IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 31 January 2025, accepted 18 February 2025, date of publication 24 February 2025, date of current version 28 February 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3544550

RESEARCH ARTICLE

CQS-Attention: Scaling Up the Standard Attention Computation for Infinitely Long Sequences

YIMING BIAN[1], (Member, IEEE), AND ARUN K. SOMANI[2], (Life Fellow, IEEE)
[1]Lewis-Sigler Institute for Integrative Genomics, Princeton University, Princeton, NJ 08540, USA
[2]Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010, USA

Corresponding author: Yiming Bian (yimingb@princeton.edu)

This work was supported in part by the Philip and Virginia Sproul Professorship at Iowa State University, and in part by the High Performance Computing at Iowa State University (HPC@ISU) equipment for computing through NSF under Grant MRI 1726447 and Grant MRI 2018594.

ABSTRACT Transformer models suffer from unaffordable high memory consumption when the sequence is long and standard self-attention is utilized. We developed a sequence parallelism scheme called CQS-Attention that can break the limit of sequence length. A long sequence is divided into multiple overlapping subsequences. The attention of each subsequence is independently computed and gathered as the final exact attention of the original long sequence. CQS-Attention is a fork-join parallel model comprising three components: Scheduler, Workers, and Tiler. The Scheduler equally partitions computation responsibility in a completely mutually exclusive manner and ensures the local subsequence length is minimum. Each worker independently computes the standard attention of the assigned subsequence and transfers local results to the Tiler, which produces the final attention. CQS-Attention makes attention computation embarrassingly parallel. Hence, it enjoys great performance regarding single-device memory and computation time consumption, mathematical stability and scalability. More importantly, it is fully compatible with all state-of-the-art attention optimizations. Our code and supplementary information (SI) are available at *https://github.com/CQS-Attention/CQS_Attention*.

INDEX TERMS Attention computation, cyclic quorum sets, parallel algorithm, transformer.

I. INTRODUCTION

Since the proposal of the vanilla Transformer in 2017 [1], transformer-based models have achieved remarkable results in various language and vision tasks. The attention mechanism plays a critical role. It enables the model to capture and utilize long-range contextual information effectively. However, transformer models face serious computational challenges in time and space since the demand for longer sequences grows, such as in the application of multi-turn conversation, long document comprehension, video generation, etc. In [2], Rabe et al. pointed out that modern hardware is often memory-constrained, while computation is relatively cheap. Thus, the device memory is often the limiting factor of modern accelerators. Unfortunately, the scalability of large transformer models is hampered by heavy memory requirements. They cannot handle very long sequences, especially when the standard self-attention is used. Because it scales quadratically with the sequence length, which makes long sequence modeling very inefficient [3], [4], especially during the training stage.

The associate editor coordinating the review of this manuscript and approving it for publication was Yangming Lee.

A. RELATED WORK

To alleviate the unaffordable memory demand issue, new attention mechanisms, especially approximate attentions, are introduced such as sparse attention [5], [6], [7], [8], [9], [10], local-global attention [11], [12], [13], [14], [15], [16], [17], dilated attention [18], etc. Optimization approaches have been developed including memory optimization [8], [19], [20], [21], IO optimization [22], exploitation of the distributive law [2], replacing batch with sequential computation [23], [24], etc. Distributed attention solutions were proposed in recent years such as data parallelism, tensor parallelism [25] and pipeline parallelism [26]. However, 3-dimensional parallelism is still insufficient to scale up the context length of large language model (LLM) [27], [28]. Sequence parallelism was developed as 4D parallelism such

as Ring Self-Attention (RSA) in [29], Ring Attention [27] and DistFlashAttn [28].

B. CONTRIBUTIONS

Our **contribution** in this research is to develop a simple fork-join module, called CQS module, using the theory of cyclic quorum sets (CQS). This module operates as a novel sequence parallelism method called CQS-Attention to compute standard attention computation. What makes it stand out from previous sequence parallelism designs and other optimizations is its completely mutually exclusivity and balanced workload partitioning. Independent local computations bring numerous advantages; the most unique one among them is universal compatibility. Thus, CQS-Attention can co-exist with any optimization because it divides the problem of attention computation for the entire sequence into independent sub-problems of attention computation for subsequences and any optimization can be applied to improve the sub-problem.

The underlying reason for memory-saving is fairly simple: Each worker device only receives a subsequence (roughly $\frac{1}{\sqrt{W}}$ of all tokens). Hence, the memory requirement for a single device is reduced to $(\frac{1}{\sqrt{W}})^2 = \frac{1}{W}$, where W is the number of workers in the system. More importantly, since the responsibility of each worker is equal and independent, the workload balance is achieved, and no communication is required among workers. Therefore, all local computations are embarrassingly parallel.

Instead of promoting a particular implementation or distributive design of CQS-Attention, the primary objective of this article is to present the algorithm and show its potential with solid theoretical analysis and sufficient simulations. The rest of the article is structured as follows. In Section II, we introduce preliminary knowledge of cyclic quorum sets and its all-pairs capability. We detail the design of CQS-Attention and provide theoretical analysis in Section III. In Section IV, we demonstrate the workflow and simulate a multi-GPU system that implements CQS-Attention. We elaborate on the advantages and limitations of CQS-Attention, and compare it with multiple distributed algorithms for attention computation in Section V. In Section VI, we discuss future research directions.

II. CYCLIC QUORUM SETS AND ALL-PAIRS PROPERTY

The quorum concept is used in the design of distributed computing systems when a mutual exclusion is required to execute an algorithm. A quorum is a subset of all sites in the system. Before a site executes a critical section, it only needs permission from the sites in its quorum instead of all other sites. Maekawa's algorithm [30] is the first quorum-based algorithm that achieves mutual exclusion using the following properties of quorum sets: non-empty intersection, equal workload, and equal responsibility. Maekawa's algorithm is symmetric and allows fully parallel operation.

Finding such a feasible group of quorum sets for a distributed system with $W = m(m-1)+1$ sites is equivalent to finding a finite projective plane of $m-1$ points, where m is the quorum size [30]. Unfortunately, no major progress has been made since Bruck-Ryser-Chowla theorem [31], [32], which states the non-existence of planes of certain orders. Lam ran an exhaustive search on CRAY-1A for more than 2 000 hours and confirmed the non-existence of finite projective plane of order 10 [33]. Thus, in a distributed system with 111 sites, a feasible set of 111 quorums whose size is 11 does not exist.

In [34], Luk et al. developed two quorum-based mutual exclusion algorithms with the aforementioned properties. One of them is based on cyclic difference sets in combinatorial theory [35]. It works for an arbitrary number of sites (W) in a distributed system. The scheme is strictly symmetric, and the quorum size (m) is close to the theoretical Lower Bound ($\mathrm{LB}(m) = \sqrt{W}$). The set of quorums found in this mutual exclusion task is called cyclic quorum sets (CQS). In [36], Kleinheksel et al. solved the intensive all-pairs computation problem in a distributed manner using CQS. This method distributes workload equally, reduces memory footprint, and minimizes data replication in all pair interactions.

CQS is a set of W quorums with the same size (m). Given any one quorum, all others can be determined cyclically. E.g., given $Q_x = \{q_{a_0}, q_{a_1}, \cdots, q_{a_{m-1}}\}$, the remaining $W-1$ quorums are computed using Eq. (1), where $i \in [1, W)$.

$$Q_{(x+i) \bmod W} = \{q_{(a_0+i) \bmod W}, \cdots, q_{(a_{m-1}+i) \bmod W}\} \quad (1)$$

The whole set of data is first evenly divided into W subsets of data (q_x). Using W workers to compute all possible pairs among $\{q_0, q_1, \cdots, q_{W-1}\}$, or $\frac{W(W-1)}{2}$ pairs, each worker receives a quorum of m data subsets and computes all pairs within it, thus $\frac{m(m-1)}{2}$ local pairs. If all possible pairs are covered by the union of all local pairs, this set of quorums, or CQS, has the all-pairs property. A trivial case is $m = W$ where each worker equally computes all possible pairs. The interest is to find a CQS of minimum quorum size (m) with all-pairs property. Moreover, the set of subscripts of the lexicographically smallest quorum in such a CQS is called an interest set ($\mathscr{I}$), thus $\{0, 1, a_2, \cdots, a_{m-1}\}$, letting $a_0 = 0$ and $a_1 = 1$ because the first two elements are always $\{0, 1\}$ [37].

	Early	worm	is	caught	by	the	bird
Early	0	0	0	0	0	0	0
worm	1	1	1	1	1	1	1
is	2	2	2	2	2	2	2
caught	3	3	3	3	3	3	3
by	4	4	4	4	4	4	4
the	5	5	5	5	5	5	5
bird	6	6	6	6	6	6	6

FIGURE 1. A naïve work partition of $P \in \mathbb{R}^{7\times7}$: Numbers are worker indices as a secondary discriminative feature to color.

