DeFT: Decoding with Flash Tree-Attention for Efficient Tree-structured LLM Inference

Anonymous Author(s)
Affiliation
Address
email

Abstract

Given the increasing demand for tree-structured interactions with LLMs, we introduce DeFT (Decoding with Flash Tree-Attention), an IO-aware tree attention algorithm tailored for tree-structured inference. Unlike traditional sequence-based decoding, tree-structured decoding better accommodates modern task requirements, including self-consistency, few-shot prompting, multi-step reasoning, and multi-model/head coordination. However, existing sequence-based inference systems are ill-suited for tree-structured decoding, resulting in redundancy in computation, memory footprints, and memory access, thereby undermining inference efficiency. To address this challenge, DeFT maintains memory-efficient attention calculation with low memory footprints through two key stages: (1) **QKV Preparation:** We propose a *KV-Guided Grouping Strategy with Tree Split* to intelligently group QKV, optimizing GPU resource utilization while minimizing memory reads/writes for KV cache between GPU global memory and on-chip shared memory; (2) **Attention Calculation:** We compute partial attention of each QKV group in a fused kernel and employ a *Tree-topology-aware Global Reduction* strategy to obtain final attention. By reducing 73-99% KV cache IO and nearly 100% IO for partial results during attention calculation (e.g., Softmax), DeFT achieves up to 2.52/3.82× speedup in the end-to-end/attention latency across three practical tree-based workloads: namely, few-shot prompting, multi-step reasoning, and speculative decoding, over state-of-the-art attention algorithms.

1 Introduction

Large language models (LLMs) [1, 34, 35] are extensively utilized across a range of tasks like chatbot [31], code generation [26], reasoning [42, 4, 28], etc. To meet the increasing demand for service quality of wide-range applications, the interactions with LLMs are more and more complex: moving from simple *sequence-structured patterns* like multi-turn chats, to *tree-structured patterns*, including self-consistency [37], few-shot prompting [25], multi-step reasoning [42, 11, 41], and multi-model/heads coordination [27, 5], etc. Unfortunately, higher service quality is not a free lunch: we sacrifice efficiency—more tokens need to be generated to provide large space for tree search [10, 23, 21] or selection, as shown in Table 1.

The mismatch between the existing sequence-based inference systems [20, 29, 16] and tree-structured interactions exacerbates the efficiency problem. Most current inference systems are designed for **sequence-based decoding**, which samples a single sequence of tokens every time, while **tree-based decoding** maintains multiple sequences with common prefixes as a tree structure, as shown in Figure 1. Since nodes in the forms of the tree can be shared computationally and in memory while that of the sequence cannot, applying tree-structured tasks directly to sequence-based decoding causes three levels of redundancy: (1) *memory storage*, especially the KV cache [20, 45]; (2) *computation*, especially the computation for common prompts among sequences in a batch [45]; (3) *memory access*.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

Existing work of tree-based inference systems [45, 9] focuses on the first two levels while largely ignoring the third yet the most important one–*memory access*, given the nature of memory-bounded LLM inference [32, 5, 19]. As for sequence-based decoding methods optimize the memory access for the aspects of partial results (i.e., $\mathbf{QK}^\top$) during attention calculations [6, 7, 15]. However, their effectiveness in tree-based decoding is limited. In particular, these optimizations are unable to address the potential bottleneck posed by the KV cache IO when dealing with a large number of tokens, as illustrated in Table 1.

Table 1: **Comparison of efficiency in sequence-based (CoT [38]) and tree-based (ToT [42]) decoding for a reasoning task.** The task is *sorting 128 numbers* from [4]. The total generated tokens of CoT is only 525 while 38,315 in ToT, resulting in inefficiency in end-to-end latency (`second`) and IO (`TB`). IO mainly consists of two parts as follows. (i) *KV cache*: `IO-KV`; (ii) *Partial results during attention calculation like QK^T and softmax*: `IO-PA`; Baselines: (i) *Flash-Decoding* [7]; (ii) *Tree Attention*: tree attention in Medusa [5].

	Metrics		
	`Latency`	`IO-KV`	`IO-PA`
Flash-Decoding + CoT	21	0.6	0
Flash-Decoding + ToT	429.65	59.96	0
Tree Attention + ToT	380.87	12.40	3.69
DeFT-Flatten(ours) + ToT	94.61	12.40	0
Speed up over best baseline	4.02×	-	-

As a remedy, in this paper, we resort to the key attention component during the decoding process. Orthogonal to the traditional attention mechanisms in sequence-based decoding, tree attention [27, 5]—specifically designed to handle hierarchical or tree-structured tokens in tasks such as parallel decoding—can reduce the kernel launching, computation and KV cache storage overheads for attention calculations. However, this line of research does not further leverage the tree topology to reduce IO when calculating attention, and thus still not fully IO-aware for both (i) partial result (i.e., $\mathbf{QK}^\top$) [5] due to the lack of tiling and kernel fusion [6]; and (ii) KV cache in a tree structure [27]. These limitations hinder their effectiveness in optimizing memory access during tree-based decoding.

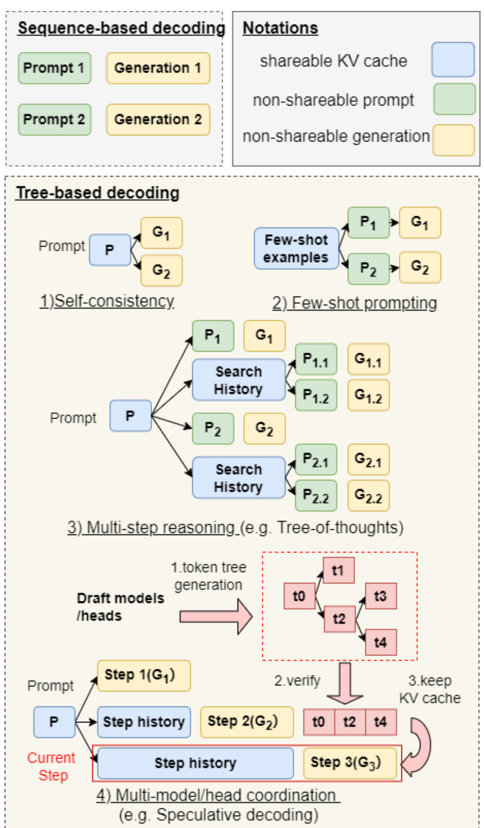


Figure 1: **An illustration of Sequence-based decoding and Tree-based decoding**.

To bridge the above gap, we propose DeFT, an IO-aware tree attention algorithm with two key insights. First, the IO overhead for queries (Q) is negligible compared to that of KV cache, primarily because the maximum query length typically corresponds to numbers of root-to-leaf paths in the tree, resulting in relatively short queries (e.g. dozens of tokens) compared with KV cache length in each node (e.g. hundreds/thousands of tokens). Second, in sequence-based decoding, each KV cache entry corresponds to a unique query, whereas in tree-based decoding, multiple queries can share their common ancestor's KV cache during attention calculation, benefiting not only in reducing KV cache storage but also in IOs.

Building upon these two insights, in the first phase of DeFT–**QKV Preparation**, we split the KV cache of the decoding tree with two choices: (i) split by node (DeFT-Node), which is simple with no need for causal mask; (ii) flatten the tree KV then split evenly (DeFT-Flatten), which ensures more stable speedup due to balanced workloads in GPUs, with little cost of bit causal mask IO. Then we group the KV cache of each node with all queries that share it in the decoding tree, to minimize the IO of KV cache with negligible IO overhead of queries. In the second phase of DeFT–**Attention Calculation**, we adopt a fused kernel to get partial attention with LogSumExp of QKV groups calculated in phase 1, and conduct tree-topology-aware global reduction inspired by Flash-Decoding [7]. We summarize our contributions as follows:

- We propose a simple but hardware-efficient tree attention algorithm–DeFT, which is IO-aware for both KV cache in a tree structure and partial results (i.e., $\mathbf{QK}^\top$ and Softmax). We offer two specific implementations: DeFT-Node is straightforward without a mask, while DeFT-Flatten ensures more stable speedup across various tree topologies, with minimal extra IO cost for masks.

- We implement DeFT on OpenAI Triton [33] to gain precise management over memory access and fuse all attention operations into a single GPU kernel.
- We theoretically justify the superiority of DeFT over the existing attention algorithms [40, 7, 5, 27] in terms of IO complexity.
- We empirically verify its effectiveness on few-shot prompting, multi-step reasoning and speculative-decoding tasks. DeFT can achieve a walk-clock time speedup of **1.3**× for few-shot prompting, **2.5**× for speculative decoding, **1.3**× for multi-step reasoning, due to an up to **3.82**× faster attention calculation, with the baseline implementations [7, 5, 45].

2 Related Work

Tree-based Decoding. Tree-based decoding, exemplified by beam search [10], has been pivotal in NLP, handling lexical and logical constraints [2, 30, 13], mitigating gender bias [24], achieving communicative goals [14], and improving alignment [21]. Based on the structure feature of queries and KV cache, we can classify tree-based decoding into two patterns: (i) tree-structured past KV with parallel queries—usually in multi-step reasoning [42, 4, 28], using search trees with parallel hypothesis generation and selection based on scoring functions. Some score candidates per token [8, 24, 23], others per reasoning step [39, 36, 41]. (ii) past KV in sequence with tree-structured queries—usually in speculative decoding [5, 27]. A token tree as queries are generated from different draft models [27] or heads [5], then these tokens will be verified in parallel via tree-based decoding. Details of these two patterns are discussed in Appendix A.2. Efficiency in tree-based decoding remains underexplored despite various search algorithms' application, such as A* [23] and Monte-Carlo Tree Search [21].

Memory-efficient Attention Algorithms. Existing memory-efficient attention algorithms target sequence-based decoding. FlashAttention [6] improves self-attention computation in LLM training via tiling and kernel fusion, reducing IOs. Flash-Decoding [7] extends this, enhancing parallelism by dividing K and V and introducing global reduction to gather partial attention results, enabling efficient decoding for long sequences. Unluckily, applying these memory-efficient algorithms to the tree-based decoding overlooks redundancy in IO of tree-structured KV cache, which is the focus of DeFT.

Tree Attention. Integrated into LLM inference, tree attention reduces computation, storage, and kernel launching overheads [27]. Tree-structured token candidates undergo parallel decoding, with SpecInfer [27] introducing a topology-aware causal masked tree attention algorithm, dynamically updating a causal mask to capture relationships among tokens. Medusa [5] uses a similar mechanism with a static causal mask, while other works [44, 22] adopt analogous approaches to enhance attention calculation efficiency. However, unlike DeFT, these existing works utilizing tree attention do not take memory access into consideration.

Storage Optimization of Tree-based Decoding. LLM frameworks optimized for tree-based decoding [20, 45] focus on memory storage efficiency. vLLM [20] enhances GPU memory utilization, allowing sequences from the same parent to share KV cache storage. SGLang [45] supports dynamic KV cache management during multi-round interactions with LLMs, improving memory efficiency.

Discussion on Concurrent Works. Some concurrent works [43, 18, 3] also recognize the importance of IO during LLM inference. However, these works have at least one of these flaws: i) they [43, 18, 3] cannot be easily extended to situations where the decoding tree has more than two levels—they target single-context batch sampling scenarios, a special case of general tree-based decoding with a system prompt as prefix and unique suffixes in the first depth; ii) they [18, 3] do not consider the efficiency issues caused by the lengths of different nodes in the decoding tree. Details of comparison for DeFT and concurrent works are discussed in Appendix A.3.

3 DeFT

In this section, we start by introducing the background knowledge of LLM inference, upon which we outline the overview of system support for DeFT. We then present DeFT including its algorithm and Attention Kernel design, which not only reduces memory access of tree KV but also adopts a fused kernel to eliminate the memory access of partial results like $\mathbf{QK}^\top$ and Softmax operations. We further theoretically analyze DeFT's IO with existing attention algorithms to justify its advances.

3.1 Preliminary

LLM inference and its bottleneck. LLM inference involves two stages: (1) prefill and (2) decoding. During the prefill stage, a prompt is tokenized to initialize LLM. The output of the prefill stage becomes the input for the decoding stage. The decoding stage is auto-regressive, with each output token from the previous step serving as the input token for the next step. Due to the sequential process of auto-regressive decoding, LLM inference is memory-bound [32, 19, 5], wherein every forward pass requires transferring all model parameters and KV cache from slower but larger High-Bandwidth Memory (HBM) to the faster but much smaller shared memory of the GPU [17] [1].

