tokens proposed before each verifier call, denoted γ in Equation 1.
Layers	integer	The total number of transformer layers in the base model.
LM vocab	integer	The vocabulary size used by the model's LM-head.
LM params	e.g. 1.2B	The total number of parameters in the loaded language model.
Head params	e.g. 262M	The number of parameters in the output LM-head.
Head portion	percent	Head params divided by LM params.
GPU	name	The GPU used for the benchmark run.
Backend	name	The attention implementation used by the loaded model.
Model dtype	dtype	The datatype used for the base model parameters during inference.
Bridge dtype	dtype	The datatype used for the loaded HVC-bridge.
Speed internals	yes/no	Whether internal body/head/verifier timers were enabled during the speed phase. For reported speedups this is disabled.
Profile prompts	integer	Number of prompts run in the profile phase.
Warmup prompts	integer	Number of prompts run in the warmup phase before measured runs.
Measured prompts	integer	Number of prompts included in the speed-phase metrics.

Table 5: Measured result and profile fields shown in the benchmark plot footers.

Plot field	Format	Meaning
Peak mem normal	GiB/MiB	Mean CUDA peak allocated memory over normal generation runs.
Mean speedup	1.23×	Overall per-token speedup for the variant. It is computed as self-speculative tokens per second divided by normal tokens per second after summing seconds and generated tokens over measured prompts.
Acceptance rate	percent	Accepted draft tokens divided by drafted tokens.
Exact match	percent	Fraction of measured prompts where the decoded self-speculative output exactly matches the decoded normal-generation output.
Peak mem self	GiB/MiB	Mean CUDA peak allocated memory over self-speculative runs for the variant.
Tokens gen	s X / n Y	Total generated tokens in the speed phase, where X is the self-speculative token count (s) and Y is the normal-generation token count (n).
Drafter split	B x% / H y% / O z%	Profile-phase split of drafter time, where x is body time (B), y is head time (H), and z is remaining overhead (O).
Verifier/normal	1.23×	Mean verifier-call time divided by normal time per generated token. The initial prompt prefill verifier call is excluded.
Drafter/normal	percent	Mean drafter-token time divided by normal time per generated token.
ANNH accuracy	percent	Acceptance rate with ANNH divided by acceptance rate without ANNH, used as a proxy for how well the ANNH preserves the dense-head drafter behavior.
ANNH head speedup	1.23×	Dense LM-head time divided by ANNH head lookup time in the profile phase.
ANNH index	C clusters; top-$k = K$	ANNH search setup, where C is the number of clusters in the loaded index and K is the number of top clusters searched per token.

3

Results

3.1 Skip ablations

To know what layers are best to skip, different skip-ablations were evaluated. The plots show different skip ablations visually: ■■■■■■■□□■■■■■■■ means that two middle layers are skipped and all other layers are activated, so a gap-jump. The format ■■■■■■■■■■■■■□□□ means that the last three layers are skipped, corresponding to an early-exit. This gives an intuitive visual overview of how skip patterns compare.

The first metric plotted for each model is KL divergence from the full model distribution to the skipped model distribution, divided by the number of skipped layers. Lower values are better. Normalizing by skipped layers makes it easier to compare ablations that remove different amounts of compute. The second metric is top-1 agreement with the full model, where higher values are better. The dataset used to measure the ablations is codelion/fineweb-edu-1B.

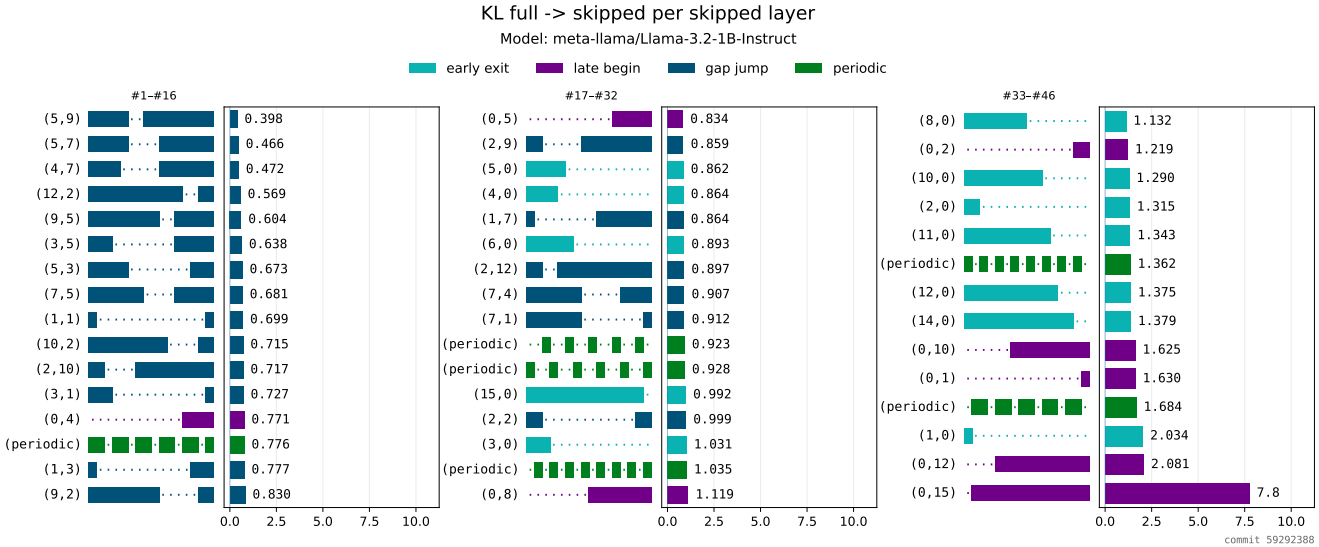


Figure 6: KL divergence per skipped layer for skip ablations of meta-llama/Llama-3.2-1B-Instruct.

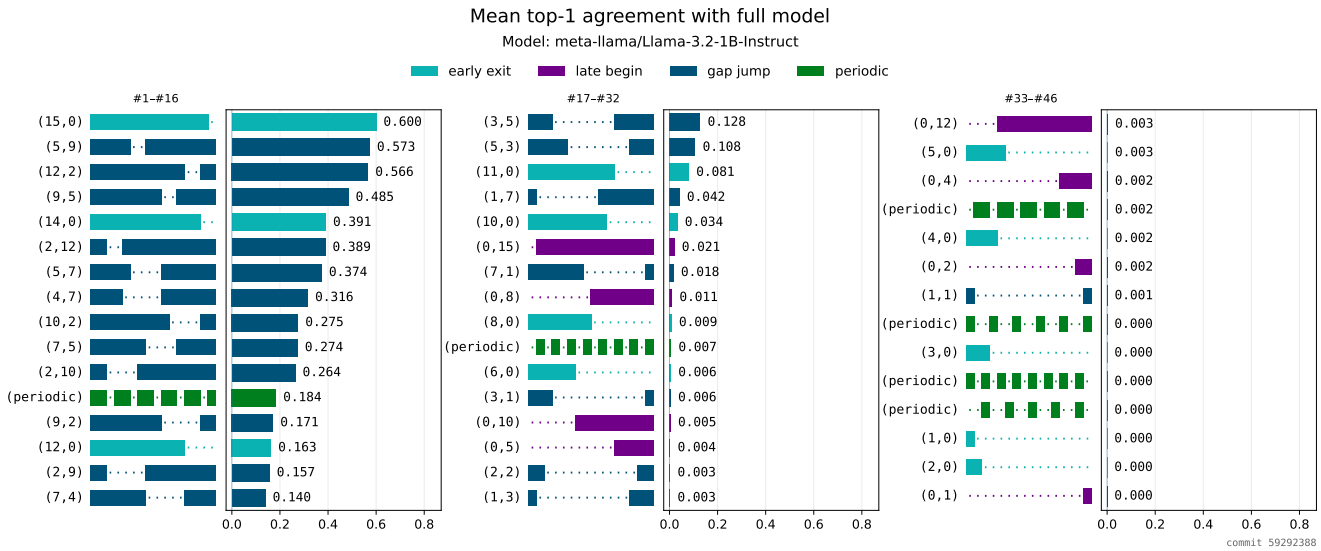


Figure 7: Top-1 agreement with the full model for skip ablations of meta-llama/Llama-3.2-1B-Instruct.

The skip ablations for Llama 3.2 1B Instruct show that per layer skipped, an internal gap seems to hurt the generation performance less. In the top-1 agreement in Figure 7, an early-exit version does produce the highest score, but it is also skipping fewer layers than any of the gap-jump ablations. The worst layer to skip seems to be the first layer. Periodic ablations seem to be able to perform okay but it heavily depends on what specific layers that were skipped. One periodic ablation is among the better ablations while another one is among the worst, even though they seemingly use the same strategy.

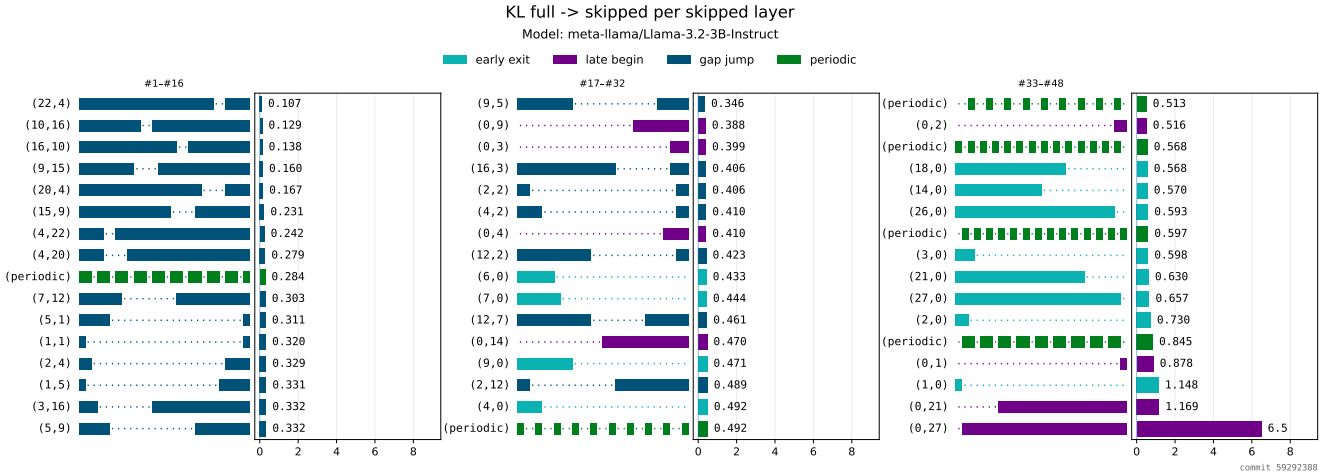


Figure 8: KL divergence per skipped layer for skip ablations of meta-llama/Llama-3.2-3B-Instruct.

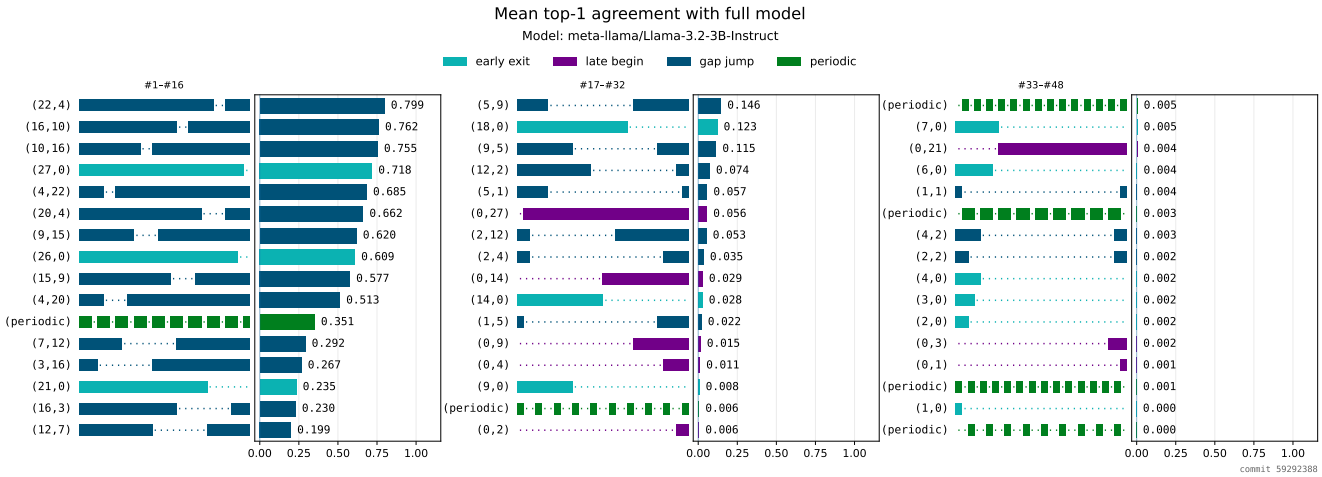


Figure 9: Top-1 agreement with the full model for skip ablations of meta-llama/Llama-3.2-3B-Instruct.

The same approximate pattern can be seen for Llama 3.2 3B Instruct. Minimal per layer degradation seems to be dominated by internal gaps. Periodic gaps again seem to produce very different results depending on exactly what holes the periodic pattern produced. Per layer skipped, neither early-exit or late-start seem competitive with an internal contiguous gap. Similar to Llama 3.2 1B Instruct, the generation quality degrades very quickly when layers are skipped. It does not look possible to use skipped layers to gain a speedup in self-speculation without a HVC-bridge.

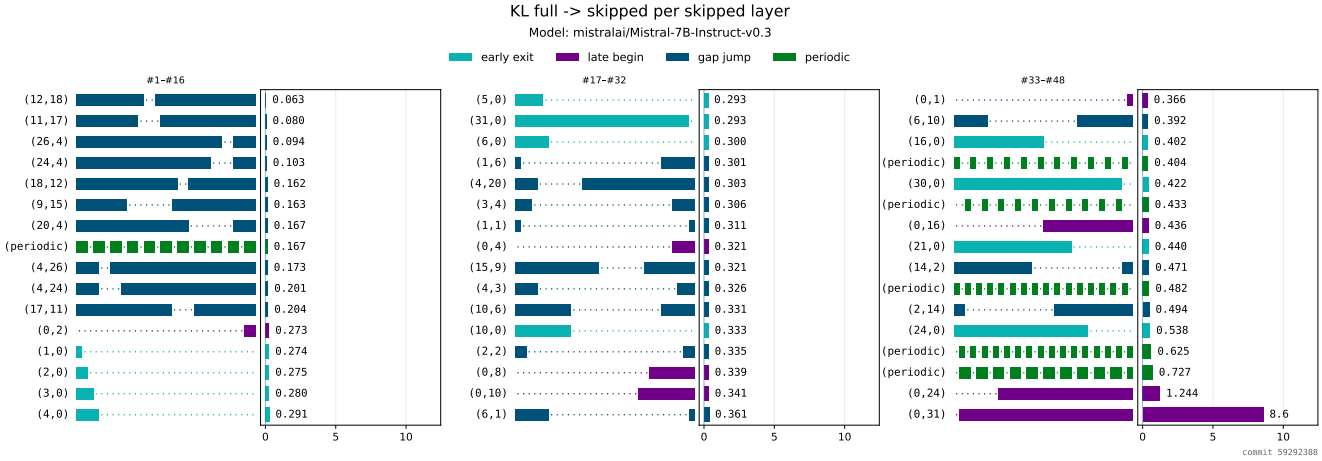


Figure 10: KL divergence per skipped layer for skip ablations of mistralai/Mistral-7B-Instruct-v0.3.

