Hydragen: High-Throughput LLM Inference with Shared Prefixes

Jordan Juravsky [* 1] Bradley Brown [* 2] Ryan Ehrlich [* 3] Daniel Y. Fu [1] Christopher Ré [1] Azalia Mirhoseini [1]

Abstract

Transformer-based large language models (LLMs) are now deployed to hundreds of millions of users. LLM inference is commonly performed on batches of sequences that share a prefix, such as few-shot examples or a chatbot system prompt. Decoding in this large-batch setting can be bottlenecked by the attention operation, which reads large key-value (KV) caches from memory and computes inefficient matrix-vector products for every sequence in the batch. In this work, we introduce Hydragen, a hardware-aware exact implementation of attention with shared prefixes. Hydragen computes attention over the shared prefix and unique suffixes separately. This decomposition enables efficient prefix attention by batching queries together across sequences, reducing redundant memory reads and enabling the use of hardware-friendly matrix multiplications. Our method can improve end-to-end LLM throughput by up to 32x against competitive baselines, with speedup growing with the batch size and shared prefix length. Hydragen also enables the use of very long shared contexts: with a high batch size, increasing the prefix length from 1K to 16K tokens decreases Hydragen throughput by less than 15%, while the throughput of baselines drops by over 90%. Hydragen generalizes beyond simple prefix-suffix decomposition and can be applied to tree-based prompt sharing patterns, allowing us to further reduce inference time on competitive programming problems by 55%.

1. Introduction

Text generation on batches of sequences is a common setting for LLM inference. In many real-world use cases, sequences in a batch share a common prefix. Examples

*Equal contribution [1]Stanford University [2]University of Oxford [3]University of Waterloo. Correspondence to: Jordan Juravsky <jbj@stanford.edu>, Bradley Brown <bradley.brown@cs.ox.ac.uk>.
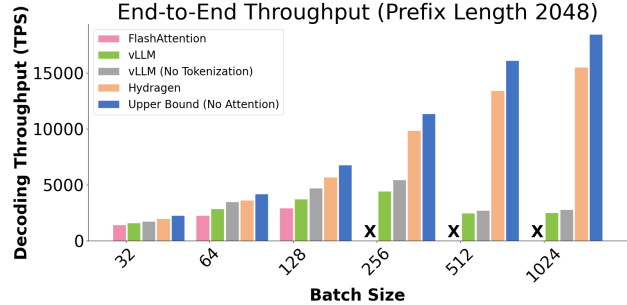
Preprint.

Figure 1. End-to-end decoding throughput in tokens per second (TPS) with CodeLlama-13b when generating multiple completions from a prompt containing 2048 tokens. An "x" indicates that FlashAttention does not have enough memory to run. As the batch size grows, Hydragen achieves a significantly higher throughput than vLLM baselines. Throughput with Hydragen always remains within 50% of the upper bound where attention is entirely removed from the model. Details are in Section 4.1.

include a chatbot serving many users with shared system instructions (Figure 2 left), an assistant model using a few-shot prompt for solving domain-specific tasks (Brown et al., 2020), and competitive programming systems that sample many candidate solutions for a single problem (Li et al., 2022). As transformer-based LLMs (Vaswani et al., 2023) are deployed at increasingly large scales (Malik, 2023), improving their efficiency with shared prefixes can have a significant impact. In this work, we use a hardware-aware perspective to analyze and optimize this inference setting.

Shared prefixes create overlaps in the attention keys and values across sequences, presenting an opportunity for specialized optimization. Existing work (Kwon et al., 2023) identifies that naive KV caching leads to redundant storage of the prefix's keys and values, and addresses this redundancy with a paged memory management strategy. While this optimization can significantly reduce GPU memory consumption, it does little to affect the speed of computing attention, which can often bottleneck end-to-end throughput with large batches. Since each sequence in the batch has a distinct (albeit overlapping) KV cache but only a single attention query when decoding, existing attention implementations like FlashAttention (Dao et al., 2022; Dao, 2023) and PagedAttention (Kwon et al., 2023) compute attention by performing many independent matrix-vector products. This
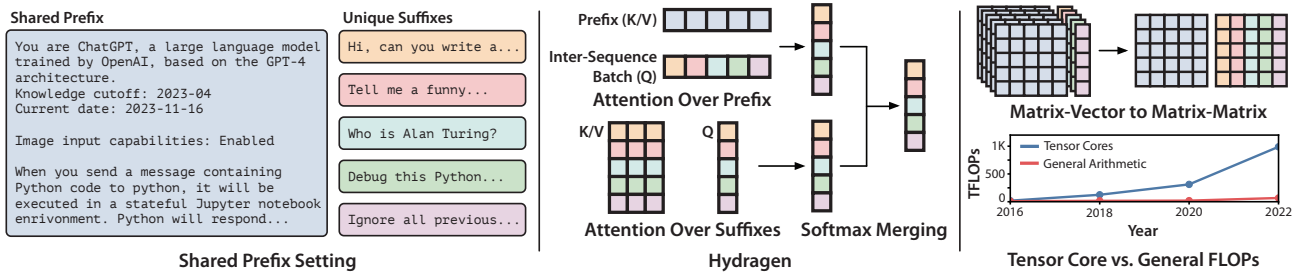
Figure 2. Left: An example LLM inference scenario where a chatbot model processes many sequences that share a large shared prefix (the system prompt). Middle: An overview of Hydragen, where overall attention is decomposed into attention over the shared prefix (batched across all queries in a batch) and attention over the remaining suffixes (independent across sequences, as is normally done). Top Right: Hydragen's attention decomposition allows many matrix vector products to be replaced with fewer matrix-matrix products. Bottom Right: Using matrix-matrix products is particularly important as GPUs dedicate an increasingly large ratio of their total FLOPs to tensor cores that are specialized in matrix multiplication.

approach is memory-bound when the KV cache is large, and moreover does not use hardware-friendly matrix multiplications. Both of these characteristics lead to poor performance on modern GPUs. Across successive hardware generations, GPU computational capability has improved at a significantly faster rate than memory bandwidth. Additionally, an increasingly large fraction of total GPU floating-point operations (FLOPs) are only available when using tensor cores, a specialized hardware feature that is dedicated to performing matrix-matrix products and not matrix-vector products (Figure 2 bottom right).

In this paper, we demonstrate that shared prefixes enable more than just memory savings, and can additionally be used to improve decoding throughput. We identify that FlashAttention and PagedAttention redundantly read the prefix's keys and values from GPU memory when computing attention, regardless of whether the prefix is redundantly stored. In order to eliminate these redundant reads, we present Hydragen, an exact implementation of attention that is specialized for shared prefixes (Figure 2 middle). Hydragen decomposes full-sequence attention into separate attention computations over the prefix and suffixes. These sub-computations can be cheaply combined to recover the overall attention result (Section 3.1). With attention decomposition, Hydragen is able to efficiently compute attention over the prefix by batching together attention queries across sequences (Section 3.2). This inter-sequence batching replaces many matrix-vector products with fewer matrix-matrix products (Figure 2 top right), reducing redundant reads of the prefix and enabling the use of tensor cores.

Experimentally, we find that Hydragen can significantly improve LLM throughput in large-batch settings with shared prefixes. In end-to-end benchmarks, Hydragen increases the throughput of CodeLlama-13b (Rozière et al., 2023) by up to 32x over vLLM (Kwon et al., 2023), a high-performance inference framework that avoids redundant prefix storage

but not redundant prefix reads. The attention operation in isolation can be accelerated by over 16x using Hydragen when compared to a state-of-the-art FlashAttention baseline, with benefits increasing as the batch size and shared prefix length grow. We also demonstrate that Hydragen's efficient processing of shared prefixes can influence algorithmic decisions on how to use LLMs most effectively. With a large batch size, Hydragen allows the shared prefix to grow from 1K tokens to 16K tokens with less than a 15% throughput penalty whereas vLLM throughput decreases by over 90%. On long document question answering tasks, we show that Hydragen can process 512 questions in less time than it takes a FlashAttention baseline to process 64 (Section 4.3). Finally, we demonstrate that Hydragen's attention decomposition and batching apply to more general patterns of prompt sharing than a single prefix-suffix split. When solving APPS competitive programming problems (Hendrycks et al., 2021), where two levels of prompt sharing occur, we apply Hydragen hierarchically to maximize sharing and reduce evaluation time by an additional 55% over a single-level of prompt sharing (Section 4.4).

