LEANATTENTION: HARDWARE-AWARE SCALABLE ATTENTION MECHANISM FOR THE DECODE-PHASE OF TRANSFORMERS

Rya Sanovar [1] **Srikant Bharadwaj** [1] **Renee St. Amant** [1] **Victor Rühle** [1] **Saravan Rajmohan** [1]

ABSTRACT

Transformer-based large language models are memory hungry and incur significant inference latencies even on cutting edge AI-accelerators, such as GPUs. Specifically, the time and memory complexity of the attention operation is quadratic in terms of the total context length, i.e., prompt and output tokens.

To that end, we propose LeanAttention, a scalable, hardware-efficient, "exact" attention acceleration mechanism for the decode-phase of transformer-based models. LeanAttention enables scaling the attention mechanism for the challenging case of long context lengths by re-designing the attention execution flow for the decode-phase. As a result, we achieve an average of 1.73x speedup in attention execution compared to FlashDecoding, with up to 2.18x speedup for 256k context length.

1 INTRODUCTION

Transformer-based (Vaswani et al., 2017) language models (Achiam et al., 2023; Touvron et al., 2023; Zhang et al., 2022; Li et al., 2023; Chowdhery et al., 2023) have revolutionized the field of natural language processing (NLP) and found applications across diverse domains (Spataro & Inc.; Inc.). These powerful models, fueled by massive amounts of data and sophisticated architectures, have become indispensable tools for tasks such as machine translation (Kenton & Toutanova, 2019), question answering (OpenAI), text generation (OpenAI), and sentiment analysis.

The core of the transformer architecture is the powerful component of self-attention. However, execution of the self-attention mechanism is slow and suffers from a large memory footprint, especially when dealing with long contexts. A standard implementation of self-attention has quadratic time and memory complexity with respect to total sequence length, which leads to scalability challenges as model sizes (Brown et al., 2020) and supported context lengths increase (Anthropic; Zhang et al., 2024; Liu et al., 2023). Despite these scalability challenges, we see a trend of state-of-the-art models supporting greater and greater context lengths, with some production models supporting contexts as long as hundreds of thousands of tokens. Support for long contexts can improve a model's utility by enabling an increasingly rich context, which is particularly beneficial in a range of applications (e.g. RAG involving numerous long documents) that improve relevance, coherence, and user experience.

To mitigate LLM scalability challenges, mechanisms like FlashAttention (Dao et al., 2022) and FlashAttention-2/3 (Dao, 2023; Shah et al., 2024) have been developed. FlashAttention brings IO-awareness to attention computation by reducing slow reads and writes to and from the GPU's global memory (Ivanov et al., 2021). Instead, it computes attention in the faster shared memory using a tiling strategy. It allows for parallelization over batch size and number of heads. FlashAttention-2 builds on FlashAttention to further optimize attention computation by enabling parallelization over input sequence length (or query length).

While these optimizations provide significant improvements, these mechanisms only give performance benefits for a subset of problem sizes (i.e. query length, context length, batch size, and number of heads). Their utilization of underlying hardware resources is mostly optimized for problem sizes encountered in the prefill-phase of transformer-based models, and often results in critically low hardware utilization for problem sizes found in the decode-phase (see Figure 1). By overlooking the distinct behavior of attention during the decode phase versus the prefill phase, these mechanisms miss out on potential performance gains that could be achieved by *efficiently* exploiting the parallelization capabilities of the underlying hardware.

In decoder-only transformer models, the inference process for a single request involves multiple forward passes through the model where output tokens are generated sequentially (Kim et al., 2023). This inference procedure

[1]Microsoft. Correspondence to: Rya Sanovar <ryasanovar6@gmail.com>, Srikant Bharadwaj <srikant.bharadwaj@microsoft.com>.

Proceedings of the 8^{th} MLSys Conference, Santa Clara, CA, USA, 2025.

inherently comprises of two distinct computational phases due to the practice of reusing (caching) the key-value tensors of the previously computed tokens (Pope et al., 2023). The first phase is the *prefill-phase* (also known as *prompt-computation phase*) where attention is computed of the entire input prompt against itself to generate the first output token. It's followed by the *decode-phase*, also known as the *token-generation phase*, which executes in an auto-regressive manner (Vaswani et al., 2017). The current token computes attention against itself and the entire cached context (*kv-cache*) of previous tokens in the sequence to produce the next token. With the push towards longer contexts, this cached context length can get extremely long, exceeding more than hundreds of thousands of tokens in length (Li et al., 2024; Zhang et al., 2024; Fu et al., 2024; Liu et al., 2023). Despite state-of-the-art batching techniques (Yu et al., 2022b) and attention partitioning mechanisms (Dao et al., 2022; Dao, 2023; Dao et al.; Ye et al., 2024), the lack of a *smart parallelized execution* of attention along this long context makes the decode-phase slow, bound by memory bandwidth (Williams et al., 2009) and capacity (Kim et al., 2023).

Efficient parallelization of attention over the context dimension is highly necessary. Although mechanisms like FlashDecoding (Dao et al.) and FlashInfer (Ye et al., 2024) enable parallelization over context length, they do it via the fixed-split partitioning strategy which provides the hardware with imbalanced loads and as a consequence suffers from hardware resource under-utilization that slows down attention computation. Further, attention optimizations (Agrawal et al., 2024) are increasingly relying on batching requests of unequal context lengths to improve overall throughput, but suffer from similar inefficiencies due of the partitioning strategies adopted by FlashAttention and FlashDecoding.

To address these limitations, we introduce LeanAttention, a generalized exact-attention mechanism that enables *efficient* parallelization across all problem size dimensions and provides the hardware with equalized loads by virtue of its streaming nature. It ensures perfect quantization efficiency, i.e., 100% GPU occupancy for all problem sizes, delivers a runtime speedup during attention for long contexts, and scales efficiently in multi-GPU scenarios with tensor parallelism.

Overall, our contributions are as follows:

- Identify the limitations of state-of-the-art attention execution optimizations on GPUs during the decode-phase of transformer-based models. (subsection 3.2)
- Identify the *associative nature* of the softmax re-scaling operator that enables it to function as a reductive operator.
- Leverage this crucial associativity property to employ a stream-K style (Osama et al., 2023) partitioning in LeanAttention that *always* provides *equal* compute loads to every compute unit in the hardware system (as shown in Figure 1) for *any* problem size, thus giving near 100% hardware occupancy and delivering speedup irrespective of problem size and hardware architecture (subsection 4.3).
- Expatiate LeanAttention's versatility and generalizability, where FlashAttention-2 and FlashDecoding can be recovered as special cases of it. (subsection 4.3)

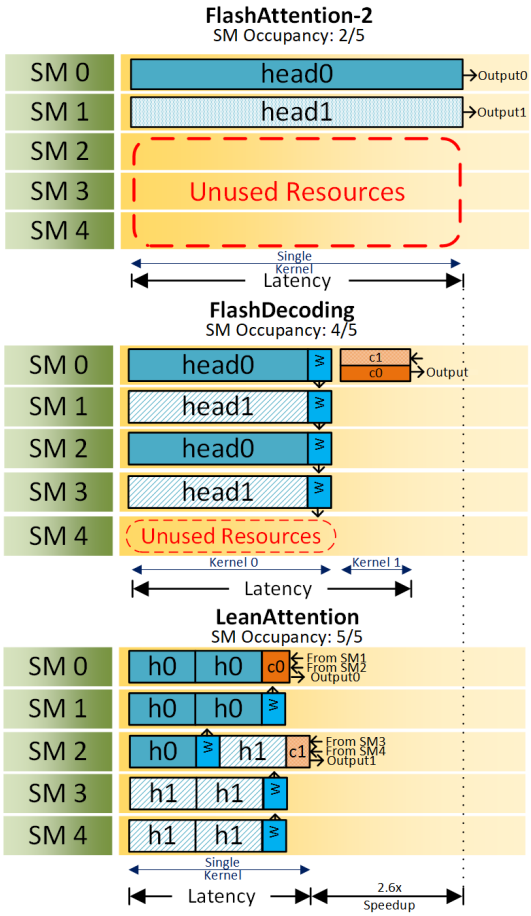

Figure 1. Attention execution schedule of FlashAttention-2 (Dao, 2023), FlashDecoding (Dao et al.) (fixed-split), and LeanAttention across a hypothetical 5 SM GPU for 2 heads. LeanAttention splits the context into optimal LeanTiles (shown with 5 tiles per head).