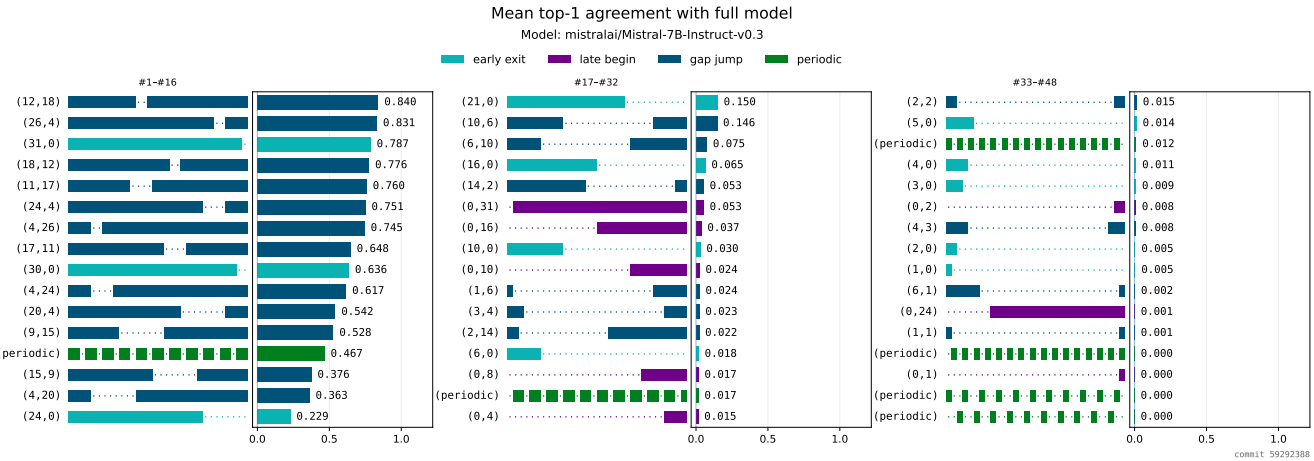


Figure 11: Top-1 agreement with the full model for skip ablations of mistralai/Mistral-7B-Instruct-v0.3.

Mistral 7B Instruct continues the pattern but with slightly more optimistic results for early-exit and late-start. An observation is that for this model, a large number of skipped layers with early-exit and late-start look more promising than large gaps with gap-jump.

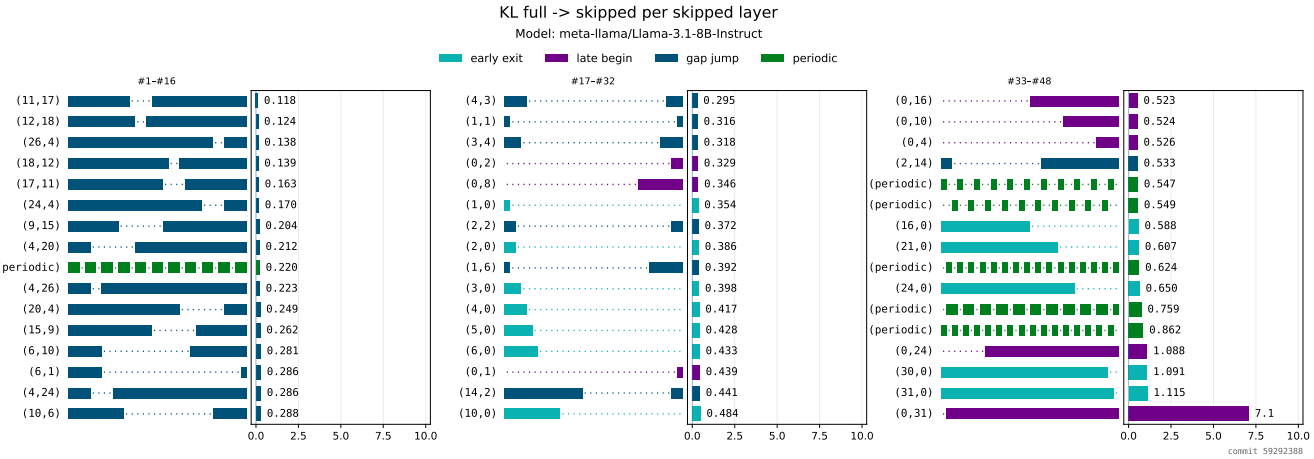


Figure 12: KL divergence per skipped layer for skip ablations of meta-llama/Llama-3.1-8B-Instruct.

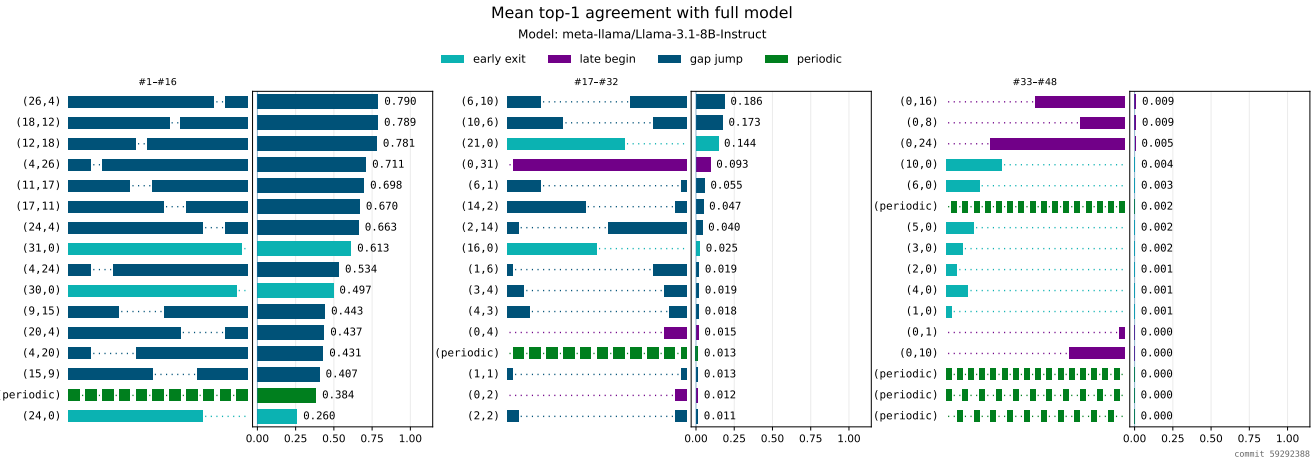


Figure 13: Top-1 agreement with the full model for skip ablations of meta-llama/Llama-3.1-8B-Instruct.

Llama 3.1 8B Instruct shows approximately the same pattern as Llama 3.2 1B and 3B Instruct.

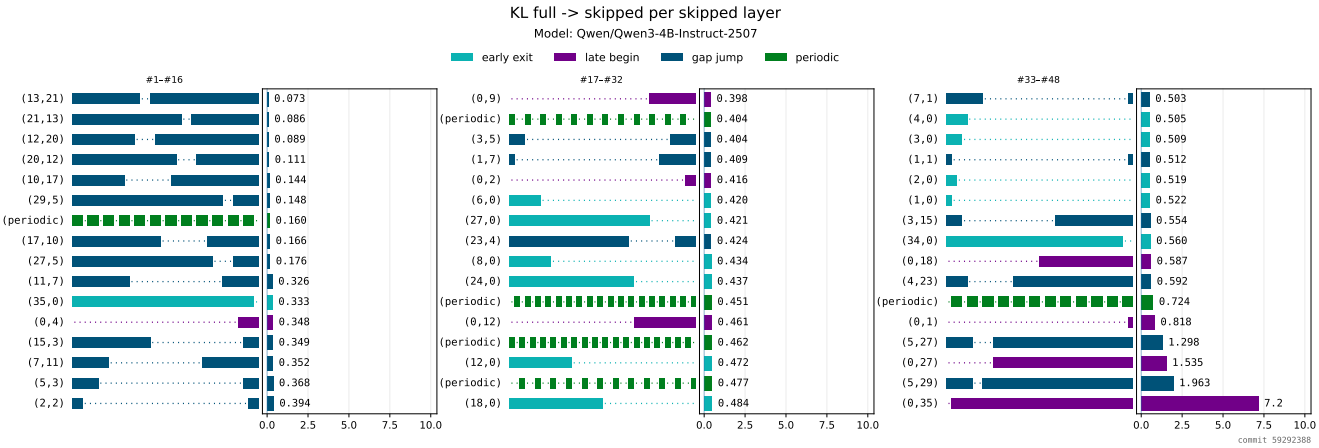


Figure 14: KL divergence per skipped layer for skip ablations of Qwen/Qwen3-4B-Instruct-2507.

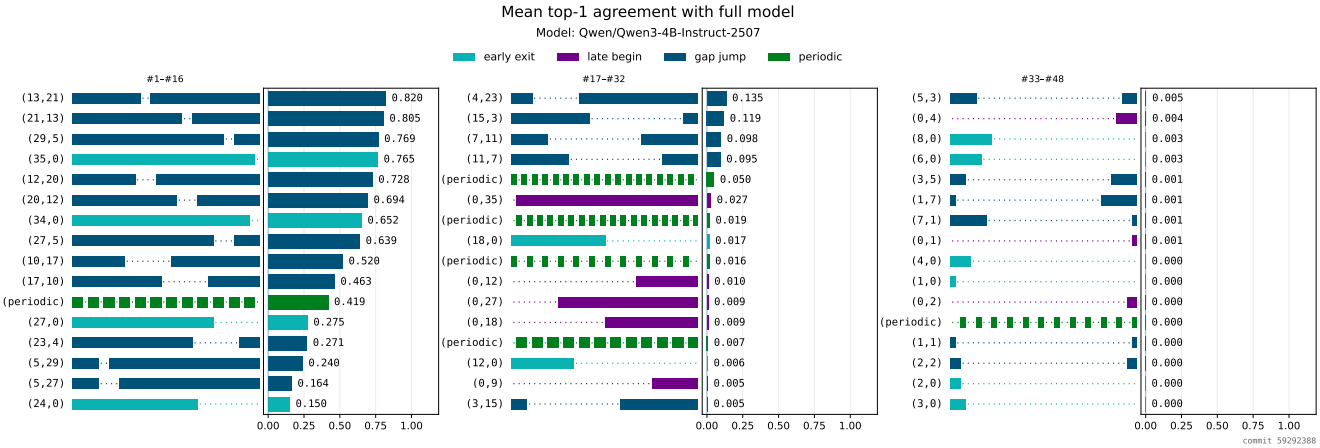


Figure 15: Top-1 agreement with the full model for skip ablations of Qwen/Qwen3-4B-Instruct-2507.

Qwen 3 4B Instruct also seems to produce the same pattern where a gap-jump is better per skipped layer than the other variants.

Taken together, the skip-ablation results in Figures 6 to 15 show a clear pattern that skipping a contiguous gap in the middle tends to do less damage to generation quality than early-exit or late-start ablations. The best-ranked ablations for all five models are in general internal gaps, while periodic patterns with multiple holes do not show an obvious advantage. This makes the gap-jump setup the most promising general solution to use when training the HVC-bridge and benchmarking speedups.

3.2 ANNH cluster building and evaluation

Here are results regarding building a cluster presented. The most relevant examples are for models that have a large head-to-body ratio, because those are where using ANNH instead of the full dense LM-head can produce the biggest speedup.

3.2.1 Llama 3.2 1B Instruct

3.2.1.1 5344 clusters

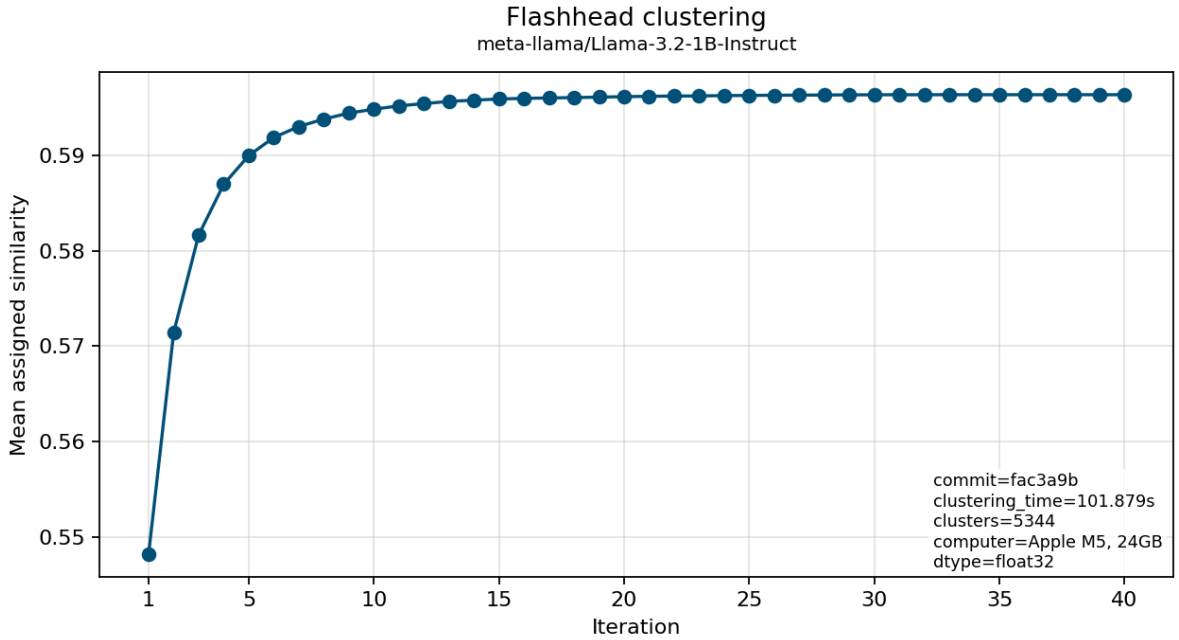


Figure 16: Mean assigned cosine similarity during ANNH clustering of meta-llama/Llama-3.2-1B-Instruct LM-head vectors over clustering iterations.

Figure 16 shows the process of clustering 5344 clusters for the 128,256 token vectors of Llama-3.2-1B-Instruct. It reaches a plateau after around 15 iterations. The total 40 iterations took around 102 seconds on an Apple M5 chip. The same type of cluster building curve was observed for all models. The clustering produces these quality metrics:

Table 6: Cluster quality metrics for the 5344-cluster ANNH index built from Llama 3.2 1B Instruct LM-head vectors.

Cluster Quality Metrics

Llama 3.2 1B Instruct, 5344 clusters

Metric	Value
Number of clusters	5344
Mean assigned similarity	0.596406
Fraction assigned to nearest centroid	0.947184
Clustering time	102 seconds
Clustering iterations	40
Minimum cluster size	24
Maximum cluster size	24

The fraction assigned to the nearest centroid being 0.947 signals that most of the vectors belong to a cluster whose centroid is the nearest centroid. The minimum and maximum cluster size also show that all clusters have the same size and the correct size for 5344 clusters with 128,256 unembedding token vectors.

By running evaluation on the cluster with different number of top-k probings these are the results:

Table 7: Match-rate sweep over top_k_clusters.

Top-1 and Top-3 Match Rate by Probed Clusters

Llama 3.2 1B Instruct

commit=2f4373e · windows=30 · window_length=200 · total_clusters=5344

top k clusters	% of all clusters probed	top1 match rate	top3 match rate
1	0.02%	0.345394	0.552429
10	0.19%	0.687772	0.873367
20	0.37%	0.812228	0.942211
50	0.94%	0.915578	0.983417
100	1.87%	0.956449	0.996147
300	5.61%	0.991457	0.999497
500	9.36%	0.996315	0.999832
1000	18.71%	0.998325	1.000000

Table 7 shows that the probability of getting the true top-1 converges to 1 when the number of top-k probed clusters increases. To find the true top-1 token more than 99% of the time a top-k of around 300 is needed which is 5.61% of all clusters probed. Top-3 containment means that given the top selected token from the cluster, is that within top-3 what the full LM-head would have selected.

The following subsections report the data for 2.6k, 8k, and 16k clusters.

3.2.1.2 2672 clusters

Table 8: Cluster quality metrics for the 2672-cluster ANNH index built from Llama 3.2 1B Instruct LM-head vectors.

Cluster Quality Metrics

Llama 3.2 1B Instruct, 2672 clusters

Metric	Value
Number of clusters	2672
Mean assigned similarity	0.553953
Fraction assigned to nearest centroid	0.936674
Clustering time	57 seconds
Clustering iterations	40
Minimum cluster size	48
Maximum cluster size	48

Table 9: Match-rate sweep over top_k_clusters.

Top-1 and Top-3 Match Rate by Probed Clusters

Llama 3.2 1B Instruct

commit=dc68e2f · windows=30 · window_length=200 · total_clusters=2672

top k clusters	% of all clusters probed	top1 match rate	top3 match rate
1	0.04%	0.327471	0.557119
10	0.37%	0.703685	0.884255
20	0.75%	0.818258	0.948911
50	1.87%	0.913400	0.986935
100	3.74%	0.955946	0.996817
300	11.23%	0.990620	0.999832
500	18.71%	0.995477	1.000000
1000	37.43%	0.998492	1.000000

3.2.1.3 8016 clusters

Table 10: Cluster quality metrics for the 8016-cluster ANNH index built from Llama 3.2 1B Instruct LM-head vectors.