2. Background

2.1. Hardware Efficiency Considerations

GPU Performance Bottlenecks: GPUs possess a limited number of processors for performing computation and a limited amount of bandwidth for transferring data between processors and memory. When a program running on a GPU is bottlenecked waiting for compute units to finish processing, it can be classified as compute-bound. Alternatively, memory-bound programs are bottlenecked accessing GPU memory. To summarize a program's use of hardware resources, we can calculate its arithmetic intensity, defined as the ratio between the total number of arithmetic operations performed divided by the total number of bytes transferred. Higher arithmetic intensities imply a greater use of compu-

[handwritten note: VLLM avoids redundant KV-cache storage. But does not avoid redundant KV cache reads.]

tational resources relative to memory bandwidth.

Batching: Batching is a common optimization that can increase an operation's arithmetic intensity and reduce memory bottlenecks. Consider the example of computing matrix-vector products. To compute one product, each element of the input matrix is read from memory but used in only a single multiply-accumulate. Therefore, the arithmetic intensity of the operation is low, and is memory-bound on GPUs. However, if many matrix-vector products need to be computed using the same matrix, we can batch the operations together into a single matrix-matrix product. In the batched operation, the cost of reading the input matrix is amortized over the batch of vectors. Each element of the input matrix is now used for many multiply-accumulates, increasing the arithmetic intensity of the overall operation and improving hardware utilization.

Tensor Cores: Modern GPUs (and other AI accelerators) are designed with specialized units for efficiently computing matrix multiplications. Effectively using these resources can be crucial for achieving good overall performance; on GPUs, tensor cores dedicated to matrix multiplications can compute over 10x more floating-point operations per second than the rest of the GPU. This further motivates batching matrix-vector products into matrix-matrix products.

2.2. Attention and LLM Inference

The focus of this work is optimizing attention in transformer-based LLMs. Scaled-dot-product (SDP) attention operates on a sequence of queries $Q \in \mathbb{R}^{N_q \times d}$, keys $K \in \mathbb{R}^{N_{kv} \times d}$, and values $V \in \mathbb{R}^{N_{kv} \times d}$, and is defined as:

$$\text{SDP}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V \qquad (1)$$

We are particularly interested in the performance characteristics of attention during LLM text generation. Generation begins with a prefill stage that processes the starting sequence of tokens that the LLM will complete. The prefill phase encodes the entire prompt in parallel using a single transformer forward pass. Therefore, during prefix attention we have $N_q = N_{kv} \gg 1$ and as a result the multiplications involving K^T and V are hardware-friendly matrix multiplications. After the prefill stage, completion tokens are iteratively decoded from the model, with one decoding step producing one new token and requiring one forward pass. Decoding is accelerated by the use of a KV cache, which stores the attention keys and values of all previous tokens in the sequence. The KV cache avoids the needed for reprocessing the entire sequence during every decoding step, and instead only the most recent token is passed through the model. However, this leads to an attention computation

where $N_q = 1$ while $N_{kv} \gg 1$, making the multiplications with K^T and V matrix vector products. Attention during decoding is therefore memory-bound and does not use tensor cores.

2.3. Batched Inference

LLM inference throughput can be increased by generating text for a batch of sequences in parallel. With batched decoding, each forward pass of the model processes the most recent token from many sequences instead of only one. This batching increases the arithmetic intensity of transformer components such as the multilayer perceptron (MLP) blocks, achieving better hardware utilization and therefore increasing overall throughput. However, batched text generation does not increase the intensity of the attention operation, since every sequence has a distinct key and value matrix. Therefore, while operations such as linear layers can be implemented with efficient matrix multiplications during batched decoding, attention instead must compute many independent matrix-vector products. With large batch sizes or long sequence lengths, computing attention becomes increasingly expensive relative to rest of the transformer, decreasing throughput. Additionally, the storage footprint of the KV cache in GPU memory can exceed that of the model parameters when the batch size is large, imposing constraints on the maximum number of sequences that can be simultaneously processed.

2.4. Shared Prefixes

In this paper, we investigate improving the throughput of batched text generation when the sequences in the batch share a common prefix. This scenario lends itself to specialized optimizations because shared prefixes lead to overlap in the key and value matrices of sequences in the batch. The causal attention mask in LLMs results in each token's activations only being influenced by previous tokens in the sequence. Therefore, if multiple sequences share a common prefix, the keys and values corresponding to the prefix tokens will be identical across sequences.

The key-value overlap introduced by shared prefixes presents two distinct directions for improving the inference process described in Section 2.3. Firstly, naive batched inference stores the KV cache separately for every sequence, leading to redundant storage of the prefix key and value vectors. Existing work has identified this redundancy and proposed an elegant virtual memory system to eliminate duplicate storage (Kwon et al., 2023).

In this work, we identify a new opportunity to optimize the attention operation itself. When GPU kernels compute attention for each sequence in the batch using independent matrix-vector products, the prefix keys and values are repeatedly read from GPU memory, regardless of whether they are

stored redundantly or not. We now propose an alternative approach to computing attention, which can simultaneously eliminate these redundant reads while enabling the use of tensor cores.

3. Hydragen: Efficient Attention with Shared Prefixes

We introduce Hydragen, an exact implementation of attention that is optimized for shared prefixes. Hydragen is a combination of two techniques:

1. **Attention Decomposition:** We split full-sequence attention into separate attention computations over the shared prefix and unique suffixes that can be cheaply combined to recover the full attention result.

2. **Inter-Sequence Batching:** We efficiently compute attention over the prefix by batching together attention queries across sequences.

Attention decomposition allows us to isolate overlapping portions of the batch's key and value matrices, while inter-sequence batching exploits this overlap by replacing many matrix-vector products with a single matrix-matrix product. Pseudocode implementing Hydragen attention is provided in Appendix B.

3.1. Decomposing Attention Across Subsequences

As discussed in Section 2.4, sequences that share a common prefix have partially overlapping keys and values when computing attention. Our goal is to separate this computation with partial overlap into two separate operations: attention over the shared prefix, where there is total key-value overlap, and attention over unique suffixes, where there is no overlap.

Consider the general case where our keys K and values V are partitioned across the sequence (row) dimension into:

$$K = K_1 || K_2 \qquad (2)$$
$$V = V_1 || V_2 \qquad (3)$$

with $||$ denoting concatenation along the row axis. We wish to avoid directly computing our desired quantity $\text{SDP}(Q, K, V)$, and instead calculate this value using the results of the sub-computations $\text{SDP}(Q, K_1, V_1)$ and $\text{SDP}(Q, K_2, V_2)$.

The challenge in partitioning attention is with the softmax operation, since the softmax denominator is calculated by summing over all exponentiated attention scores in the sequence. In order to combine our sub-computations, we use a denominator rescaling trick inspired by FlashAttention's blocked softmax computation (Dao et al., 2022). When computing $\text{SDP}(Q, K_1, V_1)$ and $\text{SDP}(Q, K_2, V_2)$, we additionally compute and store the log-sum-exp (LSE) of the attention scores (equivalently, the log of the softmax denominator):

$$\text{LSE}(Q, K) = \log\left(\text{sum}\left(\exp\left(\frac{QK^T}{\sqrt{d}}\right), \dim = 1\right)\right) \qquad (4)$$

Given the two partitioned attention outputs and their LSEs, we can calculate our final result $\text{SDP}(Q, K, V)$ by computing the full-sequence softmax denominator and rescaling the attention outputs accordingly:

$$\frac{\text{SDP}(Q, K_1, V_1)\, e^{\text{LSE}(Q,K_1)} + \text{SDP}(Q, K_2, V_2)\, e^{\text{LSE}(Q,K_2)}}{e^{\text{LSE}(Q,K_1)} + e^{\text{LSE}(Q,K_2)}} \qquad (5)$$

We provide more details on this formula in Appendix A.

3.2. Inter-Sequence Batched Prefix Attention

With attention decomposition, we are able to compute attention over the prefix as a standalone operation for every sequence. While this decomposition does not improve performance on its own (in fact, it introduces additional work in order to combine sub-computation outputs), it can allow us to compute prefix attention much more efficiently over a batch of sequences.