Motivation for DEFT. To improve efficiency, boosting the arithmetic intensity—the ratio of total floating-point operations (FLOPs) to total memory access—of the decoding process is essential. Parallel decoding frameworks [5, 27] tend to achieve this goal by introducing more calculations to generate more tokens in each decoding step, while keeping memory access nearly the same[2] in each decoding step. A sequence of tokens will be generated as token candidates by draft models [27] or fine-tuned heads [5], which is then refined by the LLM for acceptable continuation. This line of approach reduces the total number of decoding steps as well as the total amount of memory access. In the meanwhile, tree-based decoding, leveraging the *decoding tree* defined below, enables efficient parallel decoding. The tree attention is further introduced to reduce redundant KV storage, calculation, and kernel launching overheads when calculating the attention.

Definition 3.1 (Decoding tree)**.** *A decoding tree $\mathcal{T}$ is a rooted tree where the root node corresponds to the prompt and each non-root node u represents a sequence of generated tokens $\mathcal{S}_u$. For each node u, $\mathcal{B}_u$ is the path from root node to u (without u) and $P_{\mathcal{B}_u}$ is the concatenation of tokens in sequences of nodes in path $\mathcal{B}_u$ by the sequential order. For each token $n \in u$, $s_{u,n} \in \mathcal{S}_u$ represents the sequence from the first token of node u to n (including n). The last token of each leaf node represents the input token for the next decoding iteration.*

Definition 3.2 (Tree-Attention)**.** *For each token $n \in u$, where u is any non-root node in the decoding tree $\mathcal{T}$, its tree attention is defined as the output of original Transformer-based sequence attention ($\mathit{Attention}(\cdot)$) on $P_{\mathit{root}\to n}$, where $P_{\mathit{root}\to n}$ is the concatenation of P_{B_u} and $s_{u,n}$:*

$$\mathit{Tree\text{-}Attention}(n) = \mathit{Attention}(P_{\mathit{root}\to n})\,. \tag{1}$$

The existing solution of tree attention [5, 27] omits the potential IO optimization brought by the tree topology itself, thus motivating the DEFT we will explore in this paper. DEFT optimizes LLM efficiency from another perspective: it leverages the characteristics of prefix sharing in decoding trees to reduce the redundancy of KV cache IO from HBM to on-chip shared memory, then the whole arithmetic intensity will be improved with less memory access and nearly the same FLOPs.

3.2 Overview of System Design for DEFT

We can separate the execution of attention algorithms into two main phases: (1) QKV PREPARATION PHASE: group Query, Key, and Value (QKV) logically and map QKV groups to different streaming multiprocessors (SMs) of GPUs; (2) ATTENTION CALCULATION PHASE: load QKV groups to different SMs' shared memory and apply attention algorithms to each group for final attention results.

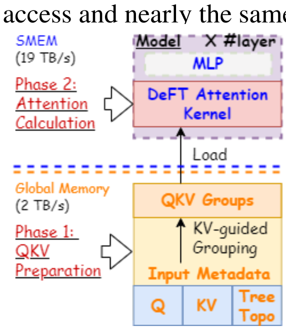


Figure 2: **Overview of DEFT**. SMEM means shared memory of GPUs. *Input Metadata* consists of 1) Query (tokens), 2) KV (KV cache of decoding tree), and 3) Tree Topo (the topology of decoding tree to map Query and KV, which are prepared by *Branch Controller*, *KV cache Manager*, and *Sequence Tree Manager* in the system elaborated in Appendix A.1, respectively.

Minimizing memory access between slow HBM and fast shared memory for memory-bound computations (e.g., attention) is crucial. DEFT aims to be a memory-efficient algorithm in both aforementioned phases to get attention for tree-based decoding. In detail, as shown in Figure 2:

① In the QKV PREPARATION PHASE, we introduce a *KV-guided Grouping* strategy with tree-topology awareness to minimize the IO of QKV.

② During the ATTENTION CALCULATION PHASE, we propose the DEFT ATTENTION KERNEL[3]. This includes (1) a *Tree-Topology-Aware Global Reduction* strategy and (2) established techniques such as *Kernel Fusion* and *Tiling* to eliminate the IO of partial results (i.e., $\mathbf{QK}^\top$ and Softmax).

Apart from efficient DEFT ATTENTION KERNEL, our system for DEFT has other two advantages: 1) efficient memory management of the KV cache in a tree structure, and 2) flexible control of the

[1] A100's HBM has 1.5-2TB/s bandwidth and 40-80GB; its shared memory has 19TB/s bandwidth and 20MB.

[2] Medusa [5] only introduces negligible memory access of KV cache for token candidates in the tree.

[3] GPUs utilize a vast array of threads to execute operations known as *kernels*

tree decoding process with arbitrary user-defined functions, to decide when and how to branch/prune. The details of key components and their coordinations in the system refer to Appendix A.1.

3.3 An Efficient Attention Algorithm with IO-awareness for Tree-structured KV Cache

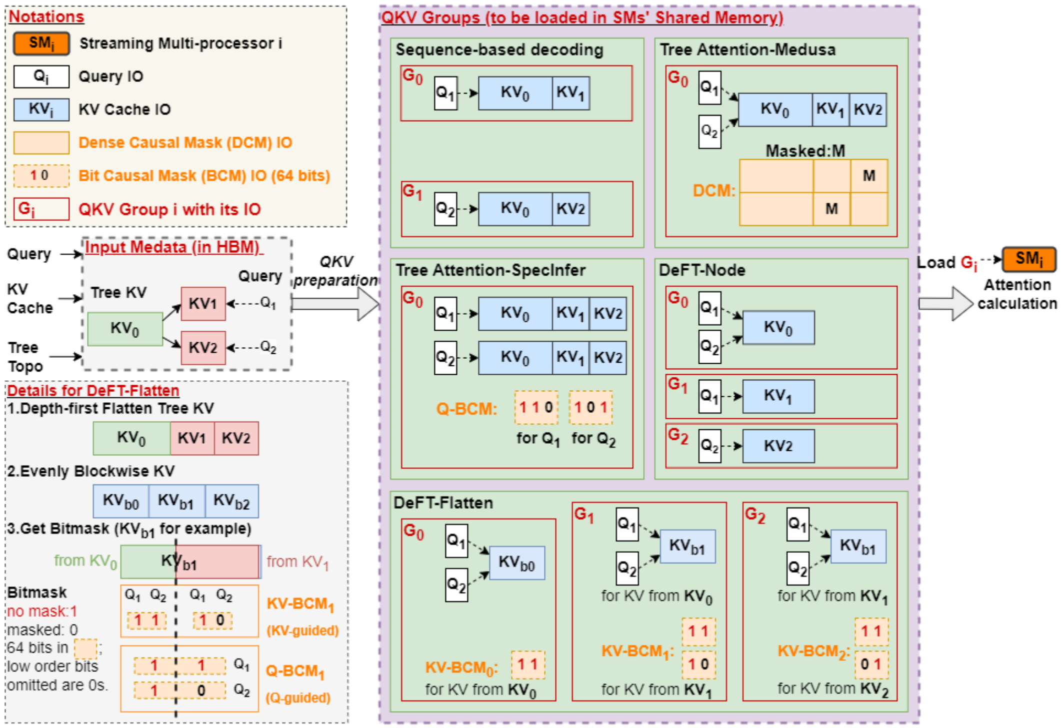


Figure 3: **Comparison of memory access from HBM to shared memory for different attention algorithms in QKV Preparation Phase**, where the amount of IO required by each is enclosed in red rectangles for each QKV group. **(Left)** *From top to bottom, there are notations, the composition of the input metadata, and, most importantly, details of the* DEFT*-Flatten algorithm:* 1) The *Depth-first Flatten* strategy aims to minimize the IOs of queries in each block obtained after splitting, as queries corresponding to child KV are a subset of those in the parent KV (e.g., Q_1 and Q_2 for KV_0 contain Q_1 for KV_1); 2) The *Evenly blockwise* strategy ensures equal lengths of KV in each QKV group for balanced workloads of streaming multiprocessors (SMs) in GPUs; 3) The Bitmask[27] is a set of 64-bit integers used to record causal information of tokens in the tree, but its IO overhead (e.g. two 64-bit integers in KV-BCM_1) is negligible compared to the dense causal mask[5]; 4) To accommodate DEFT-Flatten's KV-guided Tree Split method, we adopt the KV-guided bit causal mask (KV-BCM) instead of the Q-guided one (Q-BCM)[27]. **(Right)** *Different split and grouping strategies result in different memory access.* Q-guided grouping (e.g. sequence-based attention [7, 45] and Tree Attention-SpecInfer [27]) causes significant redundancy of KV cache; KV-guided grouping (e.g. DEFT) causes negligible additional IO of queries. The IO cost of BCM can be ignored, while DCM cannot. See more details in Table 2 and Remark 3.3.

In this section, we delve into the details of the QKV PREPARATION PHASE, which is a key design aspect of DEFT, and defer the discussion of the ATTENTION CALCULATION PHASE to Appendix A.4.

① **QKV PREPARATION PHASE of DEFT.** In sequence-based decoding, *split* strategy—namely splitting the inputs KV into blocks—is commonly deployed to generate enough QKV groups for full utilization of the GPU [7]. This technique is crucial when the parallelism (usually limited by the batch size [7]) is much smaller than the number of streaming multiprocessors (SMs) on the GPU (108 for an A100), where the operation will only utilize a small portion of the GPU. Similarly, for tree-based decoding—where a decoding tree consists of multiple nodes and each node is a sequence of tokens—the batch size of trees may also be insufficient to fully utilize the GPU when the number of tokens in the tree is large, due to memory capacity limitations.

Unfortunately, *split* the tree is not as easy as *split* the sequence [7]: it may introduce significant IOs during the QKV grouping after splits, as shown in Figure 3 and discussed in Remark 3.3.

Remark 3.3 (The effects of tree split and QKV grouping strategies in the QKV PREPARATION PHASE)**.** *In the* QKV PREPARATION PHASE*, how decoding tree is split and QKVs are grouped logically results in different memory access of QKV from HBM to shared memory for tree decoding, as shown in the right of Figure 3 and Table 2.*

Table 2: Comparison of grouping and split strategies of baselines and DeFT. For IO redundancy, these significant is in red, while these can be ignored is in blue. Detailed of IO complexity in Table 4.

Method	Sequence-based [7, 45]	Tree Attention-S [27]	Tree Attention-M [5]	DeFT-Node	DeFT-Flatten
Grouping indicator	Q-guided	Q-guided	tree-guided	KV-guided	KV-guided
Tree KV Split Granularity	by branch(query)	no split	no split	by tree node	by block
IO redundancy	KV	KV and BCM	DCM	Q	Q and BCM

- *Sequence-based decoding methods [7, 45] split the tree based on Q and group QKV based on Q without tree topology awareness, which bring redundant KV cache IO;*
- *Tree Attention-Medusa [5] groups the QKV of the entire decoding tree together with a tree topology-aware causal mask for tree attention computation based on Pytorch primitives, resulting cost of additional IO for the causal mask;*
- *Tree Attention-SpecInfer [27] groups each query with the KV of the entire tree with a causal mask for tree attention calculation, which has great redundancy in KV cache IO.*

To bridge this gap, we propose *KV-Guided Grouping Strategy with Tree Split*, offering two levels of granularity: it splits the tree by sequence nodes or blocks of the same length, and then groups the KV of each node with all queries that share it based on tree topology. This grouping strategy, with KV as the indicator for grouping, eliminates redundant IO operations for KV with negligible query IO cost, as illustrated in the bottom right of Figure 3.

Remark 3.4 (Properties of *KV-Guided Grouping Strategy with Tree Split*)**.** *The additional IO cost of Q caused by split tree KV in* DeFT *is negligible because the length of the KV often surpasses that of the Q during tree decoding, primarily due the fact that the auto-regressive decoding pattern dictates that each query in the decoding stage has a length of 1, which means the maximum query length of a decoding tree is determined by the number of branches.*

Remark 3.5 (The effects of different split granularities)**.** *We provide two algorithm choices for* DeFT *different splits granularity in KV-Guided Tree Split.*

- DeFT-Node: split by node*, which is simple without a need for the causal mask. However, it may have potentially unbalanced workloads in different SMs. For example, node A could have the KV cache of* 1000 *tokens, while node B only has that of* 2 *tokens. When nodes A and B are allocated to SM_1 and SM_2 respectively, SM_2 could finish the task much earlier and be idle.*
- DeFT-Flatten: flatten tree KV then evenly split it to blocks. *The same length of KV cache in each QKV group ensures balanced workloads in IOs and calculations for different SMs, with negligible IO cost of Bit Causal Mask, as shown in the right bottom of Figure 3.*

② **Attention Calculation Phase of DeFT.** In this phase, we design DeFT Attention kernel to load QKV splits in a memory efficient way, which is logically grouped by the QKV Preparation Phase, then to perform the attention calculation. Key techniques are as follows, whose details are discussed in Appendix A.4: 1) common *Kernel Fusion* and *Tiling* strategies avoid significant IO operations for partial results (i.e.. $\mathbf{QK}^\top$ and Softmax), which Tree Attention-Medusa [5] lacks; 2) a novel *Tree-Topology-Aware Global Reduction* inspired by Flash-Decoding [15] retrieves the final attention of each query based on partial attention results from each QKV group with tree topology.