Cluster Quality Metrics

Llama 3.2 1B Instruct, 8016 clusters

Metric	Value
Number of clusters	8016
Mean assigned similarity	0.621290
Fraction assigned to nearest centroid	0.944938
Clustering time	153 seconds
Clustering iterations	40
Minimum cluster size	16
Maximum cluster size	16

Table 11: Match-rate sweep over top_k_clusters.

Top-1 and Top-3 Match Rate by Probed Clusters

Llama 3.2 1B Instruct

commit=e83192c · windows=30 · window_length=200 · total_clusters=8016

top k clusters	% of all clusters probed	top1 match rate	top3 match rate
1	0.01%	0.349916	0.549581
10	0.12%	0.736181	0.889615
20	0.25%	0.851089	0.955276
50	0.62%	0.938693	0.987102
100	1.25%	0.971859	0.996985
300	3.74%	0.992462	0.999665
500	6.24%	0.995477	0.999832
1000	12.48%	0.998157	1.000000

3.2.1.4 16032 clusters

Table 12: Cluster quality metrics for the 16032-cluster ANNH index built from Llama 3.2 1B Instruct LM-head vectors.

Cluster Quality Metrics

Llama 3.2 1B Instruct, 16032 clusters

Metric	Value
Number of clusters	16032
Mean assigned similarity	0.668838
Fraction assigned to nearest centroid	0.936627
Clustering time	309 seconds
Clustering iterations	40
Minimum cluster size	8
Maximum cluster size	8

Table 13: Match-rate sweep over top_k_clusters.

Top-1 and Top-3 Match Rate by Probed Clusters

Llama 3.2 1B Instruct

commit=9551a4d · windows=30 · window_length=200 · total_clusters=16032

top k clusters	% of all clusters probed	top1 match rate	top3 match rate
1	0.01%	0.312060	0.531658
10	0.06%	0.758794	0.904188
20	0.12%	0.861642	0.957789
50	0.31%	0.943886	0.990787
100	0.62%	0.975712	0.997152
300	1.87%	0.992965	1.000000
500	3.12%	0.995812	1.000000
1000	6.24%	0.998325	1.000000

3.2.2 Llama 3.2 3B Instruct

3.2.2.1 8016 clusters

Table 14: Cluster quality metrics for the 8016-cluster ANNH index built from Llama 3.2 3B Instruct LM-head vectors.

Cluster Quality Metrics

Llama 3.2 3B Instruct, 8016 clusters

Metric	Value
Number of clusters	8016
Mean assigned similarity	0.621848
Fraction assigned to nearest centroid	0.943270
Clustering time	230 seconds
Clustering iterations	40
Minimum cluster size	16
Maximum cluster size	16

Table 15: Match-rate sweep over top_k_clusters.

Top-1 and Top-3 Match Rate by Probed Clusters

Llama 3.2 3B Instruct

commit=1673707 · windows=30 · window_length=200 · total_clusters=8016

top k clusters	% of all clusters probed	top1 match rate	top3 match rate
1	0.01%	0.300670	0.479899
10	0.12%	0.805695	0.932831
20	0.25%	0.910385	0.981240
50	0.62%	0.965494	0.996147
100	1.25%	0.983920	0.997990
300	3.74%	0.994472	1.000000
500	6.24%	0.997152	1.000000
1000	12.48%	0.999162	1.000000

3.2.2.2 16032 clusters

Table 16: Cluster quality metrics for the 16032-cluster ANNH index built from Llama 3.2 3B Instruct LM-head vectors.

Cluster Quality Metrics

Llama 3.2 3B Instruct, 16032 clusters

Metric	Value
Number of clusters	16032
Mean assigned similarity	0.669897
Fraction assigned to nearest centroid	0.935980
Clustering time	656 seconds
Clustering iterations	40
Minimum cluster size	8
Maximum cluster size	8

Table 17: Match-rate sweep over top_k_clusters.

Top-1 and Top-3 Match Rate by Probed Clusters

Llama 3.2 3B Instruct

commit=7bce16c · windows=30 · window_length=200 · total_clusters=16032

top k clusters	% of all clusters probed	top1 match rate	top3 match rate
1	0.01%	0.342881	0.572027
10	0.06%	0.819263	0.940369
20	0.12%	0.918593	0.981910
50	0.31%	0.970854	0.997990
100	0.62%	0.987270	1.000000
300	1.87%	0.996482	1.000000
500	3.12%	0.998660	1.000000
1000	6.24%	0.999162	1.000000

The results show that using more clusters produces a higher top-1 accuracy for a given top-k. This also means that more clusters mean a smaller fraction of the total vocabulary unembedding vectors to gather to reach a threshold accuracy. To balance still having a routing matrix that is significantly cheaper than the full LM-head, the cluster size of around 8k will be used for all models (exact number depending on divisibility) for the self-speculation benchmarks. The exception will be for Mistral 7B since that has a vocabulary of 32,768 tokens. A cluster target of 8k will be too big to make sense, so a 4096 will be used for that one.

3.3 Training HVC-bridge

Here are the results for the HVC-bridge training. It shows training for the gaps (1,1) and (2,2) since those are the gaps that have the best possibility for speedup. See the discussion for why smaller gaps were chosen to not be included.

3.3.1 Training (1,1) gap

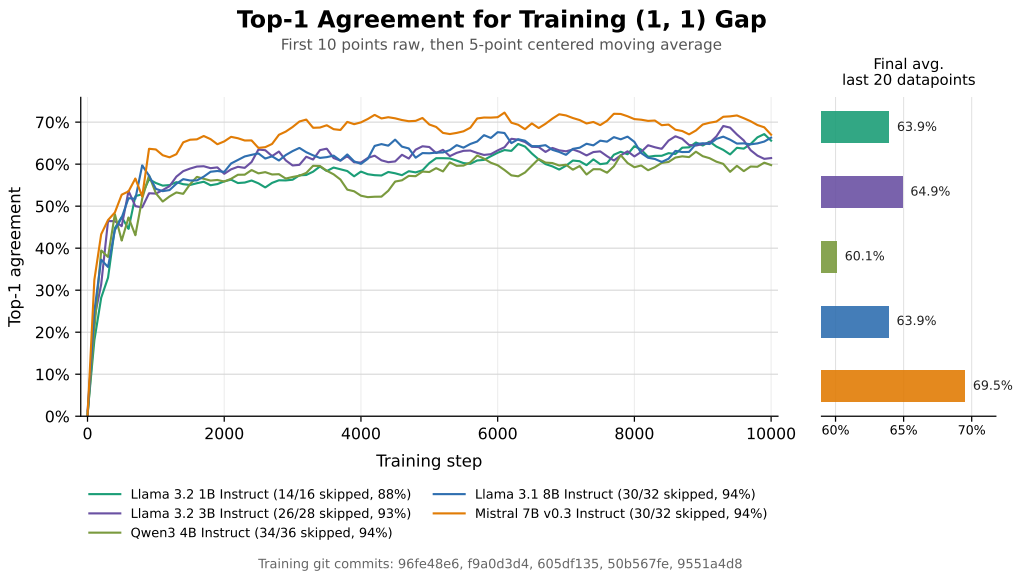


Figure 17: Top-1 agreement between drafter and verifier during training for the $(1, 1)$ gap setting. Curves show the first 10 points raw, followed by a 5-point centered moving average. The right panel reports the final average over the last 20 datapoints.

Top-1 starts at around 0% and reaches between 60–70% with the training. For a large gap like this, the HVC-bridge seems able to a significant extent compensate for the skipped layers. The Mistral 7B Instruct gets the highest top-1 result of 69.5% and the Qwen 3 4B Instruct gets the lowest top-1 result of 60.1%. The training seems to plateau after around 4,000 of 10,000 iterations.

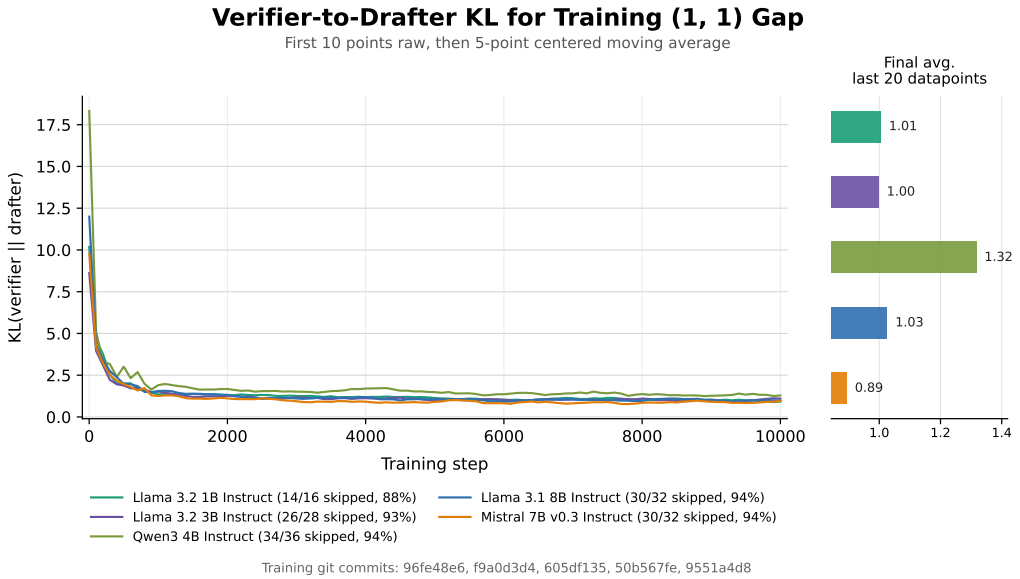


Figure 18: KL divergence from verifier to drafter during training for the $(1, 1)$ gap setting. Curves show the first 10 points raw, followed by a 5-point centered moving average. The right panel reports the final average over the last 20 datapoints.

From the same training as Figure 17, the verifier-to-drafter KL also shows a quick convergence in the first iteration and then incremental improvement over the later iterations. The training results have the same order here as from the top-1 metric perspective. Mistral 7B Instruct has the best KL at 0.89 and Qwen 3 4B Instruct has the highest at 1.32. The KL also seems to plateau around iteration 4000 to 6000.

3.3.2 Training (2,2) gap

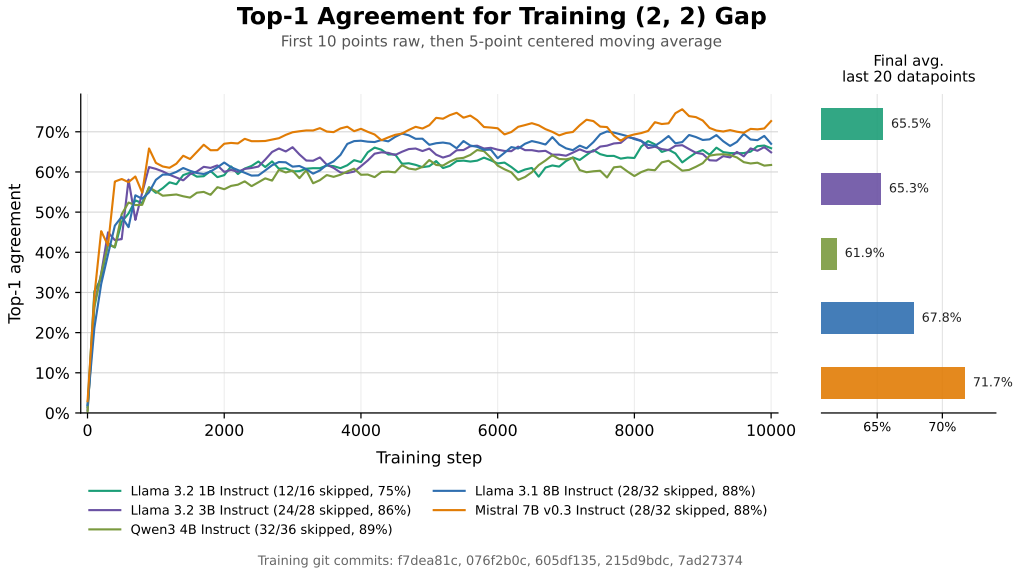


Figure 19: Top-1 agreement between drafter and verifier during training for the $(2, 2)$ gap setting. Curves show the first 10 points raw, followed by a 5-point centered moving average. The right panel reports the final average over the last 20 datapoints.

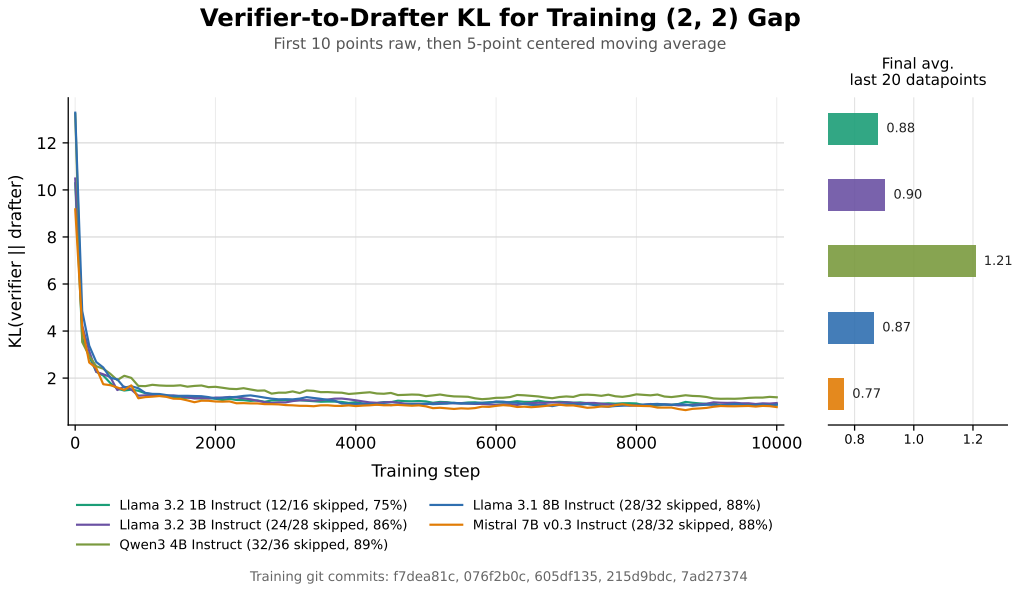


Figure 20: KL divergence from verifier to drafter during training for the $(2, 2)$ gap setting. Curves show the first 10 points raw, followed by a 5-point centered moving average. The right panel reports the final average over the last 20 datapoints.

A gap of (2,2) produces better training metrics for both KL and top-1 for all models than (1,1). The results are however not significantly better. For the $(1, 1)$ gap, the final top-1 agreement is between 60.1% and 69.5%, and the final verifier-to-drafter KL is between 0.89 and 1.32. For the $(2, 2)$ gap, final top-1 agreement is between 61.9% and 71.7%, and final KL is between 0.77 and 1.21.

3.4 Measuring speedups and memory usage

The real world speedup is benchmarked with the produced implementation. Speedup per generated token is as the main measurement but detailed numbers of measured memory usage, acceptance rate, verifier cost, LM-head compute split, ANNH accuracy, and ANNH vs full LM-head speedup is also included.

The implementation runs self-speculation for all the prompts in the prompt sets and the inference speedup for each prompt is stored. The data is presented as a histogram with a fitted normal curve. The magnitude of each bin in the histogram represents how many prompts that produced a speedup in that range.

The self-speculation is first run with the drafter skipping layers but using the normal LM-head, this is represented in blue. Then it is run with the drafter skipping layers and using ANNH instead of the full LM-head, this is represented in the color magenta.

3.4.1 Gap (1,1), block size 2, bfloat16

The benchmark uses the same five models as the skip-ablation and HVC-training sections and the prompt sets described in Table 3. Unless stated otherwise, the runs use bfloat16 model execution, bfloat16 HVC bridge execution, ANNH top-k 100, both variants, 5 warmup prompts, 15 profiled prompts, and all available prompts in each prompt set. These results are with the gap $(1, 1)$, which means that all layers are skipped except the first and the last one. No internal timing is used when measuring speedup to avoid synchronization that would affect performance.

