STREAM-CQSA: EXACT OUT-OF-MEMORY RECOVERY FOR ATTENTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-context large language models are limited not only by attention cost but also by out-of-memory (OOM) failures. A selected attention call may not fit in available device memory even when the kernel is optimized. Exact and approximate attention methods reduce memory use, but every fixed implementation still has a device-specific capacity boundary. We introduce Stream-CQSA, an attention-level OOM recovery framework based on CQS decomposition, derived from the theory of cyclic quorum sets (CQS). Stream-CQSA recursively partitions an infeasible attention call into independent subsequence tasks, executes each with a compatible inner kernel, and recomposes the local statistics to recover the full attention output. This recovery is exact relative to the wrapped attention kernel, whether that kernel is exact or approximate. Compared with FlashAttention-2, the major baseline, our native Stream-CQSA kernel improves 16-bit forward-output error relative to a dense float64 reference and matches 16-bit backward-gradient error where FlashAttention-2 fits in the GPU memory. At the longest feasible baseline length, it costs 1.5–$1.9\times$ the forward runtime and 2.1–$2.4\times$ the forward–backward runtime. Beyond that sequence length boundary, our method continues to return an output while FlashAttention-2 OOMs. Stream-CQSA is therefore not a faster attention method. Instead, it converts memory-capacity failure into a recoverable execution path by trading extra compute, host-device transfer, and recomposition for completion.

1 INTRODUCTION

Modern large language model capabilities depend strongly on model size and context length (Kaplan et al., 2020). Recent systems (Anthropic, 2024; Comanici et al., 2025; Guo et al., 2025; Yang et al., 2025) support context windows from hundreds of thousands to one million tokens, yet long inputs still degrade model behavior and remain expensive to process. A central barrier is the quadratic memory cost of exact self-attention (Vaswani et al., 2017). In practical training and inference pipelines, GPU memory often becomes the limiting resource even when the attention kernel is optimized. Cached activations, temporary buffers, and intermediate tensors can create run-time memory pressure that is hard to estimate statically. An otherwise well-defined attention computation may therefore fail at execution time and terminate the entire run. This motivates an automatic recovery mechanism that is independent of any particular sequence length or fixed memory budget.

This work accordingly addresses attention-level OOM recovery problem, in which the target attention computation has already been selected but the corresponding call cannot complete within the available memory budget. We propose CQS decomposition (Figure 1), an operation derived from cyclic quorum set (CQS) theory which partitions attention into independent subsequence computations whose local statistics recompose exactly in full-token coordinates. Stream-CQSA builds on this operation by increasing decomposition depth until the subproblems fit, executing them with a compatible inner kernel, and merging their statistics to recover the intended result. Existing attention methods can serve as inner kernels if they expose the required local statistics. Therefore, The objective of this work is neither to define a new efficient attention mechanism, nor to claim a speedup where the original computation already fits. It is to render an otherwise infeasible call executable.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

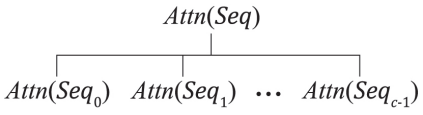


Figure 1: CQS decomposition

2 Prior work and positioning of this work

Prior work improves long-context attention by reducing data movement, changing the attention rule, or distributing the computation. Exact methods such as FlashAttention (Dao et al., 2022; Dao, 2023) reduce memory movement and intermediate storage while preserving standard attention. Approximate methods reduce complexity by changing the computation, including sparse and blockwise attention (Beltagy et al., 2020; Zaheer et al., 2020; Qiu et al., 2020; Chen et al., 2023), dilated attention (Ding et al., 2023), low-rank attention (Wang et al., 2020), kernelized attention (Kitaev et al., 2020; Katharopoulos et al., 2020; Choromanski et al., 2020), and global-local or streaming mechanisms (Xiao et al., 2023). Distributed methods shard the sequence across devices. Ring Self-Attention (Li et al., 2023b) rotates keys and values around a device ring, Ring Attention (Liu et al., 2024) extends this approach, Striped Attention (Brandon et al., 2023) adapts it to causal masking, and DeepSpeed-Ulysses (Jacobs et al., 2023) and DistFlashAttn (Li et al., 2023a) use different communication patterns. These methods improve, approximate, or distribute attention. However, under a fixed device budget, each of them still has a capacity boundary.

Stream-CQSA instead recovers a computation that has already failed under a given budget. Out-of-core attention Rabe & Staats (2021) is closest, but it only avoids materializing the quadratic attention matrix. Q, K, V, outputs, and gradients remain resident $\Theta(ND)$ terms, which account for 70–79% of FlashAttention-2's peak memory in our measurements. Once these terms exceed the budget, smaller tiles cannot help. Sequence-parallel methods Li et al. (2023b); Liu et al. (2024); Brandon et al. (2023); Jacobs et al. (2023); Li et al. (2023a) move the boundary outward by roughly the number of devices, but the degree of parallelism is fixed at launch and a shard that remains too large requires reconfiguration and restart. Stream-CQSA partitions the token set instead. A subproblem's working set shrinks with decomposition depth, every query–key pair is still computed exactly once, and subproblems communicate only during the final reduction. A failing subproblem can be decomposed further at run time (Section 3.4), and devices need not be identical. Stream-CQSA is complementary to the exact, approximate, and distributed attention families mentioned above. A method can serve as the inner attention kernels as long as it exposes recomposition statistics and supports the CQS mask. Relative to CQS-Attention Bian & Somani (2025), Stream-CQSA adds new features such as recursive decomposition, a pluggable inner-kernel interface, workload scheduling, and OOM guardrails.