Implementation details. We implement the DeFT attention kernel by OpenAI Triton [33], which enables us to control memory access from global memory to shared memory and attention calculations in a thread block granularity. DeFT-Node and DeFT-Flatten algorithms with two phases in a Python style can be found in Appendix A.7 and Appendix A.8, respectively.

3.4 Analysis: IO Complexity of DeFT

This section analyzes the IO complexity of DeFT, showing a significant reduction in HBM accesses compared to existing attention algorithms. Note that it is non-trivial to summarize the IO cost of the entire tree decoding process, thus we only compare IOs based on the decoding tree snapshot in a single iteration.

Consider a decoding tree with the features outlined in Table 3, and we summarize the corresponding IO breakdown in Table 4. It can be observed that *due to the lack of tree-topology awareness, sequence-based decoding methods, such as naive attention and Flash-Decoding, incur F_s times more memory access overheads for KV cache compared to* DeFT*-Node/Flatten and Tree Attention-Medusa [5].*

Table 4: **IO complexity breakdown for various methods.** $\mathcal{O}(1)$ denotes the IO cost for a single data in the tensor across all layers and heads, which is equivalent to $\#heads * \#layer * dtype\_size$. The best among all methods in the table is in red, while the (potential) worst is in blue. Query IO is omitted as it is $\mathcal{O}(kl_n d_{head})$ for all methods. Here, k is the number of QKV groups: for DEFT-Node $k = \#node$; for DEFT-Flatten, $k = Ntree/b_s$, where b_s is the block size of KV; for others, $k = 1$. M in Tree Attention-M is short for Medusa [5], while S in Tree Attention-S is short for SpecInfer [27].

Method	KV cache	$\mathbf{QK}^\top$	$\frac{\mathbf{QK}^\top}{s_c}$	Mask(M)	$\mathbf{M} + \frac{\mathbf{QK}^\top}{s_c}$	Softmax
Naive Attention	$\mathcal{O}(2d_{head}\sum_{i=1}^{l_n} N_i)$	$\mathcal{O}(2\sum_{i=1}^{l_n} N_i)$	$\mathcal{O}(2\sum_{i=1}^{l_n} N_i)$	0	0	$\mathcal{O}(2\sum_{i=1}^{l_n} N_i)$
Flash-Decoding	$\mathcal{O}(2d_{head}\sum_{i=1}^{l_n} N_i)$	0	0	0	0	0
Tree Attention-M	$\mathcal{O}(2d_{head}N_{tree})$	$\mathcal{O}(2l_nN_{tree})$	$\mathcal{O}(2l_nN_{tree})$	$\mathcal{O}(l_nN_{tree})$	$\mathcal{O}(2l_nN_{tree})$	$\mathcal{O}(2l_nN_{tree})$
Tree Attention-S	$\mathcal{O}(2d_{head}N_{tree}l_n)$	0	0	$\mathcal{O}(l_nN_{tree}/64)$	0	0
DEFT-Node	$\mathcal{O}(2d_{head}N_{tree})$	0	0	0	0	0
DEFT-Flatten	$\mathcal{O}(2d_{head}N_{tree})$	0	0	$\mathcal{O}(N_{tree})$	0	0

However, Tree Attention-Medusa entails higher IO overheads for partial results like $\mathbf{QK}^\top$ and Softmax due to the lack of tiling and kernel fusion[4]. What's more, a dense mask is introduced to record the causal information of tokens in the tree, with significant IO costs.

When the number of leaf nodes/queries ln is sufficiently large, the IO cost of partial results might become comparable to that of the KV cache. For instance, in the Llama models [34, 35], where $d_{head} = 128$, with $l_n = 29$, the total IO cost of $\mathbf{QK}^T$, M, $\frac{\mathbf{QK}^\top}{s_c}$, $\mathbf{M} + \frac{\mathbf{QK}^\top}{s_c}$, and Softmax matches that of the KV cache.

Table 3: **Notations**.

l_n	Number of leaf nodes in a decoding tree, which means how many queries are in this decoding iteration.
N_i	Total token length from the root node to leaf node i.
N_{tree}	Total token length the entire tree.
$\#node$	Total number of nodes in entire tree.
d_{head}	Head dimension of LLM.
s_c	Scale factor for scaled dot-product attention, typically denoted as $\sqrt{d_{\text{head}}}$.
F_s	Shared factor of reusing prefixes in tree attention, which means to which extent we can reduce IOs of KV cache: $F_s = (\sum_{i=1}^{ln} N_i)/N_{tree}$.

Remark 3.6 (KV IO in SpecInfer)**.** *Though similar to* DEFT, *SpecInfer [27] also employs a fused kernel for tree attention. No IO is sharing for KV cache among queries in SpecInfer: instead, each query will load the entire KV cache of the tree independently, bringing significant IOs of the KV cache as in Table 4.*

Remark 3.7 (Causal mask IO)**.** DEFT-*Node splits the decoding tree by nodes without the need for causal masks. For more balanced calculations among SMs in GPUs,* DEFT-*Flatten evenly splits the decoding tree into blocks, with minimal IO cost for masks inspired by SpecInfer. This design reduces the IO overhead of masks significantly compared to the dense mask design in Medusa, as shown in Table 4.*

4 Experiments

In this section, to demonstrate the effectiveness of DEFT under different tree topologies, we comprehensively conduct experiments on three types of tree-based decoding tasks, including: (1) few-shot prompting [25]: a typical case study of tree-structured interactions with two levels–a prefix and several suffixes; (2) multi-step reasoning [42, 41, 11]: tasks characterized by tree-structured past KV with parallel queries; (3) speculative decoding [5, 27]: tasks involving past KV in sequence with tree-structured queries.

4.1 Experimental Setup

Baselines. We evaluate the performance of DEFT in NVIDIA A100 (80GB) in Llama3-8B model [35] with the SOTA attention algorithms in sequence-based and tree-based decoding, as shown in Table 5. Note that we did not include the tree attention operator of SpecInfer [27] to our

[4]Note that $\mathbf{QK}^T$, $\frac{\mathbf{QK}^\top}{s_c}$, $\mathbf{M} + \frac{\mathbf{QK}^\top}{s_c}$ and Softmax will load and write, so the IO cost contains a round-trip of memory access between HBM and shared memory, as shown in Figure 9.

Table 5: **Comparison of baselines and DeFT.** Attention kernels of baselines are implemented to fit its memory management. Therefore, for a fair comparison with baselines, we implement DeFT-Node and DeFT-Flatten that fit both paged [20]/unpaged memory management.

Method	Flash-Decoding [15]	Tree Attention-Medusa [5]	Radix Attention [45]	DeFT
Memory	unpaged	unpaged	paged	unpaged/paged
Implementation	Triton	Pytorch	Triton	Triton

Table 6: **Workloads generation**. ToT-BFS is short for tree-of-thoughts [42] with breath-first-search. See more details in Table 10.

Task	**Prompt Dataset**	**Decoding Tree Source**	**Decoding Tree Collection Method**	**Stopping Criteria**
Few-shot prompting	APPS [12]	-	-	400 iterations
Multi-step reasoning	4 tasks in [4]	ToT-BFS in [4]	Reconstruct from interaction records with GPT 3.5 in [4]	End of task
Speculative decoding	APPS [12]	Medusa [5]	Record token tree shape and accepted token length per step	$\sim$ 1000 steps(max length=6000)

Table 7: **Average attention latency (second) of each tree and its influence in end-to-end latency.** b means tree width. t denotes the token tree size (i.e., the number of tree-structured queries). *Attention Speedup over the best attention* means the speedup of DeFT-Flatten over the best baseline (*Tree Attention-Medusa* in most of cases) in attention calculation. *Speedup over the best wall-clock time* means the speedup of DeFT-Flatten over the best baseline (*Radix Attention*) in end-to-end latency. *Attention Speedup over the best wall-clock* means the attention speedup of DeFT-Flatten over the best baseline (*Radix Attention*) in end-to-end latency. ⋆ means out of memory for A100 80GB, while ♠ means not supported/implemented. See details of end-to-end latency in Table 11.

Memory	Method	Few-shot Prompting			Multi-Step Reasoning				Speculative Decoding			
		`b=20`	`b=30`	`b=50`	*Sorting*	*Document*	*Keyword*	*Set*	`t=32`	`t=64`	`t=128`	`t=256`
Unpaged	Flash-Decoding	43.49	66.10	110.09	160.67	105.80	12.14	19.96	340.09	692.88	⋆	⋆
	Tree Attention-Medusa	3.93	7.51	9.57	38.64	29.10	2.62	3.96	18.05	26.31	41.10	68.28
Paged	Radix Attention	5.99	7.30	9.96	39.37	24.69	3.11	5.13	32.60	54.57	109.39	212.29
	DeFT-Node	10.51	11.41	♠	42.96	33.29	6.16	9.58	50.82	♠	♠	♠
	DeFT-Flatten .	3.47	4.07	5.87	28.41	21.45	2.57	3.83	12.68	18.18	29.97	55.58
	Attention Speedup over the best attention.	1.13×	1.63×	1.70×	1.36×	1.15×	1.02×	1.03×	1.42×	1.45×	1.37×	1.22×
	Attention Speedup over the best wall-clock	1.73×	1.63×	1.70×	1.39×	1.15×	1.21×	1.34×	2.57×	3.00×	3.64×	3.82×
	Speedup over the best wall-clock	1.24×	1.28×	1.33×	1.10×	1.03×	1.03×	1.05×	1.43×	1.70×	2.22×	2.52×

baselines as its kernel only supports at most 64 tokens in the token tree (the decoding tree except for the past seq KV part), which is unsuitable for tree-based decoding with tree-structured KV (c.f. details in Appendix A.2).

Workloads generation. To ensure fairness for workloads of different baselines, we reconstruct decoding trees from real multi-step reasoning and speculative decoding tasks, as shown in Table 6. For multi-step reasoning, we include these four tasks from [4]: (1) Sorting 128 numbers (*Sorting* in short), (2) Document merging (*Document* in short), (3) Keyword counting (*Keyword* in short), and (4) Set intersection (*Set* in short). The tree decoding process would be forced to branch and prune the tree in certain iterations to get the same shape of the decoding tree as the original decoding tree sources. See workload generation details and analysis in Appendix A.5.

4.2 Analysis of Memory Management and Bottleneck

As shown in Table 5, the kernel implementations of different attention algorithms adapt to different memory management. To fairly compare their performance of wall-clock time speedup, we need to analyze the influence of memory management and the bottleneck of the system.

A trade-off between memory storage and memory operation. For tree-based decoding, we can store the KV cache by each branch of the decoding tree in a sequence, which is quite straightforward but no storage sharing of the prefix's KV cache. Considering the limited capacity of GPU memory, ignoring the tree structure when sharing KV storage significantly restricts the number of tokens in the decoding tree. Though storing the KV cache according to each node of the decoding tree can greatly improve storage efficiency, many existing attention kernels are designed for sequence-based decoding [6, 15, 7]. To adapt these kernels, the KV caches of different nodes need to be concatenated and materialized into a single sequence tensor, incurring significant data movement costs [20].

The benefits of paged memory for tree-based decoding. To improve the efficiency of KV cache memory management, paged memory [20, 45] is the current mainstream technology. These KV cache tensors are stored in a non-contiguous, paged layout to provide token-level reuse. Besides higher storage efficiency, we note an additional benefit of paged memory management for tree-based decoding: non-contiguous storage in a memory pool is addressed by pointers, ensuring that we do not need to materialize the tree-structured KV into a single tensor before executing the attention kernel. Instead, we only need to record the memory pool addresses of each token's KV cache.

Bottlenecks and trade-offs. We provide support for DEFT and baselines with KV cache in memory management (unpaged or paged) according to their designs. We visualize the latency breakdown for (1) KV cache management, (2) attention, and (3) other operations (including MLP calculation) in Figure 13a. We observe that with unpaged KV cache management in tree-based decoding, the bottleneck (69.5-83.4%) is the data movement required to materialize the KV cache. However, when we use paged memory management, attention becomes the new bottleneck (50.5-60.0%), especially when the token tree is large.