3.4.1.1 Concrete prompt set

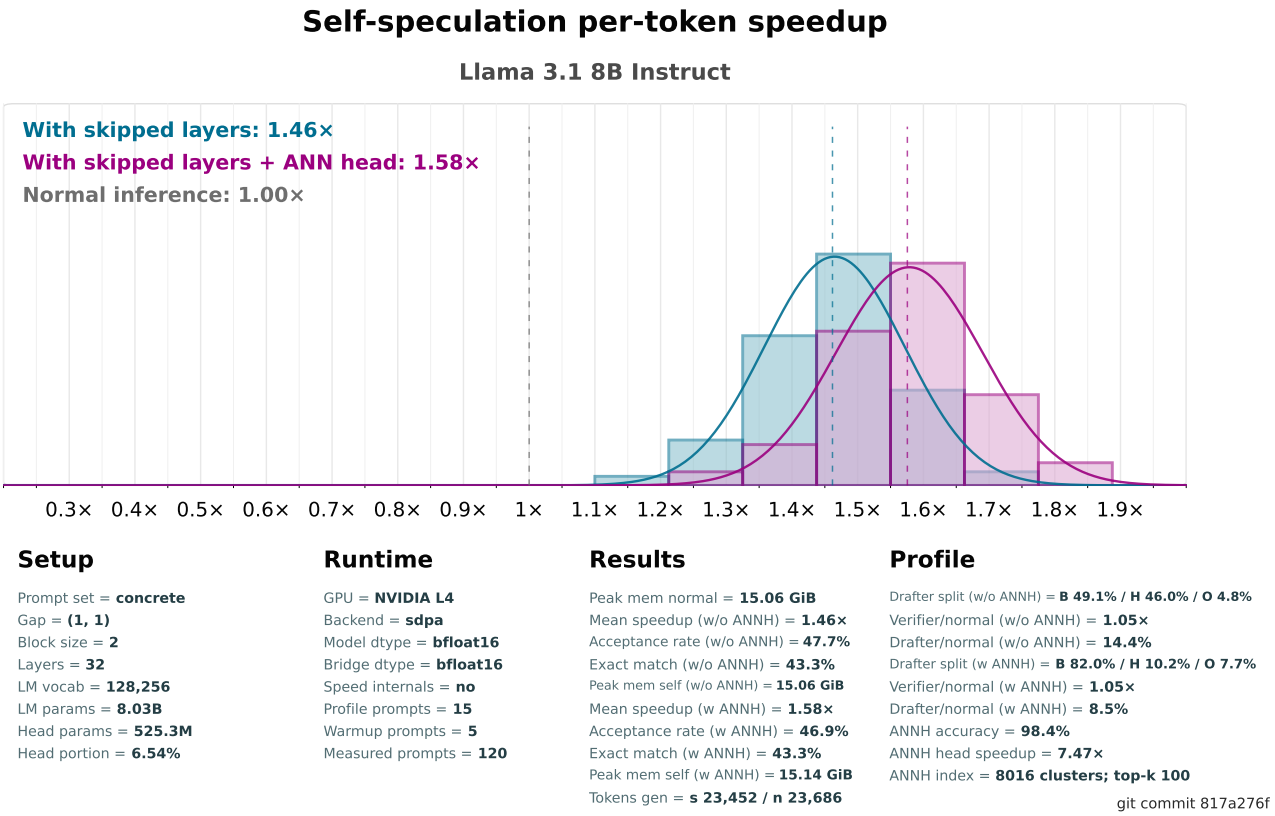


Figure 21: Per-token self-speculation speedup for Llama 3.1 8B Instruct on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt.

From Figure 21 a speedup of 1.46× with skipped layers and a speedup of 1.58× with skipped layers and ANNH can be seen. The peak memory usage remains approximately the same as normal inference for skipped layers and for skipped layers + ANNH.

The profile measurements show that the drafter head becomes 7.47× faster when replacing the dense LM-head with ANNH. This makes the acceptance rate go from 47.7% without ANNH to 46.9% with it. The fraction of when the outputs exactly match normal generation is 43.3% both with and without ANNH. This does not mean that the self-speculation is incorrect or that it is approximate. See the discussion for why this happens even without approximation.

Before the measured benchmarks, 5 warmup runs and 15 profile runs were performed. The used GPU was a NVIDIA L4. The result shows that a verifier call costs 1.05× of a normal token generation call. The block size is 2, which means that this follows the expectation that verifying two tokens and generating a bonus token is significantly cheaper than generation. The result shows that speeding up both the head and the body does increase total speedup which follows Amdahl's law reasoning. The profile shows that the drafter costs about 14.4% of normal generation with skipped layers, and about 8.5% with skipped layers and ANNH.

Using Equation 1 with the observed values $v = 1.05$, $\gamma = 2$, $a = 47.7\%$, and $d = 14.4\%$, the predicted speedup is

$$S = \frac{1 + 2 \cdot 0.477}{1.05 + 2 \cdot 0.144} = \frac{1.954}{1.338} \approx 1.460 \times,$$

which aligns exactly with the measured 1.46×. For the version with ANNH, setting $d = 8.5\%$ and $a = 46.9\%$ gives $S \approx 1.59\times$, again consistent with the measured 1.58×.

The drafter split shows that without ANNH, the share is 49.1% body, 46.0% head and 4.8% body. With ANNH this changes to 82.0% body, 10.2% head and 7.7% overhead. This shows that there is some overhead in the system, such as reconfiguration for the model between acting as a verifier and a drafter, and other costs that come with the built self-speculative system. However, if the overhead is 7.7% of the drafter and the drafter is 8.5% of the normal model, then the total overhead is quite small in absolute time.

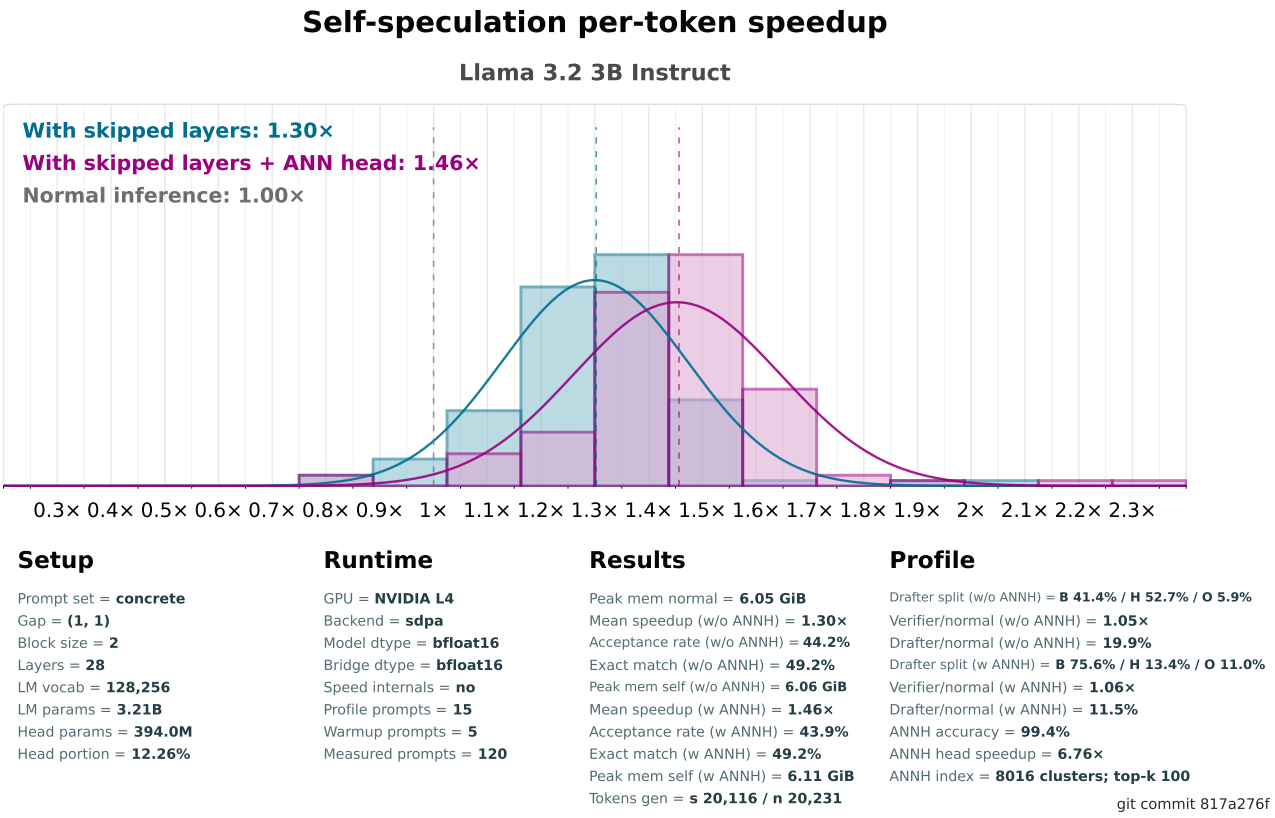


Figure 22: Per-token self-speculation speedup for Llama 3.2 3B Instruct on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt.

Figure 22 shows the same pattern of speedup as Figure 21. The speedups are here smaller, 1.30× and 1.46× respectively. This illustrates the hypothesis of overhead for easy tokens. When the model is smaller, there is less overhead for easy tokens resulting in less gain. The figure shows that the memory usage is approximately the same for all three versions.

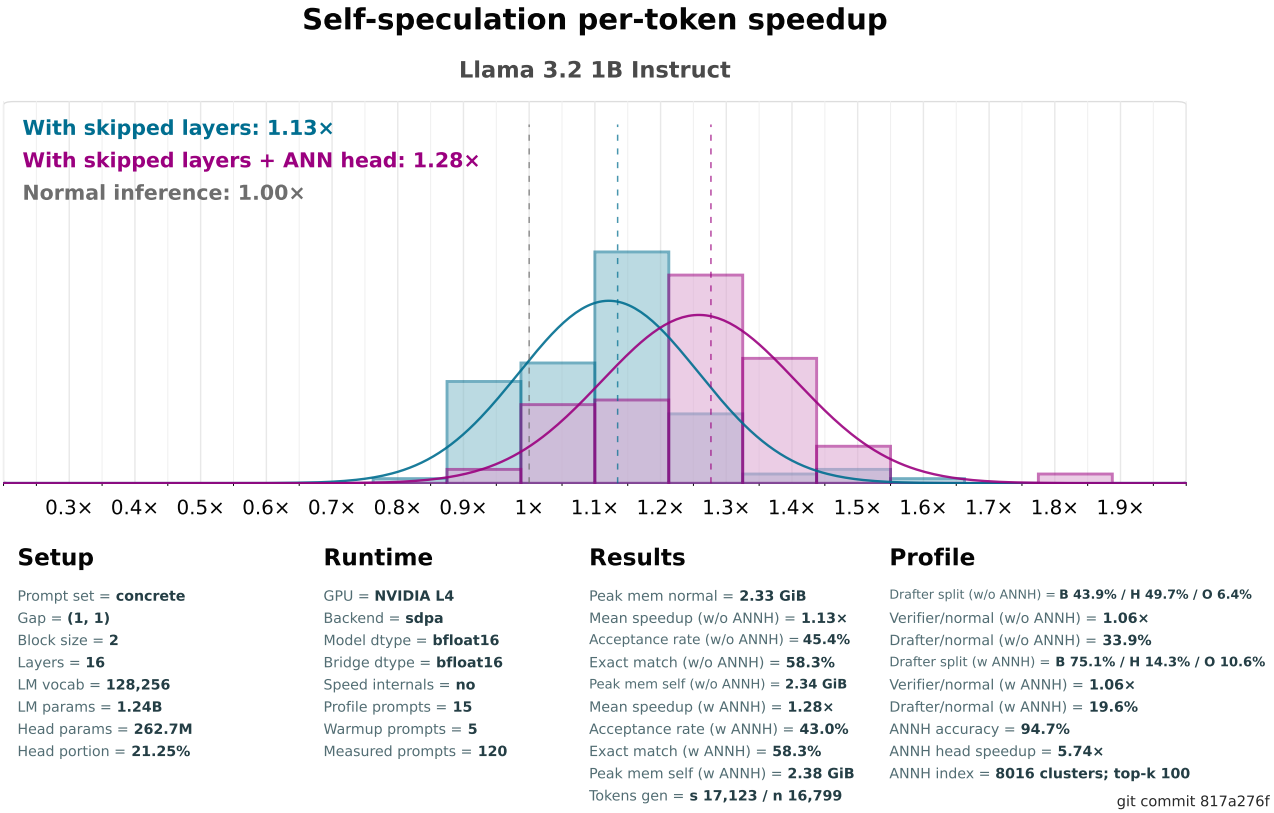


Figure 23: Per-token self-speculation speedup for Llama 3.2 1B Instruct on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt.

Figure 23 shows that the Llama 3.2 1B Instruct also gets speedups of 1.13× and 1.28×. The speedups are here smaller and again following the idea of overhead. The self-speculation runs use approximately the same amount of memory as normal inference.

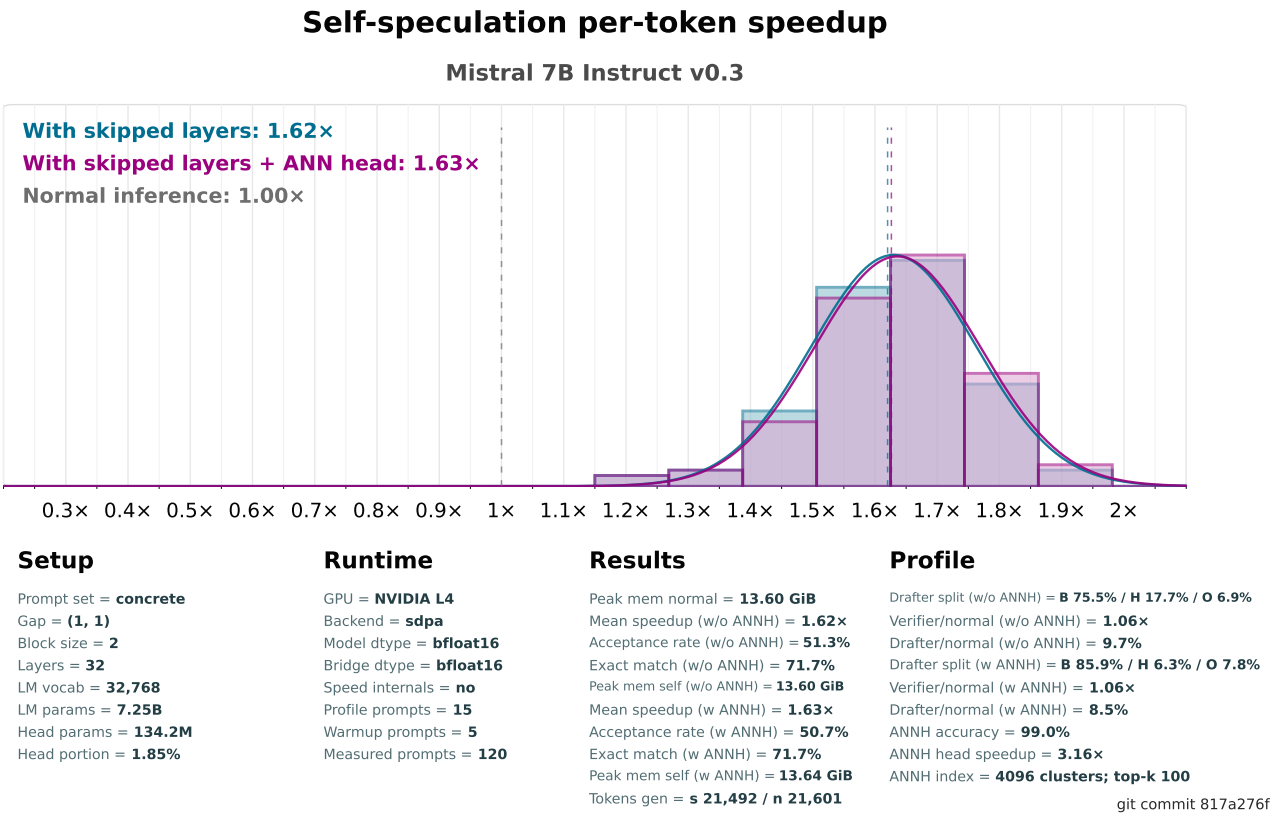


Figure 24: Per-token self-speculation speedup for Mistral 7B Instruct 0.3v on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt.

The Mistral 7B Instruct shows a relatively large speedup of 1.62× with skipped layers and a speedup of 1.63× with skipped layers + ANNH. Figure 24 shows that the LM-head portion is small compared with the body, so the head approximation contributes less to total speedup than the layer skipping does.

Using Equation 1 with $v = 1.06$, $\gamma = 2$, $a = 51.3\%$, and $d = 9.7\%$ (skipped layers only), the predicted speedup is

$$S = \frac{1 + 2 \cdot 0.513}{1.06 + 2 \cdot 0.097} = \frac{2.026}{1.254} \approx 1.62 \times,$$

which matches the measured 1.62× exactly. With ANNH, substituting $a = 50.7\%$ and $d = 8.5\%$ gives

$$S = \frac{1 + 2 \cdot 0.507}{1.06 + 2 \cdot 0.085} = \frac{2.014}{1.230} \approx 1.64 \times,$$

which is slightly above but still close to the measured 1.63×.

