Hydragen: High-Throughput LLM Inference with Shared Prefixes

Anonymous Authors[1]

Abstract

As large language models (LLMs) are deployed more broadly, reducing the cost of inference has become increasingly important. A common inference use case involves a batch of sequences that share a prefix, such as when reusing few-shot examples or sampling many completions from a single prompt. In a large-batch setting, transformer decoding 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 specialized for shared prefixes. Hydragen computes attention separately over the shared prefix and unique suffixes. This decomposition enables efficient prefix attention by batching queries together across sequences, reducing redundant memory reads and replacing matrix-vector products with hardware-friendly matrix-matrix products. In a high-throughput setting (batch size 1K, tensor parallelism across eight A100s), our method can improve end-to-end CodeLlama-13b throughput by over 3x with a prefix length of 1K, and by over 30x with a prefix length of 16K. Hydragen's efficient processing of long shared contexts lead to only a 15% drop in throughput as the sequence length grows by 16x. We extend Hydragen beyond simple prefix-suffix decomposition and apply it to hierarchical sharing patterns, which allows us to further reduce inference time on competitive programming problems by a further 55%.

1. Introduction

As LLMs grow proficient at a wide variety of tasks, reducing the cost of deploying these models becomes increasingly important. One common setting for LLM inference involves generating text for a batch of sequences that share a prefix. Examples of this use case include reusing a few-shot prompt across multiple problems (Figure 1 left), sampling many candidate solutions to a single problem (15), and long context document processing. In this work, we use a hardware-aware perspective to analyze and optimize the shared prefix setting, with a focus on large-batch, throughput-oriented applications.

In transformer-based LLMs, large-batch inference is often bottlenecked by the attention operation. Since each sequence in the batch has only a single attention query when decoding, existing high-performance attention implementations like FlashAttention (9; 8) and PagedAttention (14) compute attention using many independent matrix-vector products. For large KV caches, this approach becomes memory-bound and moreover does not use hardware-friendly matrix-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 1 bottom right).

Shared prefixes create overlaps in the attention key and value matrices across sequences, presenting opportunities for specialized optimizations. Existing work (14) exploits this overlap to avoid redundant storage of the prefix KV cache and reduce GPU memory consumption. Separate from these memory savings, in this paper we demonstrate that shared prefixes enable an alternative algorithm for computing attention - Hydragen - that is much more hardware-friendly (Figure 1 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 1 top right), reducing redundant reads of the prefix KV cache and enabling the use of tensor cores. Our entire algorithm can be implemented using existing fast attention kernels without any new CUDA code (Appendix C).

In scenarios where large batch sizes and/or long prefix

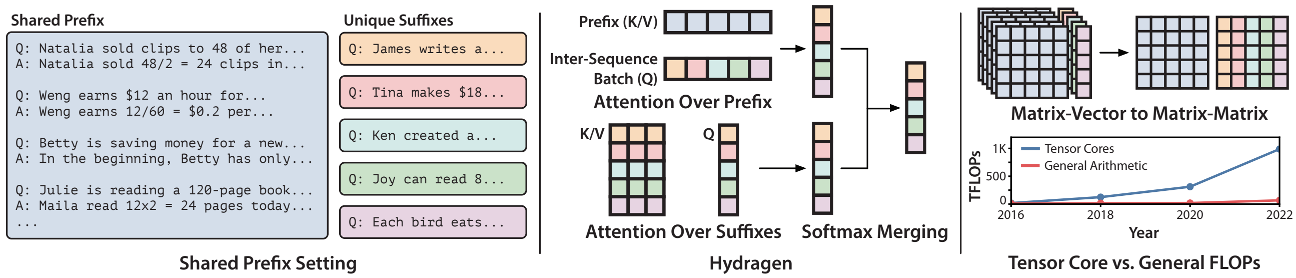


Figure 1. Left: An example inference scenario featuring a shared prefix (the few-shot examples). Middle: An overview of Hydragen, where overall attention is decomposed into attention over the shared prefix (batched across all queries in a batch) and attention over the remaining suffixes (independent 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.

lengths bottleneck decoding, we demonstrate that Hydragen can improve end-to-end LLM throughput over vLLM (14), a high-performance inference framework that avoids redundant prefix storage but not redundant prefix reads. In a high-throughput benchmarking setting (batch size 1024 with tensor parallelism across eight A100-40GB GPUs), Hydragen increases the throughput of CodeLlama-13b (21) by up over 3x with a prefix length of 1K and by over 30x with a prefix length of 16K. By efficiently attending over the shared prefix, throughput with Hydragen drops by only 15% as the prefix length grows from 1K to 16K, whereas vLLM throughput decreases by over 90%. We also benchmark the Hydragen attention operation in isolation against a state-of-the-art FlashAttention baseline (8) (Section 4.2). While Hydragen introduces additional overhead that can slow down attention with smaller inputs (e.g. prefix length 1K, batch size $\leq$ 16), performance significantly improves as the prefix length and batch size grow. With a batch size of 4K and a prefix length of 8K, Hydragen can achieve more than a 12x speedup over FlashAttention.