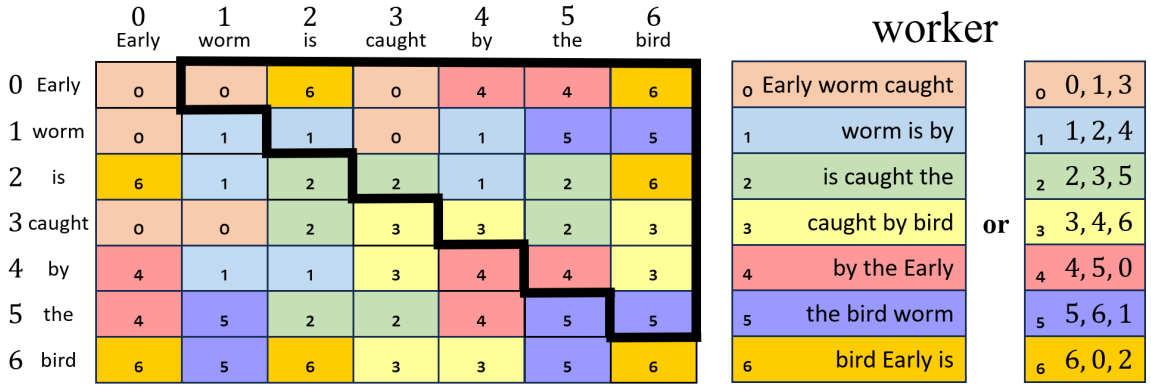

FIGURE 2. A CQS work partition of $P \in \mathbb{R}^{7\times7}$: Numbers in smaller size are worker indices. Those in larger size are token group indices, or token indices (they are different concepts but interchangeable in this example). The order of tokens in the subsequence assigned to each worker does not matter. They are ordered to display the cyclic pattern.

Finding $\mathscr{I}$ for arbitrary W is a hard problem. Researchers have been using the exhaustive search of exponential complexity [34], [38]. The search starts from the lower bound of quorum size, thus $m = \lceil\sqrt{W}\rceil$. If $\mathscr{I}$ is not found, m is increased by 1, and the search restarts. This process is repeated until an $\mathscr{I}$ is found. In [37], Bian et al. optimized the exhaustive search space by a factor of $2(W-m)^2$. They also developed a quorum construction methodology for arbitrary W without searching. It is based on factorization and $\mathscr{I}$ found for $W = 3$ to 111 in Table 1 of [34]. Nevertheless, these quorums are not cyclic but maintain all-pairs property and workload balance.

The standard self-attention computation of a sequence is bottlenecked in memory by $P = \text{softmax}(QK^T) \in \mathbb{R}^{N\times N}$, where N is the number of tokens. For instance, to parallelize the computation of P of 7 tokens with 7 workers ($N = W = 7$), thus 49 cells, a naïve way of work partition is given in Figure 1. Each worker contributes 7 cells in P, thus $\frac{1}{W}$ of all N^2 cells. Ideally, each worker only needs $\sqrt{\frac{N^2}{W}}$ tokens, thus $\frac{7}{\sqrt{7}} \approx 3$. In this scheme, however, every worker stores almost all N tokens: a row of Q and whole K to calculate P, and the whole V to calculate a row of the final output.

To find a better partition, we first prove that the partition of the strictly upper triangular matrix can be equivalent to that of the whole matrix by letting 1). the pair (i, i) on the main diagonal computed by woker i; and 2). the symmetric pair (i, j) and (j, i) computed by the same worker. In Figure 2, we show a work partition of $\frac{7\times6}{2} = 21$ cells. Each worker equally computes a minimum of 3 cells. More importantly, only 3 tokens need to be stored in the device memory instead of the whole sequence in the naïve partition scheme. Replacing tokens with their indices, a cyclic pattern can be observed (vertically). This partition achieves workload balance and a theoretical minimum amount of local tokens.

III. CQS-ATTENTION: DESIGN AND ANALYSIS

We develop a fork-join model for intense pair-wise computations called CQS module as shown in Figure 3. It consists of three parts: Scheduler, Workers, and Tiler. In this

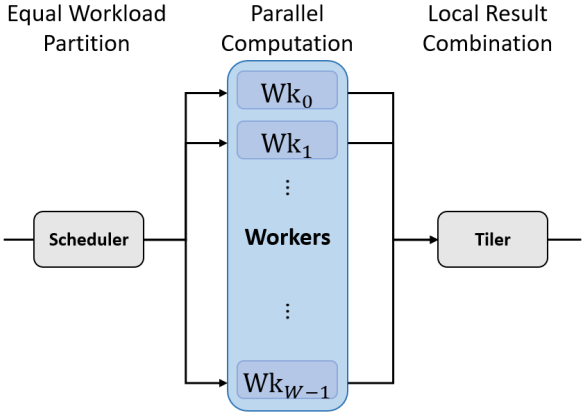

FIGURE 3. The CQS module.

study, we apply CQS module to parallelize the standard attention computation as CQS-Attention. The Scheduler decides each worker's responsibility. Workers independently compute the attention of the assigned subsequence. The Tiler combines all local results and produces the final attention. Indeed, CQS-Attention divides the problem of long-sequence attention computation into multiple independent sub-problems of shorter-sequence attention computation and these sub-problems are embarrassingly parallel.

We describe the standard attention computation as given $Q, K, V \in \mathbb{R}^{N\times d}$ of a sequence where N is the length and d is the dimension, the aim is to determine the output attention matrix $O = \text{softmax}(QK^T)V \in \mathbb{R}^{N\times d}$. The intermediate computation of $P = \text{softmax}(QK^T) \in \mathbb{R}^{N\times N}$ is the memory bottleneck. The goal of this work is to parallelize the computation of O with W worker devices while achieving the minimum single-device memory consumption.

The high-level pipeline of CQS-Attention is that, the Scheduler determines the work partition, transfers a subsequence (Q_i, K_i, V_i) and a ban list ($\texttt{BL}_i$) to worker i, and sends the Tiler a group of W material lists ($\texttt{MtrlL}$) to inform it the proper way to combine local results. Worker i computes local P_i, and nullifies those cells identified in $\texttt{BL}_i$. Then it computes a row sum vector S_i, local attention O_i, and transfers

them to the Tiler. The Tiler uses all local O_i and S_i to construct final attention O according to MtrlL sent by the Scheduler.

In the rest of this section, we elaborate design details of the Scheduler in Section III-A, the duty of Workers in Section III-B, and how the output matrix O is constructed by the Tiler in Section III-C. In addition, we analyze the performance of the CQS-Attention algorithm regarding its sacalability in single-device memory consumption and single-device computation time consumption in Section III-D.

For the simplicity of discussion, we assume there is no straggler in the system, and the communication overhead and system latency are overlooked. When discussing the computation of $P \in \mathbb{R}^{N\times N}$, the responsibility of a worker refers to cells assigned to it, while work indicates cells it actually computes. They are subtly different but interchangeable in this section. In Section A of SI, we provide the full list of terminologies mentioned in this section, their notations, and explanations. To avoid unnecessarily complex descriptions and to improve the reproducibility of all algorithms, we provide out-of-the-box Python implementations with necessary comments.

A. SCHEDULER

We mainly discuss the major challenge of work partition: $P \in \mathbb{R}^{N\times N}$, based on the CQS theory, which guarantees the workload balance and minimum single-device memory consumption. When $N \geq W$, N tokens (Tk) are divided into W token groups (TG) in an even, sequential and exclusive manner. The mapping information between TG and Tk is recorded in *TG-Tk map*. Each entry is a key-value pair (TG_idx : [Tk_idx]). The case where $N < W$ is trivial and omitted. Additionally, all pairs refer to those in the strictly upper triangular matrix **only** in the discussion of Scheduler. So the index of the first element in a pair is always smaller.

Consulting the interest set table ($\mathscr{I}$-table, provided in Section B of SI) using W for an interest set ($\mathscr{I}$), the Scheduler constructs the whole CQS. The element in each quorum is TG_idx. For most W values, total local pairs are more than all possible pairs. In other words, redundancy is inevitable in most cases. Taking advantage of the cyclic construction, the worker is able to identify pairs it is responsible for, and that too without any communication with other workers. We name this algorithm self-distillation. It decides the exclusive responsibility of each worker, and redundant TG pairs are skipped. TG_idx that only appears in redundant pairs is unnecessary, and the corresponding Tk stored in the device is a waste of memory. We apply self-distillation in the Scheduler, instead of Workers, to remove unnecessary TG_idx in CQS, or equivalently unnecessary Tk, before transferring them to Workers.

After unnecessary TG_idx are removed, CQS becomes Distilled-CQS. The length of quorum i in Distilled-CQS is denoted as m_{TGi}, and $m_{TGi} \leq m$. We retrieve the corresponding Tk_idx according to *TG-Tk map*, and obtain W new quorums. Quorum i is named a material list ($\texttt{MtrlL}_i$), whose length is denoted as m_{Tki}. $\texttt{MtrlL}_i$ is indeed a subsequence that contains m_{Tki} tokens. In this way, we construct W subsequences from the original long sequence. The Scheduler sends $Q_i, K_i, V_i \in \mathbb{R}^{m_{Tki}\times d}$ of this subsequence to worker i ($\texttt{Wk}_i$). It also sends all MtrlL to the Tiler.