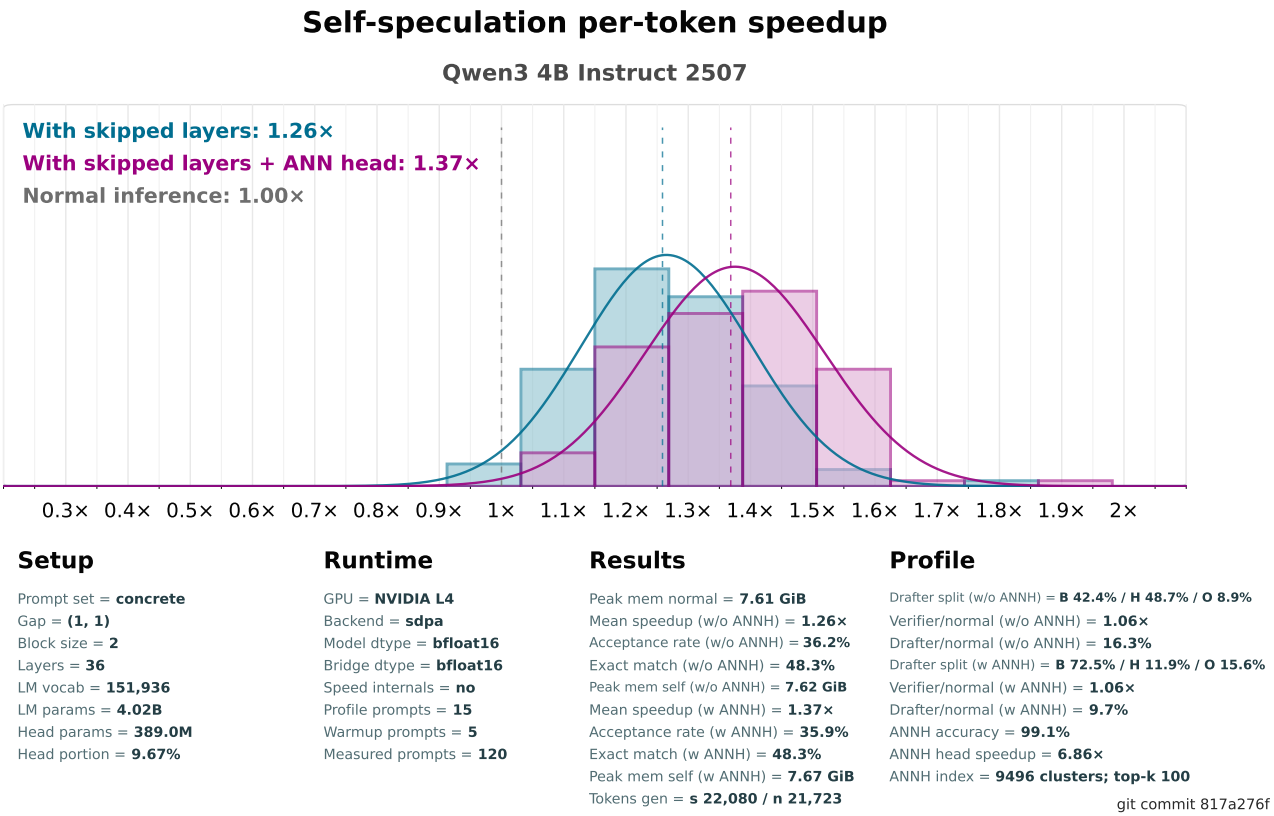


Figure 25: Per-token self-speculation speedup for Qwen3 4B Instruct on the concrete prompt set. The blue distribution is speedups for different prompts using skipped layers in the drafter. The magenta distribution is for the drafter with both skipped layers and with ANNH. The speedups reported are per-token while the histogram and normal curves are speedup per prompt.

Figure 25 shows that the implementation also gives speedups for Qwen3. The acceptance rates are relatively low at 36.2% without ANNH and 35.9% with ANNH. This manages to result in speedups of 1.26× and 1.37×. A block size of 2 can be too big for this drafter for this prompt set.

3.4.1.2 Python-diverse prompt set

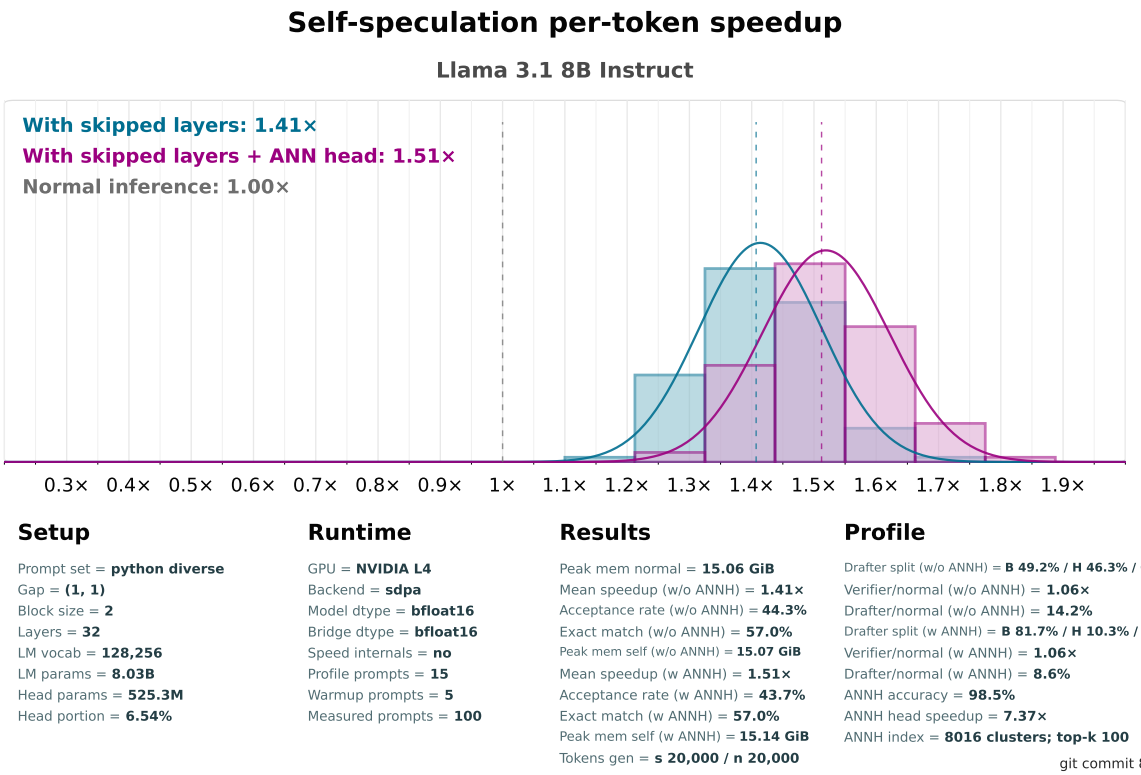


Figure 26: Per-token self-speculation speedup for Llama 3.1 8B Instruct on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2.

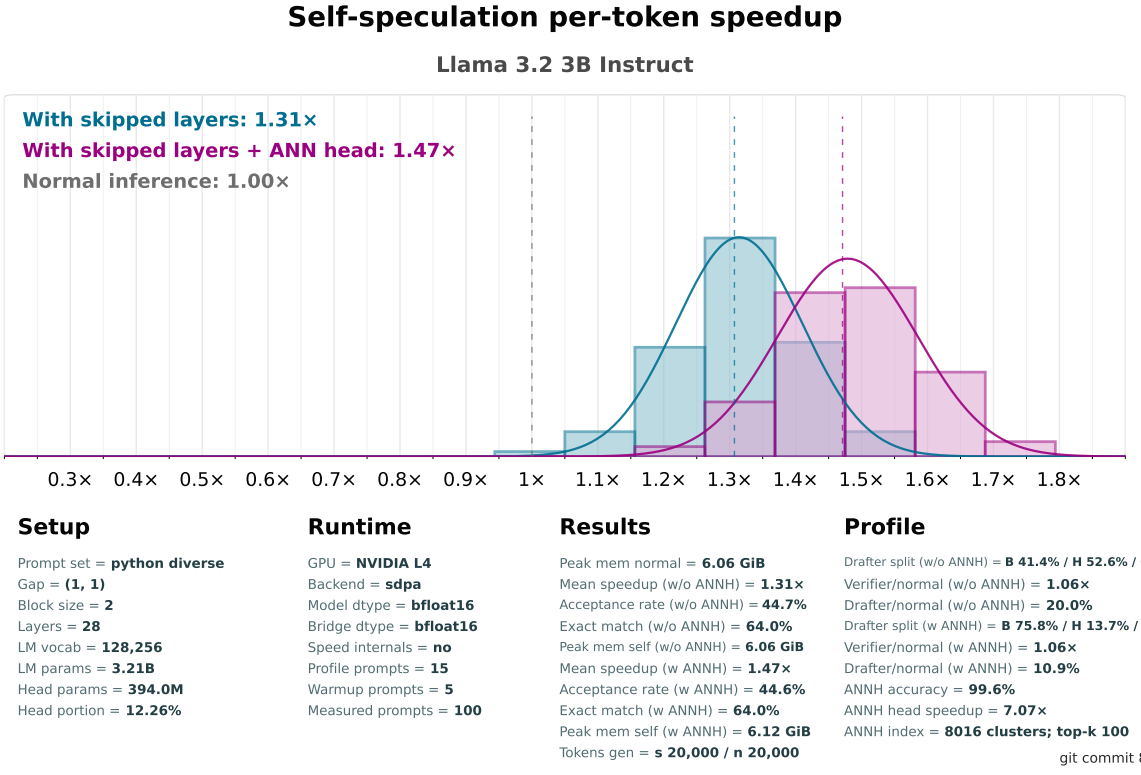


Figure 27: Per-token self-speculation speedup for Llama 3.2 3B Instruct on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2.

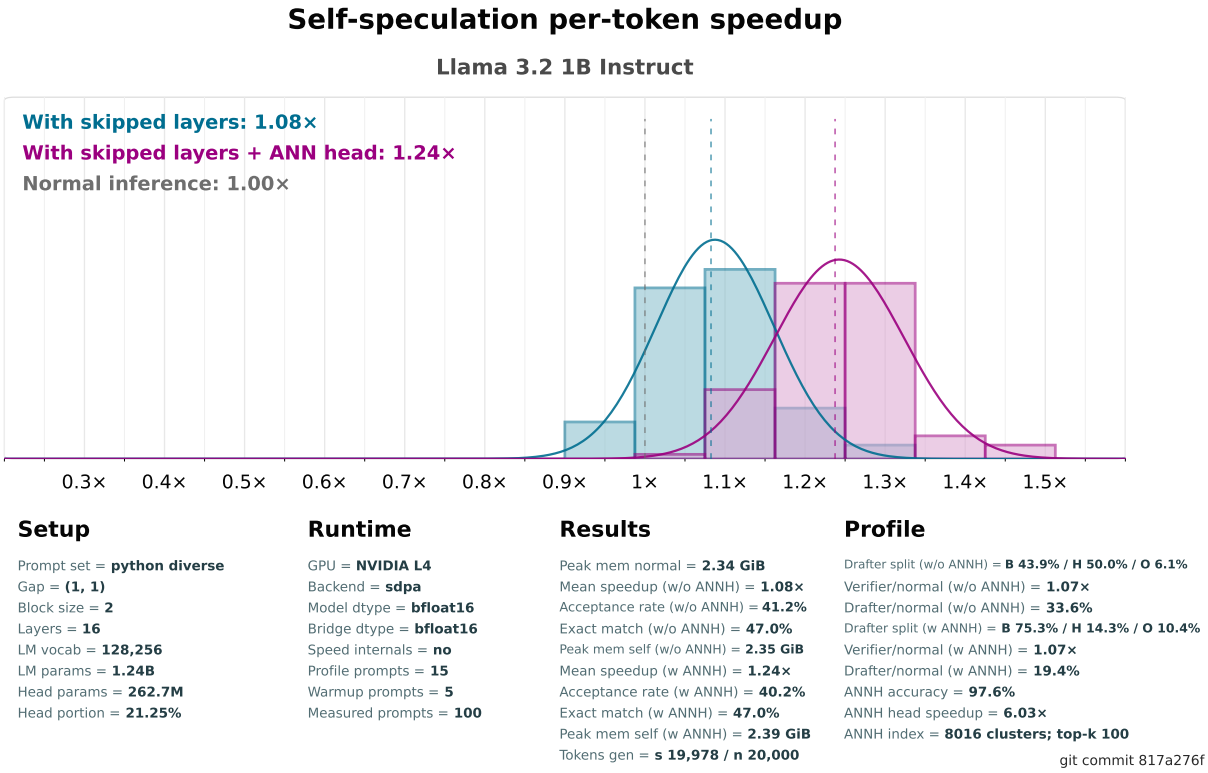


Figure 28: Per-token self-speculation speedup for Llama 3.2 1B Instruct on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2.

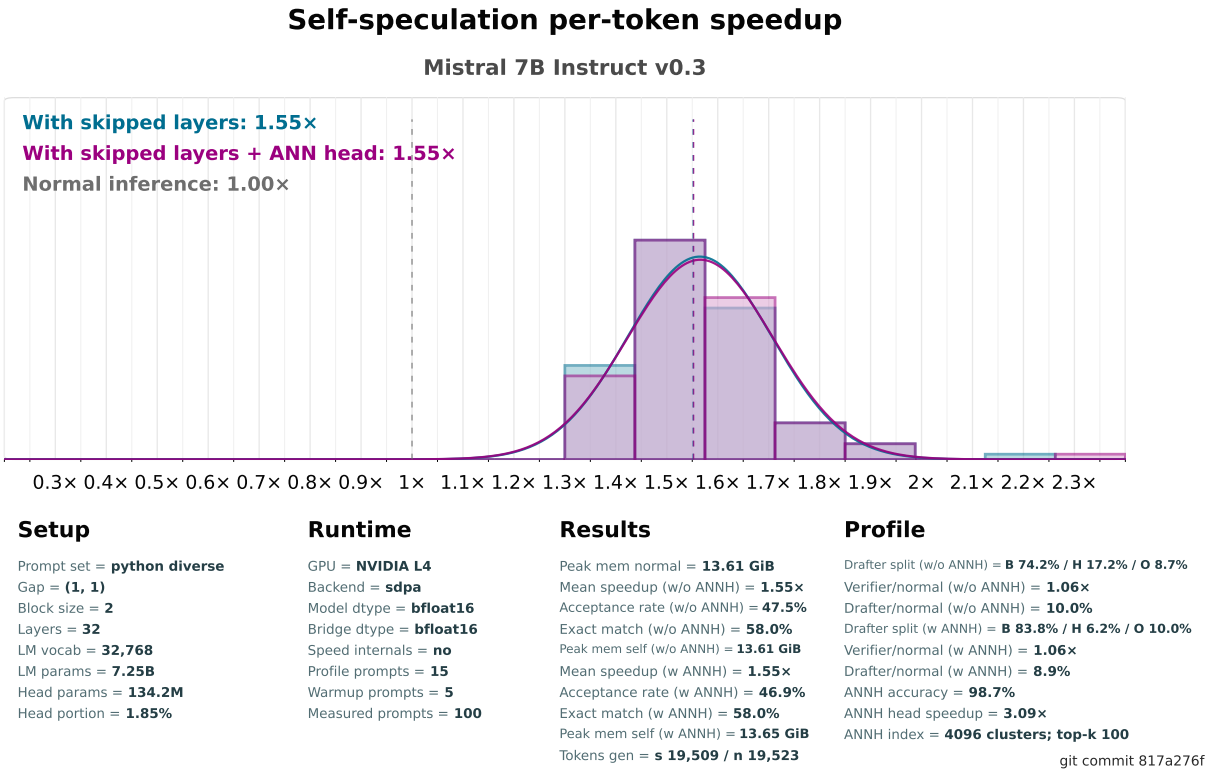


Figure 29: Per-token self-speculation speedup for Mistral 7B Instruct v0.3 on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2.