We evaluate Hydragen end-to-end on three real-world use cases. On a batched needle-in-a-haystack document processing task, we show that Hydragen can process 256 questions about a document in less time than it takes a FlashAttention baseline to process 64 questions (Section 4.3). Moreover, 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 (11), 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). Using two-level Hydragen, we also measure a 2x speedup when evaluating LLMs on GSM8k with self-consistency, relative to a FlashAttention baseline (28; 7).

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 total number of arithmetic operations performed divided by the total number of bytes transferred. Higher arithmetic intensities imply a greater use of computational 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 is 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 (FLOPS) 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 attention (SDPA) 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 $V \in \mathbb{R}^{N_{kv} \times d}$, producing an output $O \in \mathbb{R}^{N_q \times d}$ defined as:

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

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, when computing attention we have $N_q = N_{kv} \gg 1$ and as a result the multiplications in Equation 1 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 need 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 and allows these modules to use hardware-friendly matrix multiplications. However, batched text generation does not increase the intensity of attention, since every sequence has a distinct key and value matrix. Therefore, while other model components are able to use tensor cores during batched decoding, attention must be computed using 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 focus on improving the throughput of batched text generation when sequences in the batch share a common prefix. This scenario lends itself to specialized optimizations because shared prefixes create overlaps in attention key and value matrices across sequences. Using methods like PagedAttention (14), this overlap can be exploited to avoid redundant storage and save GPU memory (14). In this work, we identify an additional opportunity to use this overlap for optimizing the attention operation itself.

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 C.

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 N_{kv} (the sequence/row dimension) into:

$$K = K_1 || K_2 \tag{2}$$
$$V = V_1 || V_2 \tag{3}$$

with $||$ denoting concatenation. We wish to avoid directly computing our desired quantity SDPA (Q, K, V), and instead calculate this value using the results of the sub-computations SDPA (Q, K_1, V_1) and SDPA (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 (9). When computing SDPA (Q, K_1, V_1) and SDPA (Q, K_2, V_2), we additionally compute and store the log-sum-exp (LSE$(Q, K) \in \mathbb{R}^{N_q}$) of the attention scores (equivalently, the log of the softmax denominator):

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

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

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

We prove this formula in Appendix B.

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 over the shared prefix. Since the prefix's keys and values across sequences are identical, we can batch each 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.

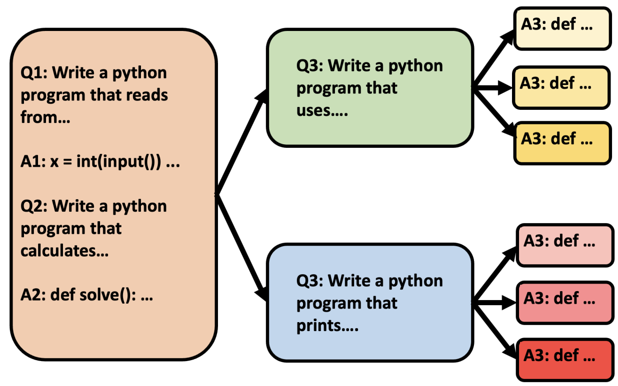


Figure 2. An example of a hierarchical sharing pattern in a competitive programming setting. The few-shot prompt (orange) is globally shared across all sequences in the batch. However, the descriptions of each problem (green and blue) are only shared across the candidate solutions corresponding to that problem.

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 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 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 3.3 for an example). These forms of sharing are increasingly relevant as LLMs are applied in more complicated inference/search algorithms (29; 4; 17).

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 with 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. Implementation

We implement Hydragen for the Llama family of models (25; 26; 21). 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 multiplications. 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 math and coding problems (21; 15). 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 incremental 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 (21) 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 4.1, 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 4.1, 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 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 much less affected 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 D.1 - for smaller models and shorter completions lengths, Hydragen's speedup can exceed 50x. Additional evaluation setup details are in Appendix E.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 with tensor parallelism across eight GPUs). We sweep over different batch sizes, prefix lengths, and suffix lengths, reporting our results in

[1] For non-hierarchical inputs, we've also written a Triton kernel for combining softmax denominators.

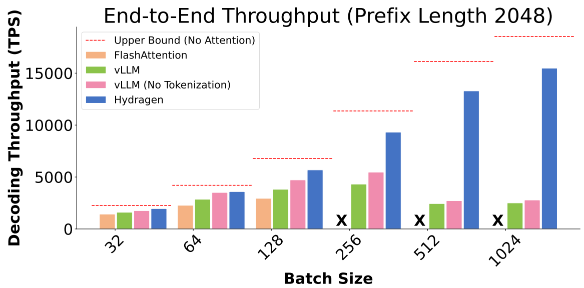


Figure 3. End-to-end decoding throughput with CodeLlama-13b when generating multiple completions from a prompt containing 2048 tokens. An "x" indicates that FlashAttention does not have enough memory to run. As the batch size grows, Hydragen achieves a higher throughput than all baselines. Throughput with Hydragen always remains within 50% of the upper bound where attention is entirely removed from the model.