The third responsibility of the Scheduler is to inform $\texttt{Wk}_i$ exact cells in P_i (computed using Q_i and K_i) it is responsible for. This list is called a task list ($\texttt{TL}_i$). The rest cells are taken care of by others, and this list is called a ban list ($\texttt{BL}_i$). Intuitively, we have $\text{len}(\texttt{TL}_i) > \text{len}(\texttt{BL}_i)$ and $\text{len}(\texttt{TL}_i) + \text{len}(\texttt{BL}_i) = m_{Tki}^2$. In practice, the Scheduler sends $\texttt{BL}_i$ to $\texttt{Wk}_i$ without generating $\texttt{TL}_i$. It will be discussed in Section III-A3.

In the end, the Scheduler transfers $\texttt{BL}_i$ and $Q_i, K_i, V_i \in \mathbb{R}^{m_{Tki}\times d}$ of a subsequence identified in $\texttt{MtrlL}_i$ to $\texttt{Wk}_i$. It also sends the full MtrlL to the Tiler for the construction of final output $O \in \mathbb{R}^{N\times d}$. In Code Block 1, we present the design of the Scheduler discussed above.

In the following paragraphs, we elaborate algorithms for *TG-Tk map* generation, self-distillation, and token index retrieval, which covers the generation of MtrlL, TL and BL.

1) *TG-TK MAP* GENERATION

Since $N \geq W$, we have $N = kW + r$, where k, r are the quotient and remainder. Grouping N tokens into W groups is even, sequential, and exclusive so that the final work partition is as balanced as possible. When N is divisible by W, $r = 0$ and each token group contains k tokens. When $r \neq 0$, we let the first $W - r$ groups have k tokens, and the remaining r groups have $k + 1$ tokens. Therefore, the maximum group size difference is 1. This difference is trivial and unavoidable when N is large. We define the map ratio as $m\_r = \frac{N}{W}$. *TG-Tk map* contains key-value pairs that record the mapping relationship between TG and Tk. The key is TG_idx, and the value is a list of, roughly m_r, Tk_idx. The *TG-Tk map* generation is presented in Code Block 2.

2) SELF-DISTILLATION

The self-distillation algorithm is named computation management logic in [36]. It states that a node can identify redundant pairs and avoid the computation by itself based on index information. The whole process does not involve any communication with other workers. In this work, we name it self-distillation. The proof is given in [36]. While this algorithm is beautifully designed, it assumes that, in our context, all token groups of a quorum reside in the corresponding worker's memory already. However, token groups that only appear in redundant pairs are not used for any computation. Therefore, we remove unnecessary TG_idx in CQS before transferring data from the Scheduler.

Self-distillation is employed in the Scheduler to avoid unnecessary data transfer. In Code Block 3, we provide a Python implementation of this algorithm. Inputs are the number of workers (W), the index of current workers (Wk_idx), and an undistilled TG_idx pair list. The output of the algorithm, a distilled list, identifies a worker's unique

TABLE 1. Pair list and CQS before and after self-distillation.

Wk_idx	CQS	Undistilled Pair List	Distilled Pair List	Distilled-CQS
0	$[0,1,2]$	$[(0,1),(0,2),(1,2)]$	$[(0,1),(0,2)]$	$[0,1,2]$
1	$[1,2,3]$	$[(1,2),(1,3),(2,3)]$	$[(1,2),(1,3)]$	$[1,2,3]$
2	$[2,3,0]$	$[(2,3),(0,2),(0,3)]$	$[(2,3)]$	$[2,3]$
3	$[3,0,1]$	$[(0,3),(1,3),(0,1)]$	$[(0,3)]$	$[0,3]$

responsibility and guarantees any of its *TG_idx* appears in at least one pair.

In Table 1, we provide the pair list of each worker before and after self-distillation in the case where $W = 4$. The undistilled pair list is generated from the corresponding quorum. Only *TG_idx* pairs in the strictly upper matrix are shown for simplicity. Pairs are required to be generated in a consistent order in this index-based algorithm, e.g., lexicographic order. After removing unnecessary *TG_idx* in each quorum, they become a Distilled-CQS. E,g., for $\texttt{Wk}_2$, *TG_idx* pair $(0, 2)$ and $(0, 3)$ are removed from the undistilled pair list because they are taken care of by $\texttt{Wk}_0$ and $\texttt{Wk}_3$ respectively. Hence, tokens in $\texttt{TG}_0$ need not be transferred, and 0 is removed from the quorum. The length of the new quorum in Distilled-CQS is denoted as m_{TGi} and $m_{TG2} = 2$.

3) TOKEN INDEX RETRIEVAL

For the computation of $P \in \mathbb{R}^{N\times N}$, CQS-Attention is able to partition worker's responsibility at cell level. We convert *TG_idx* and *TG_idx* pairs to the corresponding *Tk_idx* and *Tk_idx* pairs according to *TG-Tk map*. Since a *Tk_idx* pair is indeed a cell location in P, we restore the scale of all pairs to the whole $N \times N$ matrix. Two retrieval rules (R.r.) are stated as follows.

R.r.1: *TG_idx* $\rightarrow$ *TG-Tk map*[*TG_idx*]

R.r.2: (*TG_idx*0, *TG_idx*1) $\rightarrow$ [(*Tk_idx*0, *Tk_idx*1), (*Tk_idx*1, *Tk_idx*0) for *Tk_idx*0 in *TG-Tk map*[*TG_idx*0] for *Tk_idx*1 in *TG-Tk map*[*TG_idx*1]] (distinct)

Retrieval rules are written in a Python-like format. R.r.1 states that, for a *TG_idx*, replace it with the corresponding list of *Tk_idx* in *TG-Tk map*. R.r.2 states that, for a *TG_idx* pair, replace it with distinct *Tk_idx* pairs where *Tk_idx* comes from each *TG_idx*. Viewing *TG_idx* as a set of *Tk_idx*, this *Tk_idx* pair list (or set) is indeed the Cartesian product of *TG_idx*0 and *TG_idx*1. Retrieval rules are mainly for material ($\texttt{MtrlL}$) and task list ($\texttt{TL}$) generation.

a: MATERIAL LIST GENERATION

Material lists ($\texttt{MtrlL}$) are determined by retrieving *Tk_idx* in Distilled-CQS. Since no pair is involved, *TG_idx* is directly replaced with its corresponding *Tk_idx*, according to R.r.1. A material list ($\texttt{MtrlL}_i$) is indeed a subsequence, whose length is denoted as m_{Tki} and $m_{Tki} \approx m_{TGi} \times m\_r = \frac{m_{TGi}}{W} \times N$. The implementation is given in Code Block 1.

b: TASK/BAN LIST GENERATION

Retrieving *Tk_idx* in distilled pair lists using R.r.2, we determine a task list ($\texttt{TL}_i$) containing all the responsible cells of a worker in P. Since $\texttt{Wk}_i$ is given a subsequence whose length is m_{Tki}, it computes $P_i \in \mathbb{R}^{m_{Tki}\times m_{Tki}}$. Those cells out of $\texttt{TL}_i$ is a ban list ($\texttt{BL}_i$), and they should be masked to secure the mathematical correctness. Since $\text{len}(\texttt{BL}_i) < \text{len}(\texttt{TL}_i)$, sending a ban list is more efficient. We provide a function that generates a ban list directly from distilled pairs in Code Block 4. Task list generation is also provided for validation purposes.

B. WORKERS

Each worker ($\texttt{Wk}_i$) executes the following computations: $P_i = \text{mask}(\exp(Q_i K_i^T))$, $S_i = \text{row\_sum}(P_i)$, $O_i = P_i V_i$, where $P_i \in \mathbb{R}^{m_{Tki}\times m_{Tki}}$, $S_i \in \mathbb{R}^{m_{Tki}\times 1}$, $Q_i, K_i, V_i, O_i \in \mathbb{R}^{m_{Tki}\times d}$. P_i first computes the element-wise exponential of $Q_i K_i^T$ product as the nominator of the softmax function. Then we apply a mask that zeros cells of P_i designated in the ban list ($\texttt{BL}_i$). S_i computes the row sum of P_i, and each element is part of the exponential sum in the row *Tk_idx* of $P \in \mathbb{R}^{N\times N}$. Eventually, only S_i and O_i are transferred to the Tiler. Code Block 5 depicts the complete local computations performed by each worker.

Communication is not required among workers because their responsibilities are entirely independent. Therefore, CQS-Attention can process an infinitely long sequence by simply dividing it into shorter subsequences and adding more worker devices to the system without increasing communication complexity: because there is no communication among workers!

C. TILER

The output attention $O \in \mathbb{R}^{N\times d}$ is initialized as a zero matrix ($0_{N,d}$). The Tiler adds each row of local O_i to the row *Tk_idx* of O according to $MtrlL_i$ sent by the Scheduler. A zero vector $S = 0_{N,1}$ is also created to store the exponential sum of each row in P (i.e. S_i) as the denominator of the softmax function. Similarly, the Tiler adds each element of local S_i to the row *Tk_idx* of S. Finally, each row of O is element-wisely divided by the value in the same row of S. The implementation of Tiler is given in Code Block 6.