LeanAttention results in an average of 1.73x latency speedup over FlashDecoding for the decode phase of transformer-based models and up to 2.18x speedup for 256k context length, while maintaining a near 100% GPU occupancy irrespective of problem size.

2 Background

In this section, we provide the required background on Standard Attention (Vaswani et al., 2017) and FlashAttention-2 (Dao, 2023).

2.1 Standard Attention

For a given input tensor with dimensions of batch size B, query length N_q, key/value sequence length (also known as context length) N_k, and hidden dimension D, multi-head attention typically splits attention computation into h number of heads along the hidden dimension, with each head responsible for computing attention independently for a head dimension of size $d = D/h$.

The query and context lengths may not always be equal (Pope et al., 2023). For instance, the prefill-phase of generative decoder-only transformers such as GPT-4 (Achiam et al., 2023) or Phi-2 (Li et al., 2023) has sequence lengths $N_q = N_k = N$, but in their decode-phase the context length increments by 1 after every autoregressive step of decode generation, while the query length (for a given batch instance) is the singular token that was generated in the previous n-th time step, i.e., $N_q = 1$ and $N_k = N + n$.

The query matrix $Q \in R^{N_q \times d}$ and key K and value V matrices $\in R^{N_k \times d}$ are inputs to the following equation which is computed independently by different batch instances and heads. The output matrix $O \in R^{N_q \times d}$ is obtained in essentially three steps as shown in Equation 1. Table 1 summarises the three operations involved in self-attention along with their corresponding dimensions in both prefill and decode-phase at time step $n = 0$.

$$S = QK^T,\ P = softmax(\frac{S}{\sqrt{d}}),\ O = PV \qquad (1)$$

Standard attention implementation involves computing the large intermediate matrices, namely the attention score matrix $S \in R^{N_q \times N_k}$ and the softmax matrix $P \in R^{N_q \times N_k}$ and storing them in global memory. These intermediate matrices need to be stored in the global memory because the computation of the softmax matrix P requires *a priori* knowledge of all tokens in a given row. Specifically, the row-wise maximum and exponential sum of tokens in a row need to be computed beforehand to calculate the softmax-*ed* value of each element in the row.

The computational complexity of standard attention is on the order of $O(N_q N_k d)$, with the two matrix multiplications (MatMul's) contributing to the majority of it. Due to slow global memory access speeds, storing and retrieving these intermediate matrices (Ivanov et al., 2021) results in long latencies and incurs a large memory footprint, both in the order of $O(N_q N_k)$.

2.2 Flash Attention-2

To mitigate the memory footprint and access overheads (Ivanov et al., 2021) associated with storing the S and P matrices, FlashAttention employs kernel fusion of the three operations shown in Equation 1: $query \times key$ MatMul, softmax and $attn\_score \times value$ MatMul, effectively avoiding intermediate global memory reads and writes. To this end, it employs the tiling strategy.

Operation	Type	Operation Dimension	
		Prefill	Decode
$query \times key$	MatMul	$N \times d \times N$	$1 \times d \times N$
$softmax$	EleWise	$N \times N$	$1 \times N$
$attn\_score \times value$	MatMul	$N \times N \times d$	$1 \times N \times d$

Table 1. Operations in self-attention. Matrix multiplications are described in the $M \times N \times K$ format.

By utilizing the online softmax algorithm (Milakov & Gimelshein, 2018), FlashAttention only requires a single pass over an entire row of tokens to compute their softmax, bypassing the issue of *a priori* knowledge in standard attention. This helps leverage the tiling strategy which partitions the attention output matrix O into independent output tiles (i.e. the computation of an output tile is independent of the computation of other output tiles). A grid of *cooperative thread arrays* (CTAs)[1] is launched, each computing a given output tile of the output matrix O. The input matrices Q, K and V are partitioned into smaller tiles too. While computing the output tile corresponding to a given query tile, the necessary key/value tiles are brought into shared memory in a *sequential* manner and the output tile is iteratively updated and corrected by the right scaling factors. This on-chip updation avoids the need of storing the intermediate S and P matrices in global memory. In addition to parallelizing computation over batches and heads like FlashAttention, FlashAttention-2 further parallelizes over the query length dimension, as the attention computation of output tiles along this length is also independent. This results in a 2x speedup over FlashAttention.

Thus, the tiling strategy ensures that the extra global memory space required by FlashAttention-2 is $O(N_q)$ (needed to store the scaling factors for the backward pass), an impressive improvement in memory footprint over the $O(N_q \times N_k)$ in traditional attention. The additional parallelism over query length helps it reach 50-70% of peak theoretical FLOPS/s and increases hardware occupancy in the prefill phase. FlashAttention-2 was augmented to FlashAttention-3 (Shah et al., 2024), which is specifically fine-tuned for execution on Hopper GPU's (Luo et al., 2024) to exploit its low-precision and asynchronous hardware capabilities. Its optimizations are orthogonal to this work. Other related techniques such as Ring Attention (Liu et al., 2023) and Striped Attention (Brandon et al., 2023), optimize inter-

[1]Blocks of GPU threads are coscheduled in CTAs, which virtualize the hardware's streaming multiprocessor cores (SMs)

GPU occupancy and are also orthogonal to the technique presented in this work.

3 Challenges In The Decode Phase

Prior to outlining our methodology for LeanAttention, to set the stage for our approach, we delve into some of the challenges encountered in the decode phase of LLM inference, as well as the limitations of FlashAttention-2 optimizations in the decode phase.

3.1 Time Spent in Decode Phase

As we've discussed, modern generative LLM inference comprises of two computationally distinct phases: the prefill phase followed by the decode phase. In the prefill phase, self-attention is computed for the entire input prompt. The query length N_q in this phase is the same as the context length N_k, i.e., ($N_q = N_k = N$). Whereas, the decode phase begins generating each subsequent output token in auto-regressive iterations. For each decode iteration, the query length is a single token $N_q = 1$, and the context length N_k could be very long, in the order of more than thousands of tokens depending on the auto-regressive step and input query length.

Figure 2 depicts the processing time breakdown of prefill and decode, with their further breakdown into time spent in the attention layer and the rest of the layers in a transformer block (i.e., $Q/K/V$ activation layers, feed-forward linear layers etc).

While the large matrix multiplications found in the linear layers of the prefill phase are heavily optimized (the entire prefill phase taking up only 10% of the timeshare even for a high prompt:output ratio), the decode phase presents a different challenge. During the decode phase, where the query length is only 1 token long, linear layers perform matrix multiplications on very narrow matrices which do not provide enough work to occupy the GPU. MatMul partitioning strategies like Stream-K (Osama et al., 2023) can be leveraged to efficiently partition these narrow matrices and accelerate their computation, preventing the linear layers from becoming a bottleneck during decode. However, the attention layer, with the existing attention partitioning techniques (Dao et al., 2022; Dao, 2023; Dao et al.; Ye et al., 2024) experiences longer latencies along with significant under-utilization of hardware resources during this phase. This makes leveraging efficient parallelism along context length (N_k) during attention a crucial aspect in increasing SM occupancy and reducing decode phase processing time.

As the number of output tokens generated increases, the context length becomes longer and thus the proportion of time spent in decode relative to prefill becomes larger. Figure 2 depicts this imbalance in processing time spent in prefill and decode during attention. Even with a prompt input to output token ratio of 8:1, more than 50% of the processing time is consumed by the decode phase, taking up to nearly 80% of the timeshare for longer prompt sizes. Additionally, other layers of the decode phase such as QKV and FFN layers can be optimized using state-of-the-art MatMul partitioning techniques like Stream-K. These operations are also typically quantized to lower data formats such as INT8 to further enhance their efficiency. As a result, the decode attention operation can constitute up to 50-60% of the total duration of inference as shown.

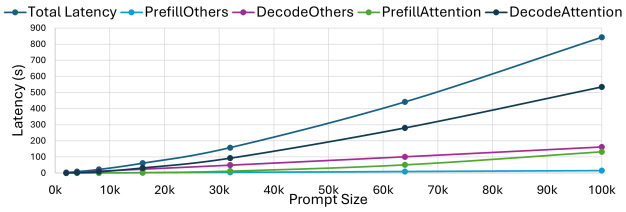

Figure 2. Timeshare of decode attention compared to other stages for different prompt sizes with 8:1 token ratio for Phi-3 Medium model with single batch size.