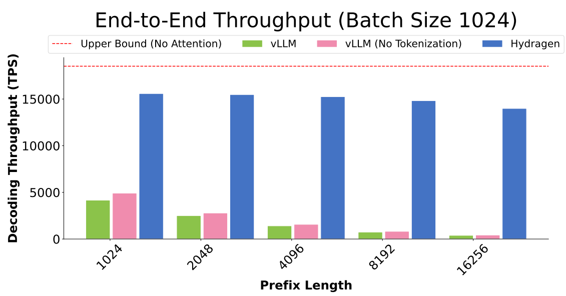
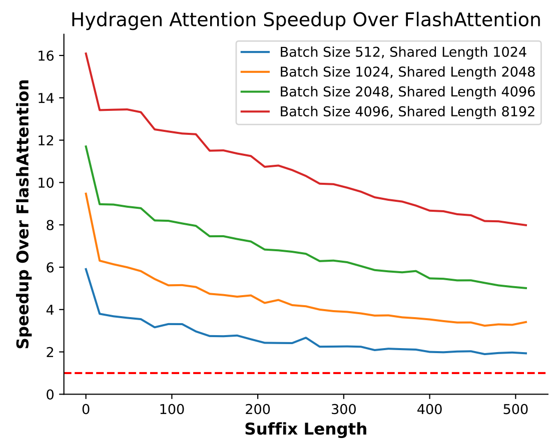


Figure 4. Comparing CodeLlama-13b decoding throughput where the batch size is fixed at 1024 and we sweep over prefix lengths. As the prefix grows from 1024 to 16256 tokens, Hydragen throughput drops by less than 15%.

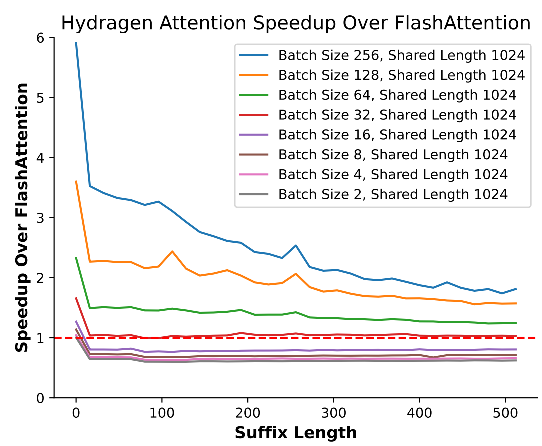
Figures 5(a) and 5(b). Our microbenchmarks corroborate our end-to-end measurements from Section 4.1, showing that speedup with Hydragen increases as the batch size and prefix lengths grow. Additionally, our results highlight the significant impact of the suffix length on inference time. Hydragen computes 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 decoding 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.

We also note that with small KV cache sizes (e.g. the results in Figure 5(b) where the prefix length is 1K and the batch size is 16 or less), the additional overhead of performing attention decomposition outweighs the benefits of shared prefix attention, resulting in a decrease in speed relative to FlashAttention. However, at these small input sizes, attention is not commonly a major contributor to decoding time (relative to other factors such as reading the model weights from GPU memory), regardless of whether Hydragen is used or not.

Our microbenchmarks are 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 D.2 and provide more evaluation details in Appendix E.2.



(a)



(b)

Figure 5. Microbenchmarking the Hydragen attention operation relative to a FlashAttention baseline on a single A100-40GB GPU. In the special case of a suffix length of zero, we only benchmark prefix attention (since there is no suffix to attend over and combine the results of).

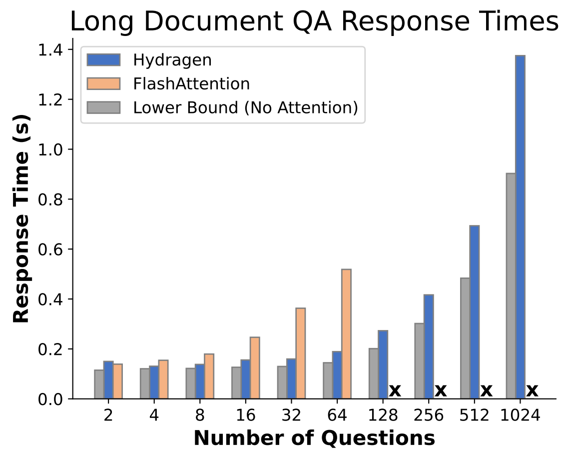


Figure 6. Measuring the time to answer questions about a 19947 token-long document when benchmarking Yi-6B-200k on four A100-40GB GPUs. An "x" indicates that FlashAttention does not have enough memory to run. Time to process the document is excluded.

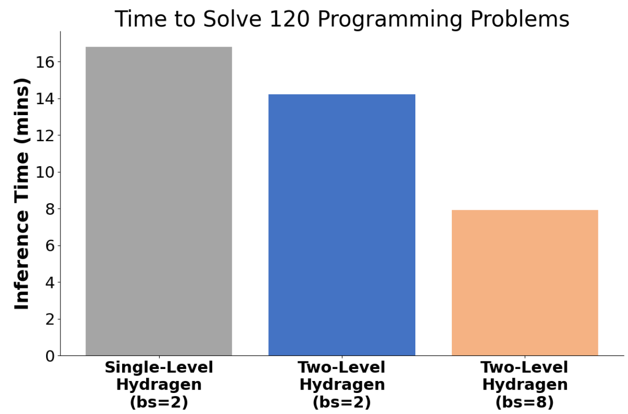


Figure 7. Measuring inference time for solving 120 programming problems. The batch size refers to the number of problems processed simultaneously. By sharing both the few-shot examples across all sequences and the problem description across generated candidate solutions, two-level Hydragen decreases overall inference time by an extra 55% over single-level Hydragen (which only shares the few-shot prompt).

