DeFT: Flash Tree-attention with IO-Awareness for Efficient Tree-search-based LLM Inference

Jinwei Yao[1,4,*]   Kaiqi Chen[2,*]   Kexun Zhang[3,*]   Jiaxuan You[4]   Binhang Yuan[5]
Zeke Wang[2,†]   Tao Lin[1,†]

jinwei.yao1114@gmail.com;   {chiaki_cage,wangzeke}@zju.edu.cn;
kexunz@andrew.cmu.edu;   jiaxuan@illinois.edu;
biyuan@ust.hk;   lintao@westlake.edu.cn
[1]Westlake University   [2]Zhejiang University   [3]Carnegie Mellon University
[4]University of Illinois Urbana-Champaign   [5]Hong Kong University of Science and Technology

ABSTRACT

Decoding using tree search can greatly enhance the inference quality for transformer-based Large Language Models (LLMs). Depending on the guidance signal, it searches for the best path from root to leaf in the tree by forming LLM outputs to improve controllability, reasoning ability, alignment, et cetera. However, current tree decoding strategies and their inference systems do not suit each other well due to redundancy in computation, memory footprints, and memory access, resulting in inefficient inference. To address this issue, we propose DeFT, an IO-aware tree attention algorithm that maintains memory-efficient attention calculation with low memory footprints in two stages: (1) **QKV Preparation**: we propose a *KV-Guided Tree Split* strategy to group QKV wisely for high utilization of GPUs and reduction of memory reads/writes for the KV cache between GPU global memory and on-chip shared memory as much as possible; (2) **Attention Calculation**: we calculate partial attention of each QKV groups in a fused kernel then apply a *Tree-topology-aware Global Reduction* strategy to get final attention. Thanks to a reduction in KV cache IO by 3.6-4.5$\times$, along with an additional reduction in IO for $\mathbf{QK}^\top$ and Softmax equivalent to 25% of the total KV cache IO, DeFT can achieve a speedup of 1.7-2.4$\times$ in end-to-end latency across two practical reasoning tasks over the SOTA attention algorithms.

1 INTRODUCTION

Large language models (LLMs) (Achiam et al., 2023; Touvron et al., 2023a;b) are extensively utilized across a range of tasks like chatbot (Roller et al., 2020), code generation (Mark et al., 2021), reasoning (Yao et al., 2023; Besta et al., 2023; Ning et al., 2023), etc. Tree search algorithms (Graves, 2012; Lu et al., 2022; Liu et al., 2023) are frequently integrated with LLMs to meet Service-Level-Objectives (SLOs) (Yao et al., 2023; Liu et al., 2023; Anderson et al., 2017; Post & Vilar, 2018; Hokamp & Liu, 2017). In order to cover a large search space, numerous tokens will be generated with significant computational and memory overhead, resulting in greater latency during inference.

Sequence-based decoding and tree-based decoding represent two prominent approaches in handling LLM inference (Yao et al., 2023). **Sequence-based decoding** 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-search-based tasks directly to sequence-based decoding causes three levels of redundancy: (1) *memory storage*, especially the

---

[*]Equal contribution. Work done during Jinwei's visit to Westlake University.
[†]Corresponding author.

KV cache (Kwon et al., 2023; Zheng et al., 2023); (2) *computation*, especially the computation for common prompts among sequences in a batch (Zheng et al., 2023); (3) *memory access*.

Existing work of tree-based decoding 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 (Shazeer, 2019; Cai et al., 2024; Kim et al., 2023). As for sequence-based decoding methods optimize the memory access for the aspects of partial results (i.e., $\mathbf{QK}^\top$) during attention calculations (Dao et al., 2022; 2023; Hong et al., 2023). 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.
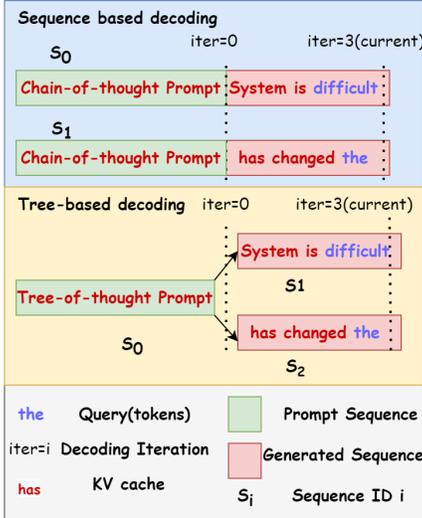
Figure 1: Comparison of Sequence-based decoding and Tree-based decoding. **An illustration of Sequence-based decoding and Tree-based decoding** with the example of Chain-of-thoughts (CoT) (Wei et al., 2022) and Tree-of-thoughts (ToT) (Yao et al., 2023) in Besta et al. (2023).

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 (Miao et al., 2023; Cai et al., 2024)—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., $\mathbf{QK}^\top$) (Cai et al., 2024) due to the lack of tiling and kernel fusion (Dao et al., 2022); and (ii) KV cache in a tree structure (Miao et al., 2023). These limitations hinder their effectiveness in optimizing memory access during tree-based decoding.

To bridge the above gap, we propose DEFT, an IO-aware tree attention algorithm with two key insights. First, the IO workload for queries (Q) is negligible compared to that of KV cache, primarily because the maximum query length typically corresponds to 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 decoding tree by nodes as each node has sequence-granularity of tokens and KV cache for high GPU utilization. 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 (Dao et al., 2023). We summarize our contributions as follows:

Table 1: **Comparison of efficiency in CoT and ToT.** The task is *document merging* from Besta et al. (2023). CoT is implemented with Sequence-based decoding while ToT is with Tree-based decoding. The total generated tokens of CoT is only 525 while 24,026 in ToT, resulting in inefficiency in end-to-end latency (`second`) and IO (`TB`). IO mainly consists of three parts as follows. (i) *KV cache*: `IO-KV`; (ii) QK^T: `IO-`QK^T; (iii) *Softmax*(QK^T): `IO-`$Softmax$. Baselines: (i) *Flash-Decoding*: attention in Flash-Decoding (Dao et al., 2023); (ii) *Tree Attention*: tree attention in Medusa (Cai et al., 2024).