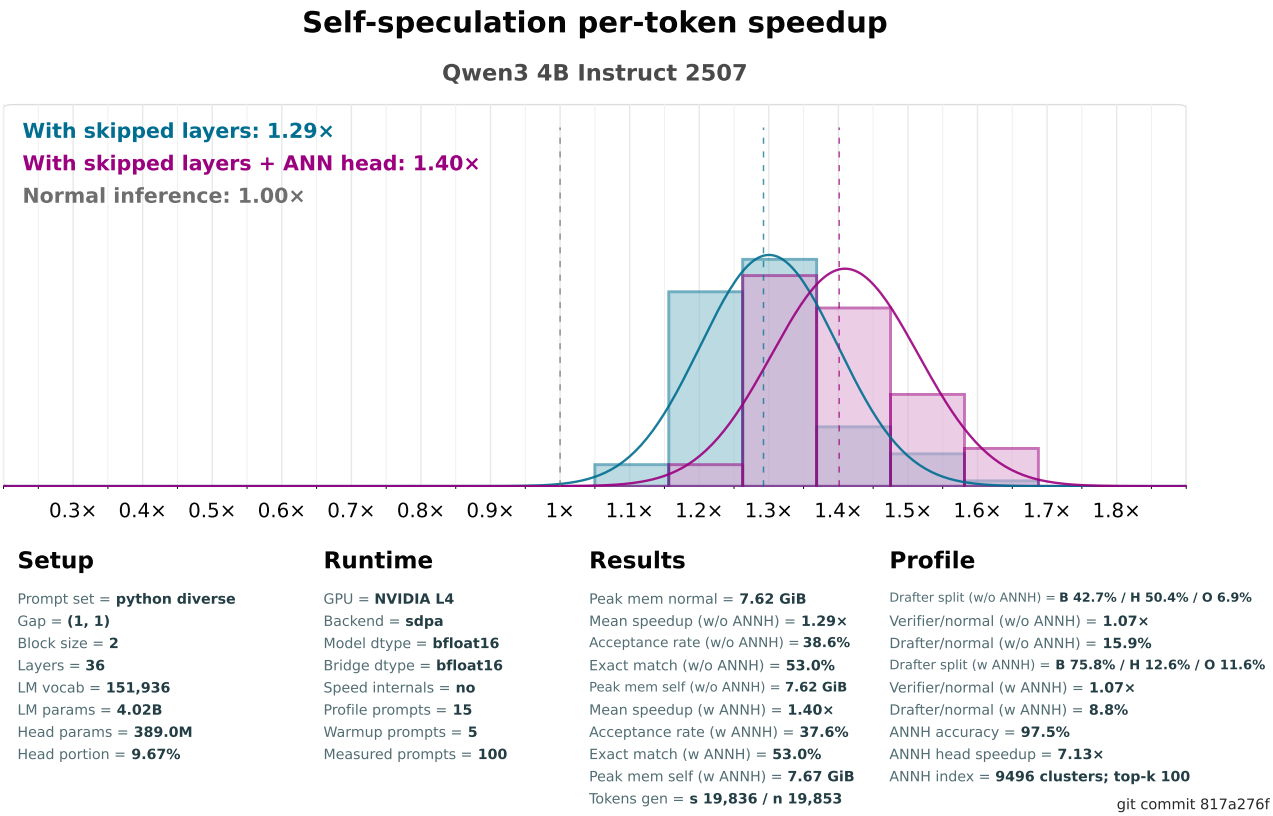


Figure 30: Per-token self-speculation speedup for Qwen3 4B Instruct on the Python-diverse prompt set, with gap $(1, 1)$ and draft block size 2.

Across the Python-diverse prompt set, the skipped-layers + ANNH variant gives speedups from 1.24× for Llama 3.2 1B to 1.55× for Mistral 7B. The larger models generally get larger speedups, but Qwen3 4B is lower than Llama 3.2 3B in this setup because its acceptance rate is lower.

3.4.2 Benchmark summary

Table 18 shows the benchmarks for all prompt sets, all models and all block sizes.

Table 18: Self-speculation benchmark results. Each cell reports the total per-generated-token speedup for the skipped-layers + ANNH variant, followed by total draft-token acceptance rate. Bold values mark the better block size for each prompt set and model, when only one block size was tested for that prompt set, that value is bold by default. All combinations in this table are runs from the main benchmark matrix.

Benchmark combination	Llama 3.2 1B-Inst	Llama 3.2 3B-Inst	Qwen3 4B-Inst	Llama 3.1 8B-Inst	Mistral 7B-Inst
$(1, 1)$, Python-diverse, block size 2	1.24× / 40%	**1.47× / 45%**	**1.40× / 38%**	**1.51× / 44%**	**1.55× / 47%**
$(1, 1)$, concrete, block size 2	**1.28× / 43%**	**1.46× / 44%**	**1.37× / 36%**	**1.58× / 47%**	**1.63× / 51%**
$(1, 1)$, open-ended, block size 2	**1.22× / 37%**	**1.40× / 39%**	1.20× / 25%	**1.45× / 39%**	**1.48× / 42%**
$(2, 2)$, Python-diverse, block size 2	1.20× / 48%	1.46× / 53%	1.43× / 47%	1.50× / 52%	1.55× / 55%
$(1, 1)$, Python-diverse, block size 1	**1.25× / 58%**	1.38× / 64%	1.36× / 58%	1.40× / 64%	1.43× / 68%
$(1, 1)$, open-ended, block size 1	1.22× / 54%	1.32× / 56%	**1.21× / 40%**	1.35× / 56%	1.38× / 60%

The table shows three broad patterns. First, all skipped-layers + ANNH combinations are faster than normal generation. Second, prompt style matters: the open-ended prompt set usually has lower acceptance rate and thus lower speedup than concrete or Python-diverse completion. Third, the $(2, 2)$ gap improves acceptance rate relative to $(1, 1)$ on Python-diverse prompts, but the extra kept layers adds more compute which makes the speedup smaller compared to (1,1) in most cases.

3.4.3 float32 debug test

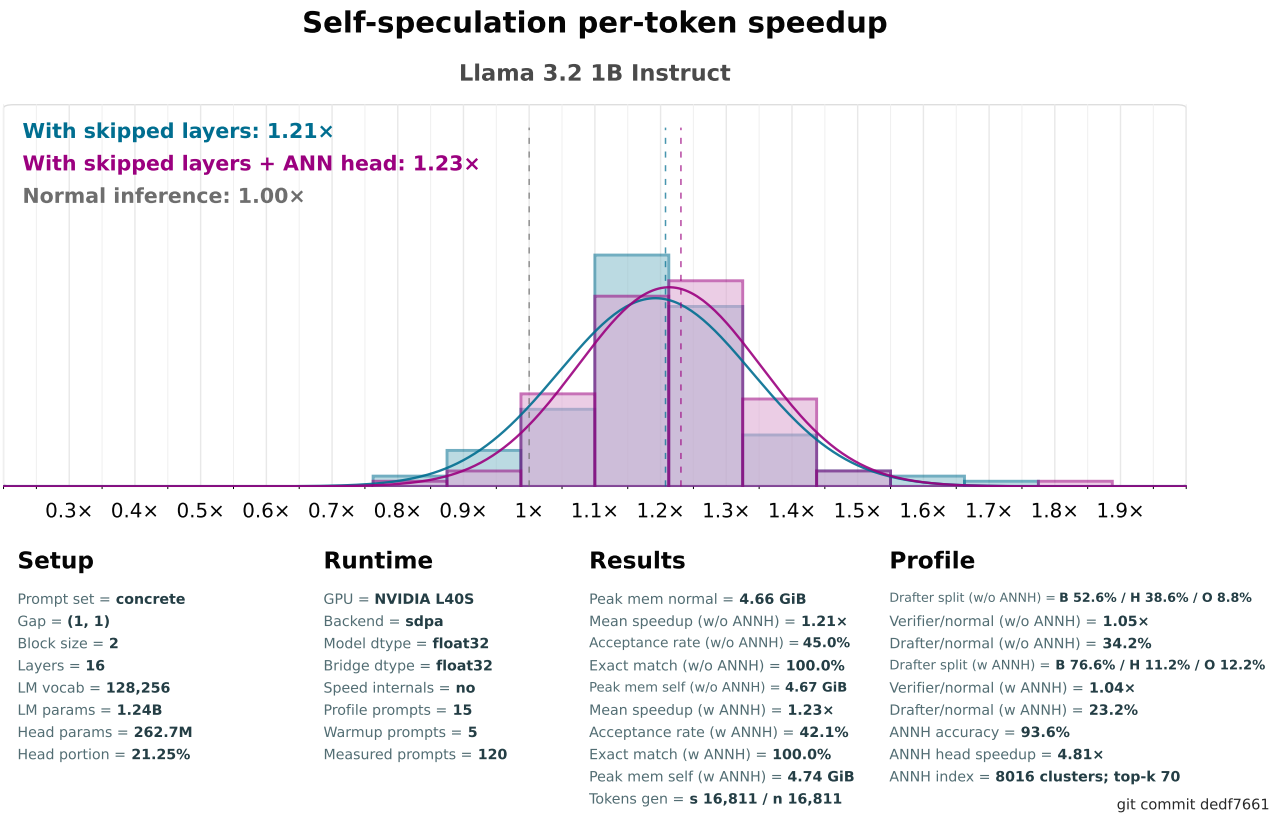


Figure 31: Per-token self-speculation speedup for Llama 3.2 1B Instruct on the concrete prompt set. This is a debug run to investigate generation correctness. The datatype for the model is float32.

Figure 31 shows a debug run with float32. Here the match between normal generation and self-speculation generation is 100% for both drafter versions. The number of tokens generated is also exactly the same with 16811 in for normal and with the self-speculation implementation.

4

Discussion

4.1 Interpreting the results

Here are thoughts and discussion about the results and approach for this project.

4.1.1 Layer-skipping ablations

The skipping ablations results showed that in general, it seems better to skip an internal gap than to do early-exit or late-start. The results also show that without HVC, the performance degrades very quickly. Doing an (N-1, 0) early exit where N is the number of model layers, meaning that only the final layer is skipped, gave top-1 agreement scores of 0.600 for Llama 3.2 1B Instruct, 0.718 for Llama 3.2 3B Instruct, 0.613 for Llama 3.1 8B Instruct, 0.765 for Qwen3 4B Instruct, and 0.787 for Mistral 7B Instruct v0.3. All these results are significantly below 1.0 and indicate that it does not seem reasonable to build a drafter without using a HVC, which aligns with the expectation from the theory about the project.

The layer skipping ablations also showed that it does not seem beneficial to do non-contiguous skips, such as periodically skipping every other layer or every third. Periodic masks sometimes appear competitive in the normalized KL ranking, but they do not lead the results and they often have very low top-1 agreement. Since those masks would require more than one HVC bridge and did not show an obvious advantage, they were not chosen as a direction for the benchmark setups.

The KL per layer results show that skipping the first layer hurts performance the most. In the current runs, keeping every layer except the first gives top-1 agreement between 0.021 and 0.093 across the five tested models, and it is the worst single-layer skip for all of them in terms of KL. This is reasonable since the first layer is the one trained to route the embedding vector.

The patterns seem somewhat stable between different model families. Some models such as Llama 3.2 1B Instruct seem to handle early-exit relatively well compared to its result for other ablations. However, per skipped layer, the internal gaps still seem to have the better results.

4.1.2 HVC bridge training

In Figures 17 and 19 a very aggressive gap of (1,1) and (2,2) respectively is trained. The figures show that the top-1 agreement starts at approximately 0% which is coherent with what the skip-ablations showed for so many skipped layers, but that the top-1 agreement converges to

around 60–72% depending on the model. This is a result that strongly shows that the HVC-bridge can partially compensate for a large gap. The same pattern shows for the verifier to drafter KL metrics in Figures 18 and 20. For gap (1,1) the KL starts very high, but converges to between 0.89 to 1.32 depending on the model.

Using a gap (2,2) results in a better KL and top-1 for all five plotted models, but not much better. Top-1 for Mistral 7B Instruct went from 69.5% to 71.7%, for Llama 3.1 8B from 63.9% to 67.8%, and for Qwen3 4B from 60.1% to 61.9%. These are improvements but the number of layers used doubled. To produce a drafter that can give speedups in self-speculation, it is highly advantageous if it is cheap.

This can be shown directly from Equation 1. Solving for the required acceptance rate gives

$$a = \frac{S(v + \gamma d) - 1}{\gamma}.$$

For a target speedup of $S = 1.4\times$, a block size of $\gamma = 1$, and a verifier cost of $v = 1.05$, Table 19 shows that the needed acceptance rate increases quickly as the drafter becomes more expensive. For sufficiently expensive drafters, the required acceptance rate is above 100%, meaning that the target speedup is impossible even if every drafted token is accepted. This means that using more layers is only beneficial if the improved drafter quality is large enough to compensate for the increased draft cost.

Table 19: Acceptance rate required to reach a theoretical self-speculation speedup of 1.4 × with block size $\gamma = 1$ and verifier cost $v = 1.05$.

Drafter cost d of full model	Acceptance needed for 1.4 ×
5%	54.0%
10%	61.0%
15%	68.0%
20%	75.0%
25%	82.0%
30%	89.0%
40%	Impossible (103.0% required)
50%	Impossible (117.0% required)
60%	Impossible (131.0% required)
70%	Impossible (145.0% required)
80%	Impossible (159.0% required)

4.1.3 ANNH cluster building

The FlashHead-alike ANNH cluster took between 57 and 309 seconds to build on an Apple M5 24GB depending on the model. The more clusters, higher hidden dimensions and larger vocabulary size, the longer the clustering takes. The clusters files are not notably big, the Llama 3.2 3B 8016 cluster has a file size of 99 MB. The clusterings in the result sections used a max iterations of 40. However, Figure 16 and the clusterings made internally during the project show that the process converges after around 15 iterations. Therefore using 40 iterations is not strictly needed.

4.1.4 ANNH accuracy

The result shows that the most important parameter to the ANNH inference accuracy is the top-k parameter. Tables 7, 9, 11, 13, 15, and 17 all show that to get a 99% accuracy, a top-k of 200–

300 is needed. An interesting observation is that this means that ANNH's with more clusters need a smaller portion of the total number of clusters probed to reach a certain threshold in accuracy. Probing 300 of 16032 is a significantly smaller portion probed than 300 of 2672. One possible reason for this is that using more clusters means fewer vectors per cluster, the centroids which are the average of the cluster-vectors thus become a less approximated representation of the content. So with more clusters, given a query hidden vector, the probability that the scoring with a centroid gives good routing increases. This means that top-k = constant returns fewer candidate vectors when using more clusters, but those candidates were routed with less approximation. This pattern should continue with increasing cluster size, the extreme is that the number of clusters is the same as the vocabulary size, at that point the accuracy will be 1.0 with top-$k = 1$. The problem then is of course that the cluster scoring matrix multiplication is as big as the one we tried to avoid in the original LM-head. So there needs to be a balance between centroid representation for its cluster vectors and having a cluster matrix that is significantly cheaper than the original LM-head.

The tables also show that Llama 3.2 3B Instruct seems to get higher accuracy than Llama 3.2 1B Instruct for the same top-k, same number of clusters and having the same vocabulary. For top-$k = 100$ with 8016 clusters the accuracy for 1B was 97.19% and for 3B 98.39%. One possible explanation for this is that the 3B version has more hidden dimensions. This means that there is more space to distribute the vocabulary and therefore possibly more separation between clusters which can make routing easier.

4.1.5 ANNH with self-speculation

Running ANNH with speculative decoding is distinctively different than running the model normally with ANNH. With speculative decoding any mistake from the ANNH (and the drafter in general) will be caught by the verifier. So mistakes from the ANNH will instead only reduce the acceptance rate, not produce any different output. For normal generation with ANNH to have high probability that the output won't diverge or degenerate, the accuracy would likely need to be around $> 99.5\%$, but in speculative decoding, it is okay if it is lower because it will just be a factor that reduces the acceptance rate. If the acceptance rate is 50% without ANNH, and the ANNH with a chosen top-k has an accuracy of 95% the acceptance rate will be $0.95 \times 0.5 = 47.5\%$, but the output is still guaranteed to be correct. Since the results show that its significantly easier to go from 0% to 95% than 95% to 99.5%, the ANNH head can be very performant during self-speculation and it is enough to use a top-k in the range of 50 to 150.