4.3. Long Document Processing

Additionally, we explore the performance of Hydragen on workloads involving very long documents. This setup resembles a "needle-in-a-haystack" evaluation, except we have embedded many needles into our document, which we retrieve in parallel in a single batch. We construct a document by embedding synthetic facts into an excerpt of *War and Peace* (24). 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 (1) on its ability to answer questions based on the embedded facts. We run this benchmark across four A100-40GB GPUs using Hydragen in addition to our FlashAttention and no-attention baselines. Results are reported in Figure 4.2. 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 remains within 60% of the no-attention optimum. Notably, Hydragen can process 256 questions in less time than it takes the FlashAttention baseline to process 64 questions. We provide additional evaluation details in Appendix E.3.

4.4. Hierarchical Sharing in Competitive Programming and Self-Consistency Evaluation

We lastly demonstrate the benefits of applying Hydragen in settings with hierarchical sharing (described in Section 3.3). Competitive programming was a motivating application for developing our method, since current state-of-the-art systems can sample thousands or more candidate programs from prompts that can contain thousands of tokens (15; 21). Self-consistency similarly harnesses multiple samples to improve model capabilities by conducting majority voting to determine a final answer (28). In both of these settings, 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 that problem's candidate solutions (see Figure 4.4).

For competitive programming, we benchmark the total time required to evaluate CodeLlama-7b (using tensor parallelism across 8 A100-40GB GPUs) on 120 problems from the APPS dataset (11). We use a two-shot prompt and 128 candidate programs per problem. We benchmark Hydragen using two approaches:

1. **Single-Level Hydragen:** We use a single-level version of Hydragen to share the few-shot prompt across all sequences in the batch, but not share problem descriptions across candidate solutions. This leads to redundant storage of the problem description across all 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 4.4 and Table 1. 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 E.4.

For benchmarking self-consistency, we follow the original paper's procedure for evaluation on GSM8k (7), using an eight-shot prompt and sampling 40 completions per problem. We evaluate the 7b and 13b models in the Llama 2 family, benchmarking a FlashAttention baseline and Hydragen with two levels of prompt sharing. We again benchmark Hydragen at the same batch size as our baseline and additionally with the new largest batch size that can fit. We report our results in Table 1. Like with code generation, we observe a speedup with Hydragen even when the batch size is held constant with the baseline, with the speedup increasing to over 2x when the batch size is further increased.

5. Related Work

Transformers and Language Models: The transformer architecture has enabled significant improvements in state-of-the-art language models (27). A defining feature of transformers is that their performance consistently improves when scaling up data and model size (20; 5; 6; 12; 18). LLM-powered assistants such as ChatGPT have been widely adopted and are currently used by over a hundred million users (16), 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 (22) and GQA (2) 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 (23; 3; 13). vLLM (14) 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. SGLang (30) 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 decoding 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 (19; 9; 8), 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 (10), which can achieve over 10x more FLOPS 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 (28) 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 (21). AlphaCode (15), a state-of-the-art competitive programming system, samples as many as a million programs to solve a single problem. Tree-of-Thoughts (29) 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.

6. Conclusion

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. In settings where the batch size is small and/or the sequence lengths are short, attention is often only a minor contributor to throughput and Hydragen's effects will be minimal (or even detrimental, see Figure 5(b)). For example, interactive chatbots may use a smaller batch size than fits in GPU memory in order to satisfy latency constraints. Therefore, although this application can feature a

Model	Attention Algorithm	Batch Size (sequences)	Eval Time (mins)	Speedup
Llama-2-7b-chat	FlashAttention	360	41.63	1
Llama-2-7b-chat	Hydragen	360	28.39	1.47
Llama-2-7b-chat	Hydragen	1200	19.94	2.09
Llama-2-13b-chat	FlashAttention	240	70.49	1
Llama-2-13b-chat	Hydragen	240	51.31	1.37
Llama-2-13b-chat	Hydragen	720	32.88	2.14

Table 1. Measuring the time required to run evaluation on the GSM8k test set using self-consistency.

significant amount of prompt sharing (system instructions which are shared across user requests can often be quite long), it may not be able to benefit from Hydragen to the same degree as purely throughput-oriented applications. We provide an extended discussion of the settings where Hydragen is applicable in Appendix A.

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.

References

[1] 01-ai. Yi, 2023. Accessed: 2024-02-01.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[3] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, 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)*, pages 646–660. IEEE Computer Society, 2022.

[4] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models, 2023.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

[8] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.

[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

[10] Daniel Y. Fu, Hermann Kumbong, Eric Nguyen, and Christopher Ré. Flashfftconv: Efficient convolutions for long sequences with tensor cores, 2023.

[11] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[12] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

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

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[15] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.

[16] Aisha Malik. 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.

[17] Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Large language models can do parallel decoding, 2023.

[18] OpenAI. Gpt-4 technical report, 2023.

[19] Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

[20] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[21] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

[22] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[23] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[24] Leo Tolstoy. *War and Peace*. 1869.

[25] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[26] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten,

Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[28] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.

[29] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

[30] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang, 2023.