	Metrics			
	`Latency`	`IO-KV`	`IO-`QK^T	`IO-`$Softmax$
Flash-Decoding + CoT	21	0.6	0	0
Flash-Decoding + ToT	450	30.7	0	0
Tree Attention + ToT	660	8.6	0.7	1.3
DeFT(ours) + ToT	272	8.6	0	0
Speed up over best baseline	1.66×	-	-	-

- 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 implement DEFT on OpenAI Triton (Tillet et al., 2019) 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 (Wolf et al., 2019; Dao et al., 2023; Cai et al., 2024; Miao et al., 2023) in terms of IO complexity.
- We empirically verify its effectiveness on practical reasoning tasks. DEFT can achieve a speedup of **1.7-2.4** times across two practical reasoning tasks compared with the SOTA attention algorithms.

2 RELATED WORK

Tree-based Decoding. Tree-based decoding, exemplified by beam search (Graves, 2012), has been pivotal in NLP, handling lexical and logical constraints (Anderson et al., 2017; Post & Vilar, 2018; Hokamp & Liu, 2017), mitigating gender bias (Lu et al., 2021), achieving communicative goals (Holtzman et al., 2018), and improving alignment (Liu et al., 2023). Recent strategies enhance LLM reasoning (Yao et al., 2023; Besta et al., 2023; Ning et al., 2023), using search trees with parallel hypothesis generation and selection based on scoring functions. Some score candidates per token (Dathathri et al., 2019; Lu et al., 2021; 2022), others per reasoning step (Welleck et al., 2022; Uesato et al., 2022; Xie et al., 2023). Efficiency in tree decoding remains underexplored despite various search algorithms' application, such as A* (Lu et al., 2022) and Monte-Carlo Tree Search (Liu et al., 2023).

Memory-efficient Attention Algorithms. Existing memory-efficient attention algorithms target sequence-based decoding. FlashAttention (Dao et al., 2022) improves self-attention computation in LLM training via tiling and kernel fusion, reducing IOs. Flash-Decoding (Dao et al., 2023) 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 (Miao et al., 2023). Tree-structured token candidates undergo parallel decoding, with SpecInfer (Miao et al., 2023) introducing a topology-aware causal masked tree attention algorithm, dynamically updating a causal mask to capture relationships among tokens. Medusa (Cai et al., 2024) uses a similar mechanism with a static causal mask, while other works (Zhao et al., 2023; Liu et al., 2024) adopt analogous approaches to enhance attention calculation efficiency. However, unlike DEFT, these existing works utilizing tree attention do not take memory access into consideration.

Discussion on Tree-based Decoding. To improve inference efficiency by generating multiple tokens in each iteration, tree-based decoding (Miao et al., 2023; Cai et al., 2024; Zhao et al., 2023; Liu et al., 2024) could have sequential past KV cache with tree-structured queries. In DEFT, we propose another tree-based decoding with tree-structured past KV cache. A general tree-based decoding could have both tree-structured past KV and queries by combining the two aforementioned tree-decoding paradigms mentioned. Details are discussed in Appendix A.2.

Storage Optimization of Tree-based Decoding. LLM frameworks optimized for tree-based decoding (Kwon et al., 2023; Zheng et al., 2023) focus on memory storage efficiency. vLLM (Kwon et al., 2023) enhances GPU memory utilization, allowing sequences from the same parent to share KV cache storage. SGLang (Zheng et al., 2023) supports dynamic KV cache management during multi-round interactions with LLMs, improving memory efficiency.

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 (Shazeer, 2019; Kim et al., 2023; Cai et al., 2024), 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 (Jia & Van Sandt, 2021) [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 (Cai et al., 2024; Miao et al., 2023) 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 (Miao et al., 2023) or fine-tuned heads (Cai et al., 2024), 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 ($Attention(\cdot)$) on $P_{root \to n}$, where $P_{root \to n}$ is the concatenation of P_{B_u} and $s_{u,n}$:*

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

The existing solution of tree attention 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 node sharing in decoding trees to reduce the redundancy of KV cache IO from HBM to on-chip shared memory. Together with the IO-awareness DEFT tree attention for KV cache and partial results (i.e., $\mathbf{QK}^\top$), 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 design the system to support DEFT Attention Kernel with two advantages: 1) efficient memory management of the KV cache in a tree structure, and 2) flexible control of the 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.

The main goal of the above system is to provide efficient interactions with the LLM during tree-based decoding. Two key interactions are 1) preparation of input metadata for the DEFT Attention Kernel (discussed in this section), and maintenance of the decoding tree after model forward (discussed in Appendix A.1).

As will be elaborated in Section 3.3, the DEFT Attention Kernel requires 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. The essential data flow of input metadata in the DEFT Attention Kernel is from HBM to shared memory in groups during the QKV PREPARATION phase outlined later. These groups are then processed by the DEFT ATTENTION KERNEL[3]. Details are discussed in Section 3.3.

---

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

[2] Medusa (Cai et al., 2024) only introduces negligible memory access of KV cache for token candidates in the tree.