Queries do not affect each other when computing attention, therefore if two sets of queries attend over identical keys and values, they can be merged into a single attention operation with a larger number of queries. With attention decomposition, this case now applies to each sequence's attention operation over the shared prefix. Since the prefix's keys and values across sequences are identical, we can batch every sequence's query vector together into one attention operation over a single sequence. Importantly, this batching significantly raises N_q and the arithmetic intensity of prefix attention, replacing many separate matrix-vector products with a single matrix-matrix product. By replacing multiple independent attention computations over the prefix with a single batched operation, we can reduce the number of times that the prefix KV cache is read from GPU memory. Additionally, we can now use tensor cores during prefix attention and significantly improve hardware utilization.

Note that we are unable to apply inter-sequence batching when computing attention over suffixes, since the keys and values in each sequence's suffix are not identical. Suffix
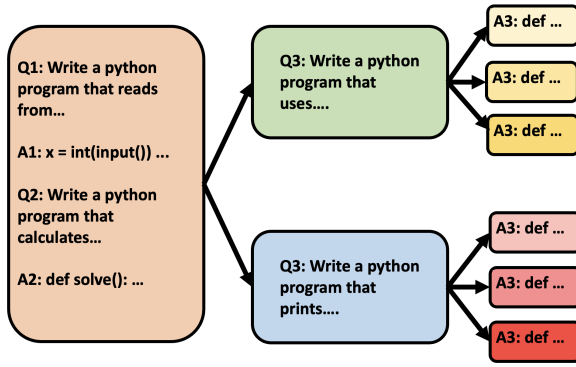
Figure 3. An example of a batch of sequences with a hierarchical sharing pattern. This diagram depicts the setting of Section 4.4, which solves competitive programming problems using a few-shot prompt and by sampling many candidate solutions per problem. The few-shot prompt (orange) is globally shared across all sequences in the batch. However, the descriptions of each problem (green and blue) are only shared across the candidate solutions corresponding to that problem.

attention is therefore computed normally, with a single query per sequence.

3.3. Hierarchical Sharing

So far, we have focused on the setting where all sequences in the batch share a common starting subsequence followed by suffixes that are entirely distinct from one another. However, this excludes other forms of sharing that appear in important use cases. Sequences in the batch may not all start with a global prefix, and instead the batch may be divided into groups of overlapping sequences. Additionally, sharing may be more fine-grained than a simple prefix-suffix decomposition, with the overlap between sequences forming a tree structure where each node contains a token sequence that is shared by all descendants (see Figure 3 for an example). These forms of sharing are increasingly relevant as LLMs are applied in more complicated inference/search algorithms (Yao et al., 2023; Besta et al., 2023; Ning et al., 2023).

Hydragen naturally generalizes to these richer forms of sharing as well. To apply Hydragen to a tree of sequences, we replace attention decomposition over the prefix and suffix into attention decomposition at every vertex in the tree. We can then use inter-sequence batching across levels of the tree, so that the keys and values associated with one node in the tree are shared across the queries of all descendant nodes.

3.4. Estimating Throughput Improvements with Hydragen

Hydragen significantly improves the efficiency of attention with shared prefixes relative to approaches that compute attention independently for every sequence (Section 4.2). However, translating this targeted efficiency into end-to-end throughput improvements depends strongly on the details of the inference setting being considered. In general, in order for Hydragen to meaningfully improve decoding speed in a particular setting, attention must be a major contributor to decoding time. For example, with small batch sizes or short sequence lengths, decoding speed is often bottlenecked not by attention, but by reading the parameters of the model from GPU memory. The benefits of Hydragen in this scenario will therefore be minimal. Similarly, given a fixed batch size and sequence length, we expect Hydragen to improve throughput more on a model that uses multi-headed attention than a similarly-sized model that uses multi-query attention (Shazeer, 2019) or grouped-query attention (Ainslie et al., 2023) in order to reduce the size of the KV cache. However, reducing the KV cache size allows for a larger batch size to fit within GPU memory constraints, which can further increase the speedup of using Hydragen.

As discussed in Section 2.3, the cost of attention becomes disproportionately high as the batch size grows, since the arithmetic intensity of most transformer operations improve while attention remains memory-bound. Hydragen greatly improves the hardware utilization of attention, making the comparison of attention FLOPs to other model FLOPs more useful when determining the maximum achievable speedup. In several experiments in Section 4, we include a "No Attention" baseline that only runs the non-attention components of the transformer in order to establish an upper bound for attainable throughput.

Another important consideration when predicting the benefits of Hydragen is the relative number of prefix (i.e. shared) tokens compared to suffix (i.e. unshared) tokens. Since Hydragen makes no optimizations to attention over suffixes, long suffixes can decrease generation throughput. We explore the impact of suffix length on attention speed in Section 4.2.

3.5. Implementation

We implement Hydragen for the Llama family of models (Touvron et al., 2023a;b; Rozière et al., 2023). We highlight that our implementation is simple: we use no custom CUDA code and write Hydragen entirely in PyTorch[1] plus calls to a fast attention primitive. This contrasts with more sophisticated algorithms like PagedAttention, which require bespoke GPU kernels to read from and update the paged KV cache. We believe that Hydragen's simplicity will allow it to be easily ported to other hardware platforms such as TPUs, which also have hardware dedicated to fast matrix multi-

---

[1]For non-hierarchical inputs, we've also written a Triton kernel to combine softmax denominators, which can improve attention performance by about 10-20%.
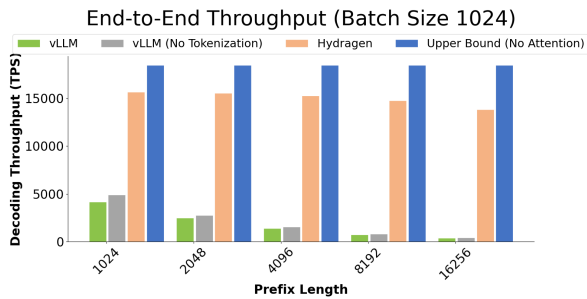
Figure 4. Comparing decoding throughput of CodeLlama-13b between Hydragen, vLLM (with and without tokenization), and "No Attention", where the attention operation is removed from the model to demonstrate the throughput ceiling. In this scenario where the batch size is fixed, Hydragen improves throughput by up to 32x over the best baseline, with speedups increasing with prefix length.

plications. In our implementation, we use version 2.3.6 of the `flash-attn` package when attending over the prefix, and a Triton kernel from `xformers` when attending over the suffix. The second kernel allows us to have changing sequence lengths in the suffix KV cache across decoding steps while still adhering to the constraints required to use CUDA graphs.

4. Experiments

4.1. End-To-End Throughput

We benchmark end-to-end LLM throughput in the setting where many completions are sampled from a single prompt. This is a common technique for improving a model's ability at solving complex problems such as math or competitive programming (Rozière et al., 2023; Li et al., 2022). Our benchmarks evaluate Hydragen against four baselines:

1. **FlashAttention:** We perform inference without any shared prefix optimizations, as if all sequences in the batch were fully distinct. We compute full-sequence attention using the Triton kernel that Hydragen uses for suffix attention, and otherwise use the same codebase as Hydragen. This baseline redundantly stores the prefix's keys and values for every sequence in the batch, causing this method to run out of memory quickly.

2. **vLLM:** We use version 0.2.7 of the `vllm` package, which uses the PagedAttention algorithm. vLLM avoids redundant storage of the prefix, allowing much larger batch sizes to be tested. Additionally, because of this non-redundant storage, PagedAttention can achieve a higher GPU cache hit rate when reading the prefix, reducing the cost of redundant reads.

3. **vLLM without Detokenization:** We disable incremen-

tal detokenization in vLLM (accomplished by commenting out one line in the vLLM codebase), which we observed to improve throughput.

4. **No Attention:** We skip all self-attention computations in the transformer. This (functionally incorrect) baseline provides a throughput ceiling and helps to illustrate the cost of different attention implementations relative to the rest of the transformer. Note that the query, key, value, and output projections in the attention block are still performed.

We run our benchmarks on CodeLlama-13b (Rozière et al., 2023) and distribute the model with tensor parallelism across eight A100-40GB GPUs, in order to have enough GPU memory to store the KV cache with large batch sizes. In Figure 1, we fix the prefix length to 2048 and sweep over the batch size while generating 128 tokens per completion. When the batch size is small, non-attention operations contribute significantly to decoding time, with all methods reaching at least half of the throughput of no-attention upper bound. At these low batch sizes, Hydragen, the vLLM baselines, and the FlashAttention baselines have similar throughputs. However, as the batch size grows and attention over the prefix becomes increasingly expensive, Hydragen begins to significantly outperform the other baselines.