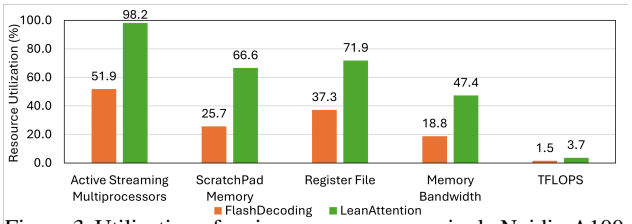

Figure 3. Utilization of various resources on a single Nvidia-A100-80GB GPU in LeanAttention compared to FlashDecoding at Heads=56 and BS=1 measured using Nsight Compute.

3.2 Limitations of FlashAttention-2 for Decode

In both the prefill and decode phase, FlashAttention-2 traverses the context length dimension (N_k) sequentially, i.e., it updates the attention output for a given query tile by bringing in the key/value tiles into shared memory in a *sequential* manner. While FlashAttention-2 does parallelize over query length (N_q) to increase SM occupancy, this additional mode of parallelism has limited parallelization capacity in the decode phase where the query is a single token ($N_q = 1$). Not parallelizing attention computation along context length makes FlashAttention-2 (Dao, 2023) suffer from extremely low SM occupancy during decode as depicted in Figure 1. This means that at any given point in time, the number of CTAs in flight on the GPU is directly proportional to the number of query tiles, and, therefore, to the query length-regardless of the context length.

More explicitly, for a single batch instance, the maximum number of heads for state-of-the-art LLMs barely occupy the compute resources of modern hardware architecture systems during the decode phase where query length $N_q = 1$. For example, for a model with 128 heads, its decode phase would

suffer from severe under-utilization of an 8 GPU A100 system that has 864 compute cores at its disposal. Unlike the prefill phase, decode phase can offer parallelization only across batch size and number of heads for FlashAttention-2.

Processor occupancy in FlashAttention-2 could be improved by increasing the batch size or number of heads, the other two modes of parallelization it addresses. Intuitively, having larger batch sizes in the decode phase could provide enough work to every compute resource to fully occupy the GPU, but this introduces other challenges and limitations. Due to increasingly large model sizes, the need to independently cache KV context for every batch instance would likely exceed the memory capacity of the hardware system. Moreover, scheduling overheads (Yu et al., 2022a) for efficiently batching queries along with the challenges of batching low SLA queries would increase inference latency and challenge utilization.

Without having to resort to larger batch sizes as the sole solution to resolving the GPU occupancy issue (which is limited by available memory capacity), the large context length in the decode phase would benefit from partitioning its workload across different SMs efficiently. This motivates the need for smarter attention decomposition techniques which can efficiently distribute the workload across the cores without resorting to larger batch sizes.

3.3 Limitations of Related Work

FlashDecoding, which is FlashAttention-2 with *fixed-split* partitioning, has recently been proposed (Dao et al.; Hong et al., 2024; git), where attention computation is also partitioned along context length N_k. FlashInfer (Ye et al., 2024) implements an identical fixed-split partitioning of attention for single-requests in the decode phase. For the case of batched-requests in decode, FlashInfer implements an optimized version of PagedAttention for efficient KV cache storing and fetching.

Fixed-split is a general matrix multiplication decomposition scheme that we briefly describe here. Given a MatMul computation problem with matrices A $(M \times K)$ and B $(N \times K)$ to obtain a matrix C $(M \times N)$ where $C = AB^T$, to optimize concurrent computation, the fixed-split mechanism partitions the K-mode of the A and B matrices into s batches based on a fixed splitting factor s provided dynamically at run time. This launches s times the CTAs as launched without fixed-split, which are computing partial products of the output tiles of the C matrix concurrently. Fixed-split utilizes the associativity of addition in the inner product of a MatMul to later reduce or "fix-up" the partially computed C matrices and produce the final C matrix. The concurrency from fixed-split reduces latency and simultaneously increases hardware occupancy at the cost of an additional reduction at the end.

FlashDecoding++ (Hong et al., 2024) achieves speedup over FlashDecoding by approximating the softmax operation to remove the sequential dependencies it creates in attention. Notably, this approach compromises on accuracy and its implementation is limited to certain model architectures. FlashDecoding++, as well as other techniques that focus on softmax approximations to achieve speedup (like ConSmax (Liu et al., 2024) and Softermax (Stevens et al., 2021)), can be seamlessly integrated into LeanAttention.

Despite these improvements, fixed-split used in these mechanisms (Dao et al.; Ye et al., 2024; Hong et al., 2024) is a non-optimal load balancing strategy. While this method of partitioning would improve speedup and occupy the GPU well for some attention workloads, it's an inefficient strategy to adopt for the entire problem space because often results in partially full waves of attention computation that suffer from quantization inefficiencies, i.e. low GPU occupancy due to imbalanced loads, and loses out on performance gains it could get from the idle resources otherwise (depicted in Figure 1). While increasing the number of splits could help occupy the GPU better, it would result in reduction overheads that scale with the split factor and would allocate minimal work to each SM making it an inefficient use of register space.

While fixed-split partitioning along the context length in FlashDecoding does occupy a larger number of compute resources on the GPU compared to vanilla FlashAttention-2, its GPU occupancy varies greatly with problem size, split factor and the number of compute units in the hardware system as shown in Figure 3, making it unlikely for FlashDecoding and its variants to reach perfect quantization efficiency for *all* problem sizes and hardware systems. Moreover, the problem of quantization inefficiencies with these mechanisms would be particularly exacerbated in the common cases of processing a batch of requests of heterogenous context lengths (Agrawal et al., 2024). In contrast, LeanAttention, with its *stream-K-style decomposition* (discussed in section 4), will *always* provide well-balanced loads to every compute unit in the hardware system to reach near 100% GPU occupancy for all problem sizes and hardware architectures, making it perfectly adept at handling batched requests of unequally sized contexts.

4 LeanAttention

LeanAttention is an optimized scalable attention execution mechanism. It provides extensive parallelism across all modes of the attention tensor, with well-balanced computation workloads to each CTA ensuring close to 100% SM occupancy while delivering a runtime speedup in attention execution.

First, we identify that the *associative property of softmax re-*

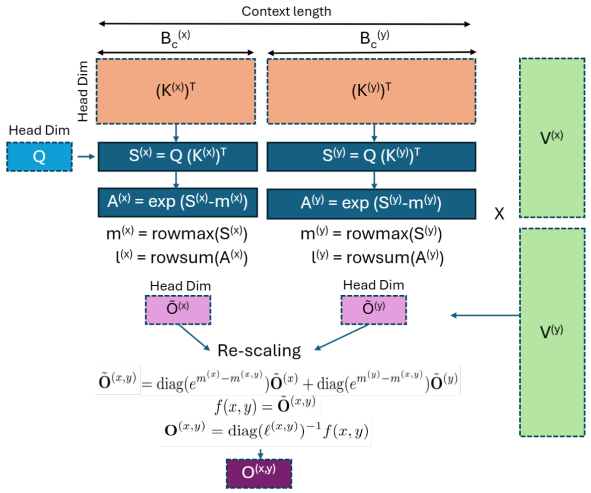

Figure 4. Illustrative diagram showing LeanAttention's partitioning strategy with two differently sized work volumes of a head assigned to different CTAs. The un-scaled outputs are independently computed and re-scaled later in a reduction operation. Note that this can be generalized to any arbitrary-sized work volume split.

scaling enables us to treat it as a reduction operator along the context length dimension and allows us to split the workload (i.e. KV tensors) of a single head into unequally sized blocks if and when needed (described in subsection 4.1), while maintaining the lossless exactness of attention.

Second, we identify the smallest optimal granularity of decomposition in attention computation, termed as *LeanTile* (subsection 4.2), which can be linearly mapped on the hardware resources in a flexible style akin to stream-k decomposition of matrix multiplications subsection 4.3). Multiple such LeanTiles belonging to either single or multiple attention outputs will constitute a workload assigned to a CTA.

4.1 Softmax Re-scaling as Reduction

LeanAttention's decomposition results in splits of work for a given SM that are not always equal in size, i.e., the K/V tensors of a given query tile are not dispatched in same-sized blocks to different SMs (unlike FlashDecoding (Dao et al.), FlashInfer (Ye et al., 2024)).

To reduce these partial attention outputs that result from differently sized blocks, we use a softmax re-scaling operator. This requires us to identify softmax re-scaling's associativity property that allows it to correctly reduce blocks of unequal sizes, i.e., application of softmax re-scaling as a reduction operator will give the same exact attention output with no loss in accuracy, *regardless of the way the work might be split, whether in same-sized blocks or arbitrary differently sized blocks*.