D. PERFORMANCE ANALYSIS

We analyze the performance of the CQS-Attention algorithm regarding the scalability in single-device memory consumption with the sequence length (N) and single-device computation time consumption with the number of workers (W).

1) MEMORY CONSUMPTION

We define the overall memory consumption as the total number of cells of all matrices, including intermediate ones. We compute the Overall Memory Consumption Ratio (OMCR) as $\frac{m_{Tk}^2+(4d+1)m_{Tk}}{N^2+4dN}$. Since $m_{Tk} \approx m \times m\_r$ and $m\_r = \frac{N}{W}$, we have $m_{Tk} \approx N \times \frac{m}{W}$. Therefore, OMCR $\approx \frac{\frac{m^2}{W^2}N+\frac{(d+1)m}{W}}{N+d} \approx \frac{m^2}{W^2}$ when $N \rightarrow \infty$. This ratio agrees with the basic understanding: CQS-Attention divides the whole

sequence into W shorter sequences. Specifically, N tokens are divided into W groups and m of them (a subsequence) are assigned to each of W workers. Since the quadratic standard attention is computed, the memory consumption ratio is $\frac{m^2}{W^2}$.

The theoretical Lower Bound (LB) of quorum size m is $\sqrt{W}$ [34]. In Figure 4 (left), we plot the actual ratio of length between the subsequence and the whole sequence for the number of workers (W) corresponding to existing interest sets, i.e. $3 \leq W \leq 111$. Since $\text{LB}(m) = \sqrt{W}$ and it must be an integer, the lower bound is indeed $\lceil\sqrt{W}\rceil$. Three curves align with each other asymptotically. Therefore, it is safe to replace m with $\sqrt{W}$. Hence, OMCR $\approx \frac{1}{W}$. In fact, it is also the peak memory consumption ratio because the peak memory consumption is dominated by the quadratic matrix when N is large. Therefore, we use an asymptotic Memory Consumption Ratio (asym-MCR) as a unified metric in the memory discussion. We plot the actual value of $\frac{m^2}{W^2}$ and the theoretical asym-MCR in Figure 4 (right). This ratio suggests that scaling the sequence length to infinity only requires a simple stack of worker devices. In other words, adopting CQS-Attention, the standard attention computation of an infinitely long sequence becomes a resource problem, instead of a technical problem.

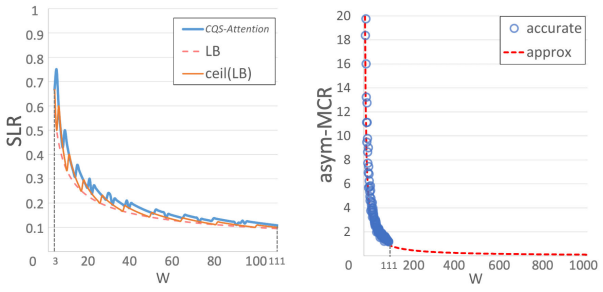

FIGURE 4. Using the size of existing interest sets (m) for feasibility check of approximating m with $\sqrt{W}$. Left: Sequence length ratio (SLR) between the subsequence (m token groups) and the whole sequence (W token groups); Right: Asymptotic memory consumption ratio (asym-MCR) with respect to the number of workers.

2) COMPUTATION TIME

The speedup of CQS-Attention is the wall-clock time ratio between the single-device (SD) and CQS implementation: $\frac{t_{SD}+t_L}{t_{Stgl}+t_O+t_L}$, where the wall-clock time of CQS-Attention considers the computation time of the straggler (t_{Stgl}), system overhead (t_O) and latency (t_L). Since t_O and t_L could vary significantly under different system configurations and test scenarios, e.g., the network overhead can dominate in a globally distributed system setup, we only focus on the computation time improvement in this work to demonstrate the benefit of shrinking the sequence length, which is the essence of CQS-Attention algorithm. Therefore, we use Computation Time Ratio (CTR $= \frac{t_{SD}}{t_{Stgl}}$). The wall-clock time of computation is obtained by profiling the actual hardware. In Section IV-B, we present a series of CTR analysis on NVIDIA A100 GPU.

IV. A DEMONSTRATION OF WORKFLOW AND SIMULATIONS

We unfold the entire pipeline of CQS-Attention in Section IV-A with a toy example where $N = 10, d = 1$ and $W = 7$. In Section IV-B, we adopt NVIDIA A100 GPU as an example of worker device and demonstrate the advantage of CQS-Attention including single-device computation time, peak memory consumption, etc.

A. THE WORKFLOW

We demonstrate the workflow of CQS-Attention using a simple case where $N = 10, d = 1, W = 7$ in Figure 5. We also provide the same workflow figure with *Wk_idx* being the discriminative feature instead of color in Section C of SI. The Scheduler generates *TG-Tk map* recording the mapping between 7 token groups and 10 tokens. It consults $\mathscr{I}$-table for an interest set for $W = 7$, i.e. $\mathscr{I} = [0, 1, 3]$, and constructs the whole CQS. Applying self-distillation, there is no redundancy detected.

The Scheduler generates a ban list ($\texttt{BL}_i$) for each worker. E.g., for $\texttt{Wk}_4$, the distilled *TG_idx* pair list is $[(4, 5), (0, 4), (0, 5)]$. The missing pairs in the upper triangular matrix are $[(0, 0), (4, 4), (5, 5)]$. Since $\texttt{Wk}_4$ must be responsible for $(4, 4)$, we retrieve all *Tk_idx* pairs for the other two *TG_idx* pairs. Thus, $(0, 0) \rightarrow [(0, 0)]$ and $(5, 5) \rightarrow [(6, 6), (6, 7), (7, 6), (7, 7)]$ given $0 : [0]$ and $5 : [6, 7]$ in *TG-Tk map*. For the convenience of local computation, we re-index the original *Tk_idx* in $\texttt{BL}_4$. The index mapping depends on its material list ($\texttt{MtrlL}_i$). E.g., $\texttt{MtrlL}_4$ is obtained by replacing each *TG_idx* in the distilled quorum $[4, 5, 0]$ with corresponding *Tk_idx* in *TG-Tk map*. Thus, $4 : [4, 5]$, $5 : [6, 7]$, $0 : [0]$, and $\texttt{MtrlL}_4 = [0, 4, 5, 6, 7]$. It is sorted in this example only for demonstration purposes. We map each *Tk_idx* in $\texttt{MtrlL}_4$ to consecutive integers from 0. Thus, 0, 6, 7 are mapped to 0, 3, 4, and $\texttt{BL}_4$ in local indices is $[(0, 0), (3, 3), (3, 4), (4, 3), (4, 4)]$. Finally, the Scheduler sends 7 material lists ($\texttt{MtrlL}$) to the Tiler, and transfers the corresponding subsequence (Q_i, K_i, V_i) to each worker. It also sends a ban list ($\texttt{BL}_i$) to each worker to mask P_i.

Each worker first computes $P_i = Q_i K_i^T \in \mathbb{R}^{m_{Tki} \times m_{Tki}}$, where $m_{Tki} \in \{3, 4, 5\}$. Masking and exponentiation can be implemented in either order. The only difference is that if masking is executed first, the masked cells are set to $-\inf$, so that the exponentiation later sets them to zero. Vector S_i records the sum of each row in P_i for the denominator computation of the softmax function in the Tiler. Finally, $\texttt{Wk}_i$ computes the local attention O_i and transfers it with S_i to the Tiler.

$\texttt{MtrlL}$ logs each of 7 subsequences and index of the corresponding worker. Local attention (O_i) is a part of the final attention of assigned tokens. The Tiler adds each row of O_i to the row *Tk_idx* in O. E.g., $\texttt{Wk}_0$ is assigned token 0, 1, 3. Each row of $O_0 \in \mathbb{R}^{3\times 1}$ is added to row 0, 1, 3 of the final output attention $O \in \mathbb{R}^{10\times 1}$. Next, each row of O is scaled by the value in the same row of S, which is determined