In Figure 4, we run a similar experiment, except now we hold the batch size constant at 1024 and sweep over the shared prefix length. The throughput of vLLM significantly decreases as the prefix grows, from just under 5k tokens/second with a prefix length of 1024 to less than 500 tokens/second with a prefix length of 16256. However, with Hydragen, throughput is largely unaffected despite the prefix growing by over 15k tokens. Moreover, across all sequence lengths tested, Hydragen throughput is always within 70% of the no-attention ceiling. We perform more in-depth sweeps over different models, prefix lengths, batch sizes, and numbers of generated tokens in Appendix C.1 - for smaller models and shorter completions lengths, Hydragen's speedup can exceed 50x. Additional evaluation setup details are in Appendix D.1.

4.2. Microbenchmarking Attention

We also perform more granular benchmarks comparing Hydragen attention against FlashAttention, in order to more precisely demonstrate the performance characteristics of our method. Our microbenchmarks run on a single A100-40GB using eight query attention heads, one key and value head, and a head dimension of 128 (matching the setting of CodeLlama-34b when distributed across eight GPUs). We sweep over different batch sizes, prefix lengths, and suffix lengths, reporting our results in Figure 5. Our microbenchmarks corroborate our end-to-end measurements
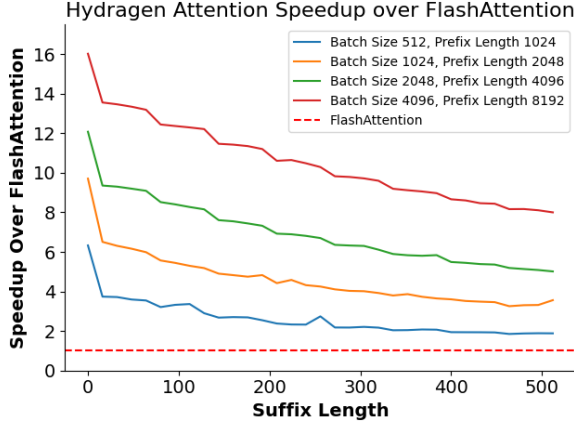
Figure 5. Measuring the speedup of Hydragen attention over FlashAttention across various batch sizes, shared prefix lengths and suffix lengths on an A100-40GB GPU. We see that Hydragen results in faster inference in all cases, in particular when the ratio of shared length to unique length is high and the batch size is large. We observe even larger performance gains when running on an L40S (a GPU with a higher compute-to-bandwidth ratio than an A100), shown in in Figure 8.

from Figures 1 and 4 that the speedup with Hydragen increases as the batch size and prefix lengths grow. However, the microbenchmarks also highlight the significant impact of the suffix length on inference time. Hydragen computes attention over suffixes using memory-bound FlashAttention (without inter-sequence batching). As the suffix lengths grow, reading this portion of the KV cache becomes an increasingly significant contributor to total execution time. When generating text using Hydragen, this means that the first tokens decoded by the model are generated the fastest, with throughput decreasing over time as the lengths of completions (and therefore the lengths of suffixes) grow.

These microbenchmarks are also influenced by the hardware platform that they are run on. GPUs with a higher ratio of compute to memory bandwidth benefit more from Hydragen eliminating memory bottlenecks when attending over the prefix. We report results on other GPUs in Appendix C.2 and provide more evaluation details in Appendix D.2.

4.3. Long Document Question Answering

Additionally, we explore the performance of Hydragen on workloads involving very long documents. We construct a document by embedding synthetic facts into an excerpt of War and Peace. Our shared prefix, totalling 19947 tokens, contains both the document as well as five few-shot examples of question/answer pairs. Our benchmark evaluates Yi-6B-200k (01-ai, 2023) on its ability to answer questions based on the embedded facts. We run this benchmark using Hydragen as well as with our FlashAttention and no-
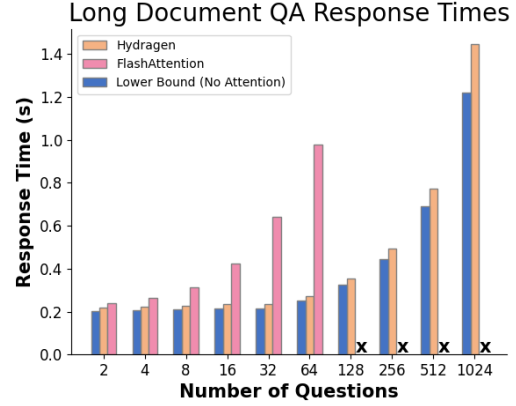

Figure 6. Time to answer questions about a 19947 token-long document. An "x" indicates that FlashAttention does not have enough memory to run. Time to process prefix excluded.

attention baselines. Results are reported in Figure 6. We observe that processing time for the FlashAttention baseline rapidly grows far beyond the time of the no-attention baseline, highlighting how attention is the dominant operation for this configuration. Meanwhile, Hydragen's processing time always remains within 75% of the no-attention optimum. Notably, Hydragen can process 512 questions in less time than it takes the FlashAttention baseline to process 64 questions. We provide additional evaluation details in Appendix D.3.

4.4. Hierarchical Sharing in Competitive Programming

We lastly demonstrate the benefits of applying Hydragen to a setting with hierarchical sharing. Competitive programming was a motivating application for developing Hydragen, since current state-of-the-art systems sample a very large number of candidate programs from prompts that can contain thousands of tokens (Li et al., 2022; Rozière et al., 2023). In this experiment, we benchmark the total time required to evaluate CodeLlama-7b on 120 problems from the APPS dataset (Hendrycks et al., 2021) using a two-shot prompt and 128 candidate programs per problem. When multiple problems are processed in a single batch, prompt overlap occurs across two levels: the few-shot prompt is shared across all sequences in the batch, while each problem's description is shared across all of the problem's candidate solutions (see Figure 3).

We run this benchmark using two methods:

1. **Single-Level Hydragen:** We use a single-level version of Hydragen to share the few-shot prompt across all sequences in the batch, and not share problem descriptions across candidate solutions. This leads to redundant storage of the problem description across all
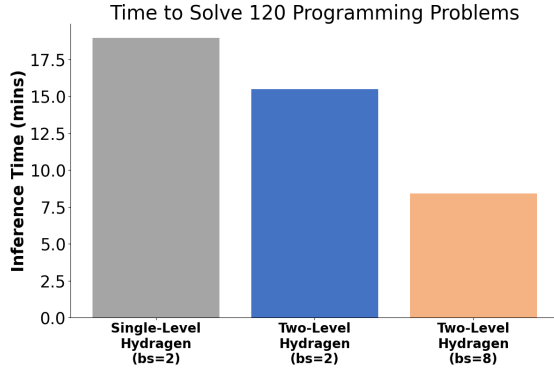
Figure 7. Time to run inference over a dataset of 120 APPS coding problems, sampling 128 solutions per problem with two few-shot examples. The batch size refers to the number of problems processed simultaneously. Across all Hydragen runs (both single and two-level), the few-shot prompt is shared across all sequences. By additionally sharing the problem description across generated candidate solutions, two-level Hydragen decreases overall inference time by an extra 55% over single-level Hydragen.

candidate solutions, reducing the maximum batch size that can be used.

2. **Two-Level Hydragen:** We apply Hydragen across both levels of prompt overlap. This has the dual benefits of improving attention efficiency (by increasing the degree of sharing) as well as avoiding redundant storage, which allows us to increase the batch size used for evaluation. We avoid conflating these benefits by evaluating two-level Hydragen twice: once with the same batch size used for single-level Hydragen, and once with an enlarged batch size.

We report our results in Figure 7. We see that even when the batch size is held constant, adding a second level of sharing to Hydragen can improve attention efficiency and decrease dataset evaluation time by 18%. Furthermore, the memory saved due to not redundantly storing the problem description allows us to increase the batch size, which in turn results in an additional 45% reduction in evaluation time. We provide additional evaluation details in Appendix D.4.