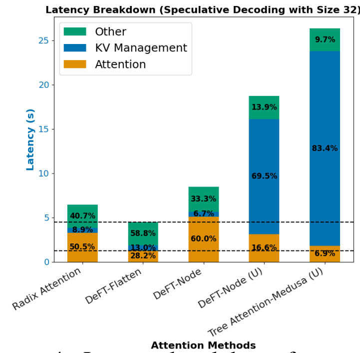


Figure 4: Latency breakdown for speculative decoding with a token tree of 32 queries, whose tree topology is from Medusa [5]. *U* means unpaged memory management.

4.3 End-to-end Behaviors: Latency and IOs.

We evaluate DEFT's performance on various tree-based decoding tasks by measuring end-to-end latency (Table 11 in Appendix A.6), attention latency (Table 7), and IO (Table 12 in Appendix A.6). This assessment demonstrates DEFT's optimization of tree attention and its acceleration of wall-clock time.

For few-shot prompting tasks, we used a prompt with 4k tokens and performed 400 decoding iterations, achieving a 1.33× end-to-end speedup thanks to 1.70× faster attention calculation and an approximately 90% reduction in IO.

For speculative decoding tasks, DEFT-Flatten achieved up to a 2.52× wall-clock time speedup due to up to a 3.82× speedup in attention, as the entire token tree (all queries) can share IO of the long prefix.

For multi-step reasoning tasks, although DEFT-Flatten can have up to 1.36× attention speedup, the end-to-end acceleration is less pronounced for two reasons: (1) the tree width is too small (only 10), making the benefits of reusing KV cache IO less significant; (2) the total number of tokens in the tree is too low, resulting in attention's end-to-end latency accounting for only about 30% of the total time (compared to approximately 50-80% in speculative decoding). Our experiments in few-shot prompting demonstrate that increasing the tree width (from 10 to 50) can result in significant end-to-end acceleration of 100 iterations from 1.2× to 1.5×, as shown in Appendix A.6).

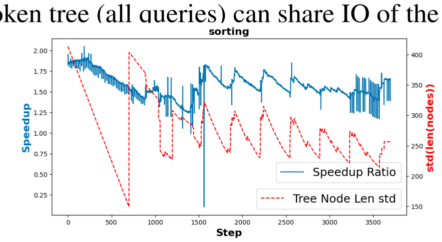


Figure 5: Comparison of split strategies DEFT-Node and DEFT-Flatten in *sorting* task. *Speedup ratio* refers to the ratio between the per iteration latency of DEFT-Node and DEFT-Flatten. *Tree Node Len std* represents the standard deviation of the tree node lengths for each iteration.

4.4 Ablation Study

The influence of split strategy in DEFT. We visualize the per-iteration latency of DEFT-Node and DEFT-Flatten for a tree in the sorting task in Figure 5, as the size and topology of the decoding tree change with each iteration. This comparison highlights the sensitivity of these two split strategies to changes in tree size. We observe a strong positive correlation between the ratio of per-iteration latency of DEFT-Node and DEFT-Flatten (Speedup Ratio) and the dispersion of tree node sizes. This correlation arises because the performance of DEFT-Flatten remains relatively stable, whereas the performance of DEFT-Node is more strongly influenced by the topology of the tree. DEFT-Flatten provides a stable speedup of approximately 1.75× compared to DEFT-Node.

5 Discussion and Limitations

Transitioning to complex tree-structured interactions demands efficient systems. DEFT optimizes memory access in tree-based decoding by wisely splitting and grouping KV cache entries, showing up to 3.82× faster attention calculation. The limitation of DEFT is that the obvious performance gain requires a relatively large token tree (e.g. few-shot prompting with a long prompt) or sufficient queries (e.g., speculative decoding scenario) to share KV cache IOs of prefixes. In future work, we will test DEFT on tasks with larger token trees, such as multi-step reasoning in coding or document analysis, to demonstrate its effectiveness in diverse scenarios.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. Guided open vocabulary image captioning with constrained beam search. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 936–945, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[3] Ben Athiwaratkun, Sujan Kumar Gonugondla, Sanjay Krishna Gouda, Haifeng Qian, Hantian Ding, Qing Sun, Jun Wang, Jiacheng Guo, Liangfu Chen, Parminder Bhatia, et al. Bifurcated attention for single-context large-batch sampling. *arXiv preprint arXiv:2403.08845*, 2024.

[4] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*, 2023.

[5] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

[6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[7] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference, 2023. PyTorch Blog.

[8] Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. Plug and play language models: A simple approach to controlled text generation. In *International Conference on Learning Representations*, 2019.

[9] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *arXiv preprint arXiv:2311.04934*, 2023.

[10] Alex Graves. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711*, 2012.

[11] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.

[12] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[13] Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1535–1546, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[14] Ari Holtzman, Jan Buys, Maxwell Forbes, Antoine Bosselut, David Golub, and Yejin Choi. Learning to write with cooperative discriminators. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1638–1649, 2018.

[15] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Hanyu Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus. *arXiv preprint arXiv:2311.01282*, 2023.

[16] Hugging Face. Text Generation Inference. `https://github.com/huggingface/text-generation-inference`. Accessed: 2024-05.

[17] Zhe Jia and Peter Van Sandt. Dissecting the ampere gpu architecture via microbenchmarking. In *GPU Technology Conference*, 2021.

[18] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024.

[19] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

[21] Jiacheng Liu, Andrew Cohen, Ramakanth Pasunuru, Yejin Choi, Hannaneh Hajishirzi, and Asli Celikyilmaz. Making ppo even better: Value-guided monte-carlo tree search decoding. *arXiv preprint arXiv:2309.15028*, 2023.

[22] Mingdao Liu, Aohan Zeng, Bowen Wang, Peng Zhang, Jie Tang, and Yuxiao Dong. Apar: Llms can do auto-parallel auto-regressive decoding. *arXiv preprint arXiv:2401.06761*, 2024.

[23] Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, et al. Neurologic a* esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799, 2022.

[24] Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Neurologic decoding:(un) supervised neural text generation with predicate logic constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4288–4299, 2021.

[25] Ben Mann, N Ryder, M Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, S Agarwal, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[26] Chen Mark, Tworek Jerry, Jun Heewoo, Yuan Qiming, Pinto Henrique Ponde de Oliveira, Kaplan Jared, Edwards Harrison, Burda Yuri, Joseph Nicholas, Brockman Greg, et al. Carr andrew n. *Leike Jan, Achiam Joshua, Misra Vedant, Morikawa Evan, Radford Alec, Knight Matthew, Brundage Miles, Murati Mira, Mayer Katie, Welinder Peter, McGrew Bob, Amodei Dario, McCandlish Sam, Sutskever Ilya, and Zaremba Wojciech*, 2021.

[27] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

[28] Xuefei Ning, Zinan Lin, Zixuan Zhou, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Large language models can do parallel decoding. *arXiv preprint arXiv:2307.15337*, 2023.

[29] NVIDIA. TensorRT-LLM. `https://github.com/NVIDIA/TensorRT-LLM`. Accessed: 2024-05.

[30] Matt Post and David Vilar. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1314–1324, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[31] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Kurt Shuster, Eric M Smith, et al. Recipes for building an open-domain chatbot. *arXiv preprint arXiv:2004.13637*, 2020.

[32] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[33] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[34] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[35] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[36] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

[37] 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. *arXiv preprint arXiv:2203.11171*, 2022.

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[39] Sean Welleck, Jiacheng Liu, Ximing Lu, Hannaneh Hajishirzi, and Yejin Choi. Naturalprover: Grounded mathematical proof generation with language models. *Advances in Neural Information Processing Systems*, 35:4913–4927, 2022.

[40] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[41] Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, James Xu Zhao, Min-Yen Kan, Junxian He, and Michael Xie. Self-evaluation guided beam search for reasoning. *Advances in Neural Information Processing Systems*, 36, 2024.

[42] 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. *arXiv preprint arXiv:2305.10601*, 2023.

[43] Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. *arXiv preprint arXiv:2402.15220*, 2024.

[44] Yao Zhao, Zhitian Xie, Chenyi Zhuang, and Jinjie Gu. Lookahead: An inference acceleration framework for large language model with lossless generation accuracy. *arXiv preprint arXiv:2312.12728*, 2023.

[45] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

Contents of Appendix

A Appendix **13**
A.1 Components of System Support for DeFT . . . 13
A.2 Discussion of Tree-based Decoding . . . 14
A.3 Discussion of Concurrent Works . . . 15
A.4 Discussion of Techniques in Efficient Attention Algorithm Design . . . 16
A.5 Discussion of Workloads Generation . . . 19
A.6 Additional Results . . . 20
A.7 DeFT-Node Algorithm . . . 21
A.8 DeFT-Flatten Algorithm . . . 23

A Appendix

A.1 Components of System Support for DeFT

The left part of Figure 6 shows the coordinations of different components for efficient and flexible tree-based decoding. The details of functions for system components of DeFT are as below:

1. **Branch Controller**: It makes the tree decoding process forced by a user-defined function (e.g. branch to two children every 3 iterations, as the example shown in the right of Figure 6). Tree-search-based algorithms can be applied here using the decoding tree's topology information.
2. **Sequence Tree Manager**: It maintains the topology of the decoding tree based on the tree operations and tokens from the Branch Controller. The tree operations like pruning and branching will be executed by *Tree Handler* in this component. *Branch Result Storage* will record token generation results of all branches in the decoding tree, and output when the decoding stops.
3. **KV cache Manager**: It will maintain KV cache with a tree structure. A map between sequence IDs in the decoding tree and KV cache index is kept, which will be updated based on KV operations[5] from the Sequence Tree Manager. We provide both paged [20] and unpaged memory management in this part to fit different attention kernels.
4. **Model Interface**: pass input metadata to DeFT Attention kernel and MLP module, then return logits and memory pointers of updated KV cache.

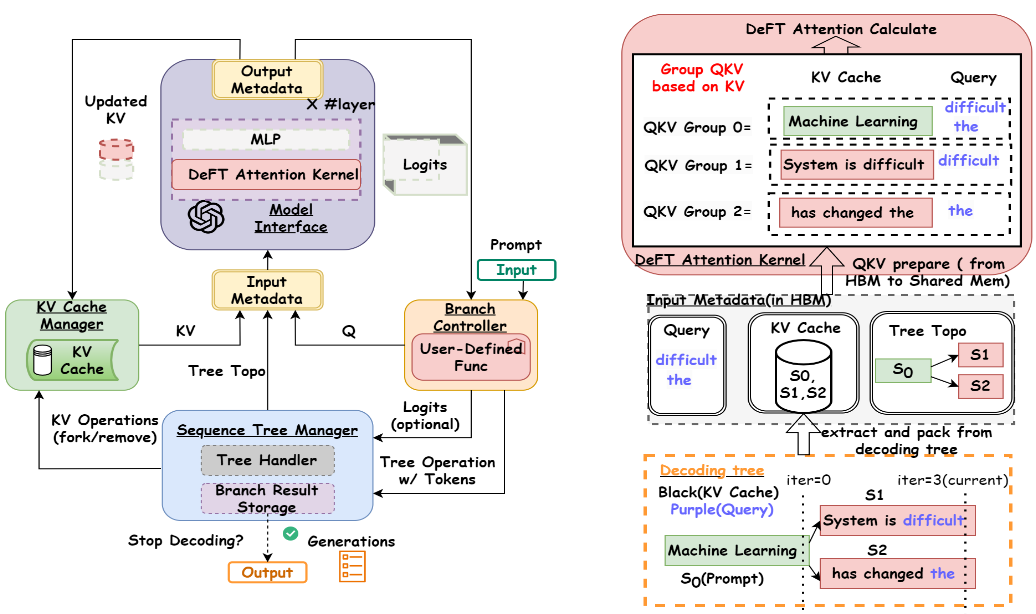


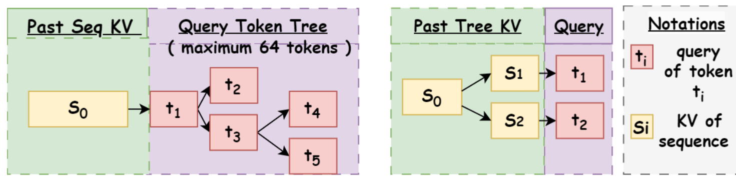
Figure 6: **Illustration of DeFT**. (Left) System overview. (Right) The data flow using a decoding tree example.