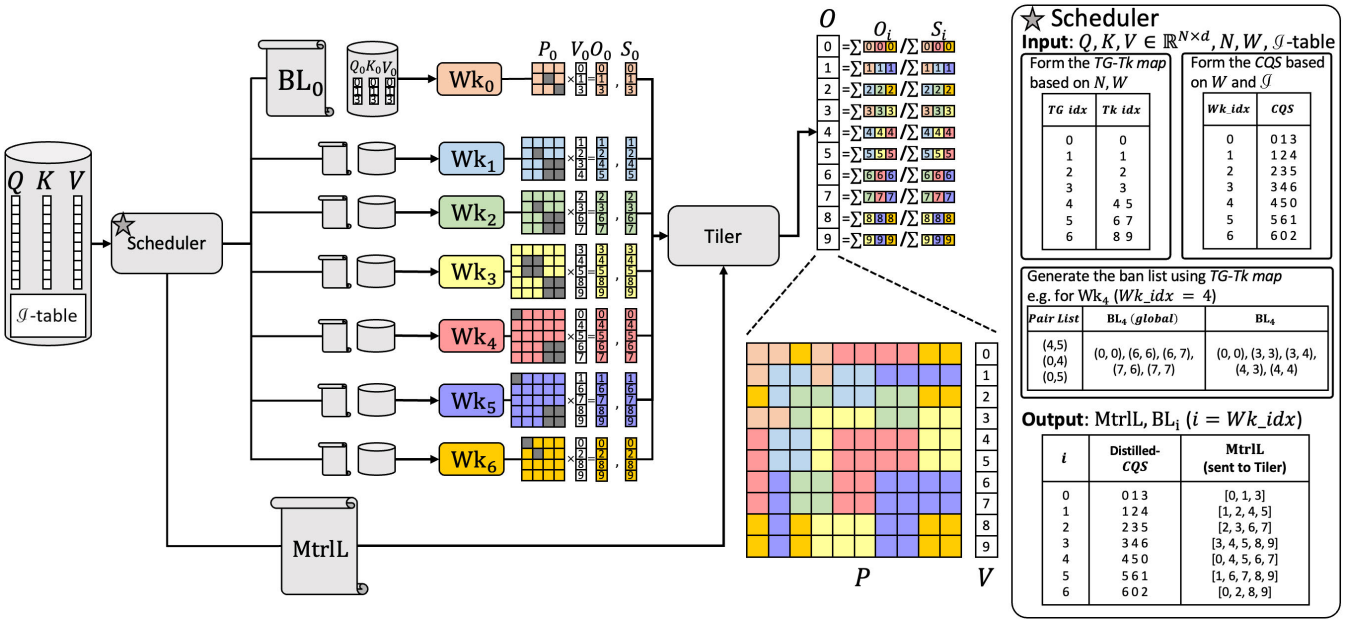

FIGURE 5. Workflow of CQS-attention for $N = 10, d = 1, W = 7, \mathscr{J} = [0, 1, 3]$: All numbers in the workflow are *Tk_idx*. The cylinder represents data such as Q, K, V of the (sub)sequence, and $\mathscr{J}$-table. The scroll represents a ban list ($\texttt{BL}_i$) or material lists ($\texttt{MtrlL}$). Local computations include P_i, S_i, O_i, but P_i is not sent to the tiler. Dark cells of P_i identified by $\texttt{BL}_i$ are zeroed. P, V are given here only for validation purpose.

TABLE 2. The maximum subsequence length (m_{Tk}), average wall-clock time (t_{wc}, in ms), and average peak memory consumption (PMC, in GB) when $W = 1, 4, 7, 8$ and 31: Each computation was performed 5 times.

N	W_1			W_4			W_7			W_8			W_31		
	m_{Tk}	t_{wc}	PMC	m_{Tk}	t_{wc}	PMC	m_{Tk}	t_{wc}	PMC	m_{Tk}	t_{wc}	PMC	m_{Tk}	t_{wc}	PMC
10K	10K	2.67	3.45	7.5K	2.29	1.95	4 287	1.32	0.65	5K	1.41	0.88	1 937	1.01	0.144
20K	20K	7.85	13.7	15K	5.01	7.71	8 572	2.64	2.54	10K	2.67	3.44	3 873	1.22	0.532
30K	30K	16.07	30.7	225K	9.63	17.3	12 858	4.04	5.68	15K	4.97	7.72	5 808	1.58	1.18
40K	40K	27.96	54.6	30K	16.07	30.7	17 144	6.08	10.1	20K	7.85	13.7	7 744	2.34	2.08
45K	45K	839.74	69.1	33 750	19.90	38.9	19 287	7.37	12.8	22.5K	9.57	17.3	8 712	2.75	2.62
46K	46K	894.64	72.2	34.5K	20.76	40.6	19 715	7.68	13.3	23K	10.02	18.1	8 904	2.82	2.74
47K	47K	914.42	75.3	35 250	21.86	42.4	20 144	7.94	13.9	23.5K	10.36	18.9	9 099	2.38	2.86
48K	48K	956.57	78.6	36K	22.40	44.2	20 572	8.23	14.5	24K	10.72	19.7	9 292	2.42	2.98
49K	49K	1 016.07	81.9	36 750	23.31	46.1	21K	8.52	15.1	24.5K	11.23	20.5	9 486	2.49	3.11

by summing S_i in the same manner as in O. We show the exact partition of $P \in \mathbb{R}^{10\times10}$ for readers to validate the independence of the work partition and, more importantly, computation correctness. Overall, each worker independently computes the attention of a subsequence, and local results are put together by the Tiler.

B. THE PERFORMANCE

We carry out a series of experiments on AMD EPYC 7502 (as the Scheduler and the Tiler) and a single piece of NVIDIA A100 GPU (as Workers). The GPU memory is 80 GiB ($\approx$ 85.89 GB). We record the average wall-clock time and active GPU memory timeline of standard attention computation configuring W and N to multiple values. The implementation of attention computation is borrowed from `scaled_dot_product_attention()` by PyTorch (version 2.1.0) but it is not directly called in our experiments to avoid built-in optimizations. Other setups include CUDA (version 11.8) and cuBLAS (version 11.11.3.6).

Taking advantage of the independent workload partition of CQS-Attention, we are able to simulate multiple GPU workers using a single piece of actual hardware. A subsequence (Q_i, K_i, V_i) and a ban list ($\texttt{BL}_i$) are sent to the GPU and local results (O_i, S_i) are sent back to the CPU after necessary computations. Then the GPU memory is flushed to simulate another worker device. This process repeats for W times.

With embedding dimension (d) fixed to 64 and precision set to FP16, we manage to fit at most 49 798 tokens on A100. Therefore, we decide to set N to multiple values between 10K and 49K. We compare the performance with W configured to 1, 4, 7, 8, 31. In Table 2, we list the length of the longest subsequence a worker receives (the straggler), the average all-clock time of computation, and the average peak memory consumption.

We plot the average wall-clock time in each experiment setup and the corresponding performance curve, regarding the computation time, of A100 GPU with the sequence length in Figure 6. A leap in computation time is observed when N increases from 40K to 45K, which is consistent with

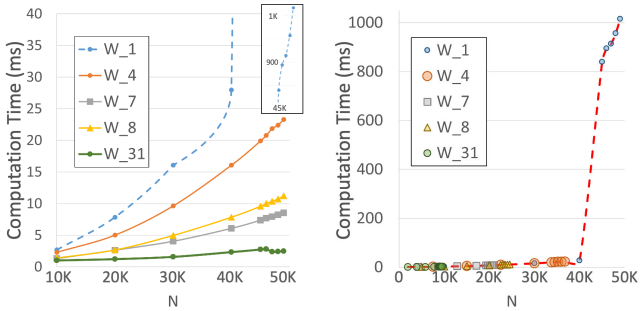

FIGURE 6. Left: The average wall-clock time of single-device computations; Right: The performance curve for A100 GPU.

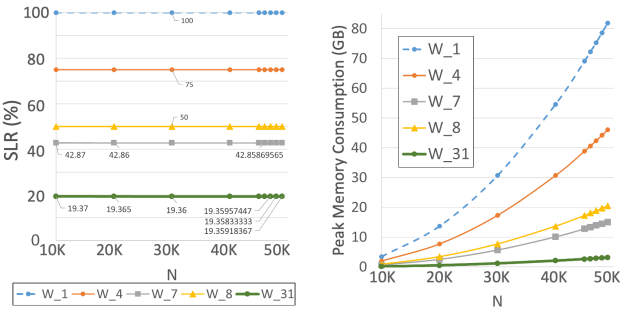

FIGURE 7. Left: The sequence length ratio (SLR); Right: The peak memory consumption (PMC), averaged.

the general trend described in Section 2.3 of the *User's Guide* by NVIDIA. Due to the non-linear performance curve, one very interesting observation is that the computation time (i.e. CTR) for 49K tokens are 43.6×, 119.3×, 90.5×, and 408.1× faster provided 4, 7, 8, and 31 workers. Nevertheless, these astounding numbers will not be achieved once local wall-clock time becomes math-limited instead of memory-limited when m_{Tk} or N is large enough, especially when scaling up to infinity. Therefore, we suggest not to fully load a single device in memory in exchange for significantly less computation time.

In Section D of SI, we provide active memory timelines of A100 for $W = 1$ and 31 in all sequence length scenarios. In Figure 7, we plot the Sequence Length Ratio (SLR $= \frac{m_{Tk}}{N}$) and the Peak Memory Consumption (PMC) of the straggler. Subtle fluctuations of SLR for $W = 7$ and 31 are because the whole sequence length (N) is indivisible by W. Nevertheless, SLR is mathematically stable as $\frac{1}{\sqrt{W}} + \epsilon$ regardless of N. Intuitively, more worker devices grant a lower single-device memory consumption and computation time since the subsequence is shorter. However, it is not always true with CQS-Attention such as $W = 7 vs. 8$ and $W = 31 vs. 32$ (not given here) due to the inevitable longer subsequence to maintain the all-pairs property which also introduces redundancy. For example, the interest set size (m) for $W = 7$ and 8 are 3 and 4 respectively, which makes the subsequence length $\frac{3}{7}$ and $\frac{4}{8}$ of the whole sequence. Thus, the subsequence is shorter given 7 workers!