[3] GPUs boast an extensive array of threads to execute an operation, known as a *kernel*. Each kernel retrieves inputs from HBM to registers and shared memory, processes them, and subsequently saves the outputs back to HBM.
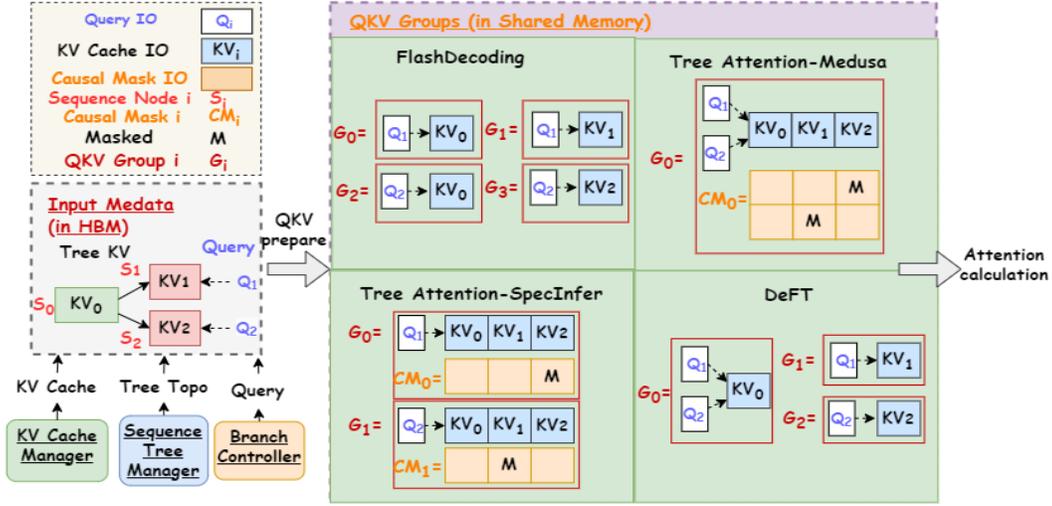
Figure 2: **Comparison of memory access from HBM to shared memory for different attention algorithms in QKV Preparation Phase**, where the amount of IO required by each is enclosed in red rectangles. *Branch Controller*, *KV cache Manager*, and *Sequence Tree Manager* are compoents to provide input medata. Different split strategies result in different memory acess: Sequence-based attention algorithms like Flash Decoding are not aware of the tree topology of KV cache, so KV_0 will be loaded twice; Tree Attention in Medusa (Cai et al., 2024) groups all queries and KV cache in a tree, with additional IO of the causal mask, then the QKV group will be allocated to streaming multiprocessors in GPU by Pytorch primitives; Tree Attention in SpecInfer will load KV cache of the whole tree for each query with the causal mask; DEFT groups QKV based on KV with tree topology information (e.g. from the decoding tree at the left, we can know Q_1 and Q_2 both share KV cache of sequence node S_0), which reduces IO of KV_0 at the expense of negligible query overhead.

3.3 An Efficient Attention Algorithm With Tiling and Reusing Tree KV cache

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

DEFT aims to be a memory-efficient algorithm in both aforementioned phases to get attention for tree-based decoding. Motivated by the heterogeneous GPU memory hierarchy (Dao et al., 2022)—namely, HBM is large but slower while the shared memory is much smaller but much faster—minimizing memory access between HBM and shared memory for memory-bound computations (e.g., attention) during the attention computation is crucial. In detail:

- In the QKV PREPARATION PHASE, we introduce a *KV-Guided Tree Split* strategy with tree-topology awareness to minimize the IO of QKV.
- During the ATTENTION CALCULATION PHASE, we propose a *Tree-Topology-Aware Global Reduction* strategy combined with the established techniques (*Kernel Fusion* and *Tiling*), to eliminate the IO of partial results (i.e.. $\mathbf{QK}^\top$ and Softmax).

QKV PREPARATION PHASE of DEFT. In sequence-based decoding, *split* strategy—namely splitting the inputs QKV into blocks—is commonly deployed to generate enough QKV groups for full utilization of the GPU (Dao et al., 2023). This technique is crucial when the parallelism (usually limited by the batch size (Dao et al., 2023)) 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 (Dao et al., 2023) may introduce significant IOs, as shown in Figure 2 and discussed in Remark 3.3.

Remark 3.3 (The effects of QKV grouping strategy in the QKV PREPARATION PHASE). *In the QKV PREPARATION PHASE, how QKVs are grouped logically results in different memory access of QKV for tree decoding, as shown in Figure 2.*
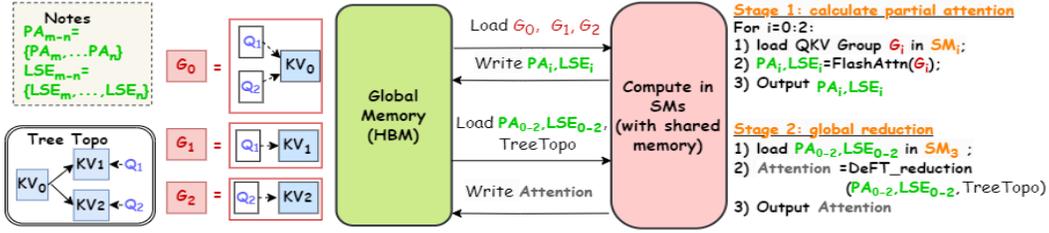
Figure 3: **Overview of two phases in DEFT Attention Kernel.** The decoding tree is the same as the one in the left of Figure 2. SM_i means the streaming multiprocessor i in GPU; G_i means the QKV group i obtained after QKV PREPARATION PHASE; PA_i, LSE_i means the partial attention and LogSumExp, respectively; $DeFT\_reduction$ means *Tree-Topology-Aware Global Reduction*. To clarify, *Tree Topo* refers to the topology of the decoding tree. Only the mappings between queries and KV cache, as well as between different nodes of KV cache, will be loaded, rather than the QKV itself. Details of two phases are discussed in Appendix A.3.

- *Flash-Decoding (Dao et al., 2023), splits long KV and group QKV based on Q without tree topology awareness, which brings redundant KV cache IO from HBM to shared memory;*
- *Tree Attention-Medusa (Cai et al., 2024) 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 (Miao et al., 2023) 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 a *KV-Guided Tree Split* strategy: it splits the tree by sequence nodes[4], and then groups the KV of each node with all queries that share it based on tree topology. This 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 2.

Remark 3.4 (Properties of *KV-Guided Tree Split* strategy). *The additional IO cost of Q in DEFT is negligible because the length of the KV often surpasses that of the Q during tree decoding, primarily due to two reasons: (1) 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; (2) In tree-search tasks, the token length of each sequence node in the decoding tree ("thought" in reasoning) is typically not exceedingly small (Yao et al., 2023), implying that the KV sequence length of a node is often much larger than the total length of all queries that share it.*

Besides, DEFT is extreme simple yet effective: during decoding, there is no need for DEFT to utilize a causal mask[5] (Miao et al., 2023; Cai et al., 2024) to calculate the attention in the incoming ATTENTION CALCULATION PHASE. Instead, only the tree topology among sequence nodes in the decoding tree is required.