5. Discussion

In this work we introduced Hydragen, an exact, hardware-aware implementation of attention for batches of sequences that share common prefixes. Our method separates attention over shared prefixes from attention over unique suffixes. This allows us to batch attention queries across sequences when attending over the prefix, reducing redundant memory reads and enabling the use of tensor cores. Hydragen can improve LLM throughput in scenarios where attention is a significant contributor to decoding time, with the greatest

speedup occurring when the batch size is large, the shared prefix lengths are long, and the unique suffix lengths are short.

We emphasize that Hydragen is an optimization that can be applied as part of a larger inference framework, and is not intended to be an end-to-end inference solution. Our proof-of-concept implementation of Hydragen requires that the user specifies where sharing occurs across the input sequences. We are excited about future work that incorporates Hydragen into systems that continuously receive requests and schedule sequences for generation (Yu et al., 2022; Kwon et al., 2023), such that overlapping sequences can be dynamically identified and exploited.

We hope that our work inspires new LLM algorithms that leverage efficient handling of shared prefixes. Hydragen's ability to significantly expand the shared prefix without a significant throughput penalty should allow models to be provided with much more context than was previously practical. Moreover, we hope that Hydragen's ability to generalize to tree-shaped sharing patterns can assist with research that uses LLMs to explore many possible solutions before deciding on a final output.

6. Related Work

Transformers and Language Models: The transformer architecture has enabled significant improvements in state-of-the-art language models (Vaswani et al., 2023). A defining feature of transformers is that their performance consistently improves when scaling up data and model size (Radford et al., 2019; Brown et al., 2020; Chowdhery et al., 2022; Hoffmann et al., 2022; OpenAI, 2023). LLM-powered assistants such as ChatGPT have been widely adopted and are currently used by over a hundred million users (Malik, 2023), motivating research into how these models can be deployed more efficiently.

KV Cache Management: Managing large KV caches is a challenge when deploying LLMs. MQA (Shazeer, 2019) and GQA (Ainslie et al., 2023) modify the transformer architecture in order to reduce the KV cache size. These techniques decrease the number of key-value attention heads and assign multiple query heads to a single key-value head. Alternative approaches operate at a systems level, dynamically moving keys and values between GPU memory, CPU memory, and disk (Sheng et al., 2023; Aminabadi et al., 2022; HuggingFace, 2022). vLLM (Kwon et al., 2023) introduces a virtual paging system that enables fine-grained KV cache management. This virtual paging can also avoid redundant storage of a prefix's keys and values. Concurrent with our work, SGLang (Zheng et al., 2023) also investigates and optimizes inference with sequences that have complicated prompt sharing patterns. Their RadixAttention algorithm

dynamically scans incoming requests to find the largest subsequence that has already been processed, avoiding the recomputation of overlapping keys and values. Importantly, while both vLLM and RadixAttention avoid redundant storage of overlapping keys and values, they do not optimize the attention computation itself.

Hardware-Aware Algorithms: Algorithms that leverage an understanding of the underlying hardware platform can significantly improve device utilization. Hardware-awareness has significantly improved the efficiency of the attention operation (Rabe & Staats, 2022; Dao et al., 2022; Dao, 2023), reducing the memory requirements from $O(N^2)$ to $O(N)$ while improving execution time by avoiding redundant memory transfers. In addition to improving input-output (IO) transfers, many GPU-aware algorithms (including Hydragen) focus on leveraging tensor cores (Fu et al., 2023), which can achieve over 10x more FLOPs per second than the rest of the GPU.

LLM Algorithms: Recent work has demonstrated that LLM capabilities can be improved when many potential solutions are explored when solving a problem. Self-consistency (Wang et al., 2023) improves performance on arithmetic reasoning tasks by sampling many solutions to a single problem and using a majority-voting protocol. On competitive programming problems, LLMs perform substantially better when many different attempts to a problem are sampled (Rozière et al., 2023). AlphaCode (Li et al., 2022), a state-of-the-art competitive programming system, samples as many as a million programs to solve a single problem. Tree-of-Thoughts (Yao et al., 2023) introduces an explicit tree-based search algorithm for solving problems that can be decomposed into discrete decision points. All of these scenarios involve performing batched text generation with overlapping prefixes, which Hydragen is specifically optimized for.

References

01-ai. Yi, 2023. URL https://github.com/01-ai/Yi.git. Accessed: 2024-02-01.

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 646–660. IEEE Computer Society, 2022.

Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Gianinazzi, L., Gajda, J., Lehmann, T., Podstawski, M., Niewiadomski, H., Nyczyk, P., and Hoefler, T. Graph of thoughts: Solving elaborate problems with large language models, 2023.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways, 2022.

Dao, T. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

Fu, D. Y., Kumbong, H., Nguyen, E., and Ré, C. Flashfft-conv: Efficient convolutions for long sequences with tensor cores, 2023.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J. W., Vinyals, O., and Sifre, L. Training compute-optimal large language models, 2022.

HuggingFace. Hugging face accelerate. https://huggingface.co/docs/accelerate/index, 2022.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science*, 378 (6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Malik, A. Openai's chatgpt now has 100 million weekly active users. https://techcrunch.com/2023/11/06/openais-chatgpt-now-has-100-million-weekly-active-users/, 2023. Accessed: 2023-11-06.

Ning, X., Lin, Z., Zhou, Z., Wang, Z., Yang, H., and Wang, Y. Skeleton-of-thought: Large language models can do parallel decoding, 2023.

OpenAI. Gpt-4 technical report, 2023.

Rabe, M. N. and Staats, C. Self-attention does not need $o(n^2)$ memory, 2022.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2023.

Shazeer, N. Fast transformer decoding: One write-head is all you need, 2019.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Fu, D. Y., Xie, Z., Chen, B., Barrett, C., Gonzalez, J. E., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023b.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models, 2023.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Efficiently programming large language models using sglang, 2023.

A. Proving the Correctness of Attention Decomposition

We start by explicitly expressing softmax as an exponentiation followed by a normalization:

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) = \frac{\exp\left(\frac{QK^T}{\sqrt{d}}\right)}{e^{\text{LSE}(Q,K)}} \tag{6}$$

Therefore we can rewrite Equation 1 as:

$$\text{SDP}(Q, K, V) = \left(\frac{\exp\left(\frac{QK^T}{\sqrt{d}}\right)}{e^{\text{LSE}(Q,K)}}\right) V \tag{7}$$

We can then expand Equation 5:

$$\frac{\text{SDP}\left(Q, K_1, V_1\right) e^{\text{LSE}(Q,K_1)} + \text{SDP}\left(Q, K_2, V_2\right) e^{\text{LSE}(Q,K_2)}}{e^{\text{LSE}(Q,K_1)} + e^{\text{LSE}(Q,K_2)}} \tag{8}$$

$$= \frac{\left(\frac{\exp\left(\frac{QK_1^T}{\sqrt{d}}\right)}{e^{\text{LSE}(Q,K_1)}}\right) V_1 e^{\text{LSE}(Q,K_1)} + \left(\frac{\exp\left(\frac{QK_2^T}{\sqrt{d}}\right)}{e^{\text{LSE}(Q,K_2)}}\right) V_2 e^{\text{LSE}(Q,K_2)}}{e^{\text{LSE}(Q,K_1)} + e^{\text{LSE}(Q,K_2)}} \tag{9}$$

$$= \frac{\exp\left(\frac{QK_1^T}{\sqrt{d}}\right) V_1 + \exp\left(\frac{QK_2^T}{\sqrt{d}}\right) V_2}{e^{\text{LSE}(Q,K_1)} + e^{\text{LSE}(Q,K_2)}} \tag{10}$$

$$= \frac{\exp\left(\frac{Q(K_1||K_2)^T}{\sqrt{d}}\right) (V_1||V_2)}{e^{\text{LSE}(Q,K_1||K_2)}} \tag{11}$$