Without loss of generality, we describe this process of reduction to obtain one row vector of the attention score matrix $\mathbf{S}$, of the form $\begin{bmatrix}\mathbf{S}^{(x)} & \mathbf{S}^{(y)}\end{bmatrix}$ consisting of some unequal length vectors $\mathbf{S}^{(x)}, \mathbf{S}^{(y)}$ where $\mathbf{S}^{(x)} \in \mathbb{R}^{1\times B_c^{(x)}}$ and $\mathbf{S}^{(y)} \in \mathbb{R}^{1\times B_c^{(y)}}$, where 1 is the query length and $B_c^{(x)}$ and $B_c^{(y)}$ are the unequal context lengths. The vectors $\mathbf{S}^{(x)}$ and $\mathbf{S}^{(y)}$ were computed from $\mathbf{Q}\times(\mathbf{K}^{(x)})^T$ and $\mathbf{Q}\times(\mathbf{K}^{(y)})^T$ (illustrated in Figure 4). Note that to generalize this procedure for blocks of any size, the context length of $\mathbf{K}^{(x)}$ and $\mathbf{K}^{(y)}$ are $B_c^{(x)}$ and $B_c^{(y)}$ and are not necessarily equal.

The attention computation is split into two key parts. The first part involves calculation of an "un-scaled" version of $\mathbf{O}^{(i)}$ (where i is either x or y) along with statistics $m^{(i)}$ and $\ell^{(i)}$:

$$\begin{aligned}
&\mathbf{S}^{(i)} = \mathbf{Q}(\mathbf{K}^{(i)})^T \in \mathbb{R}^{1\times B_c^{(i)}}\\
&m^{(i)} = \text{rowmax}(\mathbf{S}^{(i)}) \in \mathbb{R}^{1\times 1}\\
&\ell^{(i)} = \text{rowsum}(e^{\mathbf{S}^{(i)}-m^{(i)}}) \in \mathbb{R}^{1\times 1}\\
&A^{(i)} = exp(\mathbf{S}^{(i)} - m^{(i)}) \in \mathbb{R}^{1\times B_c^{(i)}}\\
&\mathbf{O}^{(i)} = A^{(i)}\mathbf{V}^{(i)} \in \mathbb{R}^{1\times d}
\end{aligned}$$

Softmax Re-scaling Operation. The second part involves re-scaling the "un-scaled" outputs $\mathbf{O}^{(i)}$ using the previously computed statistics $m^{(i)}$ and $\ell^{(i)}$.
We define the softmax re-scaling operation $f(x,y)$ for two intermediate outputs $\mathbf{O}^{(x)}$ and $\mathbf{O}^{(y)}$ as follows:

$$\begin{aligned}
&m^{(x,y)} = \max(m^{(x)}, m^{(y)})\\
&\ell^{(x,y)} = e^{m^{(x)}-m^{(x,y)}}\ell^{(x)} + e^{m^{(y)}-m^{(x,y)}}\ell^{(y)}\\
&f(x,y) = \text{diag}(e^{m^{(x)}-m^{(x,y)}})\tilde{\mathbf{O}}^{(x)} + \text{diag}(e^{m^{(y)}-m^{(x,y)}})\tilde{\mathbf{O}}^{(y)}\\
&f(x,y) = \tilde{\mathbf{O}}^{(x,y)}\\
&\mathbf{O}^{(x,y)} = \text{diag}(\ell^{(x,y)})^{-1}f(x,y)
\end{aligned}$$

Proof of Associativity The associative nature of softmax re-scaling $f(x,y)$ allows us to reduce intermediate outputs produced from key/value vectors of different lengths in LeanAttention. We shall briefly prove that $f(f(x,y),z) = f(x,f(y,z)) = f(x,y,z)$, where: $f(x,y) = \tilde{\mathbf{O}}^{(x,y)}$, $f(y,z) = \tilde{\mathbf{O}}^{(y,z)}$ and $f(x,y,z) = \tilde{\mathbf{O}}^{(x,y,z)}$.

Proving that $f(f(x,y),z) = f(x,y,z)$:

$$\begin{aligned}
f(x,y) &= \tilde{\mathbf{O}}^{(x,y)}\\
f(f(x,y),z) &= \text{diag}(e^{m^{(x,y)}-m^{((x,y),z)}})\tilde{\mathbf{O}}^{(x,y)}\\
&\quad + \text{diag}(e^{m^{(z)}-m^{((x,y),z)}})\tilde{\mathbf{O}}^{(z)}\\
&= \text{diag}(e^{m^{(x,y)}-m^{(x,y,z)}})\tilde{\mathbf{O}}^{(x,y)}\\
&\quad + \text{diag}(e^{m^{(z)}-m^{(x,y,z)}})\tilde{\mathbf{O}}^{(z)}\\
&= \text{diag}(e^{m^{(x,y)}-m^{(x,y,z)}})\\
&\quad \times (\text{diag}(e^{m^{(x)}-m^{(x,y)}})\tilde{\mathbf{O}}^{(x)}\\
&\quad + \text{diag}(e^{m^{(y)}-m^{(x,y)}})\tilde{\mathbf{O}}^{(y)})\\
&\quad + \text{diag}(e^{m^{(z)}-m^{(x,y,z)}})\tilde{\mathbf{O}}^{(z)}
\end{aligned}$$

$$
\begin{aligned}
&= \text{diag}(e^{m^{(x)} - m^{(x,y,z)}})\tilde{\mathbf{O}}^{(x)} \\
&+ \text{diag}(e^{m^{(y)} - m^{(x,y,z)}})\tilde{\mathbf{O}}^{(y)} \\
&+ \text{diag}(e^{m^{(z)} - m^{(x,y,z)}})\tilde{\mathbf{O}}^{(z)} \\
&= \tilde{\mathbf{O}}^{(x,y,z)} = f(x,y,z)
\end{aligned}
$$

Therefore, $f(f(x,y),z) = f(x,y,z)$ and similarly $\ell^{((x,y),z)} = \ell^{(x,y,z)}$. For brevity, we omit the proof of $f(x,f(y,z)) = f(x,y,z)$, but it can be deduced in a similar manner.

This associativity of softmax re-scaling is leveraged in LeanAttention to concurrently calculate the "partial" outputs produced from unequally sized KV blocks and then "reduce" them to obtain exact attention.

4.2 LeanTile

To enable us to efficiently distribute the work of computing the attention output tiles, we define the smallest granularity of a KV block as a *LeanTile*. A single LeanTile iteration computes "local attention" across a subset of tokens along the N_k dimension to generate an un-scaled attention output as shown in the grey box in Figure 5. Algorithm 2 (in Appendix B) depicts the detailed subroutine for computing the partial attention outputs for a sequence of LeanTiles. This LeanTile() subroutine is called when computing each partial output tile in a CTA launched in LeanAttention, as will be discussed later (Algorithm 1).

To efficiently split attention into smaller tiles, it is necessary to identify the smallest tile size capable of achieving the highest compute efficiency. LeanTile size depends on the computational power and memory access costs and, thus, are fixed for a particular hardware architecture. After an extensive empirical sweep through various sizes for a *LeanTile*, we found a tile size granularity of 256 and 128 tokens along the N_k dimension to be the most optimal for a head size of 64 and 128 respectively for FP16→32 problems while experimenting on an A100 GPU (Choquette et al., 2021; Jia & Van Sandt, 2021). This optimal size can similarly be identified for other head dimensions and hardware architectures.

4.3 Decomposition and Mapping of LeanTiles

Finally, LeanAttention uses a stream-K (Osama et al., 2023) style decomposition and mapping of these LeanTiles to deliver efficient execution of attention.

