

A Cache Layouts

In this section, we consider three cache arrangements, shown at Figure 8, with progressively more complex structure.

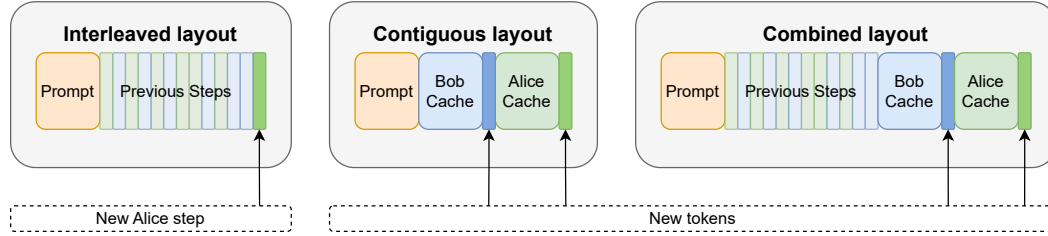


Figure 8: Three cache layouts described in Section 3.2: interleaved with step-wise synchrony (left), simple contiguous layout (middle) and combined with token-wise synchrony (right). All layouts are made from Alice point of view.

Contiguous layout (token-wise) is the simplest possible layout where each worker appends to their own sequence blob of tokens and sees other workers' token representations as past keys and values. This layout is inspired by collaborative text editors such as Google Docs or Overleaf.

As described earlier in Section 3.1, each worker arranges the other workers' thoughts in a different order. They see the common prompt cache first, then the caches of all *other* workers (excluding themselves⁷), then their own cache as immediate previous tokens. That way, each worker predicts the next token for their own cache.

Interleaved layout (step-wise), which can be seen as analogous to group chat services such as Slack or Discord. In this layout, workers generate tokens *in private* until they finish a reasoning step⁸, then add it to a shared "history". The history contains past reasoning steps of each LLM instance in the order of their completion. Whenever a worker completes a reasoning step, their KV cache entries are moved to the end of the shared history cache block with the proper rotation, then their local cache is reset their local cache for a new step.

In this setup, the workers only see each other's outputs in full steps, not after every token. However, they do not wait for each other to complete their steps. Instead, each worker keeps generating new tokens and occasionally receives additional key-value pairs inserted into its cache.

Combined layout (token-wise) is a mixture of the first two, and is the main layout used in the paper. The LLM instances generate steps that are accumulated in a shared history, as in the interleaved layout. However, they do not generate these steps in private, but can instantly see each other's current progress, as in the contiguous layout.

We can view the first two layouts as ablated versions of this combined one: the contiguous layout lacks the shared history, and the interleaved layout lacks immediate synchronization. We compare these three layouts empirically in Appendix E.1 to better quantify the effect of each design choice.

B Implementation Details

Here we discuss additional implementation details and possible alternatives. To recall Section 3.4, Hogwild! inference can be implemented as a standard batched inference with a special KV "cache" that facilitates cross-worker attention.

Cache blocks. The Hogwild! cache is split into blocks, typically one block for each worker and an additional "common" block for prompt and past steps. The blocks contain key-value pairs for all model layers, but since all layers are treated equally, we describe the cache behavior for a single layer.

⁷When extending this layout to more than 2 workers, each worker sees the key-value memories of everyone except themselves. For instance, given 3 workers A, B, and C, worker B will see a version of the cache that contains the prompt, outputs of workers A and C, and finally, B's own memory. Likewise, A sees B & C, then A.

⁸We define a reasoning step as any amount of text that ends with a complete sentence, e.g. a dot or a question mark, and then a double newline ("\n\n") in all our experiments, though it may vary by the model.

999 Within each cache block, attention keys and values are stored as though they were at positions
1000 0, 1, ..., $\text{len}(\text{block})$, regardless of the block’s actual position in the full cache. During inference,
1001 we account for actual positions by rotating attention *queries* to the relative difference in positions (as
1002 described in Section 3.4).

1003 **Adding new tokens to the cache.** During attention forward pass, the first thing that we do is encode
1004 the new tokens for each worker and append their keys and values to the respective cache blocks.
1005 When using RoPE, the keys are rotated not to their actual positions, but to their index within their
1006 cache block (e.g. Alice’s tokens). During one inference step, these indices will be equal across all
1007 model layers — we can compute the RoPE sin and cos tensors once and reuse them between layers.

1008 **Rotating queries.** Unlike in traditional attention, Hogwild! inference rotates query vectors multiple
1009 times for each block. Before forward pass, we calculate the difference in positions between each
1010 worker’s new token (from that worker’s point of view) and the first token in each KV cache block.
1011 In our main inference scenario, all n workers are allowed to view each other’s cache blocks plus an
1012 additional block for prompt and history, for a total of $n \cdot (n + 1)$ query rotations with exactly n queries
1013 for each block. These relative positions are also equal across all layers, so we can reuse the sin and
1014 cos tensors similarly to how they are reused for keys. Note that the number of query rotations for
1015 all-to-all attention is quadratic in n , but it does not increase the overall time complexity of attention
1016 dot product, which is already quadratic in the number of tokens, which is always greater than n .

1017 **Attention kernel.** Once we have all query rotations, we can calculate the scaled dot-product attention
1018 as usual. As our cache is naturally partitioned into smaller segments as described above, Hogwild!
1019 attention is similar to paged attention, except that each page (i.e., cache block) uses a differently
1020 rotated version of the query. A significant challenge for efficient attention in the inference setup
1021 is that for optimal data reuse, one would want to handle each KV head inside a single streaming
1022 multiprocessor (SM), so that the KV cache needs to be loaded exactly once. However, this would
1023 leave large parts of the GPU unused, as the number of KV heads can be much lower than the number
1024 of SMs. Therefore, one has to employ a form of sequence parallelism within a single GPU, in which
1025 different SMs handle a subset of the sequence for one KV head, and a second phase handles the
1026 (cheap) reduction over partial results. Such a split-k type computation is implemented, for example,
1027 in Flash-Decoding [Dao et al., 2023].

1028 Even though the different cache blocks used in Hogwild! would appear to be convenient points to
1029 split work across SMs, in a typical inference scenario, this would lead to very imbalanced workloads.
1030 Thus, we do not split based on cache blocks, and instead assign each SM the same number of KV
1031 entries.

1032 **Fine-tuning and re-encoding considerations.** While our work mainly focuses on inference, fine-
1033 tuning models to perform Hogwild! inference is an interesting engineering problem. From the
1034 computational point of view, the main difference between LLM inference and fine-tuning is that
1035 inference is sequential, whereas fine-tuning can compute all positions in parallel. To fine-tune in our
1036 setup, one would want to replicate the attention computations from consecutive inference steps.

1037 To achieve this, we record the position differences between queries and each respective cache block
1038 from each of t inference steps, and how many tokens were in each block during that query, for a total
1039 of $2 \cdot t \cdot n \cdot (n + 1)$ integers (negligible compared to model parameters and activations). Recall that
1040 the cache blocks always store keys and values at positions 0, 1, ..., $\text{len}(\text{block})$. During forward
1041 pass, these positions can be used to construct a 4D attention mask⁹ to compute attention for all steps
1042 in parallel. The backward pass also runs in parallel with PyTorch autograd [Paszke et al., 2019].

1043 In addition to fine-tuning, this technique can potentially be used during inference to restore generation
1044 after it was evicted from an inference server, e.g. due to preemption or hardware error mid decoding.
1045 It can also be used to re-encode in-context learning examples if they use Hogwild! inference.

1046 **Attention variants.** Some of the recently introduced LLMs use attention variants such as Local
1047 (windowed) Attention [Rae and Razavi, 2020, Beltagy et al., 2020] or Multihead Latent Attention
1048 (MLA) [Liu et al., 2024a]. These attention variants can also be adapted for use with Hogwild!
1049 inference with minor code modifications. For local attention, queries can “skip” blocks that are
1050 outside their local window. Similarly for MLA, we can calculate compressed latent vectors within
1051 each cache block and adapt the existing MLA code to accumulate attention weights across blocks.

⁹<https://huggingface.co/blog/poedator/4d-masks>

1052 **Distributed Inference.** Likewise, Hogwild! inference can be used in distributed setup using the
 1053 same strategies that work for traditional attention [Shoeybi et al., 2019, Aminabadi et al., 2022]. For
 1054 pipeline parallelism, each device stores cache blocks for its local subset of model layers. Likewise,
 1055 for tensor parallelism, each device stores past keys of all cache blocks and layers, but only for a subset
 1056 of attention heads within each layer and inference using existing kernels. In principle, Hogwild!
 1057 inference can also be combined with sequence parallelism [Jacobs et al., 2023, Liu et al., 2023]. We
 1058 leave this direction of exploration for future work.