A. Estimating Throughput Improvements with Hydragen

Hydragen can significantly improve the efficiency of attention with shared prefixes relative to approaches that compute attention independently for every sequence (see 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 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 and/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 (22) or grouped-query attention (2) 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 increase 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 (shared) tokens compared to suffix (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.

B. 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:

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

We can then expand Equation 5:

$$\begin{aligned}
&\frac{\text{SDPA}\left(Q, K_1, V_1\right) e^{\text{LSE}(Q,K_1)} + \text{SDPA}\left(Q, K_2, V_2\right) e^{\text{LSE}(Q,K_2)}}{e^{\text{LSE}(Q,K_1)} + e^{\text{LSE}(Q,K_2)}} && (8)\\
&= \frac{\left(\frac{\exp\left(\frac{QK_1^T}{\sqrt{d}}\right)}{e^{\text{LSE}(Q,K_1)}}\right) V_1 e^{\text{LSE}(Q,K_1)} + \left(\frac{\exp\left(\frac{QK_2^T}{\sqrt{d}}\right)}{e^{\text{LSE}(Q,K_2)}}\right) V_2 e^{\text{LSE}(Q,K_2)}}{e^{\text{LSE}(Q,K_1)} + e^{\text{LSE}(Q,K_2)}} && (9)\\
&= \frac{\exp\left(\frac{QK_1^T}{\sqrt{d}}\right) V_1 + \exp\left(\frac{QK_2^T}{\sqrt{d}}\right) V_2}{e^{\text{LSE}(Q,K_1)} + e^{\text{LSE}(Q,K_2)}} && (10)\\
&= \frac{\exp\left(\frac{Q(K_1||K_2)^T}{\sqrt{d}}\right) (V_1||V_2)}{e^{\text{LSE}(Q,K_1||K_2)}} && (11)\\
&= \text{SDPA}\left(Q, K_1||K_2, V_1||V_2\right) && (12)
\end{aligned}$$

as required. □

C. 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(
    out1: Tensor,
    lse1: Tensor,
    out2: Tensor,
    lse2: Tensor,
):
    """
    Combines two attention results using their LSEs.

    Out1/2 shape: [batch, seq_len, qheads, hdim]
    lse1/2 shape: [batch, seq_len, qheads]
    """
    max_lse = torch.maximum(lse1, lse2)

    adjust_factor1 = (lse1 - max_lse).exp()
    adjust_factor2 = (lse2 - max_lse).exp()

    new_denominator = adjust_factor1 + adjust_factor2

    aggregated = (
        out1 * adjust_factor1.unsqueeze(-1) + out2 * adjust_factor2.unsqueeze(-1)
    ) / new_denominator.unsqueeze(-1)

    return aggregated


def hydragen_attention(
    q: Tensor,
    prefix_k: Tensor,
    prefix_v: Tensor,
    suffix_k: Tensor,
    suffix_v: Tensor,
):
    """
    q: shape [batch, num_queries (1 during decoding), qheads, dim]

    prefix_k: shape [prefix_len, kvheads, dim]
    prefix_v: shape [prefix_len, kvheads, dim]

    suffix_k: shape [batch, suffix_len, kvheads, dim]
    suffix_v: shape [batch, suffix_len, kvheads, dim]
```

```python
    """

    b, nq, hq, d = q.shape

    # inter-sequence batching: merge attention queries
    # as if they all came from the same sequence.
    batched_q = q.view(1, b * nq, hq, d)


    # efficient attention over prefixes
    # prefix_out: shape [1, batch * nq, hq, dim]
    # prefix_lse: shape [1, batch * nq, hq]
    prefix_out, prefix_lse = attention(
        batched_q,
        prefix_k.unsqueeze(0),
        prefix_v.unsqueeze(0),
    )


    # normal attention over suffixes
    # suffix_out: shape [batch, suffix_len, hq, dim]
    # suffix_lse: shape [batch, suffix_len, hq]
    suffix_out, suffix_lse = attention(
        batched_q,
        suffix_k,
        suffix_v,
    )

    # unmerge prefix attention results and combine
    # softmax denominators
    aggregated = combine_lse(
        prefix_out.view(b, nq, hq, d),
        prefix_lse.view(b, nq, hq),
        suffix_out,
        suffix_lse,
    )