Stream-K Decomposition. Stream-K is a parallel decomposition technique for dense matrix-matrix multiplication on GPUs. Stream-K partitioning addresses the inefficiencies in fixed-split by dividing the total workload (MAC operations) equally among all CTAs using a pre-determined optimal tile size. It does this by rolling out the inner mode iterations of all output tiles and appending them to form a linear mapping. With the given grid size, it divides this total work into buckets demarcated appropriately such that each CTA has equal amount of MAC operations to perform. The grid size is fixed for a given tile size as LeanAttention provides equal work to all SMs. For example, for a tile of 256 tokens, two CTAs can be co-executed in a single wave with the available shared memory of A100 GPU. This would result in a total grid size of $108(NumSMs) \times 2 = 216$. Number of tiles to be computed by each CTA can be calculated as follows:

$$
TilesPerCTA = \frac{BatchSize \times NumHeads \times ContextLen}{TileSize \times NumSMs \times MaxCTAsPerSM} \quad (2)
$$

LeanAttention extends Stream-K style of linear mapping of iterations by rolling out LeanTile iterations in a similar fashion, assigning equal number of LeanTiles to every CTA as shown in Figure 1. Each CTA's range of LeanTile iterations is mapped contiguously into a batch size →heads →context length linearization, crossing the head and query boundary as it may. Should a given CTA's starting and/or ending LeanTile not coincide with the head's boundary, it must consolidate its partial output with those of the other CTA(s) also covering that head's output tile. In our implementation of LeanAttention, each attention output tile is consolidated by the CTA that performed that output's first LeanTile (called as a host block). Before it can do so, it must accumulate the un-scaled output tensors from the other CTA(s) (shown in Figure 1) in temporary global storage. The negligible synchronization overhead of the original stream-K implementation also extends to LeanAttention, thus leading to near 100% occupancy of SMs (not tensor core utilization) during the execution of a single CTA. Note that the temporary global storage overhead is minimal in the case of decode-phase where the output tensors are of dimensions $1 \times head\_dim$, where $head\_dim$ is typically in the range of 64 to 256.

Since we distribute the overall attention problem into optimal LeanTiles, we achieve a near 100% quantization efficiency irrespective of problem size (context length). This cohesive implementation of parallel computation and reduction happens in a single kernel launch in LeanAttention, avoiding the reduction kernel launch overheads that FlashDecoding suffers from. A difference in Stream-K decomposition in LeanAttention is in the reduction or "fix-up" phase. While Stream-K for MatMuls has addition as its reductive operator, LeanAttention has softmax re-scaling as its reductive operator.

Naturally, some CTAs will be computing LeanTile iterations of more than one independent output tile. In such cases, Stream-K's equalized partitioning makes LeanAttention more adept for problem sizes which would not occupy

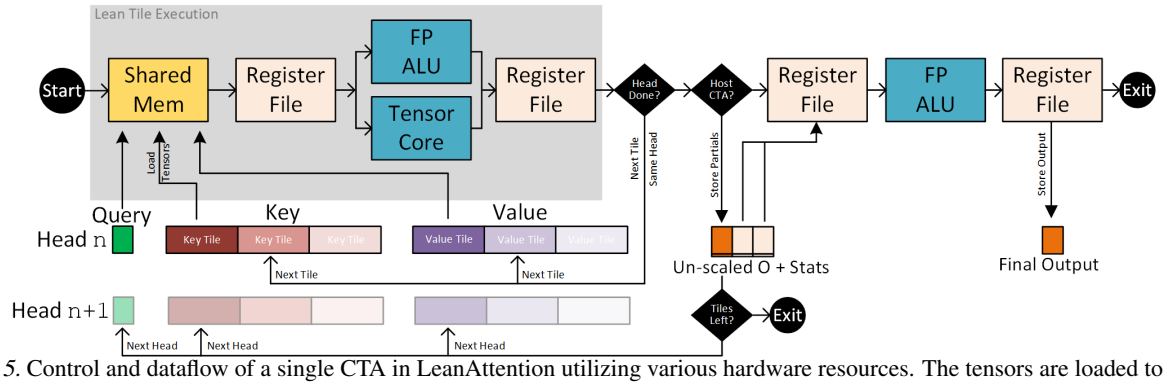

Figure 5. Control and dataflow of a single CTA in LeanAttention utilizing various hardware resources. The tensors are loaded to shared memory in a tiled manner. At the end of a head's computation, a reduction is performed if it is a host CTA, else the partial un-scaled results are written to memory before moving on to the next head.

the hardware well if executed using its counterparts, FlashAttention-2 and FlashDecoding. To enable such a smooth transition between tiles, the input tensor view is also different in LeanAttention compared to FlashAttention-2. This requires a constant stride moving between different heads as we transition from the LeanTile of one head to another requiring Q/K/V tensors be of the shape $(batch\_size, heads, query/ctx\_length, head\_dim)$ compared to FlashAttention-2's requirement of $(batch\_size, query/ctx\_length, heads, head\_dim)$.

With this execution design, we must point out that LeanAttention behaves as a versatile attention partitioning mechanism which generalizes to FlashAttention-2 when the number of output tiles is equal to grid size, and generalizes to FlashDecoding when grid size is an even multiple of number of output tiles. Finally, for all other cases (most common) LeanAttention efficiently distributes the work across the compute resources available in the system. Thus, LeanAttention will either always perform better or the same as FlashAttention-2 and FlashDecoding.

Lean Ragged Batching. For the special case of dealing with unequal context lengths within a batch of requests, the number of LeanTiles per request becomes unique, resulting in a total workload that is smaller than the non-ragged case. To account for this difference, ragged KV tensor inputs to LeanAttention are first prepared with unpadded dimensions of $(NumHeads, TotalContextLength, HeadDim)$, where $TotalContextLength$ is the sum of all distinct context lengths within the batch. The batch dimension is eliminated, and both batch indices and the true context lengths of requests are tracked through pointers to a cumulative sequence length array for each input tensor. These pointers have size $(BatchSize + 1)$ each, introducing minimal memory overhead.

With the total workload of LeanTiles correctly determined, Lean ragged batching functions identically to the non-ragged case as shown in Figure 6. The workload is distributed evenly across the grid, ensuring each CTA receives the same number of LeanTiles to process. The range of LeanTile iterations assigned to each CTA is mapped contiguously in a Heads → TotalContextLength linearization and partial outputs for each head are consolidated in the same manner as in the non-ragged case.

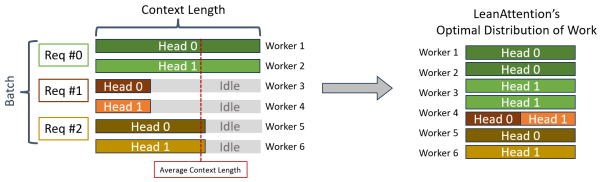

Figure 6. Illustrative diagram of LeanAttention's optimal distribution of work in the ragged batching case. Each SM receives equal amount of LeanTiles.

4.4 Execution Flow

Algorithm 1 details LeanAttention's execution. For a fixed grid size G, CTAs are launched and given equal amount of LeanTile iterations to work with (Line 7). Each CTA computes the LeanTile() subroutine for every distinct output tile that comes under its boundaries (Line 16). Figure 5 shows the execution flow of a single CTA computing partial and final attention outputs for the assigned heads.

The unique reduction phase of LeanAttention, which is characterized by its softmax re-scaling operator, is performed by the host CTA block (Lines 24-40). A host CTA (Line 17) is the CTA responsible for computing the first ever LeanTile for a given output tile, and it behaves as the reducing CTA during parallel reduction of partial un-scaled outputs.

All non-host CTAs will share their partials through a store to global memory and signal their arrival (Lines 20-23). On the other hand, a host block which is also a non-finishing block (Lines 24-25), needs to wait for other contributing peer CTA blocks to signal their completion (Line 28) and then proceed to carry out the reduction (Lines 29-35).

A CTA that's computing the attention for a head exclusively

completes all the LeanTile iterations for its output tile in a single CTA and can directly store its results from the register file to global memory (Line 38-39) without any need for further reduction.