3 Design of Stream cyclic quorum sets attention

Attention can be viewed as all pairwise token interactions. Stream-CQSA treats the selected attention call as the target and partitions it into subsequence computations whose retained interactions exactly union to the full set of token pairs. The main question is how many subsets are needed and which chunks each should contain.

Suppose the sequence is to be divided into c subsequences while preserving all interactions. We first need to partition it into c chunks, indexed from 0 to $c-1$, thus $c(c-1)/2$ distinct off-diagonal chunk pairs. Each subsequence will select l chunks and contributes $l(l-1)/2$ such pairs. Non-redundant full coverage therefore requires $c\,l(l-1)/2 = c(c-1)/2$, or $c = l(l-1)+1$. Therefore, valid c values are $\{1, 3, 7, 13, 21, \dots\}$ for $l = \{1, 2, 3, 4, 5, \dots\}$. Essentially, CQS decomposition splits a complete graph (K_c) into c smaller complete subgraphs (K_l) that every edge in K_c is covered exactly once, corresponding to a Steiner system $S(2, l, c)$ (Stinson, 2008) when it exists.

The cases $l = 1, c = 1$ and $l = 2, c = 3$ are trivial. Therefore, we use the smallest non-trivial case $l = 3, c = 7$ throughout the discussion. The sequence is partitioned into 7 chunks $(C_0, \dots, C_6)$, and each subsequence i is constructed as

$$Seq_i = \texttt{concat}(C_{(0+i) \bmod 7}, C_{(1+i) \bmod 7}, C_{(3+i) \bmod 7}) \tag{1}$$

for $0 \leq i \leq 6$. For notational simplicity, we denote subsequences by their chunk indices. The construction is cyclic, with the pattern $(0, 1, 3)$ in the first subsequence. We refer to it as an interest set ($\mathcal{I}$) for $c = 7$ (Bian & Somani, 2021) and we denote $|\mathcal{I}| = l$. We visualize the full coverage under this construction in Figure 10, and further discussion on CQS interest set is given in Appendix C.

3.1 FORWARD PASS

For each subsequence i, we gather Q_i, K_i, and V_i from the full-sequence tensors Q, K, and V. CQSA (Bian & Somani, 2025) then performs the following computations:

$$R_i = \alpha(Q_i K_i^\top) \quad P_i = \exp(R_i) \odot M_i \quad \mathrm{Num}_i = P_i V_i \quad \mathrm{Den}_i = \mathrm{row_sum}(P_i) \tag{2}$$

where $\alpha = 1/\sqrt{D}$, D is head dimension, M_i is the CQS binary mask (Section 3.3), and $\mathrm{Num}_i, \mathrm{Den}_i$ correspond to the numerator and denominator of the softmax computation.

CQSA then scatters local statistics into full-token accumulators and normalizes as

$$\mathrm{Num} = \sum_i \mathrm{scatter}(\mathrm{Num}_i) \quad \mathrm{Den} = \sum_i \mathrm{scatter}(\mathrm{Den}_i) \quad O = \mathrm{Num} \oslash \mathrm{Den} \tag{3}$$

where $\oslash$ denotes token-wise division broadcast along the head dimension (D).

The forward pass is summarized in Algorithm 1. Figure 10 provides a step-by-step illustration for $c = 7$ and $\mathcal{I} = (0, 1, 3)$, explicitly verifying full coverage of the attention matrix P.

3.2 BACKWARD PASS

The backward pass follows the same decomposition. Let $\mathbf{d}O$ denote the upstream gradient. Since $O = \mathrm{Num} \oslash \mathrm{Den}$, we have

$$\mathbf{d}\mathrm{Num} = \mathbf{d}O \oslash \mathrm{Den} \qquad \mathbf{d}\mathrm{Den} = -\frac{\langle \mathbf{d}O, \mathrm{Num} \rangle_D}{\mathrm{Den}^2} \tag{4}$$

where $\langle \cdot, \cdot \rangle_D$ denotes the inner product on the head dimension.

Using the same $(c, \mathcal{I})$, we gather $\mathbf{d}\mathrm{Num}_i$ and $\mathbf{d}\mathrm{Den}_i$ for each subsequence and compute

$$\begin{gathered} \mathbf{d}V_i = P_i^\top \mathbf{d}\mathrm{Num}_i \quad \mathbf{d}P_i = \mathbf{d}\mathrm{Num}_i V_i^\top + \mathbf{d}\mathrm{Den}_i \mathbf{1}^\top \\ \mathbf{d}R_i = \mathbf{d}P_i \odot P_i \quad \mathbf{d}Q_i = \alpha \mathbf{d}R_i K_i \quad \mathbf{d}K_i = \alpha \mathbf{d}R_i^\top Q_i \end{gathered} \tag{5}$$

Finally, $\mathbf{d}Q_i$, $\mathbf{d}K_i$, and $\mathbf{d}V_i$ are scattered into full-sequence coordinates exactly as in the forward pass

$$\mathbf{d}Q = \sum_i \mathrm{scatter}(\mathbf{d}Q_i) \quad \mathbf{d}K = \sum_i \mathrm{scatter}(\mathbf{d}K_i) \quad \mathbf{d}V = \sum_i \mathrm{scatter}(\mathbf{d}V_i) \tag{6}$$

The backward pass is stated formally in Algorithm 2, with an illustration in Figure 11 for $c = 7$ and $\mathcal{I} = (0, 1, 3)$. A full chain-rule derivation is given in Appendix D.