4.1.6 Exact match and numerical precision

As shown in Figures 21, 22, 23, 24, and 25 the exact match to the normal generation is not 100%, even though it is greedy argmax generation. This was a strange result and the project investigated why it is the case because the generation should not be approximate or lossy compared to the normal model. The reason is that when selecting the next token, there are a lot of logit-ties when using bfloat16, tokens that get the exact same score. These ties happen in both normal generation and in self-speculation. So in bfloat16 there is not enough information to select an unambiguous winner. To debug if this was really the case the project added a flag `--debug-argmax-ties` in `bench_self_spec.py` that makes the normal generation and self-speculation implementation print if there is ever a logit tie when doing argmax, see the function `argmax_debug_first_tie(..)` in the open source repository. When using this, there were usually 1–5 ties detected for each generation of at max 200 tokens. If this is the case, then there should be a 100% match rate when using float32 because then the limitation of precision is mostly

removed. As Figure 31 shows but also all internal runs, float32 generations did always get a 100% match rate which heavily suggests that the self-speculation setup and logic is not faulty but that there needs to be high enough precision to make unambiguous choices, both for the normal and self-speculative generation.

A simple sanity check supports that this is plausible. Consider logits around 12 as an example. The value 12 is not special, but just an example of a value logits might be. It will illustrate what happens when several token logits are close to each other around the same magnitude. In bfloat16, numbers around 12 are in the exponent range $[8, 16)$ and have 7 explicit mantissa bits. The spacing between representable values is therefore $2^{3-7} = 2^{-4} = 0.0625$. Between 12.0 and 12.1 there are only two bfloat16 values: 12.0 and 12.0625. The next representable value is 12.125, which is already outside the interval. Float32 has 23 mantissa bits, so the spacing in the same range is $2^{3-23} = 2^{-20} \approx 9.54 \cdot 10^{-7}$. This means that the interval from 12.0 to 12.1 contains about $\lfloor 0.1 \cdot 2^{20} \rfloor + 1 = 104858$ representable float32 values. Since the vocabulary contains many tokens and the top logits can be close to each other, it is therefore not surprising that bfloat16 can collapse distinct logits into exact ties while float32 usually separates them.

4.1.7 Why not test with fewer skipped layers

The project decided to mostly focus on very large gaps of skipped layers. The primary benchmarks are for (1,1) and (2,2) gaps. The question is then whether bigger speedups could have been achieved with smaller gaps so the drafter accuracy gets higher. From extensive internal experiments and optimization to reach as large speedups as possible, it is considered to be quite hard to compensate for a more expensive drafter from the accuracy increase of keeping more layers. Even with many layers kept, it is difficult to reach a top1 of $> 80\%$. The fundamental reason for this is probably because the choices of the verifier are not gold tokens, they are just the subjective calculation the full model happens to produce. The drafter is therefore not learning a fundamental truth but is instead guessing what the verifier would do. The project thus found that only a few layers gave the large jump in top-1 but that then adding more layers made the result incrementally better, while making the drafter linearly more expensive.

The theoretical estimation from Equation 1 can be used to check if a smaller gap will have a chance of producing a better drafter than (1,1) or (2,2). Assume the head is 5% of the compute, that ANNH makes the head 5× faster and that (1,1) with HVC gives a top1 accuracy of 65%. For a 32-layer model, assuming that body compute is linear in the number of kept layers, the drafter cost for a gap (N, N) is

$$d_N = 0.95 \cdot \frac{2N}{32} + \frac{0.05}{5}.$$

This gives $d_1 = 6.94\%$ for the (1,1) + ANNH drafter. With 65% top-1 accuracy, this baseline gives a theoretical speedup of $1.474\times$ for block size 1. For greedy self-speculation with block size 1, top-1 agreement is approximately the same as the acceptance rate because each speculative round drafts only one token. Table 20 therefore shows what acceptance rate a more expensive (N, N) drafter would need to match that baseline speedup.

Table 20: Acceptance rate needed for a larger (N, N) drafter with ANNH to match the 1.47 × theoretical speedup of a (1,1) + ANNH drafter with 65% acceptance rate, using block size $\gamma = 1$, verifier cost $v = 1.05$, and a 32-layer model.

Gap	Layers kept	Drafter cost d_N	Acceptance needed for 1.47 ×
$(1, 1)$	2	6.94%	65.0%
$(2, 2)$	4	12.9%	73.8%
$(3, 3)$	6	18.8%	82.5%
$(4, 4)$	8	24.8%	91.3%
$(5, 5)$	10	30.7%	Impossible (100.01% required)
$(6, 6)$	12	36.6%	Impossible (108.8% required)
$(8, 8)$	16	48.5%	Impossible (126.3% required)
$(12, 12)$	24	72.2%	Impossible (161.3% required)
$(16, 16)$	32	96.0%	Impossible (196.3% required)

The estimates in Table 20 show that when keeping more layers, the acceptance threshold to achieve the same speedup increases rapidly. If the top1 is 65% with (1,1) then it needs to be 73.8% for (2,2) and 82.5% for (3,3). This is not the increase we see when using smaller gaps which suggests that it will not be a viable alternative to test for gaps smaller than (2,2). Table 20 also shows that its impossible to reach the same speedup as the (1,1) with (5,5) or smaller gaps, which for this model is to keep 30.7% or more of the layers. So it doesn't matter how good a skipping ablation is that keeps more than 30.7% of the layers, it won't produce a self-speculative system faster than the (1,1) baseline.

4.1.8 Self-speculation speedups and memory usage

All combinations in the benchmark resulted in a speedup with the self-speculative system. At the precision reported in the table, adding ANNH never reduced and usually increased speedup compared with skipping layers alone. When using the best block size for each model and prompt set, the skipped-layers + ANNH speedups were in the range 1.21× to 1.63× in bfloat16. The lower end was Qwen 3 4B on open-ended prompts with block size 1, and the upper end was Mistral 7B Instruct v0.3 on the concrete prompt set with block size 2.

The measured speedups are prompt dependent and form an approximate normal distribution. With skipped layers + ANNH on the concrete prompt set, Mistral 7B Instruct had an overall speedup of 1.63× and per-prompt speedups from 1.21× to 1.90×. With the same prompt set and block size, Llama 3.2 3B Instruct had an overall speedup of 1.46× and per-prompt speedups from 0.86× to 2.27×.

On NVIDIA L4, replacing the dense LM-head with ANNH made the profiled drafter head 2.80× to 7.53× faster across the benchmark. The model with largest relative additional speedup from using ANNH in the concrete block-size-2 setup was Llama 3.2 1B, which went from 1.13× to 1.28× in average per-token speedup. Large speedups for the LM-head do not directly translate to large speedup in the self-speculative setup, but the results show that it helps most when the head is a meaningful part of the drafter cost. For Mistral 7B Instruct v0.3 the head is proportionally small, so ANNH gives little extra speedup even when the head itself becomes faster. This follows the Amdahl's law reasoning exactly.

All models seem to use approximately the same amount of VRAM as the normal inference. Loading the ANNH index into memory adds slightly to memory usage, but across the benchmarks the observed peak-allocation overhead for the skipped-layers + ANNH variant is small: about 0.9% on average, with a maximum of about 2.1%. To use the same model for the verifier

and the drafter to do self-speculation and also to share the KV-cache therefore results in a solution that does not need meaningfully more memory than normal inference.

4.1.9 How long is the total training time?

The speedup benchmarks in Figures 21, 22, 23, 24, and 25 use the setup times in Table 21 to produce the HVC-bridge and the ANNH.

Table 21: Total setup time for the HVC bridge and ANNH index used in the self-speculation speedup benchmarks.

Setup	HVC bridge	ANNH build	Total	Clusters
Llama-3.2-1B-Instruct	13 min 57 s	2 min 33 s	16 min 30 s	8016
Llama-3.2-3B-Instruct	23 min 04 s	3 min 50 s	26 min 54 s	8016
Llama-3.1-8B-Instruct	34 min 19 s	4 min 10 s	38 min 29 s	8016
Mistral-7B-Instruct-v0.3	26 min 45 s	33 s	27 min 18 s	4096
Qwen3-4B-Instruct-2507	28 min 59 s	4 min 46 s	33 min 45 s	9496

For these times, the HVC-brige is trained on a NVIDIA RTX PRO 6000 and the ANNH is built on an Apple Macbook Pro M5 24GB.

4.1.10 Does this approach make sense?

This method does seem to produce a useful set of properties:

1. No more memory usage than normal inference.
2. Lossless generation quality relative to the stock model, up to numerical tie-breaking effects.
3. Observed possible average speedups between 1.21× and 1.63×.
4. Total training time for the HVC bridge and the ANNH index of less than 1 hour.
5. A concrete recipe to turn a model into a drafter for itself.

The main drawback is that the potential speedup is dependent on the nature of the prompt. If the prompt is concrete and has a less open ended future, then the acceptance rate seems to be higher and thus resulting in a larger speedup. If the prompt is open ended like telling a story then it's harder for the drafter to generate the exact same choice as the verifier which lowers the acceptance rate and thus the potential speedup. The bridge likely also needs to be trained on a dataset that is somewhat representative of the task it will perform to achieve its full potential.

4.2 Answering the research questions

Here are the answers to the research questions that were stated in the Introduction chapter.

4.2.1 Which layer-skipping strategy minimizes damage to generation quality per layer skipped, and does this pattern hold across model families?

The experiments indicate that skipping one contiguous internal gap is the best tested strategy. The skip-ablation results show that early-exit and late-start strategies damage the output distribution more for the same number of skipped layers, while non-contiguous skip patterns such as skipping every other layer do not give a clear advantage. The useful pattern is therefore to keep the first layers, keep the last layers, and skip a block in the middle.

This pattern also makes sense regarding the role of the layers. The first layers transform the token embeddings into the model's internal representation, and the last layers prepare the hidden vector for the final unembedding. An internal gap is then likely less destructive because it leaves both the first and last layers of the model intact.

4.2.2 Can a lightweight HVC bridge recover the generation quality lost from skipping layers well enough to produce an effective drafter?

The HVC bridge can recover a large portion of the lost generation quality when skipping layers. Figures 17 and 19 show that for a large gap, (1,1) or (2,2) respectively, the HVC can increase the accuracy from around 0% top-1 to a top-1 in the range of 60% to 73%. The HVC is not powerful enough to produce a standalone model, but strong enough to often predict the same next token as the full model. The training results show that it's easier to cast a hidden vector through a smaller gap, (2,2) instead of (1,1), with KL and top-1 being better for every model tested. However, the result does not indicate a dramatic improvement from (1,1) to (2,2), so given the added compute with more layers, for self-speculation (1,1) seems to be the more attractive configuration. The answer to this research question is then that a linear HVC seems to be very effective to recover much of the degradation from skipped layers, especially for the small amount of compute the HVC needs, and it does work well enough to produce a drafter that gives a speedup in speculative decoding.

4.2.3 What is the minimum acceptance rate required for the proposed self-speculative setup to outperform normal inference, given empirically observed verifier and drafter costs?

The minimum required acceptance rate can be derived from Equation 1. The estimated self-speculative speedup is

$$S = \frac{1 + \gamma a}{v + \gamma d},$$

where γ is the draft block size, a is the draft-token acceptance rate, v is the verifier cost relative to one normal generation step, and d is the drafter cost relative to one normal generation step. To outperform normal inference, the speedup must be greater than 1:

$$\frac{1+\gamma a}{v+\gamma d} > 1.$$

Since the denominator is positive, this is equivalent to

$$1+\gamma a > v+\gamma d.$$

Solving for a gives the minimum required acceptance rate:

$$a > \frac{v+\gamma d-1}{\gamma}.$$

If the right-hand side is greater than 1, then the setup cannot outperform normal inference even if every drafted token is accepted.

Using the measured verifier and drafter costs from the concrete prompt set with gap $(1,1)$, block size $\gamma = 2$, and ANNH enabled, the required acceptance rates are relatively low, as shown in Table 22:

Table 22: Minimum draft-token acceptance rate needed to outperform normal inference, using measured verifier and drafter costs from the concrete prompt set with gap $(1,1)$, block size 2, and ANNH enabled. The measured acceptance rates and speedups are from the same benchmark runs.

Model	v	d	Required a	Measured a	Speedup
Llama-3.2-1B-Instruct	1.05	19.6%	22.1%	43.0%	1.28×
Llama-3.2-3B-Instruct	1.05	11.5%	14.0%	43.9%	1.46×
Qwen3-4B-Instruct	1.05	9.7%	12.2%	35.9%	1.37×
Llama-3.1-8B-Instruct	1.04	8.5%	10.5%	46.9%	1.58×
Mistral-7B-Instruct-v0.3	1.05	8.5%	11.0%	50.7%	1.63×

The answer to the research question is therefore that, for the skipped layers + ANNH setup with block size 2, the drafter only needed acceptance rates of about 10.5%–22.1% to break even. Equivalently, the system could reject about 78%–89% of drafted tokens and still be faster than normal inference, depending on the model. The measured average acceptance rates were significantly higher than these break-even points, ranging from 35.9% to 50.7% in the table. This explains why the setup can produce speedups even though the drafter is not a highly accurate standalone approximation of the full model.

4.2.4 To what extent can inference for LLMs be sped up by using a setup where the draft model is made computationally cheaper in both body and head by using the techniques: skipping layers + HVC + ANNH + self-speculative decoding?

Across the main benchmark matrix, the proposed setup produced speedups for all tested combinations of model, prompt set, gap, and block size. With skipped layers, HVC, ANNH, and self-speculative decoding, the measured per-generated-token speedups ranged from 1.20× to 1.63× compared to normal generation. When choosing the better tested block size for each model and prompt set, the range was 1.21× to 1.63×. The setup also kept approximately the same memory usage as normal inference because the drafter and verifier are the same model and share the KV-cache.

The answer to the third research question is therefore that the proposed combination can give meaningful real inference speedups, in this report up to about 1.6×, without increasing memory usage and keeping the output lossless compared to normal inference.

4.3 Hypothesis about easy and hard tokens

The fundamental idea behind this project is that there are easier and more difficult tokens to predict given a context. In the generation "Usain Bolt runs very fast and has the reco.." then it is likely to be statistically obvious that the next token should be "rd" to complete the word "record". But in the same generation but for another token "Usain Bolt runs very fast and has the record in 100m. ?" Then it is not necessarily obvious what the next token should be because it is the start of a new sentence and that might require more strategic and complex selection. So the idea is that there are obvious tokens and non-obvious tokens and a spectrum between these.

To then use all layers for all tokens is somewhat wasteful with this perspective. The capacity of all layers is mostly useful for key positions that decide the quality of the generation. With this, self-speculative decoding can be reframed as a practical approach to engineer the inference to not use the full capacity for all tokens. The alternative could be that you have a model that can skip layers but does so dynamically based on the perceived difficulty of the current token to generate. The problem with that solution is to design the mechanism that decides how many layers will be needed to compute the next token. If that mechanism is to conservative, then it leaves efficiency on the table, but if it is too aggressive then it is likely to underestimate the number of layers needed to produce the same token the full model would. In that setup there would be no verifier to catch the mistakes, so they would leak into the generation.

For small models, one could imagine that the overhead for easy tokens is smaller than on larger models. If you decrease the processing with X percent, then the share of tokens still possible to predict is smaller on smaller models than larger ones, because they did not have the same headroom to begin with. With that, the capacity for speedup should be larger for bigger models and that is what has been observed in this project.

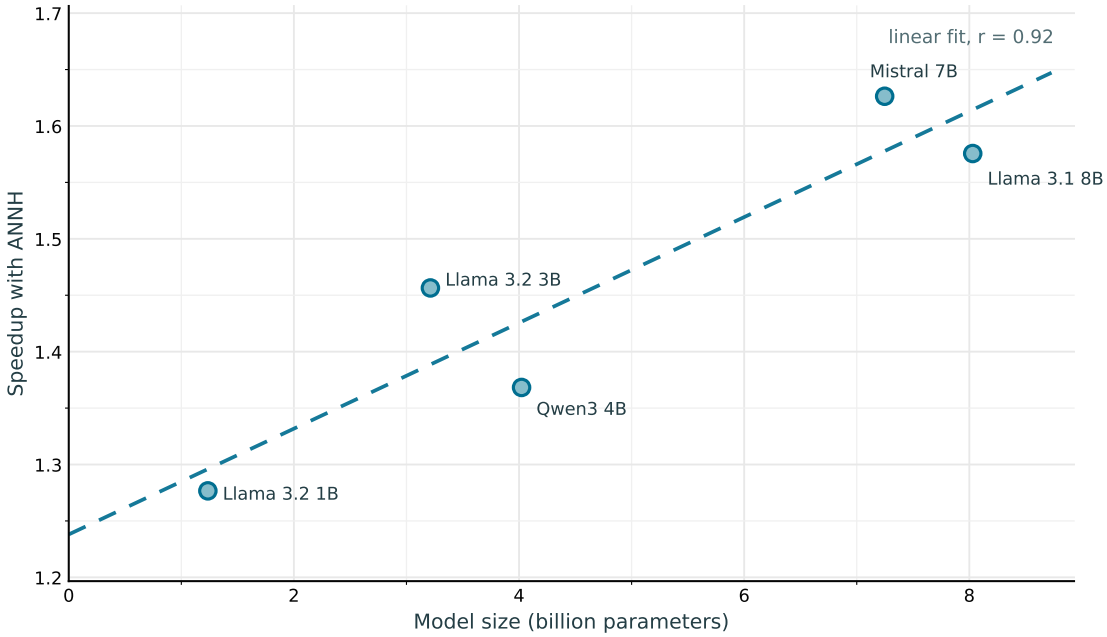


Figure 32: Observed self-speculation speedup with ANNH compared to model size on the concrete prompt set. The fitted line suggests a positive correlation between model size and speedup in these measurements, but the sample is small and mixes model families, so it should be interpreted an illustraion of what this project has seen rather than any conclusive result.