Algorithm 1 Lean Attention

1: **_shared_** $O[T_m, d]$
2: **_shared_** $m[T_m, 1]$
3: **_shared_** $l[T_m, 1]$
4: Number of output tiles: $C_m = \lceil N_q/T_m \rceil$
5: Number of iterations for each output tile: $C_n = \lceil N_k/T_n \rceil$
6: Total number of iterations: $I = C_m C_n$
7: Number of iterations per CTA: $I_G = I/G$
8: **fork** CTA$_g$ in G **do**
9: cta_start = g I$_G$ and cta_end = cta_start + I$_G$
10: **for** iter = cta_start **to** cta_end **do**
11: Index of current output tile: tile_idx = iter / C_n
12: tile_iter = tile_idx $\times C_n$
13: tile_iter_end = tile_iter + C_n
14: local_iter = iter - tile_iter
15: local_iter_end = min(tile_iter_end, cta_end) - tile_iter
16: O, m, l = LeanTile(tile_idx, local_iter, local_iter_end)
17: host-block if: iter == tile_iter
18: finishing-block if: cta_end >= tile_iter_end
19: **if** !(host-block) **then**
20: StorePartials(Op[g], O)
21: StorePartials(mp[g], m)
22: StorePartials(lp[g], l)
23: Signal(flags[g])
24: **else**
25: **if** !(finishing-block) **then**
26: last_cta = tile_iter_end / C_n
27: **for** cta = (g + 1) **to** last_cta **do**
28: Wait(flags[cta])
29: m_{cta} = LoadPartials(mp[cta])
30: l_{cta} = LoadPartials(lp[cta])
31: O_{cta} = LoadPartials(Op[cta])
32: $m^{new} = max(m_{cta}, m)$
33: $l^{new} = e^{m_{cta} - m^{new}} l_{cta} + e^{m - m^{new}} l$
34: $O^{new} = e^{m_{cta} - m_i^{new}} O_{cta} + e^{m - m_i^{new}} O$
35: Update $m = m_i^{new}, l = l_i^{new}$
36: **end for**
37: **end if**
38: Write $O = diag(l)^{-1} O$ to GMEM.
39: Write $L = m + log(l)$ to GMEM.
40: **end if**
41: iter = tile_iter_end
42: **end for**
43: **join**

5 Evaluation Methodology

Implementation. We implement LeanAttention using the CUTE abstractions (CuTe, a;c;b) provided by Nvidia's CUTLASS library (Thakkar et al., 2023). For comparative measurements we utilize FlashAttention-2's implementation of FlashDecoding as it is available on their Github repository (git)[2] and FlashInfer's implementation from their Github repository (Ye et al., 2024)[3]. For the end-to-end inference results we use Llama, Mistral and Phi-3 models as available in the HuggingFace Transfomers repository (Wolf et al., 2019) and modify them to allow execution via LeanAttention wherever necessary. Note that optimizations such as FlashAttention-3 (Shah et al., 2024) are orthogonal to this work and targeted specifically for H100s. The core computation of LeanAttention can adopt FlashAttention-3's optimizations for further benefits on Hopper GPUs (Luo et al., 2024).

System. We benchmark the attention mechanisms on a Nvidia-A100-80GB-GPU (Choquette et al., 2021) system with up to 8 GPUs. We measure runtime using a single GPU as well as 8xGPUs for larger models and context lengths. A single A100 GPU consists of 108 SMs with an 80GB HBM global memory. To demonstrate LeanAttention's versatility across hardware architectures we benchmark it similarly on a single Nvidia-H100-SXM-80GB-GPU (Luo et al., 2024) which has 132 streaming multiprocessors and an 80GB HBM global memory.

Batching. The evaluations assume the same context length for all queries in a batch working in tandem with batching techniques such as Orca (Yu et al., 2022a). However, LeanAttention supports varied context length execution including heterogeneous batching such as prefill queries with decode.

Attention Mechanisms. In addition to FlashDecoding (FD) (Dao et al.), we also benchmark FlashInfer (FI) (Ye et al., 2024) for comparison against LeanAttention (LA).

6 Evaluation Results

In this section, we evaluate the impact of LeanAttention at the attention operation-level as well as end-to-end inference performance-level.

6.1 Benchmarking Attention - Decode-Phase

We benchmark the runtime of just the attention operation at varying context lengths, number of attention heads, head dimensions (64: default and 128), and inference batch sizes

[2]Version 2.5.6

[3]Version 0.1.6 - Note that we increase the number of heads in the single batch decoding API to simulate batch size as the batch API performance is too low

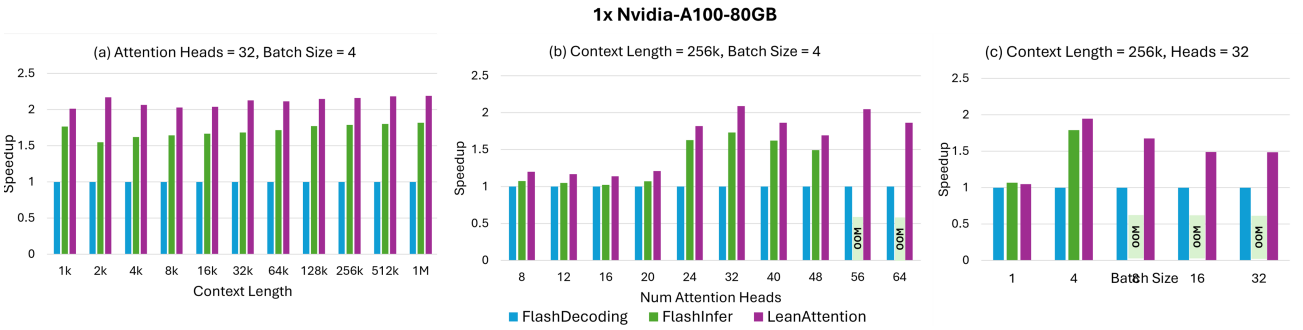

Figure 7. Speedup of LA over state-of-the-art attention execution mechanisms at different context lengths, batch sizes and attention heads with head dimension = 64 on a single Nvidia-A100-80GB GPU.

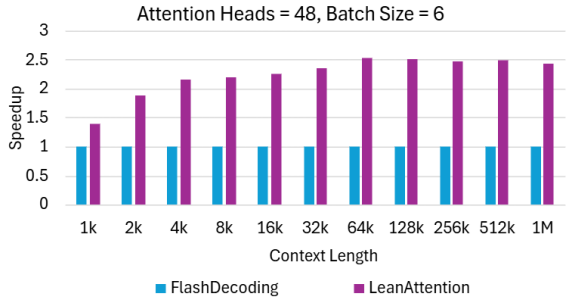

Figure 8. Speedup of LA compared to state-of-the-art Attention execution mechanisms at varying context lengths, at a fixed batch size and attention heads with head dimension = 64 on a single Nvidia-H100-SXM-80GB GPU.

on a single Nvidia A100-80GB GPU and a single Nvidia H100-SXM-80GB GPU.

Increasing Context Length. Figure 7(a) shows the speedup of different attention mechanisms for a model with 32 heads with batch size = 4 on a single A100 GPU. LA delivers close to 2x speedup compared to FD even at smaller context lengths, reaching up to 2.18x speedup as the context lengths grows to 256k tokens. When context lengths exceed 16k, we observe more than 1.46x speedup over FI. Note that while FI implements the fixed split mechanism like FD, it shows better performance over it due to efficient data movement and optimized tile size selection. Repeating a similar exercise on an H100 GPU, we observe the speedups of LA at batch size = 6 and 48 heads as shown in Figure 8. LA delivers over 2x speedup even at a 4k context length, reaching upto a maximum of 2.52x speedup at a 64k context length which more or less plateaus at the context lengths increase.

Increasing Attention Heads. Figure 7(b) shows the speedup delivered by LA over FD and FI for models with an increasing number of heads. LA delivers comparable speedups to FD and FI at smaller model sizes. With fewer heads, FD and FI's fixed-split mechanism can distribute the workload as evenly as LA. However, as the number of heads increases, FD/FI resorts to fewer splits per head, resulting in partially filled waves of attention on the SMs. In contrast, LA maintains even workload distribution at both small and large model sizes, delivering more than 2x speedup over FD when there's more than 24 heads in the model for a long 256k context at batch size = 4. This shows that LA is able to scale well for both small and large model sizes.

Effect of Batching. Figure 7(c) shows the performance improvement of LA at varying batch sizes. As expected, we observe that LA gives comparable speedup to FD and FI. This is because both FD and FI are able to employ a higher number of splits at smaller batch sizes to occupy all the SMs in the GPU. However, as batch size increases, FD selects fewer splits per head. For instance, FD doesn't split at batch sizes over 4 in Figure 7(c) because the total number of heads in the batch exceed the number of SMs available in the system. As a result, it behaves like vanilla FlashAttention-2, missing out on potential performance gains by leaving some SMs idle in its final partially full wave. Consequently, LA achieves more than 1.5x speedup compared to FD through its stream-K-ed decomposition. A comprehensive analysis on ragged batching is also detailed in Appendix A.

Overall, we benchmarked the system on more than 1,000 samples of varying batch sizes, context lengths and attention heads. On an A100 GPU, we observed an average speedup of 1.73x over FD (Max: 2.18x for 56 heads, batch size 2, 256k context). On an H100 GPU, we recorded an average speedup of 1.52x over FD (Max: 2.53x for 48 heads, batch size 6, 64k context).