3.3 CQS MASKING

Subsequences constructed by CQS decomposition exhibit structured overlap. This overlap guarantees full coverage of all token interactions but also introduces redundancy: since each chunk appears in l subsequences, interactions within diagonal chunk pairs are computed l times. Aggregating them directly would yield an incorrect result. Correctness therefore requires that each subsequence be assigned a predetermined responsibility over the whole attention matrix, with redundant contributions removed, which introduces the CQS mask. The masking rule is simple: subsequence i is responsible only for the diagonal chunk pair (i, i). For example, subsequence 0 with $\mathcal{I} = (0, 1, 3)$ contains $(0, 0)$, $(1, 1)$, and $(3, 3)$. It retains only $(0, 0)$ because the other two belong to subsequences 1 and 3.

CQS decomposition can be applied recursively because each subsequence is itself a sequence. We write `itr` for the decomposition depth. Depth `itr` yields $c^{\texttt{itr}}$ subsequences. At depth `itr` $+ 1$, each subsequence applies the same diagonal CQS masking rule and inherits the masks from earlier depths, producing structured off-diagonal masks. Figure 2 shows CQS masks at depths 1 and 2 for $N = 49$.

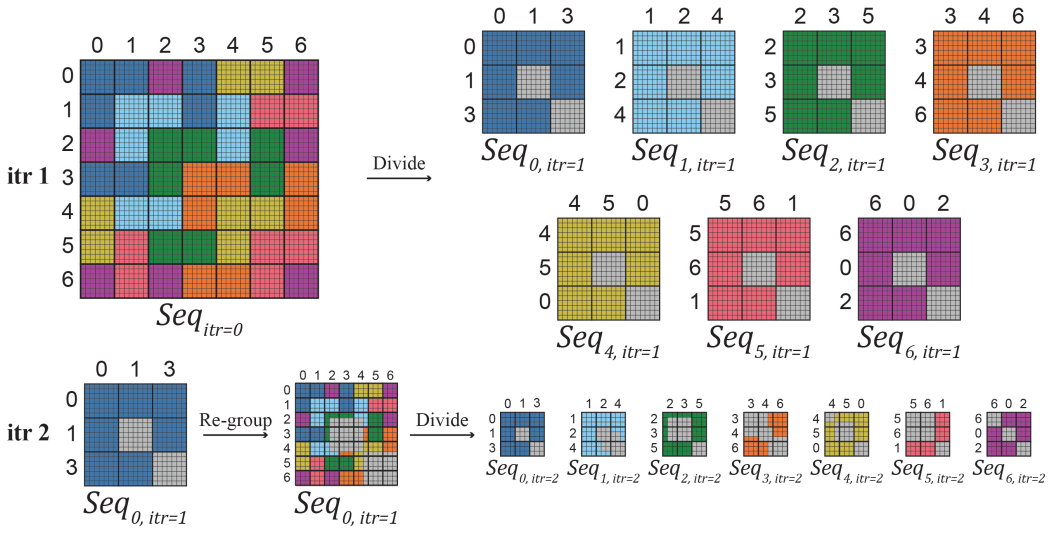


Figure 2: CQS masking on $R = QK^\top$ with $N = 49$, $c = 7$, and $\mathcal{I} = (0, 1, 3)$. Depth 1 forms 21-token subsequences and masks repeated diagonal regions. Depth 2 repartitions each subsequence into chunks of size 3, producing 9-token subsequences with local and inherited masks. Only $Seq_{0,\texttt{itr}=1}$ is shown here because other depth 1 subsequences ($Seq_{i,\texttt{itr}=1}$) follow the same pattern.

After masking, subsequence computations are non-redundant and independent. Stream-CQSA can therefore execute them in parallel, distribute them across devices, and schedule them on heterogeneous hardware (Section 3.5). The choice of c is also flexible. Different valid values, such as those listed in Table 7, may be used at each depth, affording finer control over subsequence length and thus higher utilization of the available memory.

3.4 OOM GUARDRAILS

Subsequence independence is also what makes recovery possible. If a subsequence still exceeds the memory budget, CQS decomposition can split it again into smaller independent subproblems whose local statistics recompose in the same way. This recursion continues until every subproblem fits, subject to the minimal buffers required by the inner attention kernel and the host-side storage required for recomposition. The guardrail built on this property rests on a single rule: whenever the current configuration exhausts memory, increase the decomposition depth (`itr`) by one. Since it stops at the first depth that does not fail, it returns the smallest feasible depth by construction.