The actual memory consumption in our simulations is consistent with the analysis in Section III-D1. Since the standard attention was calculated, PMC is quadratic to the subsequence length. Therefore, CQS-Attention is mathematically stable to N and highly scalable to W.

V. DISCUSSION

A. ADVANTAGES

Beyond the great performance in single-device memory consumption, computation time ratio, mathematical stability, and scalability, we highlight some other advantages of CQS-Attention as a distributed algorithm. The **independent workload partition** is the very origin of these advantages because it makes local computations independent.

1) SIMPLICITY

The architecture design of CQS-Attention is simple fork-join. When functions of the Scheduler and Tiler are performed by the same device, it becomes a centralized model. Therefore, it enjoys all the corresponding advantages such as simplified management and control.

In the rest of this section, our discussion is under the assumption of the centralized architecture.

2) ROBUSTNESS

Suppose x worker devices are down and a prompt restart is not possible; the central device can send the task to those active workers only at the cost of additional communication and computation time. Correctness is always guaranteed without introducing complex data movements as there is no data dependency. More importantly, the range of x this system can tolerate is from 0 to $W - 1$. Thus, in an extreme case where only one worker is active, it is still able to accomplish the whole computation without the Out-Of-Memory (OOM) issue. The cost is $\times(W - 1)$ more computation time and additional communication time with the central device.

3) COMPATIBILITY

CQS-Attention can be recognized as a higher level of parallelism. It is fully compatible with all optimizations because it indeed divides the whole problem of attention computation into W independent and, consequently, embarrassingly parallel sub-problems. Any optimization, including CQS-Attention itself, can be independently applied to each sub-problem. Therefore, CQS-Attention can scale the sequence length to infinity by dividing the problem into ample "small" sub-problems that each fit into a single worker device.

4) FLEXIBILITY

When dividing into W sub-problem causes OOM problem among W worker devices, CQS-Attention is flexible to resolve the issue. When OOM only happens to limited workers, we can distribute those unfinished sub-problems among capable workers. The drawback is idle workers. When OOM happens to a significant amount of workers, we re-divide the whole problem into $\tilde{W}$ sub-problems ($\tilde{W} > W$)

TABLE 3. Comparions with selected distributed algorithms for attention computation: Optimization types include approximate attention (AA), FlashAttention (FA), and sequence parallelism (SP). Sequence length ratio (SLR) and # Time Step (in the forward pass) are presented with regard to the number of worker devices (W) and, if applicable, the quorum size (m). A single time step in each method may not be the same. Hence, the number of time steps counts the "chunk" of operations that need to be executed in series. The better performance is underlined.

Method	Space Complexity	Type	Limitation	Communication	Workload Balance	SLR	# Time Step
Normalized Attention	Linear	AA, SP	Loss of chuck-wise contextual information. High design complexity.	Yes	Yes	N.A.	N.A.
DistFlashAttn	Depends	FA, SP	High communication overhead. Recomputation required.	Yes	Yes	$\frac{1}{W}$	$(\frac{W}{2}, W)$
RSA	Depends	SP	Quadratic work dependency. Low device utilization. Many times of communication.	Yes	No	$\frac{1}{W}$	W
BurstAttention	Quadratic	FA, SP	Complicated optimization mechanisms (e.g. global and local attention optimization). Recomputation required.	Yes	Yes	$\frac{1}{W}$	W
CQS-Attention	Depends	SP	Longer subsequence. Interest set searching is hard.	No	Yes	$\frac{m}{W}$	1

so that each fits. Some workers receive more than 1 sub-problem.

B. LIMITATIONS

We identify two obstacles that potentially prevent CQS-Attention from being adopted on a large scale in the current stage.

1) CENTRALIZED DESIGN

CQS-Attention suffers from most drawbacks as a centralized model i.e. the central node is extremely critical. However, the communication bandwidth limit and storage challenge can be easily resolved by constructing the central node using a group of devices such as in Figure 8, where $W = 7$, $\mathscr{I} = [0, 1, 3]$. Each device in the central node only stores a token group, whose corresponding number of tokens is roughly $\frac{N}{W}$. 3 token groups consist of a subsequence and are sent to a worker device as scheduled in the CQS. To mitigate the intrinsic drawbacks of a centralized model, one can introduce fault-tolerant mechanisms such as single redundancy and one-pipelining [39]. Overall, there is plenty of room for improving the implementation of CQS-Attention in multiple aspects, and it will be one of our works in the future.

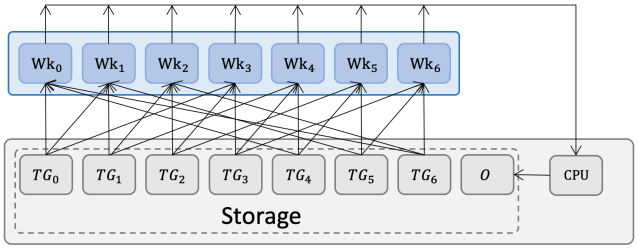

FIGURE 8. A centralized implementation where the central device (function as both the Scheduler and Tiler) has distributed storage.

2) SEARCHING FOR INTEREST SETS

Computation is never the challenge for the central node because it only computes the workload distribution (as the Scheduler) and combines all local results (as the Tiler) according to an interest set ($\mathscr{I}$). However, searching for interest sets itself is a very hard problem, and knowing an interest set in advance is indeed a very strong assumption in this work. Currently, an $\mathscr{I}$ for $W = 3$ to 111 is provided in [34]. There is no $\mathscr{I}$ provided for $W > 111$ because the search is purely exhaustive with little intellectual challenge. In [37], Bian et al. developed a quorum construction algorithm based on existing $\mathscr{I}$ and factorization. However, quorums are not cyclic and self-distillation should not work. Quorum size is not minimum either. Both drawbacks may result in inevitable redundancy among sub-problems.

The major limitation of the algorithm presentation in this work is the compromise in the experiment design for performance demonstration in Section IV-B: We only simulate using a single piece of GPU rather than (up to) 31 pieces, and we had to overlook the communication cost. As a result, we are only able to show the computation time ratio instead of speedup. Additionally, we have not implemented CQS-Attention into an LLM and this will be the focus of our future work.

C. COMPARISONS

Although CQS-Attention is universally compatible with any optimization, in Table 3, we compare it, on the high level, with multiple distributed algorithms for attention computation: normalized attention [40] inherited from Moving average Equipped Gated Attention (MEGA) [41], DistFlashAttn [28], Ring Self-Attention (RSA) [29] and BurstAttention [42]. Regarding the space complexity, normalized attention uses sparse attention (linear) and BurstAttention optimizes the standard attention (quadratic) computation. On the other hand, the space complexity of DistFlashAttn, RSA, and CQS-Attention depends on the attention choice of local computation. Thus, they enjoy the flexibility of the attention mechanism.