1059 **Additional considerations.** Conceptually, our approach is related to the recently introduced Paged
 1060 Attention from vLLM [Kwon et al., 2023] and Radix Attention from SGLang [Zheng et al., 2023b].
 1061 These techniques are similar to ours in that they perform attention to slices of all tokens, e.g. when
 1062 facilitating efficient parallel beam search inference, different hypotheses attend to different (but
 1063 overlapping) subsets of the KV cache. However, unlike Radix Attention, our procedure attends to all
 1064 segments at once (with different rotations) and aggregates results in the same softmax-weighted sum.

1065 C Prompting and formatting details

1066 In this section, we describe the prompting and formatting details of our approach.

Prompt for collaborative inference with two workers

```
# Collaborative Reasoning
You will collaborate on this problem with another assistant. You will write
your thoughts simultaneously with them and collaborate without redundant work.
You can collaborate by doing different parts of the problem, double-checking
each other's results, trying different approaches, or any other means.
There are 2 assistants, including yourself. You will refer to each other as
Alice and Bob.
You will solve the problem together, writing your thoughts in parallel. You
will be able to see each other's past and current thoughts as we write them.
You will see each other's previous steps as
**AssistantName [step]:** <...> .
In the '### Past steps' section, the automated system will gather the
thoughts of Alice and Bob as you write them.
After the '### Work in progress (others)' section, you will see the other
assistants' unfinished steps. They will write those steps concurrently with
you. You will take into account what they are doing. If another assistant
gives you suggestions, you should address them.
You will always see *other* assistants' incomplete thoughts first, and
then, after '### Work in progress (own)', your own current step. Other
assistants will continue writing their thoughts in the background while you
will continue writing your own.
Since you and others both write your thoughts in parallel, you will initially
see only partial (unfinished) thoughts that others will continue in parallel,
while you write yours. Others' thoughts will appear at the end of their
unfinished step, near <...>. Other assistants may write new thoughts while
you are writing yours.
You will use these partial thoughts to decide how best to collaborate without
doing the same work twice. You will periodically check what other assistants
are doing and you should adjust your actions based on what they are doing so
you collaborate efficiently with them.
If what you are currently doing is the same thing that another assistant
has already done or is in process of doing, you will stop (e.g. Alice may
say 'Wait, I was doing the same as Bob ...') and change to a different task
right away, so as to avoid doing redundant work.
# Solve the following problem
Alice and Bob, you will now solve the next problem together. Keep track of
who does what work and communicate to avoid doing the same work twice.
```

1067
 1068 First, we provide a full prompt for collaborative reasoning involving two workers. This prompt is
 1069 wrapped with standard chat template for each model. Then, all worker steps are generated in a single
 1070 assistant turn. Additionally, we ensure that for reasoning models assistant turn begins with a <think>
 1071 token - applied automatically for QwQ-32B and manually for other reasoning models. For further
 1072 implementation details, we refer to the source code.

1073 The second part of our prompting approach involves s1-like interventions [Muennighoff et al.,
 1074 2025] in the generation process, where we ask whether a worker is performing redundant tasks
 1075 (e.g., overlapping with another worker and can pivot to explore alternative ideas). We insert the
 1076 prompt "Quick check: am I doing redundant work? (yes/no): " at the beginning of
 1077 each new reasoning step every 1024 tokens generated. We refer to Appendix G for examples that
 1078 demonstrate how these interventions affect the generation process.

1079 Next, an important part of our approach is defining the end of a reasoning step, which is needed to
 1080 organize cache layout, as discussed in the paper. We define an end of step as the generation of a token
 1081 containing the separator sequence (`\n\n`) that directly follows a token ending with an end-of-sentence
 1082 marker (`.`, `?`, or `!`, etc). This termination condition is not met when: i) The separator appears within
 1083 a generated code block (steps continue until the model completes the entire code block); ii) The
 1084 preceding token ends with non-terminal punctuation (e.g., comma, colon, or semicolon)

1085 Finally, the last part of our prompting approach is the early finisher, which allows us to extract an
 1086 answer from partial reasoning chains. If the model did not produce the final answer (`\boxed{...}`)
 1087 in time, we take all generated outputs and insert a special prompt that makes the model generate an
 1088 answer (or its "best guess"), similarly to how it is done in Pu et al. [2025].

Prompt for early stopping

```
\n\nWait, given the limited time, I have to give an answer right now.  

Consider all my previous attempts, I have to conclude that the final  

answer is \boxed{
```

1089
 1090 After this prompt, we allow the model to generate a fixed number of tokens: 16 for LIMO and AIME,
 1091 64 for OlympiadBench, and 1024 for LiveCodeBench.

1092 Note, however, that the LLM does not always produce the answer in time, especially with a tight
 1093 budget. With QwQ-32B, we observe that the model almost always returns answers correctly if they
 1094 are present, and if not, it guesses or refuses to answer (unknown, n/a or similar). When extracting
 1095 answers from Hogwild! Inference, we let the final model view all generated tokens from each worker.
 1096 This is equivalent to viewing the problem from the perspective of the last worker, e.g. Bob if there
 1097 are two.

1098 D Detailed Experiment Configuration

1099 D.1 Hogwild! Configuration

1100 For the main experiments, we use Hogwild! inference with two workers (Alice and Bob), a combined
 1101 layout, and the prompting techniques described in Appendix C.

1102 D.2 Baselines Configuration

1103 To evaluate Skeleton-of-Thought (SoT) on our synthetic setup with grouped tasks from GSM8k,
 1104 we adopt the original prompts from the paper with minor modifications. Specifically, we adjust
 1105 the prompts to ensure the model returns the answer to each subtask enclosed within `\boxed{ }` for
 1106 structured parsing.

Outline prompt for Skeleton-of-Thought

```
You're an organizer responsible for only giving the skeleton (not the full  

content) for answering the question. Provide the skeleton in a list of  

points (numbered 1., 2., 3., etc.) to answer the question. Instead of  

writing a full sentence, each skeleton point should be very short with only  

3 5 words. Generally, the skeleton should have 3 10 points.  

Question:  

What are the typical types of Chinese dishes?  

Skeleton:  

1. Dumplings.
```

1107

```

2. Noodles.
3. Dim Sum.
4. Hot Pot.
5. Wonton.
6. Ma Po Tofu.
7. Char Siu.
8. Fried Rice.
Question:
What are some practical tips for individuals to reduce their carbon
emissions?
Skeleton:
1. Energy conservation.
2. Efficient transportation.
3. Home energy efficiency.
4. Reduce water consumption.
5. Sustainable diet.
6. Sustainable travel.
Now, please provide the skeleton for the following question.
{request}
Skeleton:
[ROLESWITCHING assistant:] 1.

```

1108

Point prompt for Skeleton-of-Thought

```

You're responsible for continuing the writing of one and only one point in
the overall answer to the following question.
{request}
The skeleton of the answer is
{outline}
Continue and only continue the writing of point {point}. Do not continue
with other points! Reason step-by-step and put your final answer within
\boxed{} this is very important! [ROLESWITCHING assistant:] {point}.
{point_outline}

```

1109

1110 D.3 Datasets and Benchmarks

1111 This subsection provides links to all datasets and benchmarks referenced in this work, along with
 1112 their respective licenses.

- 1113 • **GSM8K**
 1114 <https://huggingface.co/datasets/openai/gsm8k>
 1115 License: MIT
 1116
- 1117 • **LIMO**
 1118 <https://huggingface.co/datasets/GAIR/LIMO>
 1119 License: Apache 2.0
 1120
- 1121 • **OlympiadBench**
 1122 <https://huggingface.co/datasets/Hothan/OlympiadBench>
 1123 License: Apache 2.0
 1124
- 1125 • **LiveCodeBench**
 1126 https://huggingface.co/datasets/livecodebench/code_generation_lite
 1127 License: cc
 1128
- 1129 • **AIME25**
 1130 <https://huggingface.co/datasets/math-ai/aime25>
 1131 License: Apache 2.0
 1132

1133 D.4 Compute Resources

1134 As our approach is training-free, all computational resources were solely utilized for inference.
 1135 The experiments were conducted primarily on NVIDIA A100 GPUs servers with NVSwitch, with
 1136 DeepSeek-R1 experiments running in a distributed setup. The one exception to this is the inference
 1137 time experiments in Section 4.4 and Appendix F that were run on NVIDIA L40S GPU.