    return aggregated
```

D. Additional Results

D.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 2 and Table 3 for CodeLlama-7b, Table 4 and Table 5 for CodeLlama-13b, and Table 6 and Table 7 for CodeLlama-34b, respectively (21). 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	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	2.5 $\pm$ 0.0	2.2 $\pm$ 0.0	1.8 $\pm$ 0.0	1.3 $\pm$ 0.0	0.9 $\pm$ 0.0	2.7 $\pm$ 0.0	2.7 $\pm$ 0.0	2.6 $\pm$ 0.0	2.6 $\pm$ 0.0	2.5 $\pm$ 0.0	1.7 $\pm$ 0.0	1.8 $\pm$ 0.0	1.7 $\pm$ 0.1	0.6 $\pm$ 0.0	0.4 $\pm$ 0.0	1.6 $\pm$ 0.0	1.6 $\pm$ 0.0	1.5 $\pm$ 0.0	0.6 $\pm$ 0.0	0.3 $\pm$ 0.0	3.1 ± 0.0
64	4.2 $\pm$ 0.0	3.4 $\pm$ 0.0	2.6 $\pm$ 0.0	1.7 $\pm$ 0.0	X	5.0 $\pm$ 0.0	4.9 $\pm$ 0.0	4.9 $\pm$ 0.1	4.8 $\pm$ 0.0	4.6 $\pm$ 0.0	3.5 $\pm$ 0.1	3.5 $\pm$ 0.1	2.9 $\pm$ 0.1	0.7 $\pm$ 0.0	0.4 $\pm$ 0.0	2.9 $\pm$ 0.0	2.8 $\pm$ 0.1	2.1 $\pm$ 0.2	0.7 $\pm$ 0.0	0.4 $\pm$ 0.0	5.7 ± 0.0
128	5.7 $\pm$ 0.0	4.2 $\pm$ 0.0	2.7 $\pm$ 0.0	X	X	8.6 $\pm$ 0.0	8.5 $\pm$ 0.0	8.4 $\pm$ 0.0	8.3 $\pm$ 0.0	8.0 $\pm$ 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 $\pm$ 0.0	5.7 $\pm$ 0.0	X	X	X	13.3 $\pm$ 0.0	13.3 $\pm$ 0.0	13.1 $\pm$ 0.0	12.8 $\pm$ 0.0	12.3 $\pm$ 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	19.6 $\pm$ 0.0	19.4 $\pm$ 0.0	19.1 $\pm$ 0.0	18.5 $\pm$ 0.0	17.5 $\pm$ 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	25.3 $\pm$ 0.0	25.1 $\pm$ 0.0	24.7 $\pm$ 0.0	23.9 $\pm$ 0.0	22.4 $\pm$ 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	27.9 $\pm$ 0.0	27.5 $\pm$ 0.0	26.7 $\pm$ 0.0	25.3 $\pm$ 0.0	22.8 $\pm$ 0.0	4.9	2.8	1.5	0.8	0.4	4.2	2.5	1.4	0.7	0.4	32.9 ± 0.0

Table 2. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-7B on 8xA100 40 GB GPUs when generating 128 tokens. An x indicates the model does not have the required memory to run.

Batch	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	2.4 $\pm$ 0.0	2.2 $\pm$ 0.0	1.8 $\pm$ 0.0	1.3 $\pm$ 0.0	0.9 $\pm$ 0.0	2.6 $\pm$ 0.0	2.6 $\pm$ 0.0	2.6 $\pm$ 0.0	2.5 $\pm$ 0.0	2.4 $\pm$ 0.0	1.7 $\pm$ 0.0	1.8 $\pm$ 0.0	1.7 $\pm$ 0.0	0.6 $\pm$ 0.0	0.4 $\pm$ 0.0	1.6 $\pm$ 0.0	1.5 $\pm$ 0.0	1.5 $\pm$ 0.0	0.6 $\pm$ 0.0	0.3 $\pm$ 0.0	3.1 ± 0.0
64	3.9 $\pm$ 0.0	3.4 $\pm$ 0.0	2.5 $\pm$ 0.0	1.7 $\pm$ 0.0	X	4.8 $\pm$ 0.0	4.8 $\pm$ 0.0	4.8 $\pm$ 0.0	4.7 $\pm$ 0.0	4.5 $\pm$ 0.0	3.4 $\pm$ 0.0	3.3 $\pm$ 0.0	2.7 $\pm$ 0.0	0.7 $\pm$ 0.0	0.4 $\pm$ 0.0	2.8 $\pm$ 0.1	2.8 $\pm$ 0.0	2.3 $\pm$ 0.0	0.6 $\pm$ 0.0	0.4 $\pm$ 0.0	5.7 ± 0.0
128	5.3 $\pm$ 0.0	4.1 $\pm$ 0.0	2.7 $\pm$ 0.0	X	X	8.2 $\pm$ 0.0	8.2 $\pm$ 0.0	8.1 $\pm$ 0.0	7.9 $\pm$ 0.0	7.7 $\pm$ 0.0	6.3	5.0	2.9	0.8	0.4	4.8	4.0	2.5	0.7	0.4	10.2 ± 0.0
256	7.4 $\pm$ 0.0	X	X	X	X	12.7 $\pm$ 0.0	12.6 $\pm$ 0.0	12.5 $\pm$ 0.0	12.2 $\pm$ 0.0	11.8 $\pm$ 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	18.4 $\pm$ 0.0	18.2 $\pm$ 0.0	18.0 $\pm$ 0.0	17.5 $\pm$ 0.0	16.6 $\pm$ 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	23.4 $\pm$ 0.0	23.2 $\pm$ 0.0	22.9 $\pm$ 0.0	22.2 $\pm$ 0.0	21.0 $\pm$ 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 3. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-7B on 8xA100 40 GB GPUs when generating 256 tokens. An x indicates the model does not have the required memory to run.