Since interest sets ($\mathscr{I}$) are not unique for a given W, we also compare the SLR among different $\mathscr{I}$.

```python
import numpy as np
from random import choice
import MODULE_utils as utils
# The interest set table is implemented in MODULE_utils.
    py available at https://github.com/CQS-Attention/
    CQS_Attention/blob/main/src/MODULE_utils.py

def gen_CQS(W, interest_set):
    CQS = []
    CQS.append(interest_set)
    for i in range (1, W):
        tmp_set = []
        for ele in interest_set:
            tmp = (ele + i) % W
            tmp_set.append(tmp)
        CQS.append(tmp_set)
    return CQS

def gen_Distilled_CQS(W, CQS):
    def gen_pair_lists_before_distillation(W, CQS):
        pair_lists_before_distill = []
        for Wk_i in range(W):
            all_perm = utils.all_pairs_strict_upper(CQS[
    Wk_i])
            pair_lists_before_distill.append(all_perm)
        return pair_lists_before_distill
    def gen_distilled_pair_lists(W,
    pair_lists_before_distill):
        distilled_pair_lists = []
        for Wk_idx in range(W):
            # See Code Block 3 for Self-Disstillation()
            distilled_pair_lst = Self-Disstillation(W,
    Wk_idx, pair_lists_before_distill[Wk_idx])
            distilled_pair_lists.append(
    distilled_pair_lst)
        return distilled_pair_lists

    pair_lists_before_distill =
    gen_pair_lists_before_distillation(CQS)
    distilled_pair_lists = gen_distilled_pair_lists(
    pair_lists_before_distill)

    Distilled_CQS = []
    for pair_list in distilled_pair_lists:
        mtrlL_i_in_TG = set()
        for tk1, tk2 in pair_list:
            mtrlL_i_in_TG.add(tk1)
            mtrlL_i_in_TG.add(tk2)
        # Sort is only necessary to the cyclic pattern
        # Distilled_CQS.append(sorted(list(mtrlL_i_in_TG)
    ))
        # Otherwise, no need to sort
        Distilled_CQS.append(list(mtrlL_i_in_TG))
    return Distilled_CQS, distilled_pair_lists

# Material list generation
def gen_MtrlL(W, N, TG_Tk_map, Distilled_CQS):
    if W == N:
        MtrlL = Distilled_CQS
    else:
        MtrlL = []
        for distilled_quorum in Distilled_CQS:
            MtrlL_i = []
            for ele in distilled_quorum:
                MtrlL_i += TG_Tk_map[ele]
            MtrlL.append(MtrlL_i.copy())
    return MtrlL

def gen_BL(W, Distilled_CQS, distilled_pair_lists,
    TG_Tk_map, MtrlL):
    BL = []
    for i in range(W):
        BL_i_global = []
        material_list_in_TG_for_ban_list = Distilled_CQS[
    i].copy()
        all_TG_pair_for_ban_list = set(utils.
    all_pairs_in_upper(material_list_in_TG_for_ban_list))
        all_TG_pair_for_ban_list =
    all_TG_pair_for_ban_list - set(distilled_pair_lists[i
    ]) - {(i,i)}
        for TG1, TG2 in all_TG_pair_for_ban_list:
            BL_i_global += utils.all_inter_pairs(
    TG_Tk_map[TG1], TG_Tk_map[TG2], TG1 != TG2)
        BL_i = []
        for tk1, tk2 in BL_i_global:
            BL_i.append((MtrlL[i].index(tk1),MtrlL[i].
    index(tk2)))
        BL.append(BL_i.copy())
    return BL

def Scheduler(Q, K, V, W):
    N, d = Q.shape
    interest_set = choice(utils.All_interst_sets(W))
    m = len(interest_set)
    CQS = gen_CQS(W, interest_set)
    TG_Tk_map = gen_TK_Tk_map(N, W)
    # See Code Block 2 for gen_Distilled_CQS()
    Distilled_CQS, distilled_pair_lists =
    gen_Distilled_CQS(W, CQS)
    MtrlL = gen_MtrlL(W, N, TG_Tk_map, Distilled_CQS)
    BLi_all = gen_BL(W, Distilled_CQS,
    distilled_pair_lists, TG_Tk_map, MtrlL)
    Qi_all = [Q[MtrlL[Wk_i],:] for Wk_i in range(W)]
    Ki_all = [K[MtrlL[Wk_i],:] for Wk_i in range(W)]
    Vi_all = [V[MtrlL[Wk_i],:] for Wk_i in range(W)]
    # MtrlL is for the Tiler
    # BLi, Qi, Ki, Vi are for each worker
    return MtrlL, BLi_all, Qi_all, Ki_all, Vi_all
```

CODE BLOCK 1. Scheduler().

CODE BLOCK 1. *(Continued.)* Scheduler().

```python
def gen_TG_Tk_map(N, W):
    TG_Tk_map = {}
    k, r = divmod(N, W)
    bound = W-r
    for i in range(W):
        # First W-r TG contains k Tk
        if i < bound:
            TG_Tk_map[i] = [_ for _ in range(i*k, i*k+k)]
        # Rest r TG contains k+1 Tk
        else:
            TG_Tk_map[i] = [_ for _ in range(i*k+i-bound,
      i*k+i-bound+q+1)]
    return TG_Tk_map
```

CODE BLOCK 2. Gen_TG_Tk_map().

```python
def Self-Distillation(W, Wk_idx, Undistilled_Pair_List):
    Distilled_Pair_List = []
    # Record all the diff in Undistilled_Pair_List
    all_diff = []
    for each_pair in Undistilled_Pair_List:
        diff1 = (each_pair[0] - each_pair[1]) % W
        if diff1 in all_diff:
            # Redundancy.
            # Exclude current pair.
            continue
        else:
            # The inclusion is yet to be decided
            all_diff.append(diff1)
        diff2 = (each_pair[1] - each_pair[0]) % W
        if diff1 == diff2:
            # Redundancy.
            # Only if Wk_idx is smaller, include the pair
            if Wk_idx < W / 2:
                Distilled_Pair_List.append(each_pair)
        else:
            # Add diff2 to all_diff
            all_diff.append(diff2)
            # Include the pair
            Distilled_Pair_List.append(each_pair)
    return Distilled_Pair_List
```

CODE BLOCK 3. Self-distillation().

Different CQS schemes exacerbate the imbalance of workload partition along with self-distillation and indivisibility between N and W. As a rule of thumb, the SLR difference becomes trivial when $N > 20 \times W$. More details on $\mathscr{J}$ comparisons are given in Section E of SI.

```
from itertools import combinations
def all_pairs_including_diagonal(A):
    # All pairs in upper triangular including the
     diagonal
    return list(combinations(A, 2)) + [(ele, ele) for ele
      in A]

def all_inter_pairs(A, B, both_pair = False):
    res = []
    for ele1 in A:
        for ele2 in B:
            res.append((ele1, ele2))
            if both_pair: res.append((ele2, ele1))
    return res

def Gen_Ban_List(N, W, Wk_idx, Distilled_quorum,
     Distilled_pair_list):
    TG_Tk_map = gen_TG_Tk_map(N, W)
    BL_i = []
    TG_pair_for_BL = set(all_pairs_including_diagonal(
     Distilled_quorum))
    # Missing TG pairs
    TG_pair_for_BL = TG_pair_for_BL - set(
     Distilled_pair_list) - {(Wk_idx, Wk_idx)}
    for TG1, TG2 in TG_pair_for_BL:
        # Inclusion depends on if the TG pair is on
     diagonal
        BL_i += all_inter_pairs(TG_Tk_map[TG1], TG_Tk_map
     [TG2], TG1 != TG2)
    return BL_i

# For validation purpose
def Gen_Task_List(N, W, Wk_idx, Distilled_pair_list):
    TG_Tk_map = gen_TG_Tk_map(N, W)
    TL_i = []
    TL_i += all_inter_pairs(TG_Tk_map[Wk_idx], TG_Tk_map[
     Wk_idx])
    for TG1, TG2 in Distilled_pair_list:
        TL_i += all_inter_pairs(TG_Tk_map[TG1], TG_Tk_map
     [TG2], True)
    return TL_i

def validate(N, W, Distilled_quorum, BL_i, TL_i):
    TG_Tk_map = gen_TG_Tk_map(N, W)
    material_list_length = 0
    for TG in Distilled_quorum:
        material_list_length += len(TG_Tk_map[TG])
    return len(set(BL_i + TL_i)) == material_list_length
     * material_list_length
```

CODE BLOCK 4. Ban/task list generation and the validation.

VI. CONCLUSION AND FUTURE WORK

In this study, we develop a simple fork-join CQS module and modify it as CQS-Attention for standard attention computation. Due to the independent workload partition of CQS, CQS-Attention divides the whole computation problem into many isolated sub-problems and they are embarrassingly parallel. The length of the whole sequence can easily scale up to infinity by dividing into more sub-problems and each sub-problem fits into a single worker device. Additionally, the CQS-Attention is highly scalable because introducing more worker devices only adds minimum communication cost with the Scheduler/Tiler since it is completely communication-free among workers. The scalability of CQS-Attention can be further improved by local computation with lower complexity.

This work mainly focuses on the correctness proof, workflow demonstration and theoretical performance analysis of adopting CQS for standard attention computation. Our next step is applying CQS-Attention to optimize LLMs to demonstrate their superiority and new potential. CQS-Attention is a substantiation of CQS module. In fact, CQS module can parallelize general matrix multiplication ($C = AB \in \mathbb{R}^{N_1 \times N_2}, N_1 \neq N_2$). This topic is concisely covered with an example in Section F of SI. We will also explore more potential of CQS module to boost the computation and relax memory constraints for many important problems such as protein structure prediction in Computational Biology, many-body problems in Quantum Physics, chaotic n-body simulation in Astrophysics, etc.

```
import numpy as np
def Worker(Qi, Ki, Vi, BLi):
    # Receive Qi, Ki, Vi and BLi from the Scheduler
    Pi = Qi @ Ki.T
    for x, y in BLi:
        Pi[x, y] = -np.inf
    Pi = np.exp(Pi)
    Oi = Pi @ Vi
    Si = np.sum(Pi, axis = 1)
    return Oi, Si
```

CODE BLOCK 5. Worker().

```
import numpy as np
def Tiler(N, d, MtrlL, Oi_all, Si_all):
    # Receive MtrlL from the Scheduler
    # Receive Oi and Si from each worker
    W = len(MtrlL) # or len(Oi_all), len(Si_all)
    O = np.zeros((self.N, self.d))
    S = np.zeros(self.N)
    for Wk_i in range(W):
        O[self.MtrlL[Wk_i], :] += self.Oi_all[Wk_i]
        S[self.MtrlL[Wk_i]] += self.Si_all[Wk_i]
    return O / S[:, None]
```

CODE BLOCK 6. Tiler().