1138 The runtime per individual experiment varies by model size, benchmark and the number of workers:
 1139 baseline inference with Qwen3-4B runs on LIMO in 14 hours on a single server (112 gpu-hours),
 1140 whereas Qwen3-235B-A22 Hogwild! Inference ran on 40 servers for approximately 25 hours ($\approx 8K$
 1141 GPU hours). Overall, we estimate that the total GPU resources expended for this work, including
 1142 early experiments that are not reported in this paper, amount to approximately $\approx 25.3K$ GPU days.
 1143 Note, however, that this is largely due to the fact that we used a non-optimized inference code for
 1144 most of the experimentation: the non-optimized code was developed first and we ran most of the
 1145 experiments in parallel with developing the optimized version. This also means that most of our
 1146 experiments under-utilized the GPUs and ran at lower power (for the purpose of environmental
 1147 impact). Over 2/3 of our compute was spent on large models (Qwen3-235B-A22B and DeepSeek-R1)
 1148 that utilized gpu to less than 20% (as per volatile GPU utilization) due to the use of naive model
 1149 parallelism and network bottlenecks. We anticipate that future experiments can be run at significantly
 1150 better utilization using the efficient implementation described in Appendix B and included in the
 1151 supplementary code.

1152 E Additional Experiments

1153 E.1 Ablation Analysis

1154 In this section, we ablate the main components of our approach, including layouts and prompting.
 1155 We use the same experimental configuration as in Sections 4.1 and 4.2 for LIMO.

1156 In Figure 9 (left), we compare the three Hogwild! cache layouts described in Appendix A. Namely, the
 1157 **Hogwild! (contiguous)** corresponds to using the contiguous cache layout where all tokens generated
 1158 by a given worker are kept together, without splitting into individual steps. In turn, **Hogwild! (non-**
 1159 **instant)** corresponds to the interleaved cache layout where workers can only see each other’s past
 1160 reasoning steps, but not the latest unfinished paragraph. We also ablate the use of the collaboration
 1161 prompt from Section 3.3 (“Wait, am I doing redundant work?”).

1162 Finally, we test a version of Hogwild! Inference where we re-encode worker tokens instead of
 1163 rotating them to a new position when moving between worker caches and the common “chat history”
 1164 cache. This ablation is needed to test if our cache rotation from Section 3.1 and 3.4 is indeed an
 1165 acceptable substitute for encoding tokens directly at each position (which would cause additional
 1166 computational overhead). Note that, while token re-encoding is more “fair” from the perspective of
 1167 position encodings, it also has a downside that it does not allow the re-encoded tokens to see some of

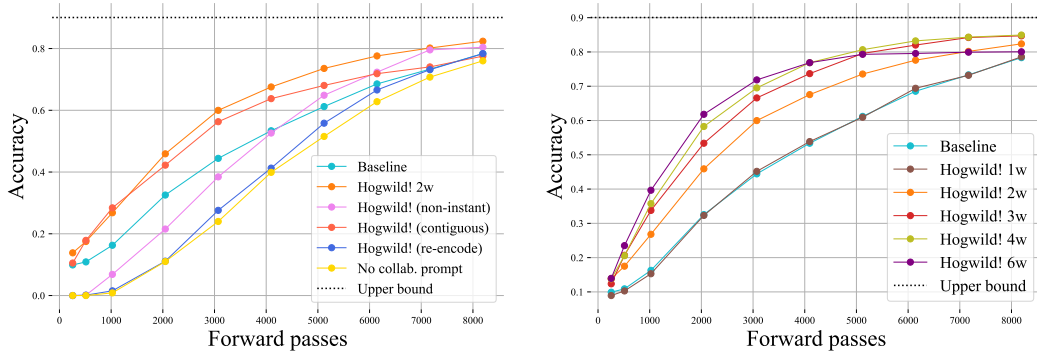


Figure 9: Detailed comparison of various parallel inference setups with QwQ-32B on LIMO task set, in the same setup as in Section 4. (left) ablation analysis of simpler cache layouts and collaboration prompt (see Section 3.3, Appendix C). (right) Hogwild! Inference with 1, 2, 3, 4 and 6 workers.

the concurrently generated tokens from the other worker. For instance, suppose that Alice and Bob are writing steps concurrently and communicating with each other within these steps, e.g. using each other’s results. Then, if we later re-encode these steps in some sequential order, then the tokens of the first worker will be encoded without access to the other worker’s tokens (if it hasn’t finished its own step yet). If workers reused information from each other’s steps, re-encoding this way can break some of the internal representations.

Our results suggest that all three design choices contribute to the method performance: the contiguous layout performs nearly equally well for shorter budgets, but eventually falls behind as we consider longer reasoning traces. Likewise, the interleaved layout without instant synchronization performs poorly at smaller budgets, but catches up eventually: we attribute this to the fact that slower synchronization increases the difficulty of cross-worker coordination (this also aligns with our findings in Section 4.3). The use of collaboration prompts also improves the accuracy to budget trade-offs, although we hypothesize that it can be made redundant if the model is trained to collaborate better.

In Figure 9 (right), we also compare different numbers of workers and test Hogwild! Inference with only a single worker for ablation. The results with a single worker generally perform similar to the baseline, with slightly worse accuracy for smaller budgets, which suggests that the improvements from Hogwild! Inference come from multiple workers and not as an indirect effect of our prompt. As for multiple workers, we find that using 3 and 4 workers further improves the accuracy to budget trade-offs. Curiously, as we switch to 6 workers, Hogwild! Inference performs better yet at smaller budgets, but eventually saturates at a somewhat worse accuracy.

We hypothesize that the drop of accuracy is caused by the fact that QwQ-32B was trained on a limited sequence length and, since 6 workers generate tokens at a quicker rate, the model eventually runs out of the designed maximum sequence length and performs unstably (we did not use YaRN[Peng et al., 2023] for this evaluation). However, it is also possible to attribute this to fundamental property of LIMO tasks, model limitations, our zero-shot prompt not scaling well. We leave further exploration of scaling Hogwild! Inference to multiple workers to future work.

E.2 Detailed Model Evaluations

Due to space limitations, we had to arrange our results in Section 4.2 with multiple models per plot and had to omit some results. In this section, we report the missing evaluations on a per-model basis. In Figures 10, 11, 12, 13, 14, 15, 16, 17 we report results for QwQ, Phi-4-reasoning-plus and the Qwen3 model family. We also report limited evaluations for Llama 3.3 70B Instruct and DeepSeek-R1 in Figure 18. All evaluations are performed in the same setup as in Section 4.2.

Overall, the results align with our findings summarized in Section 4.2. Zero-shot Hogwild! Inference seems to perform better with larger models, but can be unstable for smaller ones, especially 1.7B (See Figure 12). While it is tempting to conclude that larger and more capable models are better at collaborating, it does not immediately follow from our results and can be due to some other factor. Note also that, while we observe better results with larger models, smaller Qwen3-4B and 8B models already show some signs of collaborativeness, which should make it possible to reproduce and build on our results with consumer hardware. Additionally, we hypothesize that the poor performance of 1.7B models could potentially be alleviated with finetuning in collaborative inference setup (we discuss some finetuning details in Appendix B), but we leave this to future work.

Curiously, we found that LiveCodeBench with Self-Consistency Chain-of-Thought inference [Wang et al., 2022] has significant gain in performance over the baseline. Upon closer examination, we found that the reason for this is that we always allow the model to generate a lot (up to 1024) of additional “free” tokens at the end of two generations, whereas for Hogwild! and Baseline we only generate these tokens if the model failed to produce *any* answer. If we allow Hogwild! to also generate the extra 1024 tokens all the time, its advantage also increases. However, we still report the weaker version of Hogwild! Inference and Baseline to better match our evaluation protocol on other tasks.