The right part of Figure 6 further showcases the key data flow of the system through a decoding tree example: input metadata will be extracted by three components we mentioned above, then loaded from HBM to shared memory in a group manner after the QKV Preparation Phase discussed in

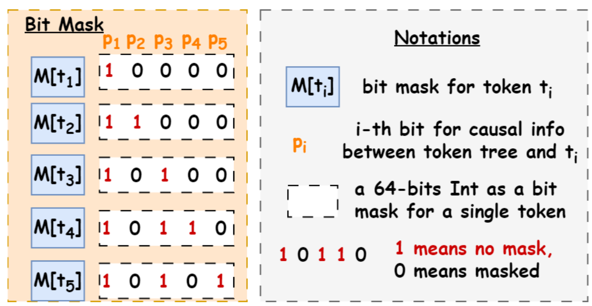
[5] e.g. when a node is pruned in the decoding tree, its KV space will be evicted using a *Remove* operation.

Section 3.3. Then QKV groups will be processed by DeFT Attention Kernel in Attention Calculation Phase of DeFT. See details of techniques in these two phases in Appendix A.4.

A.2 Discussion of Tree-based Decoding



(a) (Left) Sequence KV with queries in a tree for parallel decoding [27, 5], where a *causal mask* is applied to record the causal information among queries in a tree of tokens. (Right) Tree KV with parallel queries for shared prefixes in multi-step reasoning.



(b) *Bit Mask* in SpecInfer [27] to record the causal information between query tokens in a tree structure. The decoding tree is in the left part of 7a.

Figure 7: Discussion of **tree-based decoding** with tree queries [27] and tree KV.

Tree-based decoding could have tree-structured KV cache for storage with awareness of shared prefixes [45], or tree-structured queries in parallel/speculative decoding [27, 5], as shown in Figure 7. A general decoding could both do with tree KV and tree queries, which could reduce redundancy (e.g. IO, storage, computation, etc) of shared prefixes, as well as increase the generated tokens per decoding iteration.

The existing inference frameworks [45, 9] focused on tree-based decoding efficiency primarily aim to: (1) reduce memory footprints [45] to enable larger batch sizes for higher throughput; (2) reuse the prompt cache [9] to avoid recomputation of the KV cache for faster time-to-first-token (TTFT). However, their designs do not specifically target reducing the wall-clock time of the entire decoding process. We observe that the tree-structured feature of LLM inference could provide us some advantages to speed up the decoding itself.

Analysis of speedup potential in tree-based decoding. In tree-based decoding, KV cache and queries can be structured in a tree. Not only can we store KV cache in a tree, but also we can load QKV with awareness of tree topology during attention calculation, to minimize the expensive IO between HBM and on-chip shared memory of GPUs. We explain it in two case studies of complex scenarios with tree-structured interactions: (1) multi-step reasoning [42, 41]; (2) speculative decoding [5, 27].

Case study 1: multi-step reasoning. As shown in the left part of Figure 8, we can summarize process of multi-step reasoning [11, 42, 4] to three phases: (1) *Thought Generation*: generate k candidates for the next thought step based on a generation prompt P_g and previous steps S; (2) *Thought Evaluation*: When presented with a frontier of various thoughts, a LLM as state evaluator measures previous thoughts S based on an evaluation prompt P_e towards resolving the problem. This assessment acts as a heuristic for the search algorithm, guiding it on which states to pursue further and the sequence in which to explore them; (3) *Tree Search-based Expansion*: play different search algorithms [23, 21, 41] to explore search space, which influences the future tree topology. In both (1) and (2), we can share IO of KV cache for P_g/P_e and S during tree attention calculation.

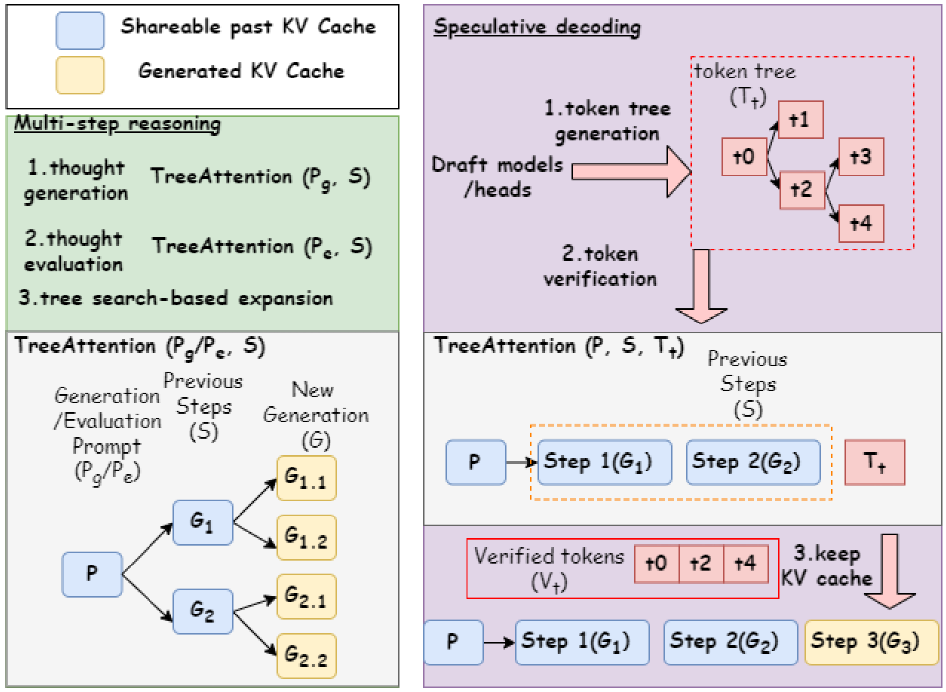


Figure 8: Analysis for two case studies of tree-based decoding. (Left) Multi-step reasoning. (Right) Speculative decoding. Blue boxes mean shareable past KV cache in storage and memory access during the tree attention calculation, while yellow boxes means the KV cache of generated context.

Case study 2: speculative decoding. As shown in the right part of Figure 8, we can summarize process of speculative decoding [5, 27] to tree phases: (1) *Token Tree Generation*: multiple small draft models [27] or fine-tuned heads [5] generate multiple sequences of tokens based on prompt P, then they are merged to a speculated token tree T_t, which is very fast (e.g. 1% of time overhead in SpecInfer [27]); (2) *Token Verification*: based on these tree-structured token candidates T_t, verify the correctness of its tokens against an LLM's output, where tree-attention calculation is the bottleneck of the process [27]. In (2), we can share IO of KV cache for P and S during tree attention calculation.

Why existing tree-attention algorithms are not enough? The existing tree-attention algorithms are either in-efficient in memory access [5, 27] or not suitable for general tree-based decoding [27] with more than 64 tokens in the token tree.

- In SpecInfer[27], as shown in Figure 7b, a *bit mask* is utilized to record the causal information among queries of a token tree. Each token t_i in queries will have a 64-bit Int as a *bit mask*, where j-th bit means the causal relationship between query of t_i and KV cache of t_j. The advantage of this mask design is that it greatly reduces IO, but it results in the maximum number of tree tokens being only 64, which is not practical for scenarios with tree-structured KV cache. What's more, it is not IO-aware for KV cache as it will load KV cache of the entire tree for each query.
- Medusa [5] is suitable for general tree-based decoding, but it is not hardware-efficient due to significant IOs of a dense causal mask and partial results during attention calculation (e.g. Softmax).

A.3 Discussion of Concurrent Works

There are some concurrent works [3, 43, 18] in attention algorithm design for single-context large-batch sampling, where the goal is to generate multiple sequences from a single context(e.g. system prompt or few-shot examples), which is a special case of tree-based decoding with a depth of 1. The design of their algorithms are based on this feature, which means they can not suit well in attention calculation of a tree with more than two levels of prefixes with efficiency.

Insights and techniques in common. Both concurrent works and DeFT have the insight that memory access is the bottleneck of LLM inference, and decomposing attention across subsequences to reduce the memory access of the prefix KV: (1) calculate attention A_p, A_s over prefix and suffixes, respectively; (2) get finial attention by online softmax merging [6, 7] based on A_p and A_s. Here are the details of the correctness proof:

- Let's say we have key tensor $K \in R^{(l_{kv},d)}$, value tensor $V \in R^{(l_{kv},d)}$, and query tensor $Q \in R^{(l_q,d)}$. Consider the general case K and V are partitioned across the sequence (row) dimension into two parts for prefix and suffixes, respectively: $K = K_p \parallel K_s$, and $V = V_p \parallel V_s$, with $\parallel$ denoting concatenation along the row axis.
- We calculate the attention A_p, A_s over prefix and suffixes, where

$$A_p = \langle Q, K_p, V_p \rangle, \quad A_s = \langle Q, K_s, V_s \rangle,$$

and

$$\langle q, k, v \rangle = \text{Softmax}\left(\frac{qk^T}{\sqrt{d}}\right) v.$$

- We calculate LogSumExp (LSE) as a weight of merging A_p and A_s. We define $LSE(q,k) = \log\left(\sum\left(\exp\left(\frac{qk^T}{\sqrt{d}}\right)\right)\right)$.
- We have

$$\langle Q, K, V \rangle = \frac{A_p e^{\text{LSE}(Q,K_p)} + A_s e^{\text{LSE}(Q,K_s)}}{e^{\text{LSE}(Q,K_p)} + e^{\text{LSE}(Q,K_s)}}. \tag{2}$$

Table 8: Comparison among DeFT and concurrent works in single-context large-batch sampling scenarios [3, 43, 18]. More $\star$ means more balanced workloads after tree split, which also shows how insensitive the acceleration is to the tree topology.

Method	Chunk-Attention [43]	Hygragen [18]	Bifurcated-Attention [3]	DeFT-Node	DeFT-Flatten
IO-aware levels	2 (depth<=1)	2 (depth<=1)	2 (depth<=1)	all(every depth)	all(every depth)
Tree KV split granularity	by node first, then by block	by tree depth	by tree depth	by tree node	flatten tree, then by block
Load-balanced level	$\star\star$	$\star$	$\star$	$\star$	$\star\star\star$
Goal metrics	throughput	throughput	latency	latency	latency

Comparison of differences. The existing works of single-context large-batch sampling are not hardware-efficient for general tree-based decoding with two reasons, as shown in Table 8:

- They are designed for decoding trees with only two levels—prefixes at the root and suffixes at depth 1. For decoding trees with multiple levels of prefixes, their algorithm can only reduce the IO of the prompt at the root of the tree. However, in scenarios such as multi-step reasoning [42, 4, 11], the token length of non-root prefixes can also be very long (e.g., thousands of tokens), and their KV cache's IO is not reused. DeFT can reuse KV IO of all non-leaf prefixes in a general decoding tree, providing greater acceleration potential.
- They have not addressed the unbalanced workload problem in tree-based decoding. Nodes in the decoding tree can vary significantly, making it crucial to split the tree and group QKV in a way that ensures balanced calculations for each QKV group. Simply dividing based on depth alone is insufficient.

A.4 Discussion of Techniques in Efficient Attention Algorithm Design

Table 9: Technique list of DeFT. What we propose is in red. The details of the first four techniques are in Section 3.3, while the details of the following techniques are discussed in this chapter.

Technique	**Goal**
KV-guided Grouping with Tree Split	High utilization of GPU and minimal KV cache IO between HBM and shared memory.
DeFT-*Node Tree Split*	High utilization of GPU and simple tree attention calculation.
DeFT-*Flatten Tree Split*	High utilization of GPU and balanced attention calculation.
Bit Causal Mask [27]	Record causal information of tokens in the decoding tree with little IO cost.
Kernel Fusion [6, 7]	Reduce partial results IO (e.g. $\mathbf{QK}^T$, Mask M, and Softmax, etc).
Tiling [6, 7]	Enable attention calculation within limited size of GPU's shared memory.
Tree-topology Aware Global Reduction	To get the correct tree attention of the entire decoding tree.

In this subsection, we summarize and discuss the common techniques in existing designs of efficient attention algorithms and kernels : (1) *Kernel Fusion* with *Tiling* strategy [6, 15, 27]; (2) *Tree-topology Aware Causal Mask* [27, 5]; (3) *KV Split* with *Global Reduction*[15]. Then we explain the details of design in DeFT Attention Kernel, where the techniques are in Table 9.