The second parameter, `n_cap`, bounds the number of subsequences resident on the GPU at once and adjusts within a depth before the depth itself changes. On a failure the offending subsequence is evicted and returned to the scheduling queue and `n_cap` is reduced, which lowers memory pressure without altering the decomposition. Depth changes only when `n_cap` reaches either limit of its range. If $\texttt{n_cap} = 1$ OOMs, no further reduction is available and the depth increases. If $\texttt{n_cap} = c^{\texttt{itr}}$ and headroom remains, the decomposition is finer than necessary and the depth decreases. After any depth change, calibration re-establishes `n_cap`. The two limits of `n_cap` thus bracket the depth from below and above, and `n_cap` is re-established by a calibration pass after any change of depth. We stress that `n_cap` bounds residency rather than concurrency. How many resident subsequences execute simultaneously is a property of the inner kernel.

3.5 WORKLOAD SCHEDULING

Once the guardrail has established a feasible decomposition, the remaining question is placement. A CQS subsequence computes independently of every other and the local statistics recompose in any order, so a monolithic attention call becomes a set of schedulable tasks. Any assignment of these tasks to devices and to time that respects the memory limit of each device yields exactly the same result. Correctness therefore constrains the decomposition but not the schedule.

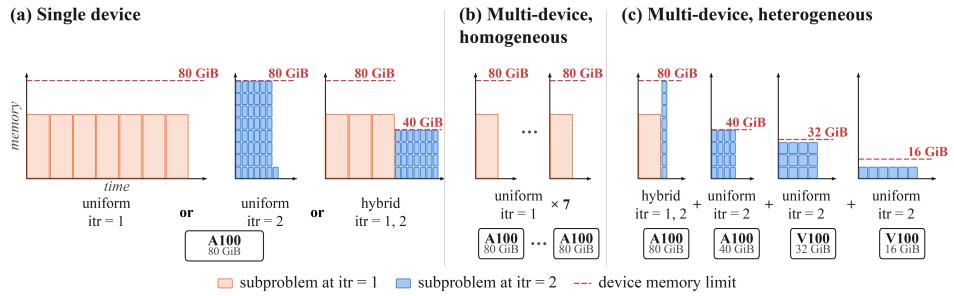


Figure 3: Schedules admitted by the decomposition. Both axes are absolute and shared across the panels so that blocks are directly comparable in the figure. The workload is hypothetical and the per-subproblem costs are chosen only so that the schedules may be drawn to scale, with the V100 assumed $1.6\times$ slower than the A100. **(a)** Three schedules for a single device, including uniform schedules at either granularity and a hybrid that mixes them. **(b)** Identical devices, with one subproblem on each. **(c)** Devices of different capacities, each scheduling under its own limit. A given subproblem has the same height but is wider on the slower device.

Figure 3 illustrates this flexibility. On a single device, a further level of decomposition produces more subproblems that are individually cheaper, so the total workload rises. It also reduces the memory each subproblem requires and allows several to be resident at once, so the schedule may complete sooner despite performing more work. A hybrid schedule mixes the two depths. It arises most often not by design but from a change in the memory available during execution. The OOM guardrail decomposes the remaining branches further while those already completed stay at the coarser depth. Across devices, independence removes communication from the critical path, no exchange being required until the final summation. Where the devices differ, two properties govern the assignment. The feasible depth is a property of the device, because a subproblem that does not fit compels further decomposition, which is what allows a smaller device to participate. The per-subproblem cost is instead a property of the hardware, unchanged in memory but greater in time on a slower device. A scheduler must account for both. All experiments in this work use the single-device uniform schedule (Figure 3a). The other schedules show what the decomposition permits, not configurations we benchmark. Scheduler design is an important problem and is left to future work.

4 Experiments

The following evaluation addresses three claims. The first is **correctness under precision**: decomposition and recomposition should not change the attention output beyond dtype-level error, in either forward or backward pass at supported precisions. The second is **the cost where the unwrapped monolithic path fits**: Stream-CQSA is expected to be slower and we quantify that overhead. The third is **recovery**: beyond the baseline OOM sequence length boundary, Stream-CQSA should return a result, with the decomposition depth increasing as the memory budget tightens.

4.1 Setup

All measurements use causal attention with $B = 1$, $H = 8$, $D = 64$ on a single NVIDIA A100 80 GiB, PyTorch 2.10.0+cu130 and `flash-attn` 2.8.3. CQS decomposition uses $c = 7$ and $\mathcal{I} = (0, 1, 3)$, and every Stream-CQSA measurement runs on the native kernel of Appendix A, which is modified based on FlashAttention-2 to evaluate the CQS mask and to skip the tiles that mask leaves empty. Synthetic Q, K, V and the upstream gradient $\mathbf{d}O$ are drawn from a zero-mean unit-variance Gaussian with fixed seeds and generated once per (N, dtype), so every method sees identical inputs at each point. We test fp16 and bf16 because the FlashAttention tensor-core path does not accept fp32, and the output, the softmax statistics and the merge are computed in fp32.