$$= \text{SDP}\left(Q, K_1||K_2, V_1||V_2\right) \tag{12}$$

as required. □

B. Hydragen Pseudocode

We provide PyTorch-style pseudocode implementing Hydragen attention below. We highlight that Hydragen can be implemented easily and efficiently in existing machine learning libraries, as long as there is a fast attention primitive that returns the LSE needed for softmax recombination.

```python
import torch
from torch import Tensor

def attention(q: Tensor, k: Tensor, v: Tensor) -> tuple[Tensor, Tensor]:
    """
    Placeholder for some fast attention primitive
    that also returns LSEs. We use the flash-attn
    package in our implementation.

    q shape: [batch, qseq_len, qheads, dim]
    k shape: [batch, kvseq_len, kvheads, dim]
    v shape: [batch, kvseq_len, kvheads, dim]
    """
    pass

def combine_lse(
```

```
17      out1: Tensor,
18      lse1: Tensor,
19      out2: Tensor,
20      lse2: Tensor,
21 ):
22      """
23      Combines two attention results using their LSEs.
24
25      Out1/2 shape: [batch, seq_len, qheads, hdim]
26      lse1/2 shape: [batch, seq_len, qheads]
27      """
28      max_lse = torch.maximum(lse1, lse2)
29
30      adjust_factor1 = (lse1 - max_lse).exp()
31      adjust_factor2 = (lse2 - max_lse).exp()
32
33      new_denominator = adjust_factor1 + adjust_factor2
34
35      aggregated = (
36          out1 * adjust_factor1.unsqueeze(-1) + out2 * adjust_factor2.unsqueeze(-1)
37      ) / new_denominator.unsqueeze(-1)
38
39      return aggregated
40
41
42 def hydragen_attention(
43      q: Tensor,
44      prefix_k: Tensor,
45      prefix_v: Tensor,
46      suffix_k: Tensor,
47      suffix_v: Tensor,
48 ):
49      """
50      q: shape [batch, num_queries (1 during decoding), qheads, dim]
51
52      prefix_k: shape [prefix_len, kvheads, dim]
53      prefix_v: shape [prefix_len, kvheads, dim]
54
55      suffix_k: shape [batch, suffix_len, kvheads, dim]
56      suffix_v: shape [batch, suffix_len, kvheads, dim]
57      """
58
59      b, nq, hq, d = q.shape
60
61      # inter-sequence batching: merge attention queries
62      # as if they all came from the same sequence.
63      batched_q = q.view(1, b * nq, hq, d)
64
65
66      # efficient attention over prefixes
67      # prefix_out: shape [1, batch * nq, hq, dim]
68      # prefix_lse: shape [1, batch * nq, hq]
69      prefix_out, prefix_lse = attention(
70          batched_q,
71          prefix_k.unsqueeze(0),
72          prefix_v.unsqueeze(0),
73      )
74
75
76      # normal attention over suffixes
77      # suffix_out: shape [batch, suffix_len, hq, dim]
78      # suffix_lse: shape [batch, suffix_len, hq]
79      suffix_out, suffix_lse = attention(
80          batched_q,
81          suffix_k,
```

```
82          suffix_v,
83      )
84
85      # unmerge prefix attention results and combine
86      # softmax denominators
87      aggregated = combine_lse(
88          prefix_out.view(b, nq, hq, d),
89          prefix_lse.view(b, nq, hq),
90          suffix_out,
91          suffix_lse,
92      )
93
94      return aggregated
```

C. Additional Results

C.1. End-to-End Throughput

We expand on the end-to-end throughput experiments discussed in Section 4.1. We report additional results with more model sizes when generating 128 and 256 tokens. These results are displayed in Table 1 and Table 2 for CodeLlama-7b, Table 3 and Table 4 for CodeLlama-13b, and Table 5 and Table 6 for CodeLlama-34b, respectively (Rozière et al., 2023). Note that in the tables where 128 tokens are generated per sequence, the "16K" column corresponds to a prefix length of 16256 tokens, while for the tables with 256 generated tokens per sequence, this corresponds to 16128 tokens (this is done to accommodate the 16384 max sequence length of the CodeLlama models).

Batch Size	FlashAttention Prefix Length					Hydragen Prefix Length					vLLM (No Tokenization) Prefix Length					vLLM Prefix Length					Upper Bound (No Attention) Prefix Length
	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	2.5 ± 0.0	2.2 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	0.9 ± 0.0	2.8 ± 0.0	2.7 ± 0.0	2.8 ± 0.3	2.6 ± 0.0	2.5 ± 0.0	1.7 ± 0.1	1.8 ± 0.1	1.7 ± 0.1	0.6 ± 0.0	0.4 ± 0.0	1.6 ± 0.1	1.6 ± 0.0	1.5 ± 0.0	0.6 ± 0.0	0.3 ± 0.0	3.1 ± 0.0
64	4.2 ± 0.0	3.4 ± 0.0	2.6 ± 0.0	1.7 ± 0.0	X	5.2 ± 0.0	4.9 ± 0.5	5.0 ± 0.0	4.9 ± 0.0	4.7 ± 0.0	3.5 ± 0.1	3.5 ± 0.1	2.9 ± 0.1	0.7 ± 0.0	0.4 ± 0.0	2.9 ± 0.1	2.8 ± 0.1	2.1 ± 0.3	0.7 ± 0.0	0.4 ± 0.0	5.7 ± 0.0
128	5.7 ± 0.0	4.2 ± 0.0	2.7 ± 0.0	X	X	8.4 ± 1.1	8.6 ± 0.3	8.4 ± 0.6	8.4 ± 0.0	8.2 ± 0.0	6.1	5.5	3.2	0.8	0.4	4.9	4.5	2.7	0.7	0.4	10.3 ± 0.0
256	8.1 ± 0.0	5.7 ± 0.0	X	X	X	13.5 ± 0.0	10.9 ± 0.0	10.9 ± 0.2	11.0 ± 0.1	12.5 ± 0.0	8.9	5.6	3.1	0.8	0.4	6.9	4.2	2.5	0.8	0.4	15.8 ± 0.0
512	X	X	X	X	X	19.8 ± 0.0	19.6 ± 0.0	19.4 ± 0.0	18.8 ± 0.0	17.7 ± 0.0	4.7	2.8	1.5	0.8	0.4	4.2	2.5	1.4	0.8	0.4	23.2 ± 0.0
1024	X	X	X	X	X	25.5 ± 0.0	25.3 ± 0.0	24.9 ± 0.0	24.0 ± 0.0	22.3 ± 0.0	4.9	2.8	1.5	0.8	0.4	4.2	2.5	1.4	0.7	0.4	30.1 ± 0.0
2048	X	X	X	X	X	27.8 ± 0.0	27.4 ± 0.0	25.4 ± 0.8	25.2 ± 0.0	22.7 ± 0.0	4.9	2.8	1.5	0.8	0.4	4.2	2.5	1.4	0.7	0.4	32.7 ± 0.0

Table 1. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-7B on 8xA100-40GB GPUs when generating 128 tokens. An "x" indicates that the model does not have the required memory to run.

Batch Size	FlashAttention Prefix Length 1K	2K	4K	8K	16K	Hydragen Prefix Length 1K	2K	4K	8K	16K	vLLM (No Tokenization) Prefix Length 1K	2K	4K	8K	16K	vLLM Prefix Length 1K	2K	4K	8K	16K	Upper Bound (No Attention) Prefix Length All
32	2.4 ± 0.0	2.2 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	0.9 ± 0.0	2.7 ± 0.0	2.6 ± 0.1	2.6 ± 0.1	2.6 ± 0.0	2.5 ± 0.0	1.7 ± 0.0	1.8 ± 0.0	1.7 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	1.6 ± 0.0	1.5 ± 0.0	1.5 ± 0.0	0.6 ± 0.0	0.3 ± 0.0	3.1 ± 0.0
64	3.9 ± 0.0	3.4 ± 0.0	2.5 ± 0.0	1.7 ± 0.0	X	5.0 ± 0.0	4.9 ± 0.0	4.9 ± 0.2	4.8 ± 0.0	4.6 ± 0.0	3.4 ± 0.1	3.3 ± 0.0	2.7 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	2.8 ± 0.1	2.8 ± 0.0	2.3 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	5.7 ± 0.0
128	5.3 ± 0.0	4.1 ± 0.0	2.7 ± 0.0	X	X	8.4 ± 0.0	8.3 ± 0.0	8.2 ± 0.0	8.1 ± 0.1	7.9 ± 0.0	6.3	5.0	2.9	0.8	0.4	4.8	4.0	2.5	0.7	0.4	10.3 ± 0.0
256	7.4 ± 0.0	X	X	X	X	12.9 ± 0.0	11.1 ± 0.4	11.5 ± 0.6	12.4 ± 0.0	12.0 ± 0.0	8.8	5.5	3.1	0.8	0.4	6.5	4.2	2.5	0.7	0.4	15.7 ± 0.0
512	X	X	X	X	X	18.6 ± 0.0	18.4 ± 0.0	18.2 ± 0.0	17.1 ± 0.9	16.8 ± 0.0	4.6	2.8	1.6	0.8	0.4	3.8	2.4	1.4	0.7	0.4	23.2 ± 0.0
1024	X	X	X	X	X	23.6 ± 0.0	23.0 ± 0.3	21.9 ± 1.0	22.2 ± 0.0	20.9 ± 0.0	4.8	2.8	1.6	0.8	0.4	3.9	2.4	1.4	0.7	0.4	30.0 ± 0.0

Table 2. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-7B on 8xA100-40GB GPUs when generating 256 tokens. An "x" indicates that the model does not have the required memory to run.