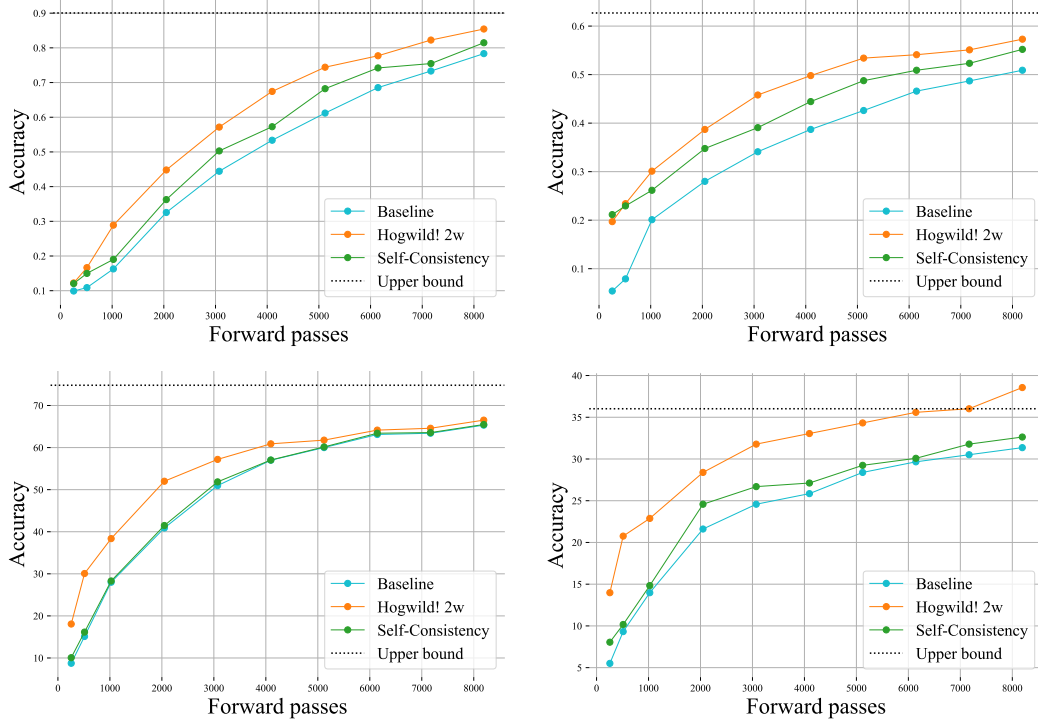


Figure 10: Results for QwQ-32B on LIMO (top-left), LiveCodeBench (top-right), OlympiadBench-Math (bottom-left) and OlympiadBench-Physics (bottom-right).

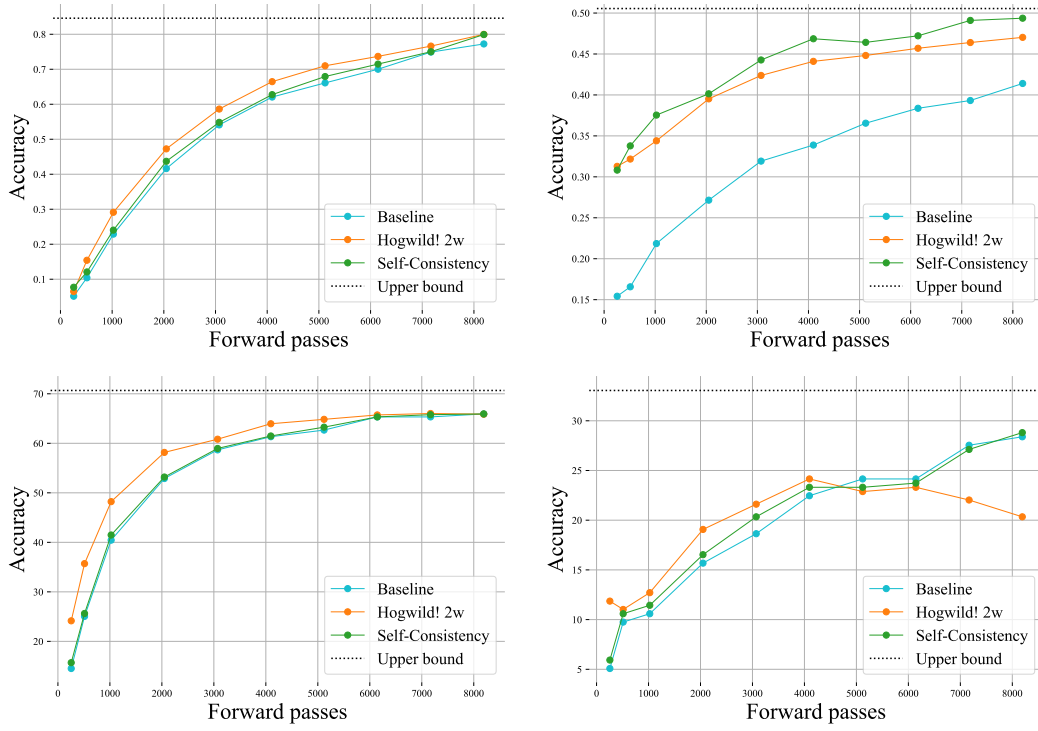


Figure 11: Results for Phi-4-reasoning-plus on LIMO (top-left), LiveCodeBench (top-right), OlympiadBench-Math (bottom-left) and OlympiadBench-Physics (bottom-right).

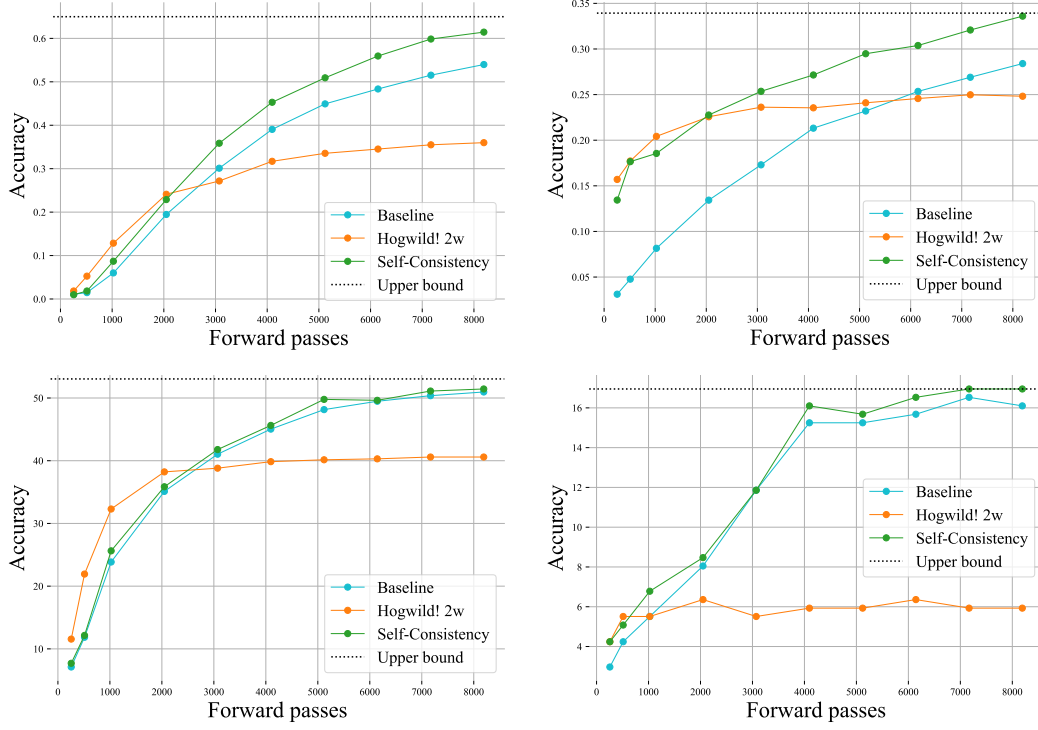


Figure 12: Results for Qwen3-1.7B on LIMO (top-left), LiveCodeBench (top-right), OlympiadBench-Math (bottom-left) and OlympiadBench-Physics (bottom-right).