Batch	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	1.7 $\pm$ 0.0	1.4 $\pm$ 0.0	1.1 $\pm$ 0.0	0.7 $\pm$ 0.0	X	2.0 $\pm$ 0.0	2.0 $\pm$ 0.0	1.9 $\pm$ 0.0	1.8 $\pm$ 0.0	1.8 $\pm$ 0.0	1.8 $\pm$ 0.0	1.8 $\pm$ 0.0	1.8 $\pm$ 0.0	0.6 $\pm$ 0.0	0.4 $\pm$ 0.0	1.6 $\pm$ 0.0	1.6 $\pm$ 0.0	1.5 $\pm$ 0.0	0.5 $\pm$ 0.0	0.3 $\pm$ 0.0	2.3 ± 0.0
64	2.9 $\pm$ 0.0	2.3 $\pm$ 0.0	1.6 $\pm$ 0.0	X	X	3.6 $\pm$ 0.0	3.6 $\pm$ 0.0	3.6 $\pm$ 0.0	3.4 $\pm$ 0.0	3.4 $\pm$ 0.0	3.5 $\pm$ 0.1	3.5 $\pm$ 0.0	2.9 $\pm$ 0.1	0.7 $\pm$ 0.0	0.4 $\pm$ 0.0	3.0 $\pm$ 0.0	2.9 $\pm$ 0.1	2.4 $\pm$ 0.0	0.6 $\pm$ 0.0	0.4 $\pm$ 0.0	4.2 ± 0.0
128	4.0 $\pm$ 0.0	2.9 $\pm$ 0.0	X	X	X	5.8 $\pm$ 0.0	5.7 $\pm$ 0.2	5.6 $\pm$ 0.0	5.6 $\pm$ 0.0	5.7 $\pm$ 0.0	5.5	4.7 $\pm$ 0.1	3.0	0.8	0.4	4.8	3.8 $\pm$ 0.1	2.6	0.7	0.4	6.8 ± 0.0
256	5.7 $\pm$ 0.0	X	X	X	X	9.6 $\pm$ 0.0	9.3 $\pm$ 0.0	9.4 $\pm$ 0.0	9.2 $\pm$ 0.0	8.8 $\pm$ 0.0	8.0	5.5 $\pm$ 0.1	3.2	0.8	0.4	6.1	4.3 $\pm$ 0.1	2.7	0.7	0.4	11.4 ± 0.0
512	X	X	X	X	X	13.4 $\pm$ 0.0	13.3 $\pm$ 0.0	13.2 $\pm$ 0.0	12.9 $\pm$ 0.0	12.3 $\pm$ 0.0	4.7	2.7 $\pm$ 0.0	1.6	0.8	0.4	4.1	2.4 $\pm$ 0.0	1.4	0.8	0.4	16.1 ± 0.0
1024	X	X	X	X	X	15.6 $\pm$ 0.0	15.5 $\pm$ 0.0	15.3 $\pm$ 0.0	14.8 $\pm$ 0.0	14.0 $\pm$ 0.0	4.9 $\pm$ 0.0	2.8 $\pm$ 0.0	1.6 $\pm$ 0.0	0.8 $\pm$ 0.0	0.4 $\pm$ 0.0	4.2 $\pm$ 0.0	2.5 $\pm$ 0.0	1.4 $\pm$ 0.0	0.7 $\pm$ 0.0	0.4 $\pm$ 0.0	18.5 ± 0.0

Table 4. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-13B on 8xA100 40 GB GPUs when generating 128 tokens. An x indicates 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	1.7 ± 0.0	1.4 ± 0.0	1.1 ± 0.0	0.7 ± 0.0	X	1.9 ± 0.0	1.9 ± 0.0	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	3.5 ± 0.0	3.5 ± 0.0	3.4 ± 0.0	3.2 ± 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 ± 0.1	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	X	5.6 ± 0.0	5.5 ± 0.0	5.3 ± 0.0	5.4 ± 0.0	5.2 ± 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	8.9 ± 0.0	8.7 ± 0.0	8.8 ± 0.0	8.7 ± 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	12.3 ± 0.0	12.3 ± 0.0	12.2 ± 0.0	12.0 ± 0.0	11.4 ± 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 5. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-13B on 8xA100 40 GB GPUs when generating 256 tokens. An x indicates 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	1.4 ± 0.0	1.2 ± 0.0	1.0 ± 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 ± 0.0	1.1 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.6 ± 0.0
64	2.5 ± 0.0	2.3 ± 0.1	2.1 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	2.6 ± 0.0	2.6 ± 0.0	2.5 ± 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 ± 0.0	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	4.1 ± 0.0	4.1 ± 0.0	4.0 ± 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	4.4 ± 0.3
256	6.0 ± 0.0	5.3 ± 0.0	4.4 ± 0.0	X	X	6.6 ± 0.0	6.6 ± 0.0	6.5 ± 0.0	6.3 ± 0.0	5.9 ± 0.0	5.1	3.9	2.8	0.8	0.4	4.4	3.3	2.4	0.8	0.4	7.2 ± 0.2