Baselines. The main experiments compare exact attention implementations only and one compatible approximate-attention case is demonstrated separately in Appendix B. *SDPA* is PyTorch's default dispatcher, which by default selects a FlashAttention backend on this hardware. *FlashAttention-2* is

Table 1: Relative error against the float64 reference. Backward error by gradient tensor is given in Table 8. $\mathcal{S}$ denotes Stream-CQSA. The $\texttt{itr} = 0$ row is the monolithic or *undecomposed* path, which is what the shipped package $\texttt{itr} = \text{AUTO}$ selects at $N = 8192$.

Method	Forward		Backward	
	fp16	bf16	fp16	bf16
SDPA	2.689×10^{-4}	2.168×10^{-3}	3.075×10^{-4}	2.458×10^{-3}
SDPA (mem-eff.)	2.685×10^{-4}	2.159×10^{-3}	3.739×10^{-4}	2.980×10^{-3}
FlashAttention-2	2.689×10^{-4}	2.168×10^{-3}	3.075×10^{-4}	2.458×10^{-3}
$\mathcal{S}$ ITR=0 (undecomposed)	2.689×10^{-4}	2.168×10^{-3}	3.075×10^{-4}	2.458×10^{-3}
$\mathcal{S}$ ITR=1	1.705×10^{-4}	1.375×10^{-3}	3.080×10^{-4}	2.462×10^{-3}
$\mathcal{S}$ ITR=2	1.681×10^{-4}	1.356×10^{-3}	3.085×10^{-4}	2.465×10^{-3}

the reference `flash-attn` package (Dao et al., 2022; Dao, 2023). We report both because they differ by 1–8% across the measured range, due to dispatch overhead and bundled-build differences. The third baseline is ***SDPA (mem-eff.)***, which pins PyTorch's memory-efficient backend and is the only baseline whose internals differ.

Stream-CQSA configurations. We test three configurations that vary in two aspects. The first is the decomposition depth `itr`, which determines the number of subproblems. The second is the residency of the fp32 accumulator, which is held either on the device or in host memory. In all three, the full-length Q, K and V remain in host memory for the whole call, and only the gathered subsequences Q_i, K_i, V_i of the subproblems currently in flight are copied to the device. Two fixed-depth configurations are $\texttt{itr} = 1$ and $\texttt{itr} = 2$, both with the accumulator on the device and the OOM guardrail disabled on purpose. These two variants aim to demonstrate the time-memory tradeoff and they are not recommended in practice. The third is $\texttt{itr} = \text{AUTO}^*$ with $\min(\texttt{itr}) = 1$, the accumulator in host memory and OOM guardrail engaged. So the depth reported is either 1 or the smallest at which no subproblem exhausts memory. The star distinguishes it from the $\texttt{itr} = \text{AUTO}$ of the shipped package, which begins at $\texttt{itr} = 0$ and returns a single monolithic call when possible to avoid the decomposition cost.

Metrics. Peak device memory (GiB) is `torch.cuda.max_memory_allocated()` over the call, with the allocator drained between measurements. Wall-clock time (s) is the median over repeats. We report two timing scopes. A forward timing measures only the forward attention call and corresponds to prefill. A forward–backward timing measures the full training-step attention cost by running the forward call followed by gradient computation. We use this scope because the gradient is not a standalone computation. It depends on intermediates produced by the same forward call, including the output, softmax statistics, and for Stream-CQSA the global log-sum-exp produced by the decomposed forward. Timing only the gradient would either reuse intermediates created outside the measured region or charge one configuration for intermediates prepared by another. Neither case is a cost a user pays in isolation. This forward–backward measurement gives every method the same scope and keeps ratios comparable. Per-stage breakdowns use CUDA events. When several subproblems are in flight, event spans include time queued behind other streams, so they show where work is issued rather than a strict runtime partition.

4.2 Output preservation under precision

All accuracy figures in the forward and backward pass reported in Table 1 are relative errors against a **float64** reference computed from the same inputs. A dense float64 is $O(N^2)$ and we fix the sequence length to $N = 8\,192$ so that a dense backward reference remains tractable. We average the difference over 10 random draws. In the forward pass, the decomposed path is more accurate than every baseline. The gain is because Stream-CQSA (denoted as $\mathcal{S}$) accumulates in fp32 and returns fp32, whereas the baselines round their output back to the 16-bit input dtype. In the backward it is comparable, differing from the baselines by at most 0.3% at either precision. Because there is no fp32-to-16-bit rounding happens in the backward for all implementations. Overall, the decomposition-recomposition mechanism of Stream-CQSA preserves the attention function to the precision of the operands.