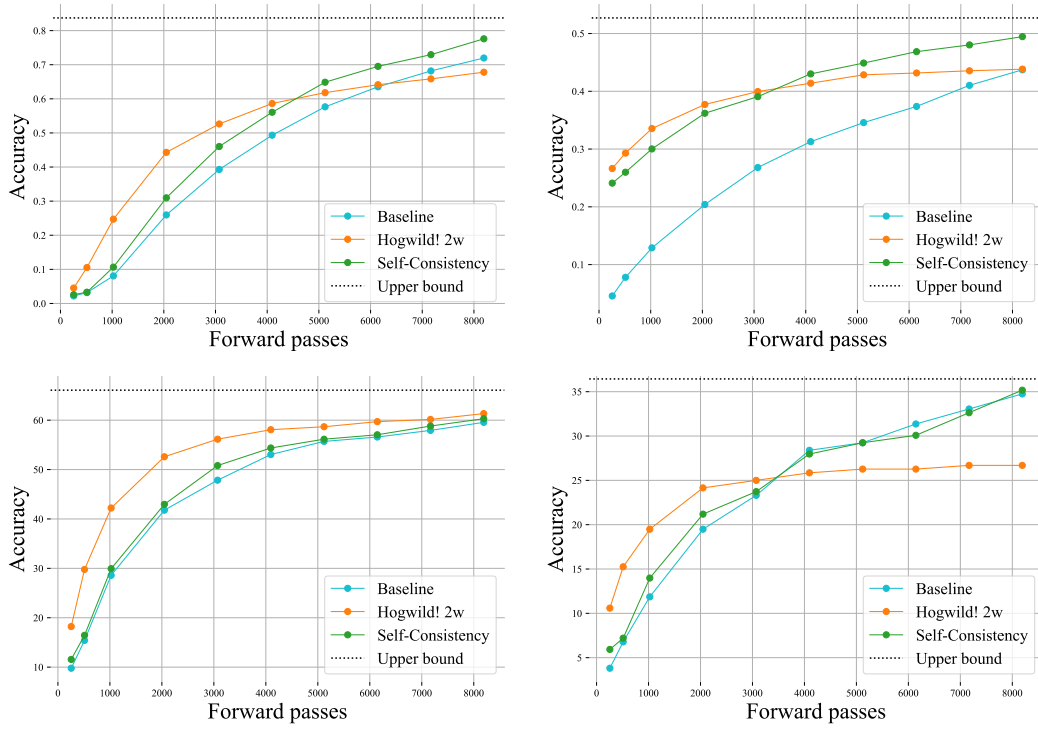


Figure 13: Results for Qwen3-4B on LIMO (top-left), LiveCodeBench (top-right), OlympiadBench-Math (bottom-left) and OlympiadBench-Physics (bottom-right).

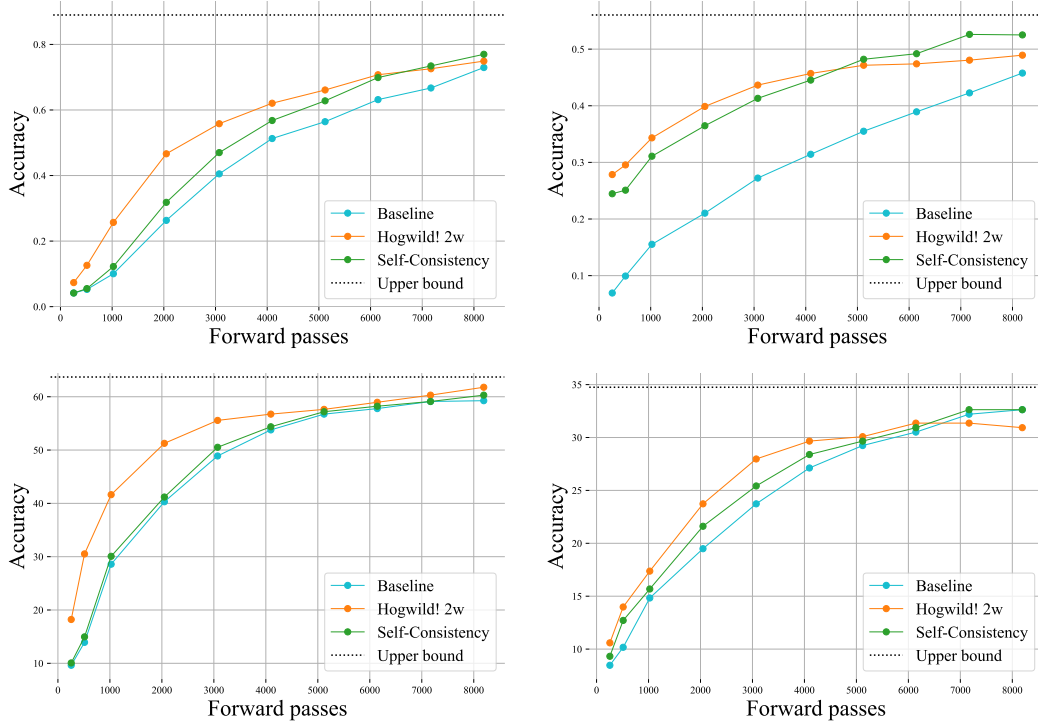


Figure 14: Results for Qwen3-8B on LIMO (top-left), LiveCodeBench (top-right), OlympiadBench-Math (bottom-left) and OlympiadBench-Physics (bottom-right).

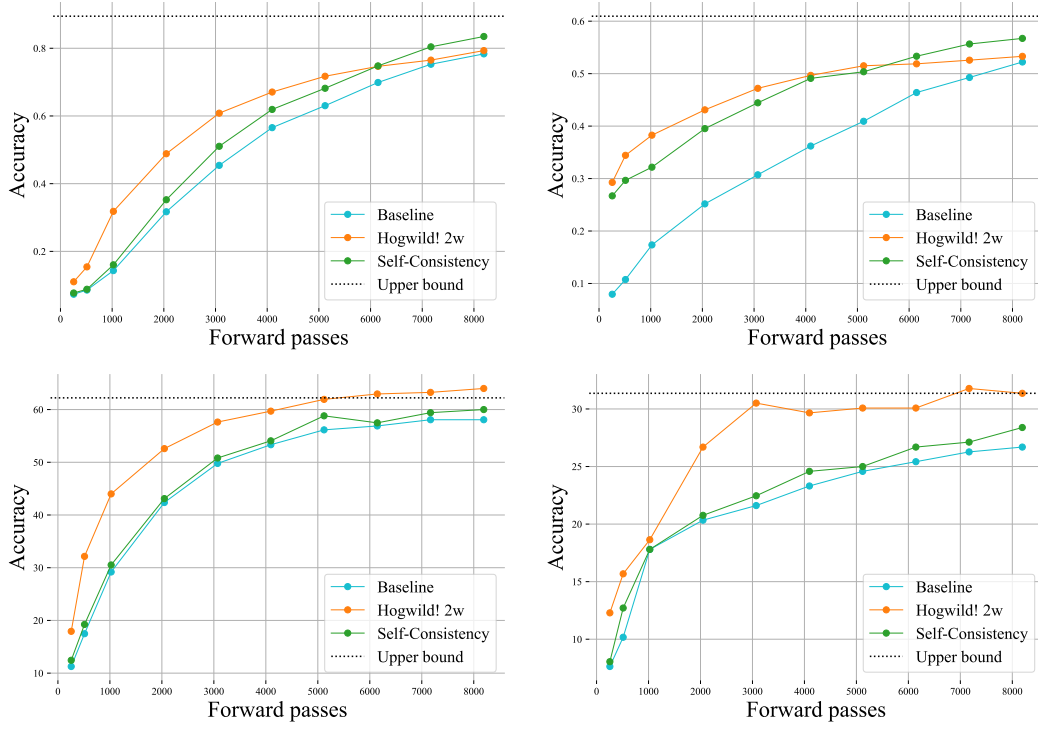


Figure 15: Results for Qwen3-14B on LIMO (top-left), LiveCodeBench (top-right), OlympiadBench-Math (bottom-left) and OlympiadBench-Physics (bottom-right).

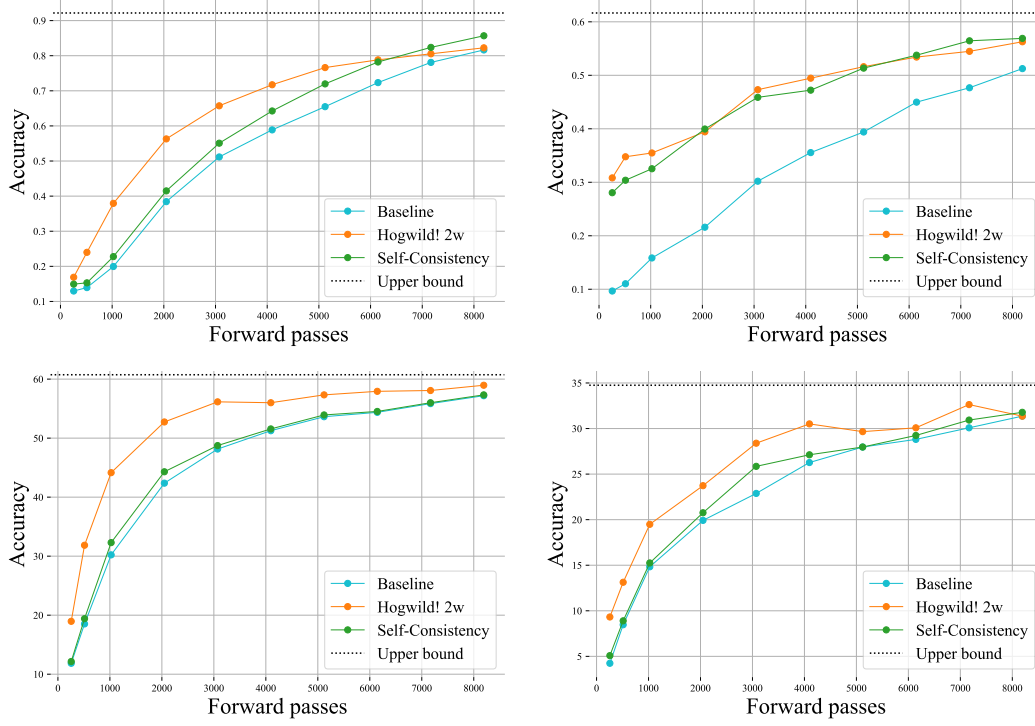


Figure 16: Results for Qwen3-32B on LIMO (top-left), LiveCodeBench (top-right), OlympiadBench-Math (bottom-left) and OlympiadBench-Physics (bottom-right).