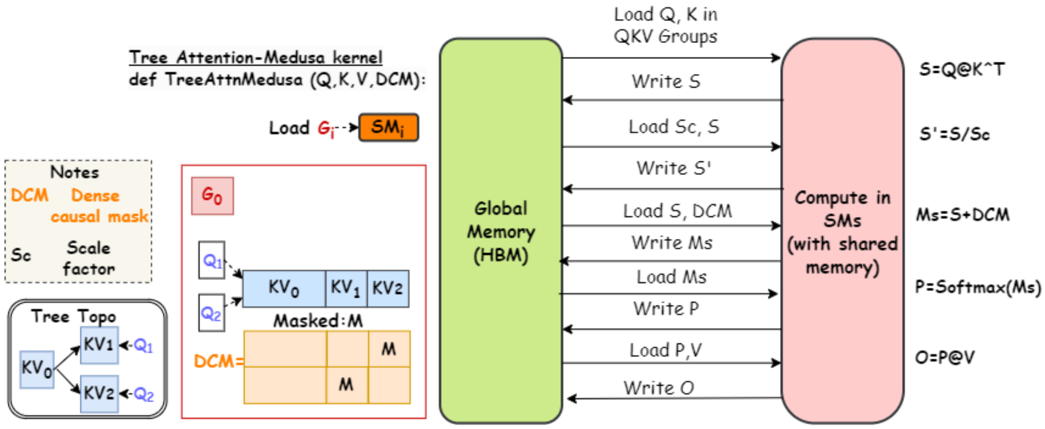


Figure 9: Operations of Tree Attention-Medusa [5]. No *Kernel Fusion* or *Tiling* strategy is applied, which introduces significant IO of partial results like $\mathbf{QK}^\top$, DCM, and Softmax between GPU global memory and on-chip shared memory.

Kernel Fusion is a common technique of IO reduction: if multiple operations are performed on the same input, it is more efficient to load the input once from HBM rather than loading it multiple times for each operation; Similarly, the same principle applies when transferring output from shared memory to HBM. To fuse all the attention operations into one GPU kernel with the limited size of shared memory, we further utilize the commonly employed *Tiling* strategy [6, 7]: split queries and KV cache within each QKV group to small blocks to prevent materialization of attention matrix in HBM by computing attention within the limited size of shared memory, then incrementally performing the softmax reduction as the formulation in Equation 2 to reconstruct the attention.

Remark A.1 (Importance of tiling and fused kernel during ATTENTION CALCULATION PHASE)**.** *Methods in this phase can be roughly divided into two categories: (1) without tiling and kernel fusion: Tree Attention in Medusa [5], which introduces significant IO operations for partial results (i.e..* $\mathbf{QK}^\top$ *and Softmax), as shown in Figure 9; (2) with tiling and a fused kernel: Flash Decoding [7], Tree Attention in SpecInfer [27] and our* DEFT.

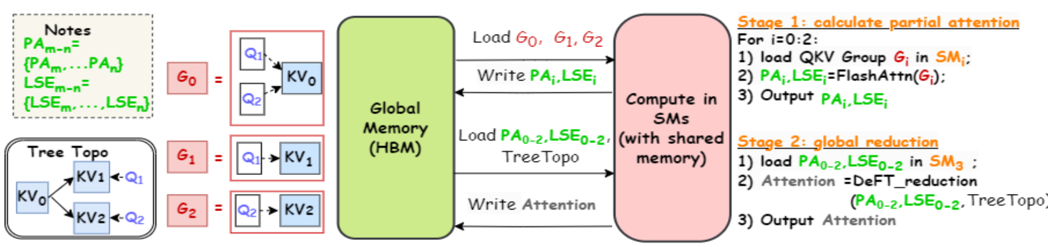


Figure 10: *Overview of two stages in* DEFT *Attention Kernel (*DEFT*-Node for example).* **Stage 1–calculate partial attentions.** Based on the QKV grouping results after *KV-Guided Grouping Strategy with Tree Split* as mentioned above, each QKV group (G_i) will be allocated to a thread block for Flash Attention [6] calculation with common *Kernel Fusion* and *Tiling* strategy. Similar to Flash-Decoding [7], we not only get partial attention (PA_i) but also return "LogSumExp" (LSE_i) as a weight parameter for the next stage's reduction. **Stage 2–global reduction.** Upon receiving PA_i and LSE_i for each QKV group G_i, DEFT now performs a *Tree-Topology-Aware Global Reduction* ($DeFT\_reduction$). Guided by the tree topology among sequence nodes of KV in the decoding tree, DEFT logically remaps the partial results of attention and LogSumExp to get the correct final attention for each query after reduction. The decoding tree is the same as the one in the left of Figure 3. SM_i means the streaming multiprocessor i in GPU.

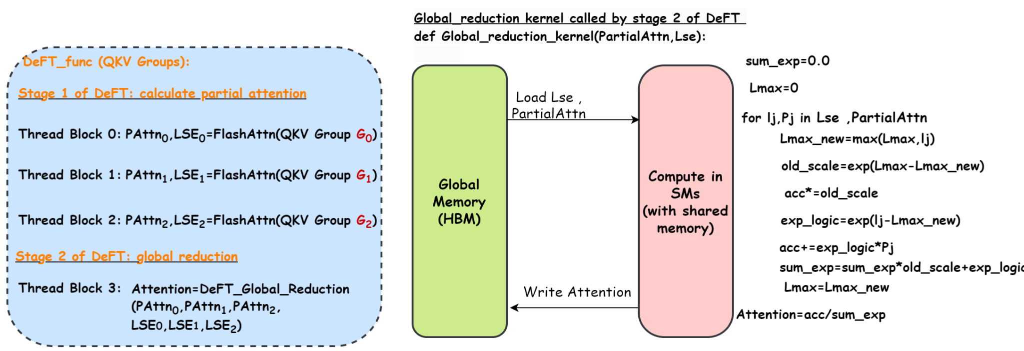
The *Tree-topology Aware Causal Mask* (*Causal Mask* for short) is introduced in speculative decoding works [27, 5] to facilitate the calculation of attention for all queries within a decoding tree using a single GPU kernel. It achieves this by recording the causal relationships among queries and KV cache in the decoding tree. As depicted in Figure 7, while originally designed for tree-based decoding with KV cache for a sequence of tokens and tree-structured queries, the *Causal Mask* can also be adapted to tree decoding with tree-structured KV cache and parallel queries—a configuration targeted by DEFT to enhance efficiency.

Remark A.2 (The effects of introducing a causal mask)**.** *Causal mask brings two parts of redundancy:*

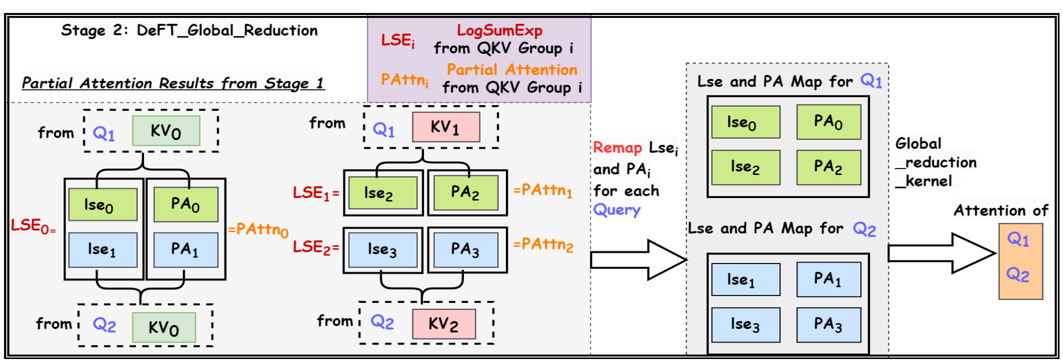
- *Memory Access. Medusa [5] materializes the dense causal mask (DCM) in HBM to record the causal information between n_q tokens in queries and n_{kv} tokens in the KV cache, thereby introducing a significant IO cost for loading this $n_q \times n_{kv}$-sized mask to shared memory. SpecInfer [27] introduces a 64-bit integer as a bit causal mask (BCM) to record the causal information among up to 64 tokens, which incurs minimal IO cost from HBM to shared memory but is not suitable for decoding trees with more than 64 tokens. Details regarding the design of the bit mask in SpecInfer are discussed in Appendix A.2.*
- *Computation. In addition to the computational cost of generating the causal mask itself, there is an additional redundancy in computation: many of the matrix multiplication results of $\mathbf{QK}^\top$ are masked out and never utilized. Both Medusa and SpecInfer have this issue.*

DEFT*-Node in Appendix A.7 does not require a causal mask and there is no IO and calculation redundancy caused by masking.* DEFT*-Flatten in Appendix A.8 adopts a bit causal mask insipred by SpecInfer [27]to minimize the IO of the causal mask. Details of the bit mask design is in the left of Figure 3.*

Split is introduced to improve GPU utilization in sequence-based decoding [15], which is necessary when the parallelism is limited by a small batch size for long-context scenarios. Flash-Decoding splits long KV and group QKV based on Q first, then these groups will be allocated to different streaming multi-processors (SMs) in the GPU to get partial attention via Flash Attention [6].



(a) Left: Illustration of DEFT-Node Attention Kernel with two stages. Right: Global reduction kernel called in DEFT stage 2 illustrated in Figure 11b. QKV Groups G_0,G_1 and G_2 are from DEFT QKV groups in Figure 3.



(b) Stage 2 of DEFT: Global Reduction. Based on tree topology in Figure 3, we can group LogSumExp and Partial Attention based on Query, then we call the Global reduction kernel in the right of Figure 11a to get the final attention.

Figure 11: **Detailed attention operations of DEFT kernel (DEFT-Node for example)**. Based on the same decoding tree in Figure 3.

To obtain the accurate final attention, partial attentions from QKV groups with identical queries need to be grouped for *Global Reduction*.

Similarly, DEFT also split the decoding tree to different QKV groups for high utilization of GPUs, which is the *KV-Guided Grouping Strategy with Tree Split* strategy we propose in subsection 3.3, as illustrated in the bottom right part of Section 3. To obtain the correct tree attention, DEFT also requires a global reduction. However, the global reduction in Flash-Decoding is for sequence-based decoding, which cannot aware the tree-topology for global reduction in tree-based decoding. Therefore, we propose *Tree-Topology-Aware Global Reduction*, as shown in the Figure 11b.

Based on the techniques mentioned above, we designed the DEFT Attention Kernel with two stages, as shown in Figure 10, to execute the attention operations after the **QKV Preparation Phase** of DEFT, which we elaborated on in Section 3.3. For more details on the DEFT Attention Kernel, see Figure 11. The attention operations of DEFT-Flatten are omitted because they are very similar to those of DEFT-Node, except for the usage of the bit causal mask for tree attention calculation.

A.5 Discussion of Workloads Generation

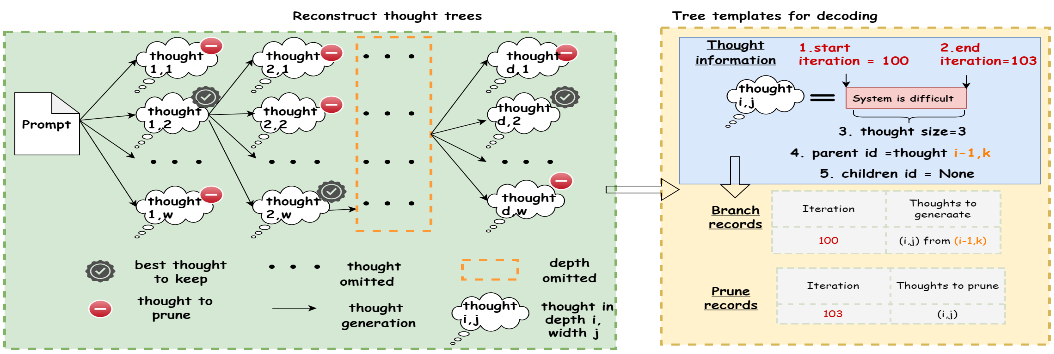


Figure 12: **The detailed procedure of reconstructing tree templates for multi-step reasoning**. (Left) Reconstructing reasoning trees from practical reasoning records as outlined in [4] involves capturing the following aspects: (1) the structure of trees, characterized by their depth d and width w; (2) the token length associated with each thought; and (3) the best thought at each depth along with its corresponding score. For the task of document merging, the tree depth is set to $d = 3$, with a width of $w = 10$ at each depth. For sorting 128 numbers, the depth is reduced to $d = 10$, while maintaining the same width of $w = 10$. See details of tree topology for other multi-step reasoning tasks in Table 10. (Right) Utilizing the extracted thought information from Left, we can generate tree templates for decoding, encompassing *branch records* and *prune records*. These records are instrumental in guiding the tree decoding process to produce decoding trees that faithfully replicate the structure of the tree-of-thoughts.