512	7.0 ± 0.0	6.0 ± 0.0	X	X	X	8.2 ± 0.0	8.1 ± 0.0	8.0 ± 0.0	7.8 ± 0.0	7.3 ± 0.0	4.2	2.7	1.5	0.8	0.4	3.6	2.4	1.4	0.8	0.4	8.8 ± 0.1
1024	X	X	X	X	X	9.4 ± 0.0	9.2 ± 0.0	9.0 ± 0.0	8.5 ± 0.0	7.6 ± 0.0	4.3	2.8	1.6	0.8	0.4	3.7	2.5	1.4	0.8	0.4	9.9 ± 0.2
2048	X	X	X	X	X	10.4 ± 0.0	10.3 ± 0.0	10.0 ± 0.0	9.4 ± 0.0	8.5 ± 0.0	4.3	2.7	1.5	0.8	0.4	3.7	2.4	1.4	0.8	0.4	11.0 ± 0.0
4096	X	X	X	X	X	11.1 ± 0.0	11.0 ± 0.0	10.7 ± 0.0	10.2 ± 0.0	9.4 ± 0.0	4.0	2.6	1.4	0.8	0.4	3.5	2.3	1.3	0.7	0.4	11.6 ± 0.0

Table 6. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-34B on 8xA100 40 GB GPUs when generating 128 tokens. An x indicates 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	1.3 ± 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	1.5 ± 0.1
64	2.5 ± 0.0	2.4 ± 0.0	2.1 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.4 ± 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 ± 0.1
128	3.8 ± 0.0	3.4 ± 0.0	2.8 ± 0.0	2.1 ± 0.0	X	4.1 ± 0.0	4.1 ± 0.0	4.0 ± 0.0	4.0 ± 0.0	3.8 ± 0.0	3.7	3.0	2.2	0.7	0.4	3.2	2.6	2.0	0.7	0.4	4.5 ± 0.1
256	5.8 ± 0.0	5.3 ± 0.0	4.3 ± 0.0	X	X	6.5 ± 0.0	6.5 ± 0.0	6.4 ± 0.0	6.2 ± 0.0	5.8 ± 0.0	5.0	3.9	2.7	0.8	0.4	4.2	3.3	2.3	0.7	0.4	7.1 ± 0.2
512	6.8 ± 0.0	5.9 ± 0.0	X	X	X	8.0 ± 0.0	8.0 ± 0.0	7.9 ± 0.0	7.6 ± 0.0	7.2 ± 0.0	3.9	2.6	1.5	0.8	0.4	3.5	2.3	1.4	0.7	0.4	8.8 ± 0.1
1024	X	X	X	X	X	9.2 ± 0.0	9.1 ± 0.0	8.8 ± 0.0	8.3 ± 0.0	7.5 ± 0.0	3.9	2.6	1.4	0.8	0.4	3.6	2.4	1.4	0.7	0.4	9.9 ± 0.0
2048	X	X	X	X	X	10.3 ± 0.0	10.1 ± 0.0	9.8 ± 0.0	9.3 ± 0.0	8.4 ± 0.0	4.0	2.6	1.5	0.8	0.4	3.6	2.4	1.4	0.7	0.4	11.0 ± 0.0

Table 7. End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-34B on 8xA100 40 GB GPUs when generating 256 tokens. An x indicates the model does not have the required memory to run.

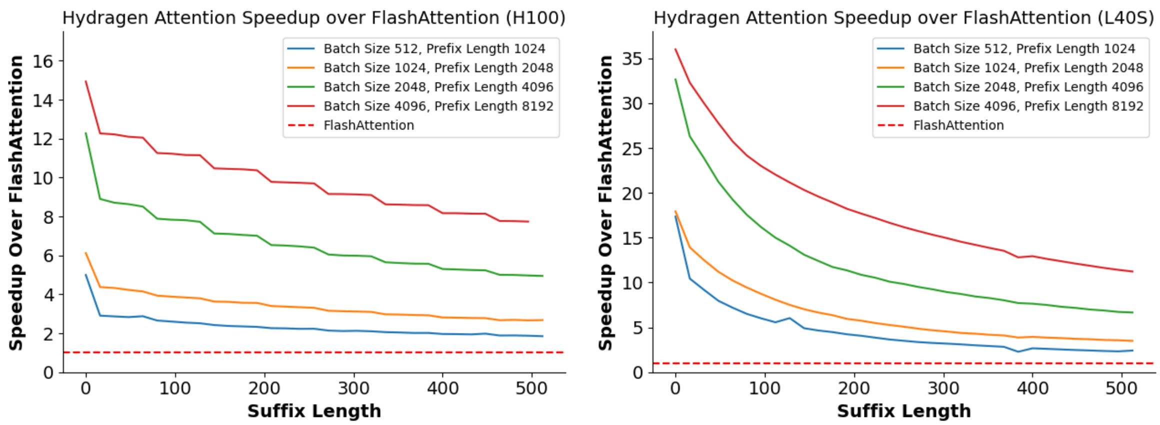


Figure 8. Speedup of Hydragen attention over FlashAttention for various batch sizes, shared prefix lengths and suffix lengths on an H100 (left) and an L40S (right) GPU.

D.2. Microbenchmarks

We repeat the A100 microbenchmark experiment from Section 4.2 on H100 and L40S GPUs, reporting our results in Figure 8. The L40S has the highest ratio of FLOPs 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 speedups on 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.

E. Experiment Details

E.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 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 we 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 4.1 and 4.1. 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 D.1, we use one warmup and one timing iteration.

E.2. Microbenchmarks

In each microbenchmark, we run 1000 iterations of warmup before reporting the mean running time across 1000 trials. Between iterations, 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.

E.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* (24), 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}?”, with the corresponding answer being {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.

E.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.