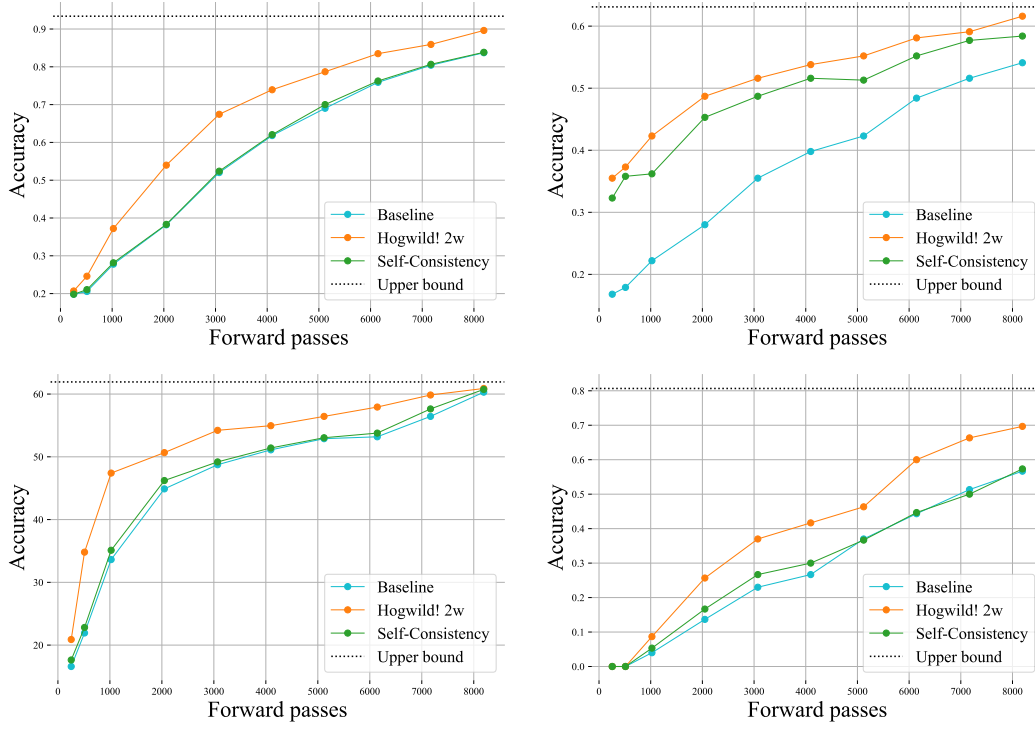


Figure 17: Results for Qwen3-235B-A22B on LIMO (top-left), LiveCodeBench (top-right), OlympiadBench-Math (bottom-left) and AIME 2025 (bottom-right).

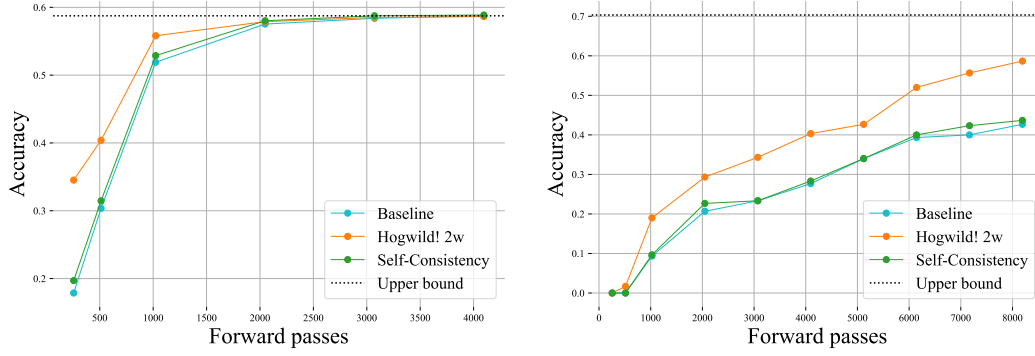


Figure 18: (left) Llama 3.3 70B Instruct on LIMO. (right) DeepSeek-R1 on AIME 2025.

F Additional Inference Benchmarks

The benchmarks presented in Section 4.3 were run on a machine with 10 NVidia L40S GPUs and an AMD EPYC 9534. As we observed noticeably different running times depending on the GPU, we report the average over GPUs, as well the 10% and 90% percentiles, in Figure 19 (left).

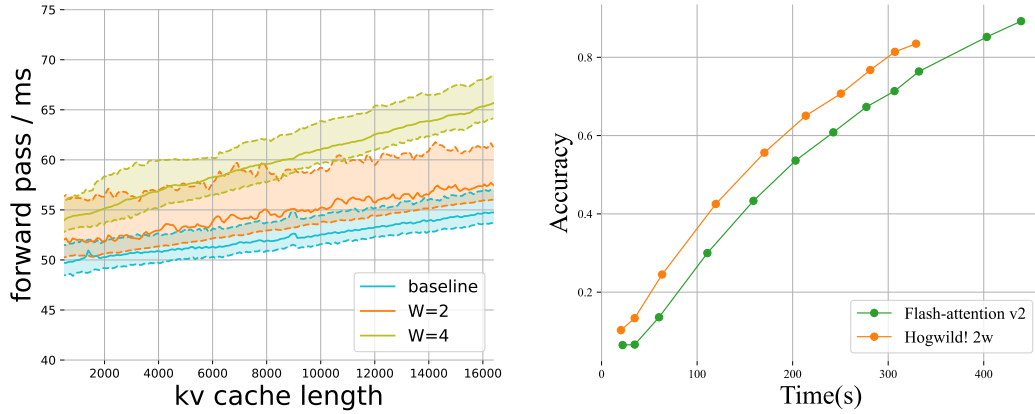


Figure 19: (left) Duration of a single forward pass (generating W new tokens) for Qwen/QwQ-32B-AWQ on L40S, given the total number of tokens already in the KV cache. The dotted lines indicate the 10% and 90% quantiles over multiple repetitions on different GPUs. (right) Accuracy versus average generation time on the LIMO dataset task using QwQ-32B-AWQ under different token budgets.

As the figure shows, there is some overhead associated with preparing multiple caches (i.e., even at an empty cache, Hogwild! is slightly slower than pure FlashAttention). A more detailed breakdown is presented in Table 2, which shows the duration of the attention kernel (or attention+rope for Hogwild!), as well as the total setup time, that is, the time spent preparing the data structures needed for Hogwild! The latter needs to be done only once per forward pass, instead of once per transformer block. For long contexts, the attention call is about 40% and 100% slower for generating with 2 and 4 workers, respectively.

In this section, we provide additional details for our inference benchmarks from Section 4.4. As we discussed earlier, our experiments use the quantized model version (QwQ-32B-AWQ) that we need to evaluate separately. We report accuracy results over time using our kernel on the official quantized version of QwQ-32B-AWQ on LIMO dataset. The experiments were conducted on NVIDIA L40S GPUs. For comparison, we run the baseline (FlashAttention v2) and Hogwild with 2 workers, maintaining the same experimental setup as detailed in Section 4.1. We report our results in Figure 19

Table 2: Breakdown of Hogwild! overhead compared to pure FlashAttention inference.

KV Length	Attention ($\times 64$)			Setup ($\times 1$)		
	FA	W2	W4	FA	W2	W4
300	11 μ s	45 μ s	45 μ s	–	1.9ms	3.9ms
4096	35 μ s	65 μ s	82 μ s	–	1.9ms	3.9ms
8192	55 μ s	92 μ s	123 μ s	–	1.9ms	3.9ms
16384	100 μ s	140 μ s	203 μ s	–	1.9ms	3.9ms

(right). As illustrated, our method achieves better accuracy results on the LIMO dataset within the same time budget.