Figure 32 shows the relation between parameter count on the x-axis and the measured speedup on the y-axis. From the few tested models in this project, a correlation of r = 0.92 is calculated. This is however not a conclusive or confident result because of the low number of datapoints and mix of model families. But it indicates that there can be a relation and that larger models can see even larger speedups than those tested in this project.

4.4 Future work

4.4.1 Adaptive block sizes

For the best possible performance on general prompts, using an adaptive block size is likely preferable. This would mean that the inference system chooses the optimal block size during generation depending on what the average acceptance rate has been for the last drafted blocks. Using Equation 1, the system could cheaply calculate what blocksize that would produce the biggest speedup and dynamically select that. This would allow for very large speedups when the prompt is very easy, but also to prevent a slowdown when the prompt is difficult. Implementing this should be straightforward but this project decided to stay with the primitive constant block size for less complexity and easier debugging.

4.4.2 Testing on larger models

A natural next step would be to investigate how it works on larger models 30B+ parameters. They could possibly see a larger speedup or they can behave similarly to the smaller models. Larger models require large amounts of VRAM to run, so to function with the available hardware, this project decided to only test for smaller models.

Larger models can be faster because of the overhead-reasoning presented in the section *Hypothesis about easy and hard tokens*, but they can also have a larger potential for speedup because the job of the linear HVC could possibly be easier with higher dimensions but approximately the same number of tokens in the vocabulary. As a thought experiment, if the number of dimensions were just 2, then the space for the hidden vectors to live in would be very small. This would mean that the hidden vectors often would overlap and use weird patterns to convey enough information to produce a suitable next token. But if the number of dimensions is very high, like 20,000, then there are much more space to cleanly separate hidden vectors that convey certain information. So the job of mapping hidden vectors before the gap to after the gap might be easier if the model is larger with more hidden space. However, it can also be the case that there are more fine grained information to map in larger models, since they are more capable. This would make the mapping more complex and the total quality of the HVC-bridge is unchanged from the smaller models.

4.4.3 Training strategy

This project began with poor results for skipping layers. With continuous refinements to the method, the HVC-bridge started to work pretty well and it converged to producing a real speedup. The key aspects to get good results have been to use the right training loss metrics (KL and CE) compared to the teacher, to use the previous position $t-1$ hidden vector as a reference, to skip an internal gap instead of early-exit, to make the training objective as close as possible to the targeted task, and to have clean training data. Many of these are somewhat obvious that they will help, but it's not obvious how they should be implemented to help maximally. The author believes that there is still unrealized potential to make the draft system work better. The current bottleneck is likely the training setup, it has a two properties that are awkward for the targeted task:

1. When training the HVC-bridge, it gets the previous reference hidden vector from the teacher at position $t-1$, not itself. This is a bit unrealistic because during inference, it will get the previous hidden vector from the verifier at the first draft step, but not for the draft steps after that. The reason for doing this is that the training can be one prefill call instead of individual calls per token position to gather the hidden vector to feed the next call. When attempting to give the previous drafter hidden vector to itself during training, it also appears difficult to learn because the training starts in a state where the drafter at most positions will be given completely unusable previous hidden vectors. So the drafter will give to itself, but before training it is not really capable of giving a hidden vector of any quality, so it is also not capable of learning any patterns. This seemed to create something close to a soft deadlock with a bad learning curve.
2. The training window is divided into sections where the student starts from where the teacher left off. But these sections are currently much larger than the block sizes that will be used. So it is a compromise to get efficient training, but also simulate the task of starting from the verifiers KV-cache and the last final hidden vector.

For future work, if a training method can be found that solves these issues but also keeps the advantages of the current solution, then it should be possible to see better training that is more aligned with the targeted objective.

4.5 Related work

This thesis builds on three main lines of related work: efficient LM-head inference, layer-skipping and self-speculative decoding, and methods that study or translate intermediate transformer representations.

4.5.1 Efficient LM heads

FlashHead [5] is the paper that proposed the idea to use ANN for the LM-head to reduce the computational work. This project has used that idea as one component to produce a cheap drafter from an LLM. This project produced an implementation from the paper's general description. The results such as speedup, accuracy and dynamics have been similar to the official paper. The exact clustering implementation is not revealed in the paper, so this project did come up with an algorithm that likely uses the same general approach. The clustering time seems to be significantly shorter with this project's implementation, 2 minutes and 30 seconds on CPU compared to the reported 4 hours on GPU in the FlashHead paper. This might indicate that their implementation uses a more fine grained greedy algorithm or some other details that increases the needed compute.

FlashHead used their ANN LM-head without speculative decoding. They reported a total speedup of 1.08× for Llama 3.2 3B in bfloat16 with 8016 clusters and top-$k = 512$, with the LM-head getting 3.13× faster. This shows that their LM-head did get much faster, but due to the head only being around 12% of the total model, the total speedup is still not huge. Performing the Amdahl's Law math, the theoretical speedup of only speeding up the head is for that model thus: $\frac{1}{(0.88+\frac{0.12}{3.13})} = 1.088\times$ which is close to their reported speedup.

This report has shown an average speedup of 1.3× with skipping layers, and 1.46× with skipping layers + ANNH for the Llama 3.2 3B. This shows that FlashHead can give a significant speedup when used in self-speculative decoding together with an approximated body.

An important observation is that $1.46/1.3 = 1.123$ is larger than the speedup from 1× to 1.08×, even though the speculative decoding has a verifier that uses the full LM-head. Intuitively, the potential for speedup would be smaller when the drafter is equipped with ANNH but not the verifier. However, ANNH/FlashHead is more important for the drafter because its body is approximated, so the head becomes a significantly larger portion to speedup. In Llama 3.2 3B, if the head is normally 12% of the compute and the body has 28 layers, if all layers except 2 are skipped with gap (1,1), then only around 2/28 of the body-compute is left. This means that the head becomes

$$\frac{0.12}{0.12 + 0.88 \cdot \frac{2}{28}} = 0.656 \approx 65.6\%$$

of the approximated drafter compute. Therefore, using ANNH/FlashHead becomes even more useful than in normal inference. This shows that FlashHead works well with the aimed Amdahl's law perspective and is an essential component to produce a cheap drafter.

4.5.2 Speculative decoding

Speculative decoding is an established technique that was created to reduce the sequential inefficiency for autoregressive generation. The idea is, as presented in the introduction, to use the full model to verify a block of tokens, which can be performed in parallel. To have something to verify, a less expensive model is used to produce a draft block that has a good chance of being

the same as the full model would have produced. The paper *Fast Inference from Transformers via Speculative Decoding* [6] from 2023 formulates a method to use a smaller model that drafts multiple tokens and then using a larger model to verify them in parallel. This would be regular speculative decoding and not self-speculative decoding since it is two different models. The paper shows how many proposed tokens can be accepted in one verifier call and by this give an inference speedup.

Accelerating Large Language Model Decoding with Speculative Sampling [7] also from 2023 presents a similar speculative sampling algorithm. It also uses a draft model and a verifier/ target model. Together these two papers formulated the approach of speculative decoding which this thesis uses.

4.5.3 Layer skipping and self-speculative decoding

LayerSkip [13] by Meta is the closest related work to the layer-skipping + self-speculation part of this thesis. LayerSkip trains language models so that intermediate layers can be used for early-exit, and then uses that to create a drafter for self-speculative decoding. In that setting, the same model can act as both drafter and verifier: the drafter exits early, while the verifier runs the full model. This avoids loading a separate draft model and makes speculative decoding more memory efficient.

This thesis shares the goal of using one model as both drafter and verifier, but differs in how the drafter is obtained. LayerSkip relies on retraining the base model so it can early-exit with high quality. This thesis keeps the LLM frozen and instead produces a drafter by training a HVC-bridge and skipping an internal gap.

The LayerSkip paper reports a speedup for Llama 2 7B of 1.54× to 1.86× for continual pre-training versions [13], while a later Hugging Face implementation benchmark reported 1.297× for facebook/layerskip-llama2-7B on summarization [14]. Assuming the dynamics is similar to what this project found, the exact speedup likely depends on hyper-paramters and the prompt. This project measured speedups between 1.45× to 1.58× for Llama 3.1 8B which is the closest model for comparison to Llama 2 7B.

This thesis and LayerSkip come with different sets of advantages and drawbacks. The strong feature with this project is that the model is frozen. This means that the upgrade is close to drop-in without any LLM retraining, but it also means that the output is the same as the stock model. When touching the parameters of the model, the LLM is changed and there is no guarantee that the output will be of the same quality as the original. This makes this a quick and no risk inference speedup. The advantage of the LayerSkip approach is that by allowing the LM model to change its parameters, there is likely more potential for speedup. The model can converge into a state where early-exits can be handled with less impact.

The LayerSkip paper reports that to train the Llama 2 7B for layer skipping via continual pretraining, 64 A100 80GB GPUs were used. The paper doesn't state the time needed for training. This is significantly more compute needed than the single RTX PRO 6000 that trained the HVC-bridge for Llama 3.1 8B in 34 minutes.

What is more useful therefore depends on the inference situation. However, unless large amounts of compute is available, using LayerSkip approach is likely inaccessible. The amount of compute to turn a large model into a LayerSkip version could also be significantly higher than for the Llama 2 7B model.

4.5.4 Intermediate representations and Tuned Lens

The HVC bridge is also related to existing research. A paper called *Eliciting Latent Predictions from Transformers with the Tuned Lens* [15] uses the similar idea of transforming a hidden vector to the output prediction space. It shows that intermediate states can have information about the next token prediction, but that the information might not be represented in the same geometry as the final layer. A learned transformation can make it easier to compare results between different layers.

As presented in the introduction, this thesis uses similar intuition to skip layers. By using a learned transformation, the intermediate result from an internal layer can be translated to the geometry of the entrance layer. The training results from this thesis seem to support the idea that a significant part of the degradation in generation quality is from geometry mismatch. Figures 17 and 19 show a drastic increase in top-1 match, from around 0% to the range of 60–70% by translating the geometry with the HVC bridge. An important difference to the Tuned Lens paper is that the HVC bridge also gets the previous final hidden vector as additional information.

4.5.5 Position of this thesis

The existing research has many of the individual components that this thesis uses. The self-speculation is similar to LayerSkip and the speculative decoding papers, the HVC bridge follows the same intuition as the Tuned Lens paper but used with some modifications, and the ANNH uses the technique proposed by FlashHead. The novelty of this thesis is the combination of these existing research domains, and implementing the ideas in a modified way to produce a complete inference system.

5

Conclusion

5.1 Summary

This thesis has investigated if an LLM can be transformed into a lightweight drafter for itself to produce a self-speculative setup that is more performant than normal inference. The key idea was to keep the LLM frozen and to have an inference drafter mode that uses approximations for both the body and the LM-head. The approximation for the body was to skip layers, and the approximation for the head was to use a FlashHead style approximate nearest neighbors head. To skip layers effectively, a mechanism here called HVC was used. This was a normed linear transformation that transformed the hidden vector from the representation of the exit layer to the representation of the entrance layer. The thesis has also investigated what ablation of skipped layers that is preferable to maximize the number of skipped layers while minimizing the damage to the generation quality.

The thesis found that an internal contiguous gap of skipped layers seemed to be preferable. It found that using fewer skipped layers made the quality higher, but inherently also increased the needed compute for the drafter. The chosen skipped layer ablation to maximize the self-speculative speedup was therefore an aggressive gap of (1,1), which means to skip all layers except the first and the last one. The thesis found an average speedup between 1.2× to 1.63× for all models, block sizes and prompt sets. The largest speedups were observed for larger models, for example Mistral 7B Instruct, 0.3v together with concrete prompts. The smaller speedups were observed for smaller models, such as Llama 3.2 1B Instruct, together with more open-ended prompts.

The thesis found that the self-speculative inference used approximately the same amount of memory as normal inference. Since the original frozen model is verifier, the output will also be exactly the same as normal inference up to what the selected floating point precision allows for.

5.2 Contributions

This thesis makes the following contributions:

1. It implements a systematic way to evaluate different ablations of skipped layers for an LLM.
2. It implements a FlashHead style ANNH that makes the operation of the LM-head significantly faster.
3. It implements training and inference code to use HVC to skip layers in the LM-body.
4. It implements a drafter inference mode for a frozen LLM where the LM-body skips a subset of its layers with HVC and the head uses ANNH.
5. It implements a self-speculative setup where an LLM is used both as verifier and drafter by switching between an approximated drafter inference mode and the original inference mode.
6. It implements a comprehensive benchmark for this self-speculative inference where skipped layers are used and optionally ANNH. The benchmark reports things like VRAM usage, acceptance rates, the computational split in the head, exact match rate and more.
7. It provides the complete implementation as open source on GitHub.

Bibliography

[1] OpenAI, "Introducing ChatGPT." [Online]. Available: https://openai.com/index/chatgpt/

[2] A. Chatterji *et al.*, "How People Use ChatGPT," technical report, 2025. [Online]. Available: https://cdn.openai.com/pdf/a253471f-8260-40c6-a2cc-aa93fe9f142e/economic-research-chatgpt-usage-paper.pdf

[3] W. Kwon *et al.*, "Efficient Memory Management for Large Language Model Serving with PagedAttention," *arXiv preprint arXiv:2309.06180*, 2023, [Online]. Available: https://arxiv.org/abs/2309.06180

[4] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," *arXiv preprint arXiv:2305.14314*, 2023, [Online]. Available: https://arxiv.org/abs/2305.14314

[5] W. Tranheden, S. Ahmed, D. Dubhashi, J. Matthiesen, and H. von Essen, "FlashHead: Efficient Drop-in Replacement for the Classification Head in Language Model Inference," *arXiv preprint arXiv:2603.14591*, 2026, doi: 10.48550/arXiv.2603.14591.

[6] Y. Leviathan, M. Kalman, and Y. Matias, "Fast Inference from Transformers via Speculative Decoding," in *Proceedings of the 40th International Conference on Machine Learning*, in Proceedings of Machine Learning Research, vol. 202. PMLR, 2023, pp. 19274–19286. [Online]. Available: https://proceedings.mlr.press/v202/leviathan23a.html

[7] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, "Accelerating Large Language Model Decoding with Speculative Sampling," *arXiv preprint arXiv:2302.01318*, 2023, [Online]. Available: https://arxiv.org/abs/2302.01318

[8] Hugging FaceTB, "Cosmopedia-100k." [Online]. Available: https://huggingface.co/datasets/HuggingFaceTB/cosmopedia-100k

[9] codelion, "FineWeb-Edu-1B." [Online]. Available: https://huggingface.co/datasets/codelion/fineweb-edu-1B

[10] MBZUAI, "LaMini-Instruction." [Online]. Available: https://huggingface.co/datasets/MBZUAI/LaMini-instruction

[11] flytech, "Python-Codes-25k." [Online]. Available: https://huggingface.co/datasets/flytech/python-codes-25k

[12] R. Eldan and Y. Li, "TinyStories." [Online]. Available: https://huggingface.co/datasets/roneneldan/TinyStories

[13] M. Elhoushi *et al.*, "LayerSkip: Enabling Early Exit Inference and Self-Speculative Decoding," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. doi: 10.18653/v1/2024.acl-long.681.

[14] A. R. Gosthipaty, M. Elhoushi, P. Cuenca, and V. Srivastav, "Faster Text Generation with Self-Speculative Decoding." [Online]. Available: https://huggingface.co/blog/layerskip

[15] N. Belrose *et al.*, "Eliciting Latent Predictions from Transformers with the Tuned Lens," *arXiv preprint arXiv:2303.08112*, 2023, doi: 10.48550/arXiv.2303.08112.