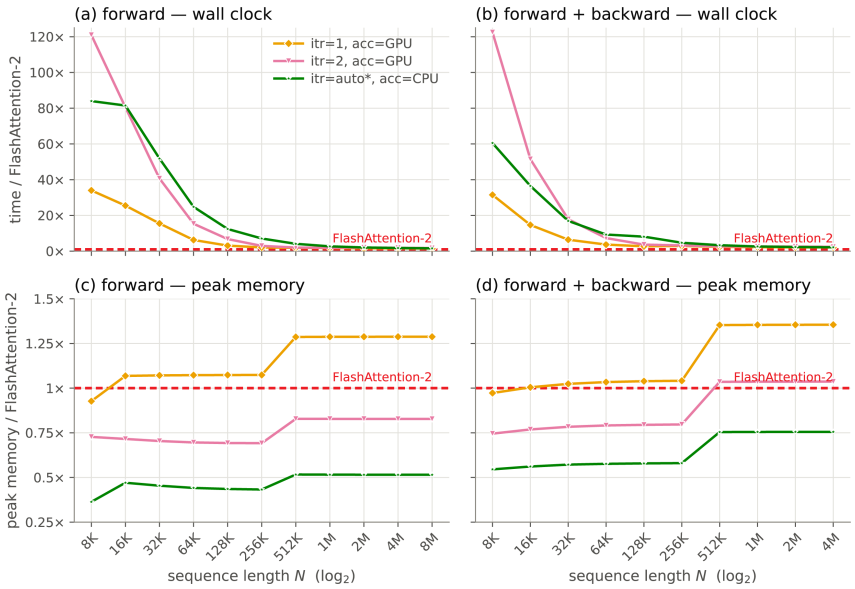


Figure 4: Cost of three Stream-CQSA configurations relative to FlashAttention-2. The forward panels run up to 8M and the forward–backward panels to 4M. The dashed line marks parity with FlashAttention-2. The underlying measurements are given in Tables 9 and 10.

4.3 Cost when the monolithic call fits

This subsection and Section 4.4 use one sweep over twelve sequence lengths from 8K to 16M in powers of two. Here we report the range where FlashAttention-2 completes, with ratios defined up to 8M for forward and 4M for forward–backward. Figure 4 shows that overhead is large at short lengths and falls quickly. In the forward pass, `itr` = AUTO* costs 84× at 8K, 2.68× at 1M, 1.76× at 4M, and 1.59× at 8M. Fixed-depth configurations follow similar curves. Forward–backward timing settles slightly higher, at 2.17× at 4M. The decline occurs because gather, staging, and recomposition costs scale mainly with the number of subproblems, while attention arithmetic grows with N^2. Thus overhead is worst where recovery is least useful and it becomes smallest near the sequence length boundary. Memory shows the opposite tradeoff. From $N = 512$K upward with the accumulator on device, Stream-CQSA converges to 1.29× forward and 1.35× forward–backward memory at `itr` = 1, and to 0.83× and 1.04× at `itr` = 2. Moving the accumulator to host is the only configuration below baseline memory in both passes, at 0.52× forward and 0.76× forward–backward. Stream-CQSA therefore pays for subsequence construction, transfers, repeated kernel launches, and recomposition. Therefore, it is an OOM recovery mechanism, not the preferred path when a monolithic call fits.

4.4 Recovery beyond the memory boundary

We next increase N to 16M (Figure 5), beyond the baseline GPU-memory boundary. Peak memory is linear in N, so each curve reaches the 79.3 GiB device limit at a predictable length. All baselines and Stream-CQSA `itr` = 1 fail at 16M forward and 8M forward–backward. Fixed `itr` = 2 completes 16M forward but fails at 8M forward–backward. Stream-CQSA `itr` = AUTO* with the OOM guardrail completes all measured lengths in both directions. Table 2 summarizes the performance at the memory boundary. The baselines fail first, ACC=GPU survives further up to the maximum capacity of the chosen depth, and ACC=CPU never fails and has a slower memory growth rate because the OOM guardrail is engaged and the accumulator is relocated to host memory.

One table entry of PyTorch's memory-efficient forward–backward path takes 15 421 s at 4M, versus 407 s for FlashAttention-2. Below 1M, the same ratio is only 2.8–3.1×. The jump occurs between 1M and 2M, where runtime grows by 49.6× instead of the expected quadratic 4×, before quadratic scaling resumes from 2M to 4M (Figure 5d). The cause is a CUDA grid-dimension limit. That backend tiles queries into 32-row blocks mapped to a grid dimension capped at 65 535, creating

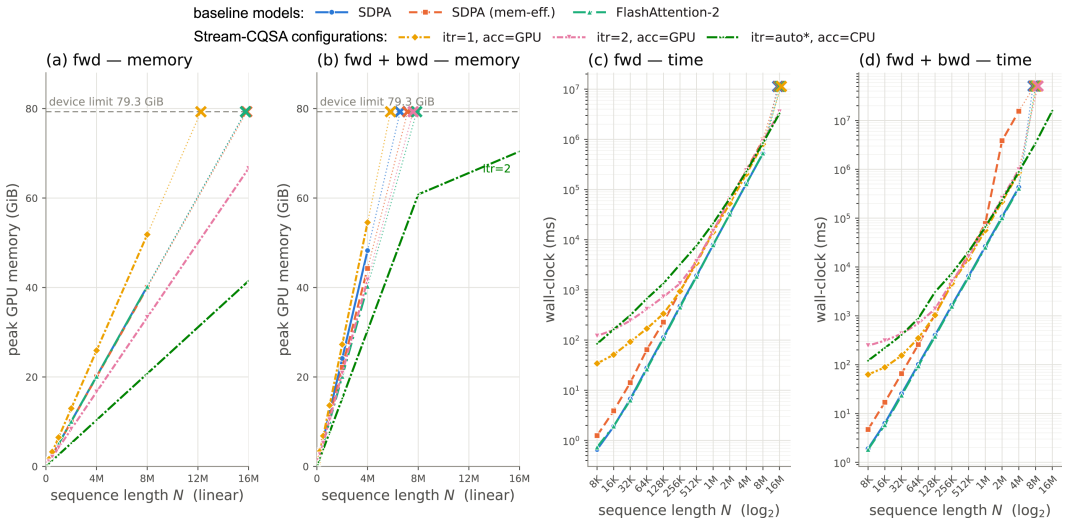


Figure 5: Peak device memory and wall-clock time versus sequence length

Table 2: Memory boundary and recovery. Entries are time (s) / peak memory (GiB). The final row shows the depth reached by `itr` = AUTO*. A star marks a length where the shipped `itr` = AUTO would execute a monolithic call (i.e., `itr` = 0).