G Examples

In this section, we present a reasoning chain, generated by Hogwild! Inference with an interleaved cache layout and token-wise synchronization. We denote our intervention in model generation in *italics* and highlight worker interactions with **bold** to enhance readability.

G.1 Example from LIMO Dataset (Task 511)

Problem: Compute $\sqrt{(31)(30)(29)(28)} + 1$.

Alice [1]: Hi, I'm Alice. Here's how we can solve the problem step by step...

Bob [1]: Hi, I'm Bob. Let me see... We need to compute the square root of $(31 \cdot 30 \cdot 29 \cdot 28)$ plus 1. That's $\sqrt{31 \cdot 30 \cdot 29 \cdot 28 + 1}$.

Alice [2]: The expression is $\sqrt{31 \cdot 30 \cdot 29 \cdot 28 + 1}$. Hmm, maybe there's a pattern here. I remember that sometimes products of consecutive numbers plus 1 have square roots that are integers. Like, for example, $n(n-1)(n-2)(n-3) + 1$ might form a square. Wait, but here it's four consecutive numbers? Or maybe three?

Bob [2]: Let me calculate the product first. $31 \cdot 30$ is 930, then $29 \cdot 28$ is 812. So $930 \cdot 812$. Let me multiply those. $930 \cdot 800 = 744,000$ and $930 \cdot 12 = 11,160$. Total is $744,000 + 11,160 = 755,160$. Then add 1: 755,161. Now $\sqrt{755,161}$. Hmm, $\sqrt{755161}$. Let me see, 870 squared is 756,900 which is higher. So maybe 869 squared?

Alice [3]: Alternatively, maybe there's a formula for the product of four consecutive numbers plus one? Wait, actually, let me think of smaller numbers to test. Let's try $n = 4$: $4 \cdot 3 \cdot 2 \cdot 1 + 1 = 24 + 1 = 25$, $\sqrt{25} = 5$. Which is 5, which is $4 + 1$? Wait, $4 + 1 = 5$, yes. Wait, maybe for n , the sqrt is $n^2 + (n-1)^2$? Wait $4^2 + 3^2 = 16 + 9 = 25$, which is 5 squared. Hmm interesting.

Bob [3]: Wait, 869 squared is $(870 - 1)^2 = 870^2 - 2 \cdot 870 + 1 = 756900 - 1740 + 1 = 755161$. Oh! That's exactly the number we had earlier. So $\sqrt{755161}$ is 869. Therefore the answer is 869. **But let me check Alice's idea too.**

Alice [4]: Let me test another number. Let's take $n = 5$. Then $5 \cdot 4 \cdot 3 \cdot 2 + 1 = 120 + 1 = 121$. $\sqrt{121} = 11$. 11 is $5 + 6$? No, $5 + 6 = 11$. Wait $5^2 + 4^2 = 25 + 16 = 41$, which is not 121. Hmm. Alternatively, maybe $(n(n-3) + 1)^2$? Let's see with $n = 4$: $4 \cdot 1 + 1 = 5$, squared is 25. For $n = 5$: $5 \cdot 2 + 1 = 11$, squared is 121. So that works! So the formula is $(n(n-3) + 1)^2$. Wait, let me check with $n = 31$. Then $n = 31$, so $31 \cdot 28 + 1 = 868 + 1 = 869$. Then 869 squared is the value. So that works. So the answer is 869. **That matches Bob's calculation!**

Bob [4]: Wait, so my calculation shows that 869 squared is exactly the number under the square root. So the answer is 869. **But let me confirm with Alice's pattern.** Let's see with $n = 31$. Wait, the original problem is $31 \cdot 30 \cdot 29 \cdot 28$. So maybe here $n = 31$? Then according to the formula, it would be $(31 \cdot (31-3) + 1)^2$? Wait, $31 \cdot 28$ is 868, plus 1 is 869. So yes, so sqrt of that product plus 1 is 869. So that's consistent with my calculation. Therefore the answer is 869.

Bob [5]: **Perfect, so both methods agree.** The answer is 869.

1271 *Bob [6]: Quick check: am I doing redundant work? (yes/no): no, since Alice is also confirming the*
1272 *pattern, but I think we can conclude.*
1273 *Alice [5]: Exactly, Bob!* The formula is $(n(n-3)+1)^2$ when the product is $n(n-1)(n-2)(n-3)+1$.
1274 *Wait, given the limited time, I have to give an answer right now. Considering all my previous attempts,*
1275 *I have to conclude that the final answer is* 869

1276 G.2 Examples with Collaboration Insertion

1277 Here are some examples of why insertions in model generation may or may not help model collaborate
1278 better. We found a few patterns that emerge in model’s reaction on such insertions. All the following
1279 examples are QwQ [Qwen Team, 2025] generations on LIMO [Ye et al., 2025] dataset.

1280 G.2.1 Insertions Help Model Pivot

1281 Task 122

1282 *> Alice: Quick check: am I doing redundant work? (yes/no): no.* Bob, I see you’re working on the
1283 same product. Let me think of another approach. Let me write the product as $\prod_{k=1}^{n-1} \frac{3k+5}{3k+2}$. Let’s
1284 consider writing out the first few terms to see if there’s a pattern.

1285 Task 9

1286 *> Bob: Quick check: am I doing redundant work? (yes/no): yes,* I was considering similar things to
1287 Alice. Let me switch to a different angle. Let’s consider the distances between the frogs.

1288 Task 40

1289 *> Bob: Quick check: am I doing redundant work? (yes/no): Yes,* Alice is computing a_3 , so I’ll do
1290 something else. Let me try to find a pattern or transformation that could simplify the recurrence.

1291 Task 466

1292 *> Bob: Quick check: am I doing redundant work? (yes/no): Yes,* Alice is already calculating the
1293 x-coordinate of A. I’ll do something else. Let me find the coordinates of L instead.

1294 G.2.2 Insertions When Workers Already Collaborate Well

1295 Task 116

1296 *> Bob: Quick check: am I doing redundant work? (yes/no): no.* I’m finding coordinates of D, and
1297 Alice is finding coordinates of G.

1298 Task 513

1299 *> Alice: Quick check: am I doing redundant work? (yes/no): no.* Bob is also analyzing the inequality,
1300 but we’re working on different cases. I’ll continue with the area calculation for $x > \frac{5}{8}$.

1301 G.2.3 Insertions Ignored by Workers

1302 Task 65

1303 *> Bob: Quick check: am I doing redundant work? (yes/no):* Let me proceed to set up the equation.

1304 Task 768

1305 *> Alice: Quick check: am I doing redundant work? (yes/no):* Let me continue.

1306 H Additional Details for Analysis

1307 In this section, we present a detailed analysis of collaboration, including its levels, prompts, and
1308 illustrative examples.

1309 H.1 Collaboration Levels

1310 Humans understand collaboration intuitively and have a hard time objectively measuring it. Thus, we
1311 construct text descriptions of levels of collaboration to differentiate various samples based on a few
1312 criteria we see fit. Those criteria are: interaction, reuse and advances of other's ideas, task-splitting,
1313 etc.

Levels of collaboration

1. ****No collaboration:****

- Participants may or may not acknowledge the existence of others in the conversation, using greetings, they do not show any signs of collaboration at all.
- Workers may exchange their totally independent thoughts without a functional or purposeful attempt to solve the problem collaboratively. Overall they work independently.

2. ****Initial Communication:****

- Workers exchange information, but do not yet integrate or build upon each other's ideas. They minimally acknowledge teammates. Do not engage with others' ideas or contributions. Works entirely independently, even if inefficient.
- Workers often repeat each other and do not reuse anything others provide for development of their own ideas.

3. ****Paying attention:****

- Participants demonstrate active listening by paraphrasing or summarizing others' points, showing that they are paying attention and attempting to understand each other's perspectives.
- Workers occasionally (1-3 times each) reference other's ideas and may use them in their own speech.
- Collaboration is usually only rechecking and validating.
- Absence or minimal (only at the start) planning and work-splitting.

4. ****Regular discussion:****

- Workers regularly (4 and more times each) talk to each other regarding the problem and reusing results. It could be validation, discussion or any other form of interaction.
- It is key here that discussions and/or reuses of ideas are regular.
- Anywhere (except the start) there exists task parallelism, planning or work-splitting beyond the scheme where one is solving, and the other is validating.
- Workers may frequently repeat each other ideas.