Batch Size	FlashAttention Prefix Length 1K	2K	4K	8K	16K	Hydragen Prefix Length 1K	2K	4K	8K	16K	vLLM (No Tokenization) Prefix Length 1K	2K	4K	8K	16K	vLLM Prefix Length 1K	2K	4K	8K	16K	Upper Bound (No Attention) Prefix Length All
32	1.7 ± 0.0	1.4 ± 0.0	1.1 ± 0.0	0.7 ± 0.0	X	2.1 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	1.9 ± 0.0	1.9 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	1.6 ± 0.0	1.6 ± 0.0	1.5 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	2.3 ± 0.0
64	2.9 ± 0.0	2.3 ± 0.0	1.6 ± 0.0	X	X	3.7 ± 0.0	3.6 ± 0.2	3.6 ± 0.0	3.4 ± 0.0	3.4 ± 0.0	3.5 ± 0.1	3.5 ± 0.0	2.9 ± 0.1	0.7 ± 0.0	0.4 ± 0.0	3.0 ± 0.1	2.9 ± 0.1	2.4 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	4.2 ± 0.0
128	4.0 ± 0.0	2.9 ± 0.0	X	X	X	5.9 ± 0.0	5.7 ± 0.4	5.5 ± 0.3	5.7 ± 0.0	5.5 ± 0.0	5.5	4.7 ± 0.1	3.0	0.8	0.4	4.8	3.8 ± 0.1	2.6	0.7	0.4	6.8 ± 0.0
256	5.7 ± 0.0	X	X	X	X	9.7 ± 0.0	9.9 ± 0.7	9.5 ± 0.0	9.4 ± 0.0	8.9 ± 0.0	8.0	5.5 ± 0.1	3.2	0.8	0.4	6.1	4.3 ± 0.1	2.7	0.7	0.4	11.4 ± 0.0
512	X	X	X	X	X	13.5 ± 0.0	13.4 ± 0.0	12.3 ± 1.3	13.0 ± 0.0	12.3 ± 0.0	4.7	2.7 ± 0.0	1.6	0.8	0.4	4.1	2.4 ± 0.0	1.4	0.8	0.4	16.1 ± 0.0
1024	X	X	X	X	X	15.6 ± 0.0	15.5 ± 0.0	15.3 ± 0.0	14.8 ± 0.0	13.8 ± 0.0	4.9 ± 0.0	2.8 ± 0.0	1.6 ± 0.0	0.8 ± 0.0	0.4 ± 0.0	4.2 ± 0.0	2.5 ± 0.0	1.4 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	18.4 ± 0.0

Table 3. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-13B on 8xA100-40GB GPUs when generating 128 tokens. An "x" indicates that the model does not have the required memory to run.

Batch Size	FlashAttention Prefix Length 1K	2K	4K	8K	16K	Hydragen Prefix Length 1K	2K	4K	8K	16K	vLLM (No Tokenization) Prefix Length 1K	2K	4K	8K	16K	vLLM Prefix Length 1K	2K	4K	8K	16K	Upper Bound (No Attention) Prefix Length All
32	1.7 ± 0.0	1.4 ± 0.0	1.1 ± 0.0	0.7 ± 0.0	X	2.0 ± 0.0	1.9 ± 0.1	1.9 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	1.7 ± 0.0	1.8 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.6 ± 0.0	1.6 ± 0.0	1.5 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	2.3 ± 0.0
64	2.8 ± 0.0	2.2 ± 0.0	1.6 ± 0.0	X	X	3.6 ± 0.0	3.5 ± 0.0	3.5 ± 0.0	3.3 ± 0.0	3.3 ± 0.0	3.4 ± 0.1	3.4 ± 0.0	2.9 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	3.0 ± 0.2	2.7 ± 0.2	2.2 ± 0.1	0.6 ± 0.0	0.4 ± 0.0	4.2 ± 0.0
128	3.8 ± 0.0	2.8 ± 0.0		X	X	5.6 ± 0.0	5.6 ± 0.1	5.4 ± 0.0	5.5 ± 0.0	5.3 ± 0.0	5.4	4.6	3.0	0.8	0.4	4.6	3.7	2.4	0.7	0.4	6.8 ± 0.0
256	5.4 ± 0.0	X	X	X	X	9.1 ± 0.0	8.6 ± 0.3	8.9 ± 0.0	8.8 ± 0.0	8.4 ± 0.0	7.6	5.5	3.1	0.8	0.4	5.9	4.3	2.5	0.7	0.4	11.3 ± 0.0
512	X	X	X	X	X	12.5 ± 0.0	12.4 ± 0.0	12.4 ± 0.0	12.1 ± 0.0	11.5 ± 0.0	4.4	2.7	1.5	0.8	0.4	3.8	2.4	1.4	0.7	0.4	16.1 ± 0.0

Table 4. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-13B on 8xA100-40GB GPUs when generating 256 tokens. An "x" indicates that the model does not have the required memory to run.

Batch Size	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
	Prefix Length					Prefix Length					Prefix Length					Prefix Length					Prefix Length
	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	1.4 ± 0.0	1.3 ± 0.0	1.2 ± 0.0	1.0 ± 0.0	0.8 ± 0.0	1.5 ± 0.1	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.5 ± 0.0	1.4 ± 0.0	1.2 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.5 ± 0.0	1.3 ± 0.0	1.1 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.6 ± 0.0
64	2.5 ± 0.0	2.4 ± 0.0	2.1 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	2.6 ± 0.1	2.6 ± 0.0	2.6 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.6 ± 0.0	2.3 ± 0.0	1.9 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	2.4 ± 0.0	2.1 ± 0.1	1.6 ± 0.1	0.6 ± 0.0	0.4 ± 0.0	2.9 ± 0.0
128	3.8 ± 0.0	3.4 ± 0.0	2.8 ± 0.0	2.1 ± 0.0	X	4.2 ± 0.0	4.2 ± 0.0	4.1 ± 0.0	4.1 ± 0.0	3.9 ± 0.0	3.8	3.0	2.3	0.8	0.4	3.4	2.7	2.0	0.7	0.4	4.6 ± 0.0
256	6.0 ± 0.0	5.3 ± 0.0	4.3 ± 0.0	X	X	6.3 ± 0.4	6.7 ± 0.0	6.6 ± 0.0	6.3 ± 0.0	6.0 ± 0.0	5.1	3.9	2.8	0.8	0.4	4.4	3.3	2.4	0.8	0.4	7.2 ± 0.0
512	7.0 ± 0.0	6.0 ± 0.0	X	X	X	7.8 ± 0.5	8.1 ± 0.0	8.0 ± 0.0	7.7 ± 0.0	7.1 ± 0.0	4.2	2.7	1.5	0.8	0.4	3.6	2.4	1.4	0.8	0.4	8.8 ± 0.0
1024	X	X	X	X	X	9.0 ± 0.2	9.0 ± 0.0	8.8 ± 0.0	8.3 ± 0.0	7.5 ± 0.0	4.3	2.8	1.6	0.8	0.4	3.7	2.5	1.4	0.8	0.4	9.7 ± 0.0
2048	X	X	X	X	X	10.1 ± 0.0	10.0 ± 0.0	9.7 ± 0.0	9.2 ± 0.0	8.3 ± 0.0	4.3	2.7	1.5	0.8	0.4	3.7	2.4	1.4	0.8	0.4	10.7 ± 0.0
4096	X	X	X	X	X	10.7 ± 0.0	10.6 ± 0.0	10.4 ± 0.0	9.5 ± 0.1	8.9 ± 0.0	4.0	2.6	1.4	0.8	0.4	3.5	2.3	1.3	0.7	0.4	11.3 ± 0.0

Table 5. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-34B on 8xA100-40GB GPUs when generating 128 tokens. An "x" indicates that the model does not have the required memory to run.