Method	Forward			Forward–Backward		
	4M	8M	16M	4M	8M	16M
SDPA	131 / 20.1	520 / 40.3	OOM	429 / 48.3	OOM	OOM
SDPA (mem-eff.)	245 / 20.0	966 / 40.0	OOM	15421 / 44.3	OOM	OOM
FlashAttention-2	130 / 20.1	532 / 40.3	OOM	407 / 40.3	OOM	OOM
$\mathcal{S}$, ITR=1, ACC=GPU	203 / 25.9	802 / 51.8	OOM	847 / 54.5	OOM	OOM
$\mathcal{S}$, ITR=2, ACC=GPU	240 / 16.7	914 / 33.3	3544 / 66.6	983 / 41.7	OOM	OOM
$\mathcal{S}$, ITR=AUTO*, ACC=CPU	228 / 10.4	846 / 20.7	3277 / 41.5	885 / 30.4	3437 / 60.8	15847 / 70.4
depth `itr` reached	1*	1*	1	1*	1	2

a boundary at $65\,535 \times 32 = 2\,097\,120$ tokens. In a targeted check of the backward kernel, that backend took 36.5 s at $N = 2\,097\,088$ (65 534 blocks) but 4250 s at $N = 2\,097\,120$ (65 535 blocks), while FlashAttention-2 stayed around 13 s in both cases. This explains the wall-clock time leap from $N = 1$M (32 768 blocks) to $N = 2$M (65 536 blocks).

Figure 6 profiles three Stream-CQSA configurations by stage. Figure 12 shows the same measurements in absolute time on log-log axes. We report six stages. `gather` assembles subsequence Q_i, K_i, V_i tensors, `scatter` returns local results to full-token coordinates, `h2d` and `d2h` transfer data, `compute` runs attention, and `merge` recomposes max-shifted partial statistics. The costs separate by order. `compute` is quadratic, while gather, scatter, and transfers are linear in N. Accumulator residency determines how much linear work remains. With a device accumulator, partial results are merged in place and `merge`/`d2h` are nearly constant. With a host accumulator, each subproblem's fp32 statistics cross the bus and merge on the host, which are both linear costs. While the sequence becomes longer, linear work measured against the quadratic work becomes negligible and the attention computation (`compute`) dominates the breakdown. For `itr` = AUTO* with ACC=CPU, `compute` accounts for 96.5% of total stage time in the forward panel and 97.2% in the backward panel at $N = 16$M. Stage totals sum CUDA-event spans across subsequences. They can exceed wall-clock time when streams overlap. At short lengths the sum reaches $3.18\times$ elapsed time. From $N = 2$M onward, only one subsequence is resident, and stage sum and wall clock differ by at most 3%. For example, they differ by 17 s out of 3 248 s at $N = 16$M. Therefore, in the very long sequence regime where a monolithic call fails and Stream-CQSA is designed for, the attention computation dominates the runtime and extra cost is almost negligible.

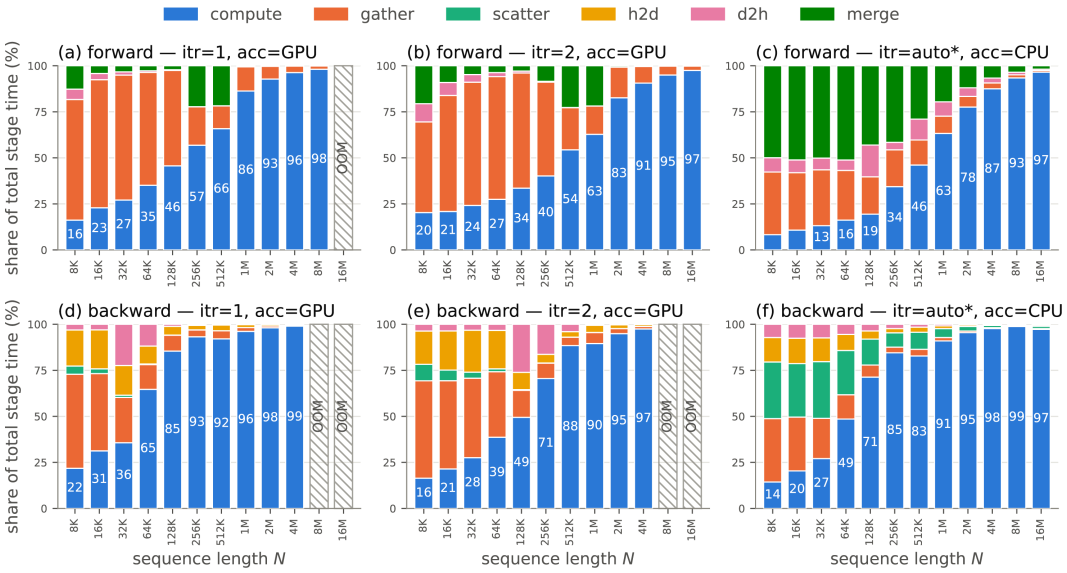


Figure 6: Stage breakdown for three Stream-CQSA configurations. The backward panels cover the backward pass alone, unlike forward–backward wall-clock and memory measurements elsewhere.