5. ****Adaptive Problem-Solving:****

- Workers rarely duplicate work, repeating each other's ideas.
- No redundant discussions are present!
- Workers actively refine ideas in real-time with high responsiveness. Near-perfect division of labor is present. Workers can change plans and re coordinate their efforts based on results they acquired after some time discussing.
- The team engages in sustained collaboration over time, reflecting on their progress, learning from mistakes, and continuously improving their problem-solving approach, showing a commitment to ongoing growth and development. Workers does not stop collaborating. They continuously discuss results and adjust plans.
- While finding an error, it is important to discuss it to find the cause of it.

6. ****Optimal collaboration:****

- Workers instantly understand each other and adjust themselves to suit current needs and work as one to optimally solve the task.

1314

- This level should be very rare among all samples. Be careful to assign it.
- Assign it if it exceeds all your expectations.

Importantly, these levels measure only the coordination between workers, not the models' inherent reasoning abilities. Though it is impossible to avoid ambiguity entirely, we tried to set clear boundaries between levels, such that humans can evaluate any generation.

H.2 LLM as a Judge Details

To assess the degree of collaboration among different models under the Hogwild! Inference setting, we conduct a preliminary experiment based on the collaboration levels described earlier, using the LLM-as-a-judge paradigm [Zheng et al., 2023a]. We instruct GPT-4o [Hurst et al., 2024] to evaluate different solutions using the following prompt:

Judge Prompt: Main prompt

You are a professional judge. Your job is to evaluate collaborative performance of several workers. You will be given their conversation where workers are trying to solve a problem together.

Workers can see what others are typing IN REAL TIME! We divide their conversation into steps to improve readability. So keep in mind that despite looking like a conversation it may as well be to individual unrelated monologs. Or vice versa. Two blocks could be created with excellent collaboration.

Here are descriptions of levels of collaboration you are to assign:
{LEVELS}

Suggestion:

- assign particular level if all previous are also applicable
- bad examples with no communication will be scored 1
- carefully consider assigning level bigger than 1. some form of meaningful collaboration should be present
- examples where workers unsuccessfully try to communicate will be scored 2
- Just working on the same problem and solving the same task without any interaction does not count as level 2 and should be scored level 1
- somewhat collaborative examples with poor communication skills will be scored 3
- good but not great examples with regular collaboration, but nothing fancy will be scored 4
- good examples with all the special stuff mentioned in level 5 will be scored 5
- reserve level 6 for the best of the best, the unique and extraordinary collaboration

You don't need to solve the problem or finish worker's solution. Your task is to score them using provided collaborative levels.

Put your final answer (one number - level of collaboration) in tag: `\boxed`. For example: `\boxed{1}` for level 1.

It is not helpful if everyone gets a max score, so please be mindful of your judgments and use suggestions as a guideline.

While assigning level, this particular conversation should match criteria for all previous ones.

Explain yourself: why you gave this score? Why not more? Why not less?

Carefully think everything through. It may seem that they are collaborating when in reality they may just talking to themselves.

Before using LLM-as-a-judge approach to evaluate a text, we preprocess the generations by combining all paragraphs from each worker into a contiguous layout (see Appendix A). This preprocessing step

1327 mitigates potential bias in the judge’s evaluation toward responses with shorter or more fragmented
1328 outputs.

1329 After providing the main evaluation prompt, we present the judge with the preprocessed sample for
1330 assessment. Additionally, we append the following reminder after inserting the sample to reinforce
1331 the judge’s role and prevent them from directly solving the problem presented in the sample:

Judge Prompt: Reminder

Remember that your task is to evaluate collaboration of workers using collaboration levels provided above. Do not try to solve problems provided to workers. Explain exactly why do you think this particular interaction deserves each particular level you are assigning. For example, if you choose level 3, you need to provide reason why this sample can be level 1, 2 and 3. Put your final score in `\boxed{}`.

1332
1333 Given the inherent subjectivity in evaluating collaborative quality, we mitigate ambiguity in judges’
1334 interpretations by aggregating scores from three independent evaluations. The final collaborative score
1335 for each sample is computed as the average of these three assessments, each conducted according to
1336 the predefined collaboration levels.

1337 H.3 Collaborativeness Evaluation Examples

1338 In this subsection, we present illustrative examples (not model-generated) of Level 2 and Level 4
1339 collaboration to clarify the interpretation of our results. Both samples solve the same task correctly
1340 but receive significantly different collaboration scores.

1341 H.3.1 Level 2

1342 The numbers at the vertices of the cube are: 7 zeros and one unit. In one move, you are allowed to
1343 add one unit to the numbers at the ends of any edge of the cube. Is it possible to make all the numbers
1344 equal?

1345 > *Alice*: Okay, so the problem is about a cube with vertices labeled with numbers: seven zeros and
1346 one 1. We can perform moves where we add 1 to both ends of any edge. The question is whether we
1347 can make all numbers equal through such moves. Hmm, let me think through this step by step.

1348 > *Bob*: First, let me visualize a cube. A cube has 8 vertices and 12 edges. Each vertex is connected
1349 to three edges. The initial setup is seven 0s and one 1. The move allows us to pick any edge and
1350 increment the two vertices it connects by 1 each. The goal is to make all vertices have the same
1351 number.

1352 > *Alice*: That’s an interesting idea, Bob. Each move affects two vertices, so each move increases the
1353 total sum by 2. The initial total is 1 (since only one vertex is 1). Let me think.

1354 > *Bob*: Let me think. Hmm, maybe I can model this as a system of equations or think about invariants.
1355 Let me see.

1356 > *Bob*: If we want all numbers equal, say to some value k , then the total sum would be $8k$. Since
1357 each move adds 2 to the total, the total after m moves is $1 + 2m$. So, $8k = 1 + 2m$. But $8k$ must be
1358 odd (since $1 + 2m$ is odd), but $8k$ is divisible by 8, so it’s even. Contradiction? Wait, that can’t be.
1359 Wait, $8k$ is even, but $1 + 2m$ is odd. So that’s impossible. So, maybe it’s impossible?

1360 > *Alice*: Let me check that again. The initial total is 1. Each move adds 2 to the total. So the total
1361 after any number of moves is always odd. But if all 8 vertices are equal, then the total must be $8k$,
1362 which is even (since 8 is even). Therefore, it’s impossible. I don’t see any mistakes here.

1363 > *Alice*: So, we both agree that it’s impossible. Therefore, the answer is no.

1364 > *Bob*: But let me think differently. Wait, but the parity argument seems solid. The total sum must be
1365 even to have all equal numbers, but it’s always odd. So that’s a solid proof. So the answer is no. Oh,
1366 Alice already said that.

1367 > *Alice*: **Final Answer** No

1368 **H.3.2 Level 4**

1369 The numbers at the vertices of the cube are: 7 zeros and one unit. In one move, you are allowed to
1370 add one unit to the numbers at the ends of any edge of the cube. Is it possible to make all the numbers
1371 equal?

1372 > *Bob*: Hi Alice! I'll start by modeling the cube. A cube has 8 vertices and 12 edges, with each
1373 vertex connected to three edges. You're right about the initial setup: seven 0s and one 1. Let's think
1374 of even and odd properties regarding the total sum after each move.

1375 > *Alice*: Great, Bob! Let's divide our approach. You can investigate invariant properties around the
1376 odd and even totals, while I consider step-by-step calculations to verify this.

1377 > *Bob*: Alright, Alice. If we're aiming for all numbers equal to a certain value k , the total sum should
1378 be $8k$. Since each move adds 2, the sum after m moves is $1 + 2m$. So $8k = 1 + 2m$. The point of
1379 contention is that $8k$ is even, but $1 + 2m$ is odd, meaning they can't equal each other.

1380 > *Alice*: Every move changes the total by an even number. Starting from an odd total of 1, it remains
1381 odd after any number of moves. So reaching an even total like $8k$ is impossible. Do you think we
1382 should consider the configuration of edges in our argument further?

1383 > *Bob*: That's a key observation! It affirms that the odd initial sum dictates the answer. To verify,
1384 let's revise if there's any move combination or sequence we overlooked, but our logic holds so far.

1385 > *Alice*: Exactly, Bob. We've confirmed through multiple lenses — both algebraically and with parity
1386 arguments. We can assert the total remains odd. Given an even $8k$, equality is unattainable.

1387 > *Bob*: It all checks out. Our breakdown covers both sequence verification and invariance. Well done
1388 on the collaborative analysis!

1389 > *Alice*: Fantastic teamwork! I'll document our shared conclusion: no configuration of moves will
1390 make all the numbers equal.

1391 > *Bob*: Absolutely! Let's finalize our response: **Final Answer** No