ATTENTION CALCULATION PHASE of DEFT. In this phase, we design DEFT Attention kernel to load QKV splits in a memory efficient way, which are 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.3: 1) common *Kernel Fusion* and *Tiling* strategies avoid significant IO operations for partial results (i.e.. $QK^\top$ and Softmax), which Tree Attention-Medusa (Cai et al., 2024) lacks; 2) a novel *Tree-Topology-Aware Global Reduction* inspired by Flash-Decoding (Hong et al., 2023) retrieves the final attention of each query based on partial attention results from each QKV group with tree topology.

DEFT Attention Kernel consists of two stages, as shown in Figure 3:

1. **Stage 1–calculate partial attentions.** Based on the QKV grouping results after *KV-Guided Tree Split* as mentioned above, each QKV group will be allocated to a thread block for Flash Attention (Dao et al., 2022) calculation with common *Kernel Fusion* and *Tiling* strategy. Similar to Flash-Decoding (Dao et al., 2023), we not only get partial attention but also return "LogSumExp" as a weight parameter for the next stage's reduction.

---

[4]When the sequence length of a node is substantial, it can be split into blocks similar to Flash-Decoding (Dao et al., 2023) to ensure the more balanced KV lengths among QKV groups.

[5]In Appendix A.6, we also provide a variant of the DEFT algorithm with split-granularity of subtrees, which could have more balanced QKV groups by split the decoding tree to evenly subtrees, with the cost of introducing causal masks.

Table 3: **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 $\#heads * \#layer * dtype\_size$. Red ticks mean the best among all methods in the table, while red crosses mean the (potential) worst.

Method	Query	KV cache	$\mathbf{QK}^{\top}$	Softmax($\mathbf{QK}^{\top}$)
Naive Attention	$\mathcal{O}(l_n d_{head})$	$\mathcal{O}(d_{head} \sum_{i=1}^{l_n} N_i)\times$	$\mathcal{O}(2\sum_{i=1}^{l_n} N_i)$	$\mathcal{O}(2\sum_{i=1}^{l_n} N_i)$
Flash-Decoding	$\mathcal{O}(l_n d_{head})$	$\mathcal{O}(d_{head} \sum_{i=1}^{l_n} N_i)\times$	$0\checkmark$	$0\checkmark$
Tree Attention-Medusa	$\mathcal{O}(l_n d_{head})$	$\mathcal{O}(d_{head} N_{tree})\checkmark$	$\mathcal{O}(2l_n N_{tree})\times$	$\mathcal{O}(2l_n N_{tree})\times$
Tree Attention-SpecInfer	$\mathcal{O}(l_n d_{head})$	$\mathcal{O}(d_{head} N_{tree} l_n)\times$	$0\checkmark$	$0\checkmark$
DEFT (Ours)	$\mathcal{O}(l_n d_{head})$	$\mathcal{O}(d_{head} N_{tree})\checkmark$	$0\checkmark$	$0\checkmark$

2. **Stage 2–global reduction.** Upon receiving partial attention and LogSumExps for all QKV groups—recall that we grouped QKV based on KV before attention calculation—DEFT now performs a *Tree-Topology-Aware Global 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.

Implementation details. We implement the DEFT attention kernel by OpenAI Triton (Tillet et al., 2019), which enables us to control memory access from global memory to shared memory and attention calculations in a thread block granularity. DEFT algorithm with two phases in a Python style can be found in Appendix A.5.

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 2, and we summarize the corresponding IO breakdown in Table 3. 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 *and Tree Attention-Medusa (Cai et al., 2024).*

Table 2: **Notations**.

l_n	Number of leaf nodes in a decoding tree, which means how many queries 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.
d_{head}	Head dimension of LLM.
F_s	Shared factor of reusing prefixes in tree attention, which means to which extent we can reduce IOs of KV cache: $F_s = (\sum_{i=1}^{l_n} N_i)/N_{tree}$.

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[6].

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 (Touvron et al., 2023a;b), where $d_{head}=128$, with $l_n=32$, the total IO cost of $\mathbf{QK}^T$ and Softmax matches that of the KV cache.

Remark. Though similar to DEFT, SpecInfer (Miao et al., 2023) 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 3.

4 EXPERIMENTS

In this section, we evaluate DEFT and other methods on tree-search-based tasks like reasoning.

---

[6]Note that both $\mathbf{QK}^{\top}$ 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.

Figure 4: **The detailed procedure of reconstructing tree templates**. (Left) Reconstructing reasoning trees from practical reasoning records as outlined in Besta et al. (2023) 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 $w = 10$. (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.

4.1 EXPERIMENTAL SETUP

To ensure a fair and feasible comparison, we use tree KV management illustrated in Section 3.2 throughout our evaluations. We omit results with sequence-based KV cache management, due to the inefficient memory footprint and out-of-memory issue[7].

Baselines. We evaluate the performance of DEFT in NVIDIA A100 (80GB) in Llama2-7B-HF model (Touvron et al., 2023b) with SOTA attention algorithms in sequence-based and tree-based decoding: (1) *Flash-Decoding* (Dao et al., 2023), using the CUDA implementation; (2) *Tree Attention-Medusa*, where we extend it to the PyTorch implementation suitable for general tree attention, inspired by the Medusa (Cai et al., 2024) with fixed tree masks. [8]

Workloads generation. Using the interaction records of Tree-of-Thoughts methods with GPT 3.5 Turbo from Besta et al. (2023), we reconstruct the decoding trees for two tasks: (1) *sorting 128*—sorting 128 numbers; (2) *doc merging*—document merging. We elaborate on the procedure of extracting decoding trees from interaction records with LLM from Besta et al. (2023) in Figure 4, which could be tree templates[9] to guide the tree decoding process with the same function as the tree-search algorithm. For the sake of simplicity, we ignore the evaluation prompt template and its overhead to select the best thought when generating the tree of thoughts. We profile the token length distribution of "thought" nodes in reconstructed trees from these two reasoning tasks. We find that 79.5% and 100% of thoughts' KV cache in the tasks of *sorting 128* and *document merging*, respectively, have a greater length than the maximum number of queries among 100 trees during tree decoding. This provides strong evidence that the IO of queries introduced by our *KV-Guided Tree Split* strategy can be ignored compared with KV Cache IO. See detailed distribution of token length and workload analysis in Appendix A.4.