The rationality of workload settings. To validate DEFT's acceleration across various decoding tree topologies, we compiled decoding trees from real tasks, covering the following three aspects:

- Few-shot prompting: This involves a two-level tree with a prompt prefix and multiple branches for suffix generation. As a case study, we fixed the prompt length at approximately 4000 tokens and varied the number of branches.
- Multi-step reasoning [42, 11, 4]: We recorded the tree shapes, prompts, and lengths of all thoughts from real reasoning task interactions [4], using these as guidance for tree decoding to validate DEFT's acceleration in thought generation of reasoning (the thought evaluation phase follows a similar pattern). See details of generation in Figure 12.
- Speculative decoding [5, 27]: We used the token tree topology from Medusa [5] and recorded real interaction data with APPS [12] as prompt dataset, including the length of accepted tokens at each step. This served as guidance to simulate the bottleneck of speculative decoding—the attention computation during the token verification phase.

Table 10: **Details of generated workloads**. For multi-steps reasoning, we include these 4 tasks from [4]: (1) Sorting 128 numbers (*sorting* in short); (2) Document merging (*document* in short); (3) Keyword counting (*keyword* in short); (4) Set intersection (*set* in short). d, w means depth and width of the tree, respectively. t means the token tree size for speculative decoding, where the tree topology is from Medusa [5].

Task	Tree Shape	Decoding Tree Source	Records Contents
Multi-step reasoning	*sorting*: $d = 10, w = 10$ *document*: $d = 3, w = 10$ *keyword*: $d = 5, w = 10$ *set*: $d = 8, w = 10$	ToT-BFS in [4]	Prompt [4],tree shape, thought size, branch records, prune records
Few-shot prompting	$d = 1, w = 10, 20, 30$	–	–
Speculative decoding	$t = 32, 64, 128, 256$	Medusa [5]	APPS [12] Prompt, token tree shape, accepted token length per step

Table 11: **Average end-to-end latency (second) of each tree.** b means tree width. t denotes the token tree size (i.e., the number of tree-structured queries). *Speedup Upper-bound(no attention)* means the wall-clock time speedup we could obtain for the best baseline (Radix Attention) if we remove the attention calculation. ⋆ means out of memory for A100 80GB, while ♠ means not supported/implemented.

Memory	Method	Few-shot Prompting			Multi-Step Reasoning				Speculative Decoding			
		`b=20`	`b=30`	`b=50`	*Sorting*	*Document*	*Keyword*	*Set*	`t=32`	`t=64`	`t=128`	`t=256`
Unpaged	Flash-Decoding	78.96	131.19	191.09	429.65	241.20	32.75	51.76	574.50	1128.45	⋆	⋆
	Tree Attention-Medusa	52.58	103.90	144.07	380.87	236.86	33.52	50.10	263.40	483.35	924.97	1881.51
Paged	Radix Attention	12.37	14.08	16.54	104.79	69.61	11.25	17.03	64.57	86.12	145.88	263.76
	DEFT-Node	17.53	21.19	♠	114.06	81.87	15.20	22.55	84.72	♠	♠	♠
	DEFT-Flatten	9.98	10.99	12.48	94.67	66.95	10.90	16.10	44.94	50.48	65.44	104.65
	Speedup of DEFT-Flatten	1.24×	1.28×	1.33×	1.10×	1.03×	1.03×	1.05×	1.43×	1.70×	2.22×	2.52×
	Upper-bound(no attention)	1.71×	2.08×	2.51×	1.96×	1.82×	1.70×	1.76×	2.01×	2.72×	3.99×	5.12×

Table 12: **Average end-to-end IO (TB).** Data format is Left/Right: *(Left)* KV Cache IO; *(Right)* partial results IO, including $\mathbf{QK}^T$,$\mathbf{QK}^\top/s_c$, Mask M, $\mathbf{M}+\mathbf{QK}^\top/s_c$ and Softmax. b means tree width. t denotes the token tree size (i.e., the number of tree-structured queries).⋆ means out of memory for A100 80GB, while ♠ means not supported/implemented.

Method	Few-shot Prompting			Multi-Step Reasoning				Speculative Decoding			
	`b=20`	`b=30`	`b=50`	*Sorting*	*Document*	*Keyword*	*Set*	`t=32`	`t=64`	`t=128`	`t=256`
Flash-Decoding	17.62/0.00	26.43/0.00	44.05/0.00	59.96/0.00	39.74/0.00	4.68/0.00	7.01/0.00	128.72/0.00	255.16/0.00	⋆	⋆
Tree Attention-Medusa	1.68/1.05	2.10/1.98	2.94/4.61	12.40/3.69	10.57/3.24	0.58/0.18	1.04/0.27	4.02/4.03	4.15/8.33	4.18/16.77	4.32/34.70
Radix Attention	17.62/0.00	26.43/0.00	44.05/0.00	59.96/0.00	39.74/0.00	4.68/0.00	7.01/0.00	131.45/0.00	256.79/0.00	522.05/0.00	1044.10/0.00
DEFT-Node	1.68/0.00	2.10/0.00	♠	12.40/0.00	10.57/0.00	0.58/0.00	1.04/0.00	4.05/0.00	♠	♠	♠
DEFT-Flatten	1.68/0.00	2.10/0.00	2.94/0.00	12.40/0.01	10.57/0.01	0.58/0.00	1.04/0.00	4.10/0.00	4.11/0.00	4.16/0.00	4.35/0.00
IO reduction of DEFT-Flatten(%)	90.47/100.00	92.1/100.00	93.33/100.00	79.32/99.73	73.40/99.70	87.61/100.00	85.16/100.00	96.88/100.00	98.40/100.00	99.20100.00	99.58/100.00

The rationality of our experiment paradigm. Our experimental paradigm involves: first, obtaining decoding trees from real tree-based decoding tasks, and second, replicating these decoding trees exactly within the same framework by enforcing LLM inference, to investigate the impact of attention acceleration on wall clock time performance. This paradigm has two advantages:

- We can utilize decoding trees from real tasks as a benchmark within a unified system, enabling fair comparison of different attention algorithms in terms of wall-clock time performance. This comparison is possible despite the algorithms being based on distinct systems, such as variations in memory management implementations for their kernels.
- We consider both the unique characteristics of tasks with diverse tree structures and the broader applicability of general tree-based decoding. See details of generated workloads for other multi-step reasoning tasks in Table 10.

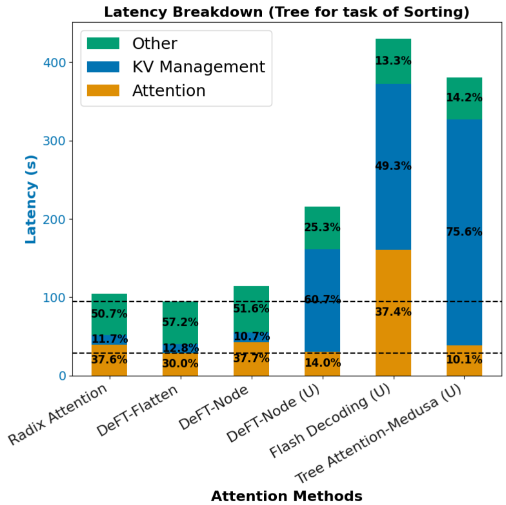
A.6 Additional Results

End-to-end latency and IOs with breakdowns. The details of end-to-end latency and IO comparsion among DEFT and baselines are in Table 11 and Table 12,respectively. We provide IO breakdowns of multi-step reasoning tasks, where the attention occupies 27.7-37.6% overhead of Radix Attention with a paged memory management. Unpaged memory will introduce about 40-75.6% overhead in end-to-end latency, due to the materialization of QKVs for tree-based decoding with a sequence-based attention kernel [6, 7].

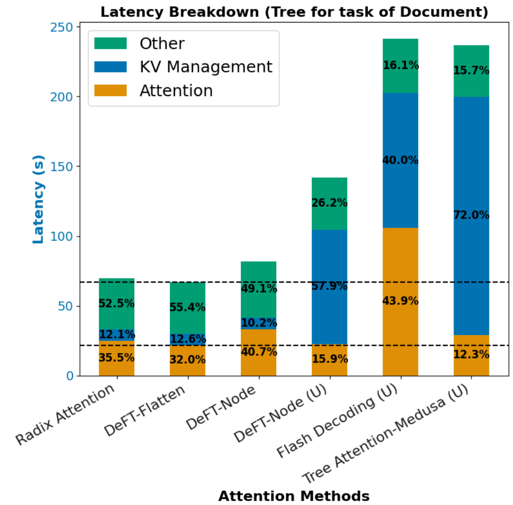
The influence of width in decoding trees. We observe that the effectiveness of attention speedup varies with different decoding tree topologies. Considering the simplest tree structure, a prompt with several suffixes—given a prompt that is not very short, one of the most important factors for speedup is the extent to which we can reuse its KV cache IO. This can be measured by the width of the tree. More specifically, it is determined by the number of queries per iteration. Therefore, we fix the prompt length at 4000 and vary the width of the decoding tree in few-shot prompting (which also indicates how many requests share the same prompt). Then, as shown in Figure 14, we evaluate DEFT-Flatten with the best baseline in attention calculation– Tree Attention-Medusa [5] (Medusa-Attn in the figure), as well as the best baseline in wall-clock time– Radix Attention [45], for the per-iteration latency over time.

We have the following observations:

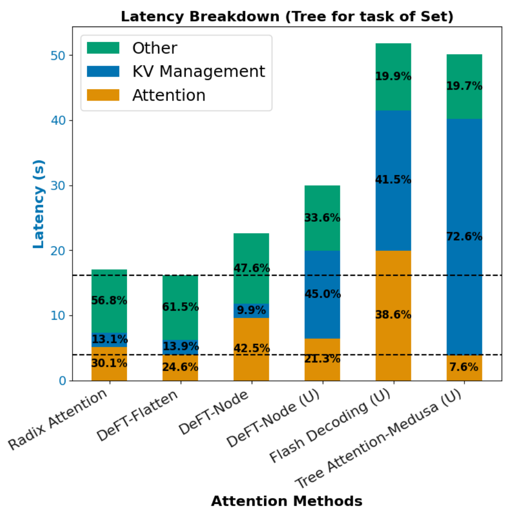
1. When the tree width is 10, the attention overhead of DEFT-Flatten is nearly the same as Tree Attention-Medusa because the IO overhead of the dense causal mask (DCM) is small compared to



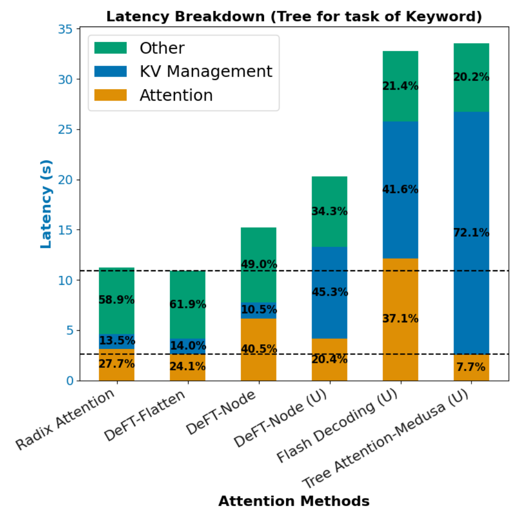
(a) Latency breakdown for task *sorting*.



(b) Latency breakdown for task *document*.



(c) Latency breakdown for task *set*.



(d) Latency breakdown for task *keyword*.

Figure 13: Latency breakdown for 4 multi-step reasoning tasks [4].

that of the KV cache, but it is still $2\times$ faster in attention latency than Radix Attention thanks to the KV IO reuse.

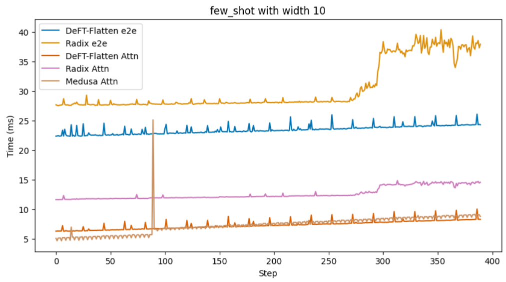
2. As the tree width increases, the attention computation overhead of Tree Attention-Medusa grows faster because the size of the DCM is directly related to the tree width. A larger tree width means the IO of the DCM grows rapidly.
3. Since the tree topology consists of a fixed prefix with several suffixes, a larger tree width allows the prompt prefix's KV cache to be reused more frequently during IO. This leads to a more significant end-to-end speedup—$1.24\times$ with a width of $w = 20$, and $1.33\times$ with a width of $w = 50$—compared to Radix Attention.
4. As iterations progress, the length of the suffixes gradually approaches the length of the prefix, leading to a decrease in the speedup of DEFT-Flatten compared with Radix Attention.