Multi-GPU Execution. Repeating a similar benchmarking process on an 8xA100 GPU system, we vary the context lengths from 1k to 1M, with 256 heads at a batch size of 4 as shown in Figure 9(a). LeanAttention reaches a speedup of more than 2x even at smaller contexts. This is because parallelizing only over the batch and heads (total heads $= 256 \times 4 = 1024$) does not provide sufficient work for each SM (total SMs $= 8 \times 108 = 864$) as $864 - (1024 - 864) = 704$ SMs remain idle in the last wave. Furthermore, FD behaves identically to vanilla FlashAttention-2, opting for a split factor of 1. In contrast, LeanAttention computes attention in fully quantized waves for all problem sizes.

To observe this effect in greater detail, we evaluate across a varying number of heads in Figure 9(b) with a context

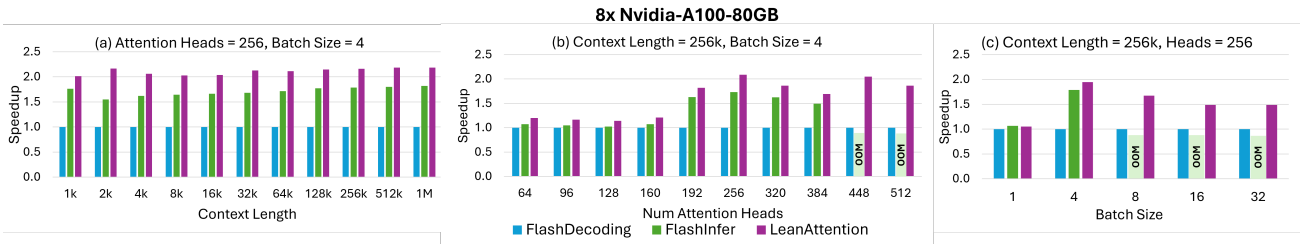

Figure 9. Speedup of LA over state-of-the-art attention execution mechanisms at different context lengths, batch sizes, attention heads with head dimension = 64 on an 8x Nvidia-A100-80GB system.

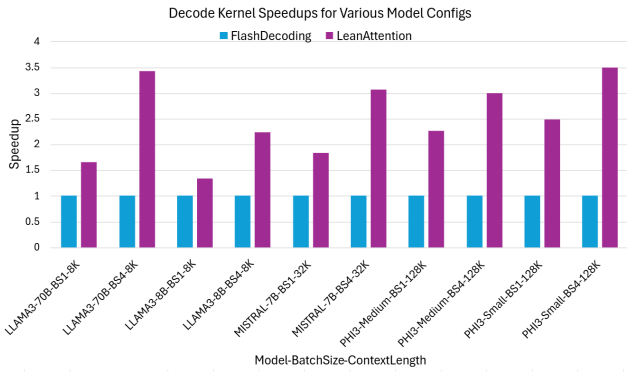

Figure 10. Speedup offered by LA for decode attention across models, batch sizes, context lengths integrated with ONNXRT (Note that ONNXRT has an older version of FlashDecode)

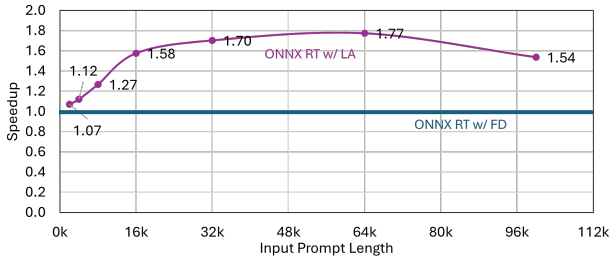

Figure 11. End-to-End Speedup of LA compared to FD in ONNXRT running Phi-3 Medium model at different context lengths, batch size = 1, prompt size : output tokens = 8 : 1

length of 256k and batch size = 4. We observe comparable speedups to FD/FI at a smaller number of heads (64, 160). This is because at these dimensions, fixed-split is able to produce enough splits to occupy most of the SMs. We can also clearly see that FD resorts to vanilla execution when we increase the number of heads from 160 to 512. LA, on the other hand scales well as we increase the number of heads, showcasing its hardware-aware scalable execution algorithm. In Figure 9(c), as we increase the batch size from 1 to 32, we can see that LA starts to outperform FD/FI as batch size increases.

Effect of Head Dimension. Figure 10 shows the speedup offered by LA for models with a LLAMA-3, Mistral and Phi-3-like config with a head dimension of 128. We utilize a 128-token wide LeanTile for decomposition instead of 256. We observe a similar trend in performance, where LA delivers a speedup of 3.5x compared to FD at 128k context length. Even at smaller context lengths of 8k tokens, we observed an improved performance of 1.34x over FD.

6.2 End-to-End Inference Performance

We measure the end-to-end inference runtime using Phi-3 Medium model (with 40 heads) as shown in Figure 11 at different prompt sizes with a prompt to output token ratio of 8:1. This includes the prefill stage latency as well as the total runtime of decode phase. LeanAttention offers a 1.12x speedup with Phi-3 Medium as compared to FlashDecoding for first 1k output tokens. However, LA offers higher speedups as the output tokens increase beyond 16k delivering an average of 1.73x speedup compared to FD. As we note, the inference-level runtime improvement delivered by LA will depend heavily on the number of heads, total context length, batch size etc.

7 Conclusion

The attention mechanism in transformer-based language models is a slow and memory hungry process. State-of-the-art optimization kernels (Dao et al.; Ye et al., 2024; Dao, 2023) have cleverly addressed this challenge but fail to adapt to the computationally distinct phases of inference at long context lengths. To address this, we propose *LeanAttention*, a generalized exact attention execution mechanism that ensures lower runtimes and almost 100% hardware occupancy during attention irrespective of problem size. By leveraging the associative property of softmax re-scaling, we employ a "Stream-K"-style partitioning of attention that provides the hardware with well-balanced loads.

Our measurements indicate that LeanAttention delivers an average speedup of 1.73x over FlashDecoding, with up to 2.18x speedup for a 256k context size.

Acknowledgements

We thank Daniel Madrigal Diaz at Microsoft for their help with LeanAttention's end-to-end inference benchmarking setup during internal demos. We also thank the ONNX team for their help with integrating LeanAttention into ONNXRuntime's main repository.

REFERENCES

GitHub - Dao-AILab/flash-attention: Fast and memory-efficient exact attention — github.com. https://github.com/Dao-AILab/flash-attention. [Accessed 19-04-2024].

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024. URL https://arxiv.org/abs/2403.02310.

Anthropic. Introducing the next generation of claude. https://www.anthropic.com/news/claude-3-family. [Accessed 19-04-2024].

Brandon, W., Nrusimha, A., Qian, K., Ankner, Z., Jin, T., Song, Z., and Ragan-Kelley, J. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431*, 2023.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Choquette, J., Lee, E., Krashinsky, R., Balan, V., and Khailany, B. 3.2 the a100 datacenter gpu and ampere architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pp. 48–50. IEEE, 2021.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

CuTe. Cute layouts. https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/01_layout.md, a. [Accessed 19-04-2024].

CuTe. Cute's support for matrix multiply-accumulate instructions. https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/0t_mma_atom.md, b. [Accessed 19-04-2024].

CuTe. Cute tensors. https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/03_tensor.md., c. [Accessed 19-04-2024].

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dao, T., Haziza, D., Massa, F., and Sizov, G. Flashdecoding: Stanford CRFM — crfm.stanford.edu. https://crfm.stanford.edu/2023/10/12/flashdecoding.html. [Accessed 22-04-2024].

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Fu, Y., Panda, R., Niu, X., Yue, X., Hajishirzi, H., Kim, Y., and Peng, H. Data engineering for scaling language models to 128k context. *arXiv preprint arXiv:2402.10171*, 2024.

Hong, K., Dai, G., Xu, J., Mao, Q., Li, X., Liu, J., Dong, Y., Wang, Y., et al. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems*, 6:148–161, 2024.

Inc., G. An important next step on our AI journey — blog.google. https://blog.google/technology/ai/bard-google-ai-search-updates/. [Accessed 31-03-2024].

Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

Jia, Z. and Van Sandt, P. Dissecting the ampere gpu architecture via microbenchmarking. In *GPU Technology Conference*, 2021.

Kenton, J. D. M.-W. C. and Toutanova, L. K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, pp. 2, 2019.

Kim, S., Hooper, C., Wattanawong, T., Kang, M., Yan, R., Genc, H., Dinh, G., Huang, Q., Keutzer, K., Mahoney, M. W., et al. Full stack optimization of transformer inference: a survey. *arXiv preprint arXiv:2302.14017*, 2023.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Li, T., Zhang, G., Do, Q. D., Yue, X., and Chen, W. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060*, 2024.