4.2 RESULTS

End-to-end behaviors: latency and IOs. We compare DEFT with Flash-Decoding and Tree Attention-Medusa in average latency and IOs for tasks of *sorting 128* and *doc merging* in Table 4. The optimization in KV cache results in a significant reduction of IOs, achieving a $3.6-4.5\times$ reduction

---

[7]When applying sequence-based KV cache management (no sharing in storage for prefixes), we find sequence-based decoding algorithms—e.g., i) naive sequence-based attention in HuggingFace Transformers (Wolf et al., 2019), and ii) original Flash-Decoding implementation—meet out-of-memory in the middle of decoding ($\sim 2,000$ iterations for workloads in this section) even for A100 with 80GB memory capacity.

[8]We didn't include the tree attention operator in SpecInfer (Miao et al., 2023) because its kernel only support at most 64 tokens in the decoding tree. Details in Appendix A.2.

[9]The decoding trees would be forced to branch and prune in certain iterations to get exactly the same shape of the decoding tree as the original tree-of-thoughts process, to ensure fairness for workloads of different baselines.

Table 4: **Average end-to-end latency (second) and IO breakdown (TB) when decoding with 100 trees of two reasoning tasks from (Besta et al., 2023):** (i) (Left) *sorting 128 numbers*; (ii) (Right) *document merging*. IO mainly consists of three parts as follows. (i) *KV cache*: IO-KV; (ii) QK^T: IO-QK^T; (iii) *Softmax*(QK^T): IO-$Softmax$. Baselines: (i) *Flash-Decoding*: attention in Flash-Decoding (Dao et al., 2023); (ii) *Tree Attention-Medusa*: causal masked tree attention in Medusa (Cai et al., 2024).

	Task of sorting 128 numbers				Task of document merging			
	Latency	IO-KV	IO-QK^T	IO-$Softmax$	Latency	IO-KV	IO-QK^T	IO-$Softmax$
Flash-Decoding	548	35.1	0	0	450	30.7	0	0
Tree Attention-Medusa	653	7.9	0.6	1.3	660	8.6	0.7	1.3
DeFT(ours)	305	7.9	0	0	272	8.6	0	0
Speed up over best baseline	1.80×	-	-	-	1.66×	-	-	-
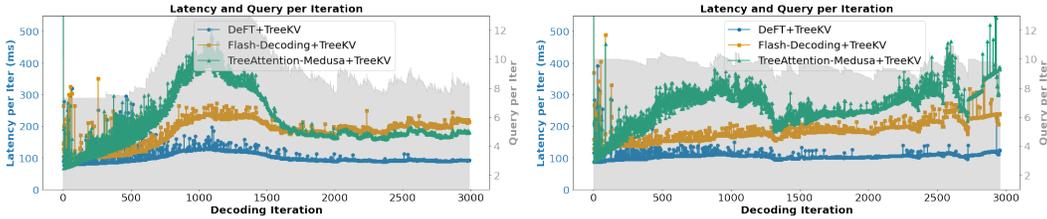
(a) Average latency and query length when decoding with 100 trees of sorting 128 numbers per iteration.

(b) Average latency and query length when decoding with 100 trees of document merging per iteration.

Figure 5: **Average per iteration results for 100 trees in each reasoning task.**

compared to Flash-Decoding's KV cache IO. Additionally, the reduction of IOs in partial results during attention calculation leads to a reduction of $1.9 - 2$TB (equivalent to 25% of the total KV cache IO) compared to Tree Attention-Medusa. *These optimizations jointly contribute to a notable speedup of our* DEFT*, achieving* $1.7 - 1.8×$ *improvements over Flash-Decoding and* $2.2 - 2.4×$ *improvements over Tree Attention-Medusa, respectively.* We will discuss why the reduction of KV cache IO does not have a significant acceleration effect on causal masked tree attention below.

Dynamic behaviors: latency per iteration. The size of the decoding tree dynamically changes due to branch and prune operations among iterations. To capture this dynamic behavior, we track the latency changes across tree decoding iterations in Figure 5 for DEFT, Flash-Decoding, and Tree Attention-Medusa. Notably, Tree Attention-Medusa exhibits a strong sensitivity to query length: as evident in iteration $1,000$ of Figure 5a, the sensitivity arises as the size of the partial result is directly proportional to the query number. Consequently, this not only leads to increased IO but also results in a larger memory footprint, with the GPU's peak memory usage reaching 95% compared to that of 40% in DEFT. However, when the tree size is relatively small due to pruning, Tree Attention-Medusa can outperform Flash-Decoding, as observed in iteration $2,500$ of Figure 5a. DEFT *significantly outperforms these two baselines and exhibits stable performance across a variety of tree sizes.*

5 DISCUSSION

In conclusion, we have proposed DeFT, a novel IO-aware tree attention algorithm to accelerate large language models combined with tree search algorithms. DeFT can aware topology of decoding tree to ease the great IO of KV cache, and a fused kernel is adopted to eliminate the IO of partial results during attention calculations. We have demonstrated in two practical reasoning tasks, DeFT can obtain $1.7 - 2.4×$ speedup thanks to $3.6 - 4.5×$ reduction of KV cache IO and $1.9 - 2$TB reduction (equivalent to 25% of the total KV cache IO) of $\mathbf{QK}^\top$ and Softmax.

Key advantages of DeFT are IO-aware and simple, as no tree attention mask is required during the decoding. DeFT is also not sensitive to query numbers of the tree, which shows great potential to support a large search space with multiple branches.

REFERENCES

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.

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 (eds.), *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 936–945, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1098. URL https://aclanthology.org/D17-1098.

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.

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.

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.

Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference, 2023. URL https://pytorch.org/blog/flash-decoding/. PyTorch Blog.

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.

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

Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In Regina Barzilay and Min-Yen Kan (eds.), *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1535–1546, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1141. URL https://aclanthology.org/P17-1141.