APPENDIX
CODE BLOCKS

See Code Blocks 1–6.

ACKNOWLEDGMENT

This work was accomplished during Yiming Bian's Ph.D. program at Iowa State University. We would like to thank all reviewers for their valuable and insightful comments and constructive feedback.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1–11.

[2] M. N. Rabe and C. Staats, "Self-attention does not need $O(n^2)$ memory," 2021, *arXiv:2112.05682*.

[3] X. Wang, M. Salmani, P. Omidi, X. Ren, M. Rezagholizadeh, and A. Eshaghi, "Beyond the limits: A survey of techniques to extend the context length in large language models," 2024, *arXiv:2402.02244*.

[4] Y. Zhou, U. Alon, X. Chen, X. Wang, R. Agarwal, and D. Zhou, "Transformers can achieve length generalization but not robustly," 2024, *arXiv:2402.09371*.

[5] J. Rae, J. J. Hunt, I. Danihelka, T. Harley, A. W. Senior, G. Wayne, A. Graves, and T. Lillicrap, "Scaling memory-augmented neural networks with sparse reads and writes," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 1–12.

[6] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," 2019, *arXiv:1904.10509*.

[7] S. Sukhbaatar, E. Grave, P. Bojanowski, and A. Joulin, "Adaptive attention span in transformers," 2019, *arXiv:1905.07799*.

[8] G. Lample, A. Sablayrolles, M. Ranzato, L. Denoyer, and H. Jégou, "Large memory layers with product keys," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–13.

[9] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 17283–17297.
[10] Y. Gao, Z. Song, X. Yang, and R. Zhang, "Fast quantum algorithm for attention computation," 2023, *arXiv:2307.08045*.
[11] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 2020, *arXiv:2004.05150*.
[12] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," 2020, *arXiv:2001.04451*.
[13] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2021, pp. 10012–10022.
[14] X. Chu, Z. Tian, Y. Wang, B. Zhang, H. Ren, X. Wei, H. Xia, and C. Shen, "Twins: Revisiting the design of spatial attention in vision transformers," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, 2021, pp. 9355–9366.
[15] Y. Li, K. Zhang, J. Cao, R. Timofte, M. Magno, L. Benini, and L. Van Gool, "LocalViT: Analyzing locality in vision transformers," 2021, *arXiv:2104.05707*.
[16] J. Yang, C. Li, P. Zhang, X. Dai, B. Xiao, L. Yuan, and J. Gao, "Focal self-attention for local–global interactions in vision transformers," 2021, *arXiv:2107.00641*.
[17] Z. Tu, H. Talebi, H. Zhang, F. Yang, P. Milanfar, A. Bovik, and Y. Li, "MaxViT: Multi-axis vision transformer," in *Proc. 17th Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2022, pp. 459–479.
[18] J. Ding, S. Ma, L. Dong, X. Zhang, S. Huang, W. Wang, N. Zheng, and F. Wei, "LongNet: Scaling transformers to 1,000,000,000 tokens," 2023, *arXiv:2307.02486*.
[19] J. W. Rae, A. Potapenko, S. M. Jayakumar, and T. P. Lillicrap, "Compressive transformers for long-range sequence modelling," 2019, *arXiv:1911.05507*.
[20] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," 2020, *arXiv:2006.04768*.
[21] B. Lin, C. Zhang, T. Peng, H. Zhao, W. Xiao, M. Sun, A. Liu, Z. Zhang, L. Li, X. Qiu, S. Li, Z. Ji, T. Xie, Y. Li, and W. Lin, "Infinite-LLM: Efficient LLM service for long context with DistAttention and distributed KVCache," 2024, *arXiv:2401.02669*.
[22] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 16344–16359.
[23] Z. Liu, H. Hu, Y. Lin, Z. Yao, Z. Xie, Y. Wei, J. Ning, Y. Cao, Z. Zhang, L. Dong, F. Wei, and B. Guo, "Swin transformer V2: Scaling up capacity and resolution," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2022, pp. 12009–12019.
[24] O. Lieber et al., "Jamba: A hybrid transformer-mamba language model," 2024, *arXiv:2403.19887*.
[25] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv:1909.08053*.
[26] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–10.
[27] H. Liu, M. Zaharia, and P. Abbeel, "Ring attention with blockwise transformers for near-infinite context," 2023, *arXiv:2310.01889*.
[28] D. Li, R. Shao, A. Xie, E. P. Xing, X. Ma, I. Stoica, J. E. Gonzalez, and H. Zhang, "DISTFLASHATTN: Distributed memory-efficient attention for long-context LLMs training," in *Proc. 1st Conf. Lang. Model.*, 2024, pp. 1–18.
[29] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence parallelism: Making 4D parallelism possible," 2021, *arXiv:2105.13120*.
[30] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst. (TOCS)*, vol. 3, no. 2, pp. 145–159, 1985.
[31] R. H. Bruck and H. J. Ryser, "The nonexistence of certain finite projective planes," *Can. J. Math.*, vol. 1, no. 1, pp. 88–93, Feb. 1949.
[32] S. Chowla and H. J. Ryser, "Combinatorial problems," *Can. J. Math.*, vol. 2, pp. 93–99, Mar. 1950.
[33] C. W. H. Lam, "The search for a finite projective plane of order 10," *Amer. Math. Monthly*, vol. 98, no. 4, pp. 305–318, Apr. 1991.
[34] W.-S. Luk and T.-T. Wong, "Two new quorum based algorithms for distributed mutual exclusion," in *Proc. 17th Int. Conf. Distrib. Comput. Syst.*, 1997, pp. 100–106.
[35] M. Hall, *Combinatorial Theory*, vol. 71. Hoboken, NJ, USA: Wiley, 1998.
[36] C. J. Kleinheksel and A. K. Somani, "Efficient distributed all-pairs algorithms: Management using optimal cyclic quorums," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 2, pp. 391–404, Feb. 2018.
[37] Y. Bian and A. K. Somani, "An efficient systematic approach to find all cyclic quorum sets with all-pairs property," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Mar. 2021, pp. 197–206.
[38] C. J. Kleinheksel and A. K. Somani, "Enhancing fault tolerance capabilities in quorum-based cycle routing," in *Proc. 7th Int. Workshop Reliable Netw. Design Model. (RNDM)*, Apr. 2015, pp. 27–33.
[39] B. R. Sklar and A. K. Somani, "Ray tracing: Parallelization via image decomposition and performance impact," in *Proc. ICPP Workshop Challenges Parallel Process.*, vol. 2, 1996, pp. 108–115.
[40] X. Ma, X. Yang, W. Xiong, B. Chen, L. Yu, H. Zhang, J. May, L. Zettlemoyer, O. Levy, and C. Zhou, "Megalodon: Efficient LLM pretraining and inference with unlimited context length," 2024, *arXiv:2404.08801*.
[41] X. Ma, C. Zhou, X. Kong, J. He, L. Gui, G. Neubig, J. May, and L. Zettlemoyer, "Mega: Moving average equipped gated attention," 2022, *arXiv:2209.10655*.
[42] A. Sun, W. Zhao, X. Han, C. Yang, Z. Liu, C. Shi, and M. Sun, "BurstAttention: An efficient distributed attention framework for extremely long sequences," 2024, *arXiv:2403.09347*.

YIMING BIAN (Member, IEEE) received the B.S. degree in information and computing science from Tianjin University of Technology, Tianjin, China, in 2018, and the M.S. and Ph.D. degrees in computer engineering from Iowa State University, Ames, IA, USA, in 2020 and 2024, respectively.

He was a Research Assistant with the Dependable Computing and Network Laboratory and a Graduate Research Scientist with the Digital Ag Innovation Laboratory, Iowa State University, from 2019 to 2023 and from 2023 to 2024, respectively. Since 2024, he has been a Postdoctoral Research Associate with the Akey Laboratory, Lewis-Sigler Institute for Integrative Genomics, Princeton University. His research interests include computer vision, deep learning, and high-performance computing, specifically on object recognition in medical imaging modalities, robustness improvement of classification models, and distributed implementation of large language models.

ARUN K. SOMANI (Life Fellow, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from McGill University, Montreal, in 1983 and 1985, respectively. He was a Scientific Officer with the Government of India, New Delhi, and a Faculty Member with the University of Washington, Seattle. He is currently an Anson Marston Distinguished Professor in electrical and computer engineering with Iowa State University. He received a Canadian Commonwealth Scholarship for his graduate work. His research interests include designing and implementing scalable and dependable high-performance computing and networking systems, optical fiber networks, critical infrastructure protection, image-based navigation, and neural network architecture design. He is a fellow of AAAS. He served as an IEEE Distinguished Visitor, an IEEE Distinguished Tutorial Speaker, and an IEEE Communication Society Distinguished Visitor. He has delivered several keynote speeches and distinguished and invited talks worldwide. He has been recognized as a Distinguished Alumnus of the Birla Institute of Technology and Science, Pilani, India. He is a Distinguished Engineer of ACM and an Eminent Engineer of Tau Beta Pi.