Li, Y., Bubeck, S., Eldan, R., Del Giorno, A., Gunasekar, S., and Lee, Y. T. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*, 2023.

Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

Liu, S., Tao, G., Zou, Y., Chow, D., Fan, Z., Lei, K., Pan, B., Sylvester, D., Kielian, G., and Saligane, M. (eds.). *ConSmax: Hardware-Friendly Alternative Softmax with Learnable Parameters*, 2024. URL https://arxiv.org/abs/2402.10930.

Luo, W., Fan, R., Li, Z., Du, D., Wang, Q., and Chu, X. Benchmarking and dissecting the nvidia hopper gpu architecture, 2024. URL https://arxiv.org/abs/2402.13499.

Milakov, M. and Gimelshein, N. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

OpenAI. Introducing ChatGPT — openai.com. https://openai.com/blog/chatgpt. [Accessed 01-04-2024].

Osama, M., Merrill, D., Cecka, C., Garland, M., and Owens, J. D. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 429–431, 2023.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Spataro, J. and Inc., M. Introducing Microsoft 365 Copilot – your copilot for work - The Official Microsoft Blog — blogs.microsoft.com. https://blogs.microsoft.com/blog/2023/03/16/introducing-microsoft-365-copilot/-your-copilot-for-work/. [Accessed 31-03-2024].

Stevens, J. R., Venkatesan, R., Dai, S., Khailany, B., and Raghunathan, A. Softermax: Hardware/software co-design of an efficient softmax for transformers. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 469–474. IEEE, 2021.

Thakkar, V., Ramani, P., Cecka, C., Shivam, A., Lu, H., Yan, E., Kosaian, J., Hoemmen, M., Wu, H., Kerr, A., Nicely, M., Merrill, D., Blasig, D., Qiao, F., Majcher, P., Springer, P., Hohnerbach, M., Wang, J., and Gupta, M. CUTLASS, January 2023. URL https://github.com/NVIDIA/cutlass.

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

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

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Ye, Z., Chen, L., Lai, R., Zhao, Y., Zheng, S., Shao, J., Hou, B., Jin, H., Zuo, Y., Yin, L., Chen, T., and Ceze, L. Accelerating self-attentions for llm serving with flashinfer, February 2024. URL https://flashinfer.ai/2024/02/02/introduce-flashinfer.html.

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, Carlsbad, CA, July 2022a. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

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, 2022b.

Zhang, P., Liu, Z., Xiao, S., Shao, N., Ye, Q., and Dou, Z. Soaring from 4k to 400k: Extending llm's context with activation beacon. *arXiv preprint arXiv:2401.03462*, 2024.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V.,

et al. Opt: Open pre-trained transformer language models.
arXiv preprint arXiv:2205.01068, 2022.

A RAGGED BATCHING IN DECODE.

Ragged Batching in Decode. For the purpose of our evaluations, we quantify the heterogeneity of a ragged batch as the ratio of average context length to the maximum context length present in the batch. Figure 12 shows the speedup of LA over FD. We observe that as the heterogeneity of batch increases, LA delivers a higher speedup because of better distribution of work across SMs.

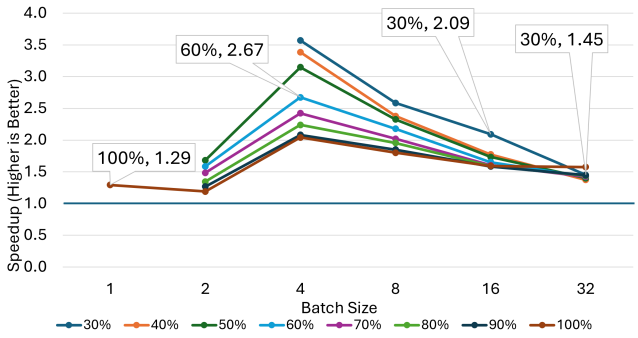

Figure 12. Speedup offered by LA over FD at different batch sizes with heterogeneous context lengths. Batch context ratio(%) shows the ratio of average context length over maximum context length

B LEANTILE ALGORITHM

Algorithm 2 LeanTile() for a sequence of lean tile iterations.

function LeanTile(tile_idx, iter_begin, iter_end)
shared $O_{acc}[T_m, d]$
shared $Q_f[T_m, d]$
shared $K_f[T_n, d]$
shared $V_f[T_n, d]$
shared $m[T_m, 1]$
shared $l[T_m, 1]$
Initialize O_{acc} to $(0)_{T_m \times d} \in R^{T_m \times d}$ in SMEM.
Initialize m to $(-\infty)_{T_m \times 1}$ and l to $(0)_{T_m \times 1} \in R^{T_m \times 1}$ in SMEM.
$mm = T_m \times$ (tile_idx / 1)
$nn = d \times$ (tile_idx % 1)
Perform lean tile iterations for this output tile.
for $iter = iter\_begin$ **to** $iter\_end$ **do**
    $kk = iter \times T_n$
    load fragments from GMEM to SMEM
    $Q_f = LoadFragment(Q, mm, nn)$
    $K_f = LoadFragment(K, nn, kk)$
    $V_f = LoadFragment(V, nn, kk)$
    Compute on chip:
    $S_f = Q_f K_f$ where $S_f \in R^{T_m \times T_n}$
    $m^{new} = max(m, rowmax(S_f))$
    $P_f = exp(S_f - m^{new})$ where $P_f \in R^{T_m \times T_n}$
    $l^{new} = e^{m - m^{new}} l + rowsum(P_f)$
    $O_{acc} = P_f V_f + diag(e^{m - m^{new}}) O_{acc}$
    $l = l^{new}, m = m^{new}$
end for
return O_{acc}, l, m
end function

C Energy Consumption Evaluation

Fixed-split partitioning results in imbalanced workloads across the SMs, leaving many of them idle during the final stages of computation. This inefficiency makes fixed-split attention mechanisms energy-inefficient. As shown in Figure 13, the disparity in energy consumption between FlashDecoding, FlashInfer, and LeanAttention increases as context lengths grow over 128k. LeanAttention, with its well-balanced load partitioning strategy, ensures more consistent utilization of SMs, making it significantly more energy-efficient.

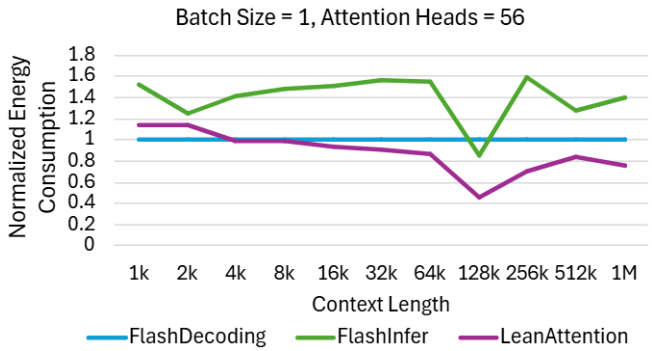

Figure 13. Ratio of Energy consumed by attention kernel to energy consumed by FlashDecoding kernel of different attention mechanisms for batch size = 1, number of heads = 56, head dimension = 64 on a single Nvidia-A100-80GB GPU as measured using NVML APIs.

D REDUCTION OVERHEAD ANALYSIS

We have noticed negligible reduction overheads in our experiments (when reducing 8 or less partial outputs per host CTA) as detailed in Table 2, with the total latency being dominated by the LeanTile computation within attention.

Reduction overheads might be more conspicuous in cases with very few heads (around 3-4 heads) or very short contexts (<1k), where more splits per head would be required to fully occupy the processor. To mitigate this, LeanAttention strategically optimizes grid size (i.e., the number of CTAs launched) to ensure that each CTA processes at least two LeanTiles. As grid size increases, so does the number of splits per head which consequently increases overheads from reduction. Thus, for shorter contexts where total number of LeanTiles are not enough to cover the entire processor width, LeanAttention chooses a smaller, more efficient grid size to maintain performance. This prevents an unnecessary increase in splits, effectively reducing reduction overheads while still leveraging parallelism across context length for speedups.

Since most open-sourced model configurations have greater than 12 heads, we rarely ever exceed 8 splits per head with LeanAttention in our experiments.

Number of Splits	Reduction Overhead %
4	0.41
8	2.85
16	6.84
32	16.47

Table 2. Reduction overhead as a percentage of timeshare for different number of splits per head.