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)*, pp. 1638–1649, 2018.

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.

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

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.

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.

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.

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.

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*, pp. 4288–4299, 2021.

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*, pp. 780–799, 2022.

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.

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.

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.

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 (eds.), *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1314–1324, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/ N18-1119. URL https://aclanthology.org/N18-1119.

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.

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

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*, pp. 10–19, 2019.

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*, 2023a.

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*, 2023b.

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.

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.

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.

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.

Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, Xu Zhao, Min-Yen Kan, Junxian He, and Qizhe Xie. Self-evaluation guided beam search for reasoning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=Bw82hwg5Q3`.

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.

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.

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.

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[10] from the Sequence Tree Manager.
4. **Model Interface**: pass input metadata to DeFT Attention kernel and MLP module, then return logits and memory pointers of updated KV cache.

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 during the QKV PREPARATION phase discussed later. Then QKV groups will be processed by DEFT ATTENTION KERNEL in Section 3.3.

A.2 DISCUSSION OF TREE-BASED DECODING

Tree-based decoding could have tree-structured KV cache for storage with awareness of shared prefixes (Zheng et al., 2023), or tree-structured queries in parallel decoding (Miao et al., 2023; Cai et al., 2024), as shown in Figure 7.

In SpecInfer(Miao et al., 2023), 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

---

[10]e.g. when a node is pruned in the decoding tree, its KV space will be evicted using a *Remove* operation.
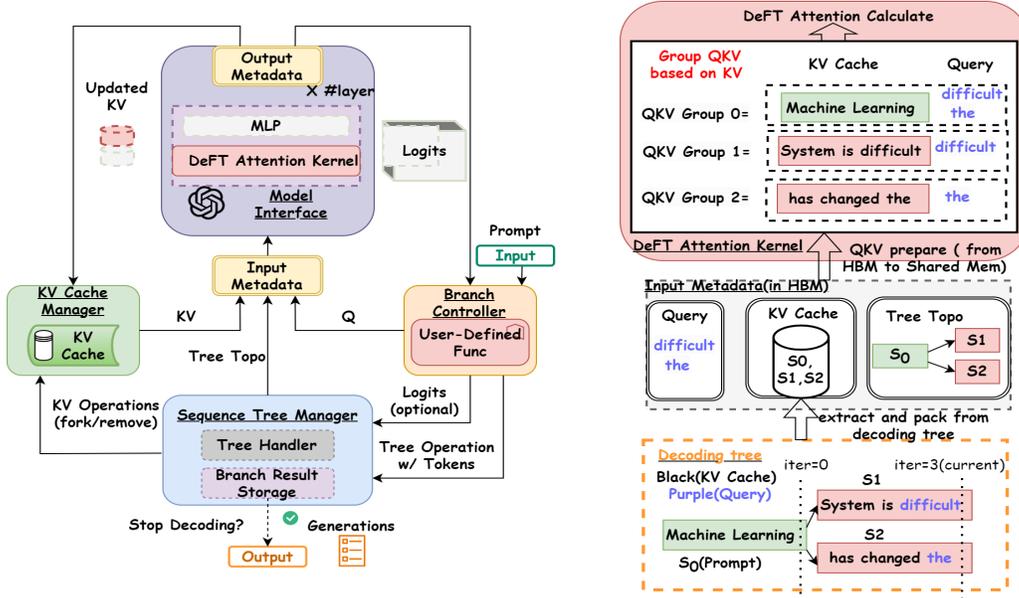
Figure 6: **Illustration of DEFT**. (Left) System overview. (Right) The data flow using a decoding tree example.
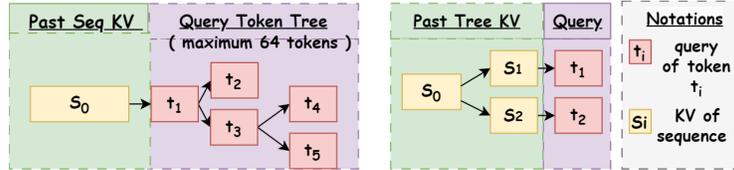
(a) (Left) Sequence KV with queries in a tree for parallel decoding (Miao et al., 2023), where a *bit mask* in Figure 7b is applied to record the causal information among queries in a tree of tokens. (Right) Tree KV with parallel queries for shared prefixes in DEFT.
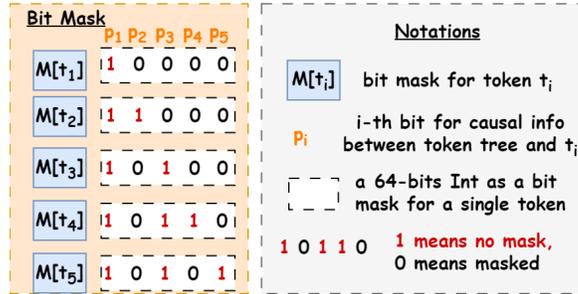
(b) *Bit Mask* in SpecInfer (Miao et al., 2023) to record the causal information between query tokens in a tree structure. The decoding tree is in the left part of 7a.

Figure 7: Discussion of **tree-based decoding** with tree queries (Miao et al., 2023) and tree KV.

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.