A.7 DeFT-Node Algorithm

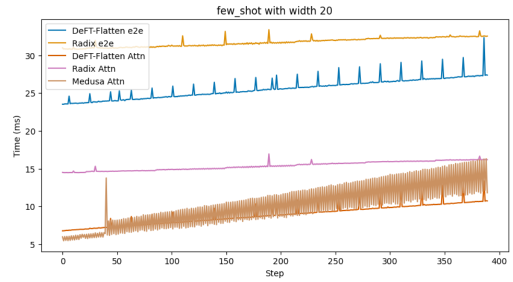
DEFT-Node has two phases-**Phase 1-QKV Preparation** and **Phase 2-Attention Calculation**.

Phase 2-Attention Calculation of DEFT has two stages.

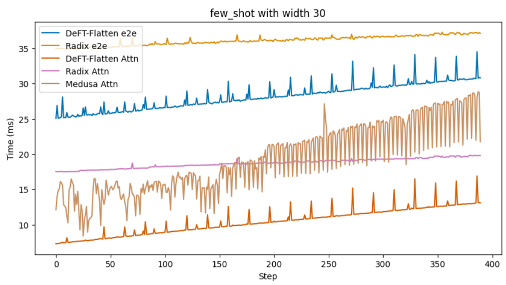
1. **Stage 1: Calculate Partial Attentions.** We will apply Flash Attention of all QKV groups obtained after **Phase 1-QKV Preparation** of DEFT, to get partial attention and LogSumExp.



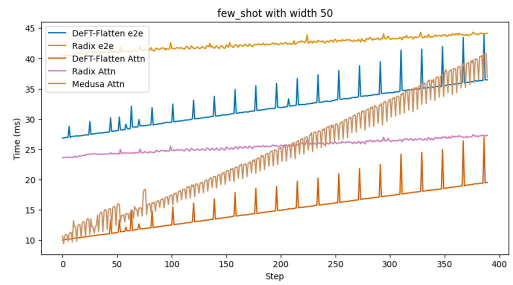
(a) Tree width is 10.



(b) Tree width is 20.



(c) Tree width is 30..



(d) Tree width is 50.

Figure 14: Per iteration latency for few-shot prompting tasks with different tree width. *e2e* means end-to-end result, while *Attn* means only the attention overhead.

Algorithm 1 DEFT-Node Algorithm-Phase 1: QKV Preparation.

Input: query $Q \in R^{(b_q,d)}$, Key cache list $KL = (K_0, ...K_{N-1})$, Value cache list $VL = (V_0, ...V_{N-1})$ for each sequence node in the tree, where N is the total number of sequences in a tree, and Tree T with its topology information.
for each q in Q with its global index idx **do**
  /*Get KV indices of all prefixes' for a query.*/
  $QMapKV[idx]$=GetPrefixKVIndices(q, KL, VL, T)
end for
for each seq's KV cache K_i, V_i in KL, VL with its KV indice i **do**
  /*Group each sequence's KV with all queries that share it.*/
  Q_i= GroupQueryToKV(Q, K_i, V_i, T) $\in R^{b_i,d} \subset Q$
  $KVMapQ[i] = Q_i$
end for
Return QMapKV, KVMapQ

2. **Stage 2: Global Reduction.** We will remap partial attention and LogSumExp based on each query, and get final attention based on global reduction similar to Flash-Decoding [7].

Algorithm 2 DEFT-Node Algorithm-Phase 2: Attention Calculation.

Input: query $Q \in R^{(b_q,d)}$, Key cache list $KL = (K_0, ...K_{N-1})$, Value cache list $VL = (V_0, ...V_{N-1})$ for each sequence node in the tree, where N is the total number of sequences in a tree, and Tree T with its topology information. QKV group information $QMapKV$, $KVMapQ$ from **QKV Preparation Phase**.

for each q in Q with its global index idx **do**
  /*Allocate to store LogSumExp of $Q@K^T$ grouped by query.*/
  $LogSumExp[idx] = \{\}$
  /*Allocate to store partial results of $SoftMax(Q@K^T)V$ for each query.*/
  $O[idx] = \{\}$
end for
/*Allocate space for output after reduction.*/
$FO = (0)_{b_q \times d} \in R^{(b_q,d)}$
for each seq's KV cache $K_i, V_i \in R^{(b_{kv},d)}, R^{(b_{kv},d)}$ in KL, VL with its KV indice i **do**
  # Unroll for loop to SMs
  Q_i= $KVMapQ[i] \in R^{(b_i,d)}$
  /*Get partial attention o_i for each QKV group, LogSumExp lse_i of $Q@K^T$ in row for reduction.*/
  o_i, lse_i = FlashAttention(Q_i, K_i, V_i)
  $\in R^{(b_i,d)}, R^{b_i}$
  /*Map the partial results back to each query for reduction.*/
  for each query q in Q_i with its group index gp_idx and global index idx in Q **do**
    if $i \in QMapKV[idx]$ **then**
      $LogSumExp[idx].append(lse_i[gp\_idx])$
    end if
  end for
end for
for each q in Q with its global index idx **do**
  # Unroll for loop to SMs
  if len($O[idx]$)==len($QMapKV[idx]$) **then**
    /*Global reduction after collecting all partial results from QKV groups that contains q.*/
    LSE_{cat}= CatTensor($LogSumExp[idx]$)
    LSE_{max}=RowMax(LSE_{cat})
    $Mid\_L = 0, Mid\_O = 0^{(1,d)}$
    for each lse_j in $LogSumExp[idx]$ **do**
      $new\_exp = e^{lse_j - LSE_{max}}$
      $Mid\_L = Mid\_L + new\_exp$
    end for
    for each lse_j, o_j in $LogSumExp[idx], O[idx]$ **do**
      $new\_exp = e^{lse_j - LSE_{max}}$
      $Mid\_O = Mid\_O + new\_exp@o_j/Mid\_L$
    end for
    $FO[idx] = Mid\_O$
  end if
end for
Return FO

A.8 DEFT-Flatten Algorithm

The algorithm (noted as DEFT-Node) in Appendix A.7 adopts a node-granularity split strategy, which is quite simple. However, when the token lengths of different nodes in a decoding tree are very unbalanced, it might introduce inefficient calculation due to the unbalanced workload in on-chip SMs of GPUs.

Therefore, we can split the decoding tree in a more balanced way– in subtree-granularity. We show the DEFT-Flatten algorithm as follows, which also consists of two stages similar to DEFT-Node.

Algorithm 3 DEFT-Flatten Algorithm-Phase 1: QKV Preparation.

Input: query $Q \in R^{(b_q,d)}$, Key cache list $KL = (K_0, ...K_{N-1})$, Value cache list $VL = (V_0, ...V_{N-1})$ for each sequence node in the tree, where N is the total number of sequences in a tree, and Tree T with its topology information. Subtree size S_t, which means each subtree after tiling contains at most S_t tokens.
/*Evenly slice/blockwise the Tree KV cache (with n_T tokens in the tree) to subtrees.*/
SubInfo, KSub, VSub =Slice(KL, VL, S_t, T)
/*Notes: (1) subtree number $m = Ceil(n_T/S_t)$;
(2) subtrees' KV cache $KSub = (Kb_0, ..., Kb_{m-1})$, $VSub = (Vb_0, ..., Vb_{m-1})$;
(3) subtree information $SubInfo = (Sb_0, ..., Sb_{m-1})$, where each subtree i has $Sb_i = (ofs_0, ...ofs_{n_{b_i}-1})$ to record the offset of each node in the subtree KV cache, with n_{b_i} as the total number of nodes in subtree i. */
for each subtree's KV cache Kb_i, Vb_i in $KSub, VSub$ with its subtree ID i **do**
  /*Group each subtree's KV with all queries that share it.*/
  Q_i= GroupQueryToKV(Q, Kb_i, Vb_i, T) $\in R^{b_i,d} \subset Q$
  $KVMapQ[i] = Q_i$
  for each query q in Q_i with a global index idx in Q **do**
    $QMapKV[idx].append(i)$
  end for
  /*Add a causal mask as different nodes in a subtree could be shared by different queries.*/
  $CausalMask[i] = GetBitMask(Q_i, Kb_i, Vb_i, T)$=$(CM_0, ...CM_{n_{b_i}-1})$
  where n_{b_i} is the total number of nodes in the subtree, and CM_i is a 64-bit int bit mask for node i.
  /*E.g, $100....00$ with 1 in bit 0, means the $Q_i[0]$ does not share KV cache of node i in the subtree.*/
end for
Return QMapKV, KVMapQ, CausalMask,SubInfo

Algorithm 4 DEFT-Flatten Algorithm-Phase 2: Attention Calculation.

Input: query $Q \in R^{(b_q,d)}$, Key cache list in subtree-granularity KSub=(Kb_0,...,Kb_{m-1}), Value cache list in subtree VSub = (Vb_0,...,Vb_{m-1} for m subtrees after tiling based on Tree T with its topology information. QKV group information $QMapKV$, $KVMapQ$, causal mask $CausalMask$ and subtree information $SubInfo$ from **QKV Preparation Phase**.

for each q in Q with its global index idx **do**
  /*Allocate to store LogSumExp of $Q@K^T$ grouped by query.*/
  $LogSumExp[idx] = \{\}$
  /*Allocate to store partial results of $SoftMax(Q@K^T)V$ for each query.*/
  $O[idx] = \{\}$
end for
/*Allocate space for output after reduction.*/
$FO = (0)_{b_q \times d} \in R^{(b_q,d)}$
for each subtree's KV cache $Kb_i, Vb_i \in R^{(b_{kv},d)}, R^{(b_{kv},d)}$ in $KSub, VSub$ with subtree ID i **do**
  # Unroll for loop to SMs
  Q_i= $KVMapQ[i] \in R^{(b_i,d)}$
  /*Reconstruct mask for attention calculation based on $CausalMask$ and $SubInfo$*/
  $bitmask = CausalMask[i] \in R^{n_{b_i}}$,where n_{b_i} is the total number of nodes for subtree i.
  $SubOfst = SubInfo[i] \in R^{n_{b_i}}$
  $mask = ReconstructMask(bitmask, SubOfst) \in R^{(b_i,b_{kv})}$
  /*Get partial attention o_i for each QKV group, LogSumExp lse_i of $Q@K^T$ in row for reduction.*/
  o_i, lse_i = FlashAttention($Q_i, Kb_i, Vb_i, mask$)
  $\in R^{(b_i,d)}, R^{b_i}$
  /*Map the partial results back to each query for reduction.*/
  for each query q in Q_i with its group index gp_idx and global index idx in Q **do**
    if $i \in QMapKV[idx]$ **then**
      $LogSumExp[idx].append(lse_i[gp\_idx])$
    end if
  end for
end for
for each q in Q with its global index idx **do**
  # Unroll for loop to SMs
  if len($O[idx]$)==len($QMapKV[idx]$) **then**
    /*Global reduction after collecting all partial results from QKV groups that contains q.*/
    LSE_{cat}= CatTensor($LogSumExp[idx]$)
    LSE_{max}=RowMax(LSE_{cat})
    $Mid\_L = 0, Mid\_O = 0^{(1,d)}$
    for each lse_j in $LogSumExp[idx]$ **do**
      $new\_exp = e^{lse_j - LSE_{max}}$
      $Mid\_L = Mid\_L + new\_exp$
    end for
    for each lse_j, o_j in $LogSumExp[idx], O[idx]$ **do**
      $new\_exp = e^{lse_j - LSE_{max}}$
      $Mid\_O = Mid\_O + new\_exp@o_j/Mid\_L$
    end for
    $FO[idx] = Mid\_O$
  end if
end for
Return FO

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Abstract and introduction accurately reflect the paper's contributions and scope.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: See section 5.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: We have IO complexity analysis in section 3.4, but it is easy and straightforward. Technique correctness proof of softmax merging in Equation 2.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: See workload generation and datasets in Appendix A.5. See algorithm in Appendix A.8.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [NA]

Justification: Datasets are open-sourced. See workload generation and datasets in Appendix A.5. See algorithm in Appendix A.8. We will release code soon.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: Datasets are open-sourced. See workload generation and datasets in Appendix A.5.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: We verify our experiments based on more than 100 traces in datasets and show average results. See workload generation and datasets in Appendix A.5.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: See section 4.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: I promise.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: This work can accelerate LLM inference.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We do it properly.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.

- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: This paper does not release new assets.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: Does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.