Batch Size	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
	Prefix Length					Prefix Length					Prefix Length					Prefix Length					Prefix Length
	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	1.4 ± 0.0	1.3 ± 0.0	1.2 ± 0.0	1.1 ± 0.0	0.8 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.5 ± 0.0	1.4 ± 0.0	1.2 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.5 ± 0.0	1.3 ± 0.1	1.1 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.6 ± 0.0
64	2.5 ± 0.0	2.4 ± 0.0	2.1 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	2.6 ± 0.1	2.6 ± 0.0	2.6 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.6 ± 0.0	2.3 ± 0.0	1.8 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	2.3 ± 0.1	2.0 ± 0.0	1.6 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	2.8 ± 0.0
128	3.8 ± 0.0	3.4 ± 0.0	2.8 ± 0.0	2.1 ± 0.0	X	4.1 ± 0.1	4.1 ± 0.0	4.1 ± 0.0	4.0 ± 0.0	3.9 ± 0.0	3.7	3.0	2.2	0.7	0.4	3.2	2.6	2.0	0.7	0.4	4.5 ± 0.0
256	5.8 ± 0.0	5.3 ± 0.0	4.3 ± 0.0	X	X	6.6 ± 0.0	6.5 ± 0.0	6.4 ± 0.0	6.2 ± 0.0	5.9 ± 0.0	5.0	3.9	2.7	0.8	0.4	4.2	3.3	2.3	0.7	0.4	7.2 ± 0.0
512	6.7 ± 0.0	5.9 ± 0.0	X	X	X	7.9 ± 0.0	7.9 ± 0.1	7.9 ± 0.0	7.6 ± 0.0	7.1 ± 0.0	3.9	2.6	1.5	0.8	0.4	3.5	2.3	1.4	0.7	0.4	8.8 ± 0.0
1024	X	X	X	X	X	9.0 ± 0.0	8.9 ± 0.0	8.6 ± 0.0	8.1 ± 0.0	7.4 ± 0.0	3.9	2.6	1.4	0.8	0.4	3.6	2.4	1.4	0.7	0.4	9.7 ± 0.0
2048	X	X	X	X	X	10.0 ± 0.0	9.8 ± 0.0	9.6 ± 0.0	9.1 ± 0.0	8.2 ± 0.0	4.0	2.6	1.5	0.8	0.4	3.6	2.4	1.4	0.7	0.4	10.6 ± 0.0

Table 6. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-34B on 8xA100-40GB GPUs when generating 256 tokens. An "x" indicates that the model does not have the required memory to run.

C.2. Microbenchmarks

We repeat the A100 microbenchmark experiment from Section 4.2 on an H100 and L40S GPU, reporting our results in Figure 8. The L40S has the highest ratio of compute capability to memory bandwidth of the three GPUs and therefore derives the most benefit from Hydragen's elimination of memory bottlenecks. While the compute-to-bandwidth ratio is higher on an H100 than on an A100, we measure similar Hydragen speedups between both cards. This stems from the fact that the `flash-attn` package that we use is not currently optimized for Hopper GPUs, and therefore achieves a lower device utilization on an H100 vs an A100.

D. Experiment Details

D.1. End-to-End Benchmarks

Our end-to-end benchmarks only measure decoding throughput and exclude the time required to compute the prefill. We measure "decode-only" time by initially benchmarking the time required to generate one token from a given prompt and subtracting that value from the time it takes to generate the desired number of tokens. This subtraction is particularly
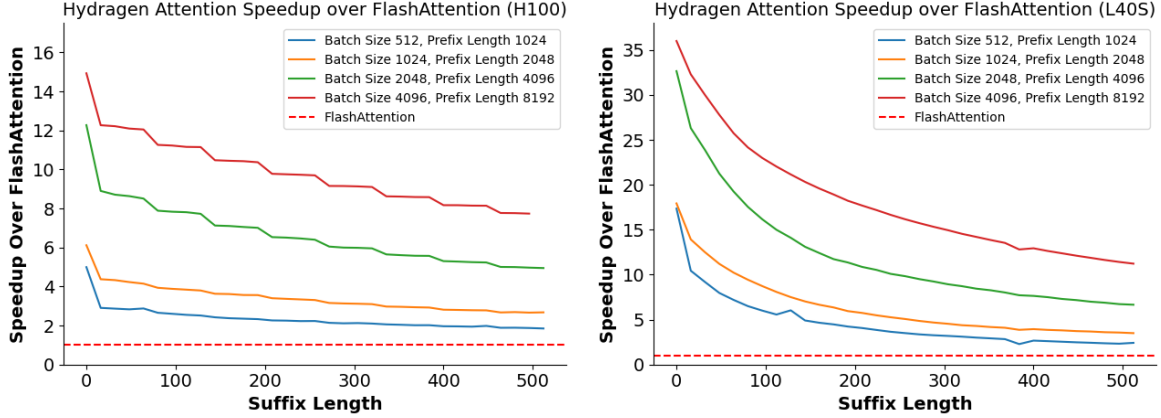
Figure 8. Speedup of Hydragen attention over FlashAttention for various batch sizes, shared prefix lengths and suffix lengths on an H100 (left) and an L40S (right) GPU.

important in order to fairly evaluate vLLM baselines, since it appears that vLLM redundantly detokenizes the prompt for every sequence in the batch at the beginning of inference (this can take minutes for large batch sizes and sequence lengths). For our "vLLM no detokenization" baseline, we disable incremental detokenization in vLLM by commenting out this line.

For all FlashAttention and No Attention datapoints, we run 10 warmup iterations and use the following 10 iterations to compute throughput. For Hydragen datapoints, we run 10 warmup and 10 timing iterations when the batch size is less than 256, and for larger batch sizes use three warmup and three timing iterations. We observe that shorter-running Hydragen benchmarks (those with smaller batch sizes, sequence lengths, model sizes, or completion lengths) can occasionally produce longer outlier times. This seems to be related not to decoding time itself, but to variations in prefilling time before decoding. For vLLM baselines (both with and without incremental detokenization), we use three warmup and timing iterations for all batch sizes below 128, as well as for all datapoints that are used in Figures 1 and 4. The longest-running vLLM runs can take many minutes to complete a single iteration, so for baselines above a batch size of 128 that only appear in the supplementary tables of Appendix C.1, we use one warmup and one timing iteration.

D.2. Microbenchmarks

In each microbenchmark, we run 1000 iterations of warmup before reporting the mean running time across 1000 trials. Between trials, we flush the GPU L2 cache by writing to a 128MiB tensor. We use CUDA graphs when benchmarking in order to reduce CPU overhead, which can be important since some benchmarks can complete a single iteration in tens of microseconds.

D.3. Long document retrieval

To demonstrate the throughput benefits of using Hydragen to answer questions about a long document, we construct a document (with 19974 tokens) that contains arbitrary facts from which question/answer pairs can be easily generated.

Prefix and Suffix Content: The content of the document is a subset of *War and Peace*, modified to include procedurally generated facts of the form "The dog named {name} has fur that is {color}". The questions are of the form "What color is the fur of the dog named name?", where the answer is color. We construct 261 questions (256 testable questions plus five for the few-shot examples) and interleave these throughout sentences of the document. When benchmarking with a greater number of questions than 256, we duplicate questions when querying the model - this is instead of adding more questions to the document in order to constrain total document length.

Model and Accelerator Choice: We choose the Yi-6B-200k model because it is small enough to fit a large KV cache in memory (important when running baselines that redundantly store the document) while also supporting a long enough context to process our document. We distribute the model across four A100-40GB GPUs in order to maximize possible KV cache size (the model only has four key/value attention heads, preventing us from easily using tensor parallelism across more GPUs).

Our reported measurements use the mean of five timing runs after ten warmup iterations.

D.4. Hierarchical Sharing in Competitive Programming

The dataset of 120 problems that we use for this benchmark comes from the introductory difficulty split of APPS. We filter out problems that include starter code. We use two few-shot examples (2400 tokens long) that come from the training split of APPS, while all of the eval examples come from the test split. We sample 512 tokens for every completion. We run this experiment using CodeLlama-7b on eight A100-40GB GPUs. We measure the total time to run inference on all 120 questions, excluding tokenization and detokenization time.