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.
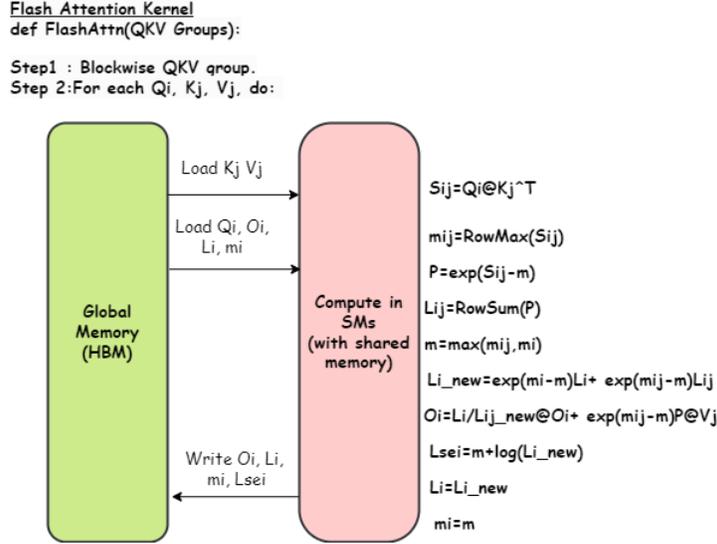
Figure 8: Operations of FlashAttention (Dao et al., 2022). *Kernel Fusion* with *Tiling* strategy is adopted to eliminate the IO of partial results like $\mathbf{QK}^{\top}$ (S_{ij} in the right part of the figure) and Softmax.
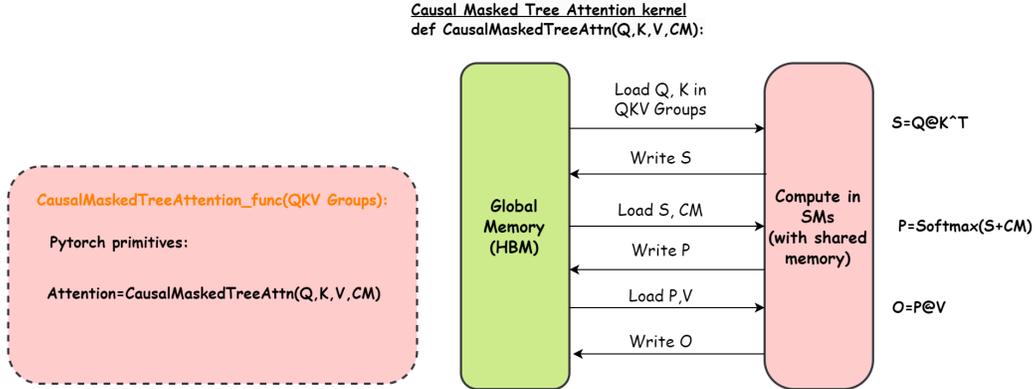
Figure 9: Operations of Tree Attention-Medusa (Cai et al., 2024). No *Kernel Fusion* or *Tiling* strategy is applied, which introduces significant IO of partial results like $\mathbf{QK}^{\top}$ (S in the right part of the figure and Softmax between GPU global memory and on-chip shared memory.

A.3 Discussion of Techniques in Efficient Attention Algorithm Design

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 (Dao et al., 2022; Hong et al., 2023; Miao et al., 2023); (2) *Tree-topology Aware Causal Mask* (Miao et al., 2023; Cai et al., 2024); (3) *Split* with *Global Reduction*(Hong et al., 2023). Then we explain the details of design in DEFT Attention Kernel.

Kernel Fusion is a common technique of IO reduction, as shown in Figure 8: 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 (Dao et al., 2022; 2023): 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 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:*
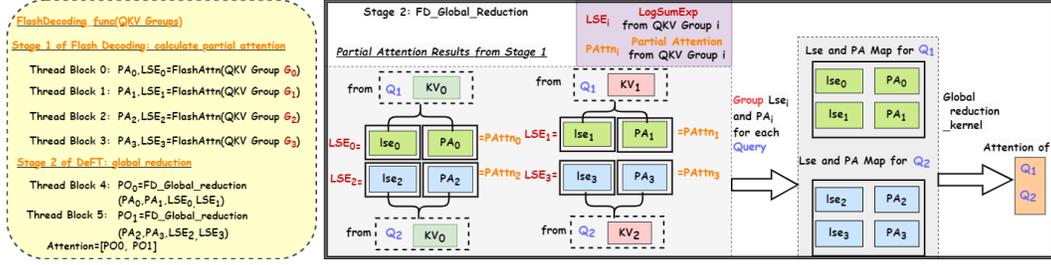
Figure 10: Operations of Flash-Decoding (Dao et al., 2023). (Left)Overview of two stages in Flash-Decoding. (Right) Details of stage 2–global reduction in Flash-Decoding. The QKV groups G_0–G_3 are from the top left of Figure 2.
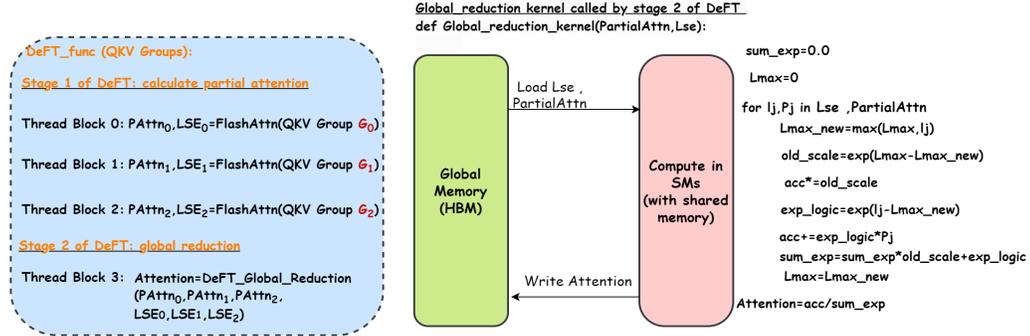
(a) Left: Illustration of DEFT kernel with two stages. Right: Global reduction kernel called in DEFT stage 2 illustrated in Figure 11b. QKV Groups G_0,G_1 and G_2 are from DEFT QKV groups in Figure 2.
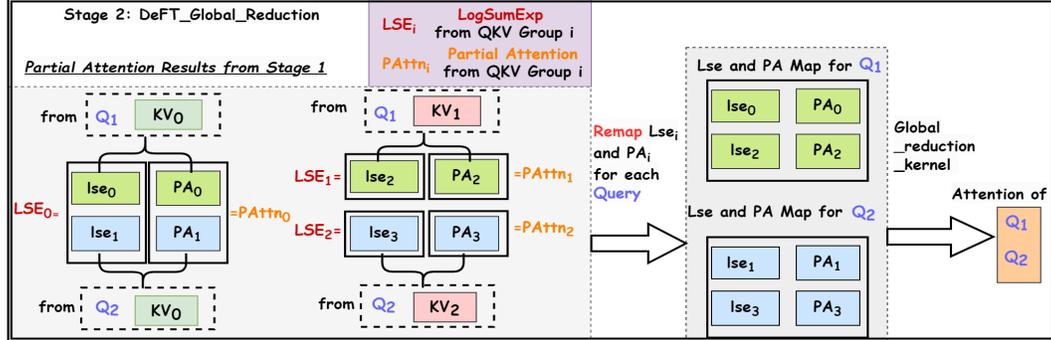
(b) Stage 2 of DEFT: Global Reduction. Based on tree topology in Figure 2, we can group LogSumExp and Partial Attention based on Query, then we call the Global reduction kernel in the right of Figure 11a to get the final attention.

Figure 11: **Attention operations of DEFT kernel**. Based on the same decoding tree in Figure 2.

Tree Attention in Medusa (Cai et al., 2024), 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 (Dao et al., 2023), Tree Attention in SpecInfer (Miao et al., 2023) and our DEFT.

The *Tree-topology Aware Causal Mask* (*Causal Mask* for short) is introduced in speculative decoding works (Miao et al., 2023; Cai et al., 2024) 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 (Cai et al., 2024) materializes the causal mask 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 (Miao et al., 2023) introduces a 64-bit integer as a bit mask 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.5 does not require a causal mask and there is no IO and calculation redundancy caused by masking.* DEFT-*Subtree in Appendix A.6 adopts a bitmask insipred by SpecInfer (Miao et al., 2023)to minimize the IO of the causal mask. Detials of this bitmask design will be in the future version of this paper.*

Split is introduced to improve GPU utilization in sequence-based decoding (Hong et al., 2023), which is necessary when the parallelism is limited by a small batch size for long-context scenarios. As shown in Figure 10, 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 FlashAttention kernel.

To obtain the accurate final attention, partial attentions from QKV groups with identical queries need to be grouped for *Global Reduction*, as depicted in the right section of Figure 10.

Similarly, DEFT also split the decoding tree to different QKV groups for high utilization of GPUs, which is the *KV-Guided Tree Split* strategy we propose, as illustrated in the bottom right part of Figure 2. To retriviel 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 11b.

A.4 DISCUSSION OF WORKLOADS GENERATION

The distribution of thought token lengths over 100 trees across various depths for two reasoning tasks–sorting 128 numbers and document merging from Besta et al. (2023), is shown in Figure 12. We can find that 79.5% and 100% of thoughts' KV cache in the tasks of *sorting 128* and *document merging* respectively, have a larger length than the maximum query numbers among 100 trees during tree decoding.

While the above observation mentioned in multi-step reasoning trees (Yao et al., 2023) generally holds true, we aim to optimize efficiency for a broader range of tree-based tasks. In scenarios where the token length of queries may be comparable to that of KV cache, there may still be room for optimization with DEFT-Node (see Appendix A.5). However, for situations like these, DEFT-Subtree (refer to Appendix A.6) is adept at evenly splitting the decoding tree into subtrees, ensuring that the token length of KV cache is significantly larger than that of queries in each QKV group.

A.5 DEFT-NODE ALGORITHM

DEFT-Node[11] 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.
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 (Dao et al., 2023).

---

[11]If there is no special explanation of the DEFT Attention algorithm in the text, it refers to DEFT-Node Algorithm.
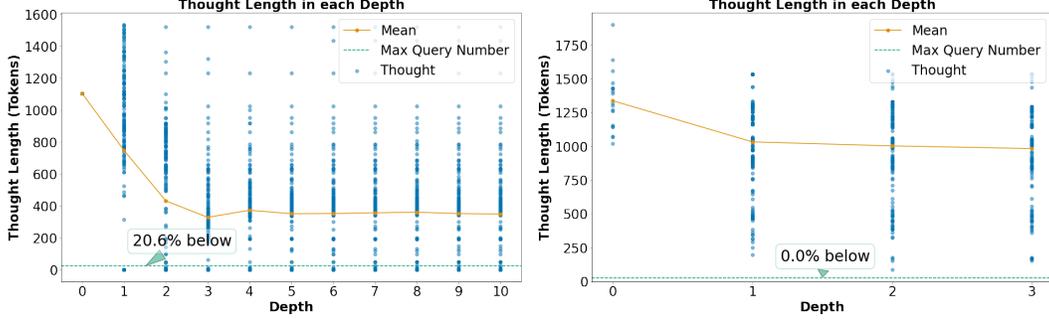
(a) Thought token length distribution in *sorting 128*. 79.5% of thoughts' KV cache length is larger than the maximum query numbers.

(b) Thought token length distribution in *document merging*. 100% of thoughts' KV cache length is larger than the maximum query numbers.

Figure 12: **The distribution of thought token lengths** over 100 trees across various depths for two tasks. We compare the thought token length with maximum query numbers during decoding for all tree templates and find that the query length during tree decoding is significantly smaller than the thought token length of KV. This observation highlights that 79.5% and 100% of thoughts' KV cache in the tasks of *sorting 128* and *document merging* respectively, have a larger length than the maximum query numbers among 100 trees during tree decoding. Note that the reason some thoughts are smaller than the maximum query length is attributed to GPT3.5's output not adhering to the prompt regulations, which may cause the graph-of-thought (Besta et al., 2023) parser to inaccurately parse the results.

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

A.6 DEFT-SUBTREE ALGORITHM

The algorithm (noted as DEFT-Node) in Appendix A.5 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-Subtree algorithm as follows, which also consists of two stages similar to DEFT-Node[12].

---
[12]We are testing DeFT-Subtree Algorithm and will add performance comparison in subsequent versions.

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

Algorithm 3 DEFT-Subtree 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-Subtree 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.***/

  $o_i, lse_i = \text{FlashAttention}(Q_i, Kb_i, Vb_i, mask)$

  $\in R^{(b_i,d)}, R^{b_i}$

  /***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.*/

    $LSE_{cat} = \text{CatTensor}(LogSumExp[idx])$

    $LSE_{max} = \text{RowMax}(LSE_{cat})$

    $Mid\_L = 0, Mid\_O = 0^{(1,d)}$

    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