5 Limitations and Future Work

This work addresses attention-level out-of-memory recovery through a decomposition-recomposition mechanism. The evaluation shows that Stream-CQSA preserves the attention output to within the precision of its operands and continues to return results beyond the baseline OOM boundary. Nevertheless, it is not yet an optimized attention engine and the present implementation should be reckon as a reference recovery system. Below we specify three limitations and each marks a direction we regard as open rather than a defect of the mechanism.

Kernel coverage. The native kernel covers the head dimensions FlashAttention-2 does in the forward pass from 32 to 256. But the backward currently supports only 64 and 128. Expanding that range is necessary for a production version. Two further directions follow: a faster kernel with better memory scaling and approximated kernels compatible to Stream-CQSA. In addition, converting an attention method manually is labor-intensive, and automating it with a tool or an agent that rewrites an attention implementation into a compatible one is another promising direction.

Scheduling. The evaluation in this work uses the simplest single-device uniform schedule, which leaves plenty of room for optimization especially in large-scale systems. The subproblems are independent by construction, so a scheduler may execute several concurrently, overlap transfer with computation, and place them across multiple or heterogeneous devices. A distributed implementation is the natural next step and we regard this as an important engineering and system design problem.

System integration. The experiments isolate attention, while real OOMs can arise from activation caches, temporary buffers, optimizer state, and other model components. Stream-CQSA relocates some memory pressure rather than removing it, so end-to-end integration in a language model is needed to decide where recovery should trigger. Edge devices are another relevant setting, where fixed and modest memory budgets may make a time-for-memory tradeoff worthwhile.

Repeated decomposition can make each in-flight subproblem as small as the budget permits, aided by host-resident fp32 accumulators and a cap on resident subproblems. What can not be shrunk is the full-length data itself such as Q, K and V and the final attention output O. Stream-CQSA relocates those tensors into host memory, which is larger and cheaper but still finite. Therefore, extremely long sequences can exhaust host memory or produce outputs that fill the GPU. This intrinsic limit is outside what this mechanism can reach.

AI USE STATEMENT

In this work, we used generative AI tools to aid and polish writing, including improving clarity, readability, and LaTeX formatting. We also used generative AI tools for retrieval and discovery, including searching for and identifying potentially relevant related work. In addition, we used coding-agent tools to assist with implementation and research execution by writing and modifying code for the Stream-CQSA implementation, experiments, evaluation, plotting, and manuscript preparation under author direction.

The core idea, method design, experimental design, scientific claims, and interpretation of results were developed by the authors, not generated by AI tools. We did not use generative AI tools to generate synthetic datasets, prove mathematical claims, formulate the core theoretical claims, design the experimental methodology, or interpret the experimental results. Survey/interview question generation and transcription of research recordings are not applicable to this work.

We reviewed all AI-assisted work. AI-suggested writing edits were checked by the authors for accuracy and consistency with the paper's claims. AI-suggested related work was verified by manually inspecting the relevant papers before inclusion. Coding-agent-generated code was reviewed, run, and checked against the intended implementation and experimental protocols, and the resulting outputs were validated by the authors before being used in the paper. We take responsibility for the final content of this work, including text, claims, code, results, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We support reproducibility by specifying the Stream-CQSA decomposition, masking rule, forward recomposition, and backward computation in Section 3, with additional implementation details and pseudocode in Appendix A and the supplementary algorithms. Section 4.1 describes the hardware, software versions, precision settings, tensor shapes, random input generation, baselines, Stream-CQSA configurations, and measurement protocol used in the experiments. All reported experiments use a single GPU, so reproducing the results does not require distributed infrastructure or multi-device scheduling. We will release a package containing the implementation, experiment scripts, plotting scripts, and instructions needed to reproduce the reported experimental results. The appendix further provides raw timing and memory tables, additional reference-kernel experiments, forward/backward comparisons, and the CQS interest-set construction needed to verify the decomposition topology.

REFERENCES

Anthropic. The claude 3 model family: Opus, sonnet, haiku. `https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf`, 2024.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Yiming Bian and Arun K Somani. An efficient systematic approach to find all cyclic quorum sets with all-pairs property. In *2021 IEEE International Conference on Big Data (Big Data)*, pp. 197–206. IEEE, 2021.

Yiming Bian and Arun K Somani. Cqs-attention: Scaling up the standard attention computation for infinitely long sequences. *IEEE Access*, 2025.

William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431*, 2023.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. Longlora: Efficient fine-tuning of long-context large language models. *arXiv preprint arXiv:2309.12307*, 2023.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

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

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.

Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. Longnet: Scaling transformers to 1,000,000,000 tokens. *arXiv preprint arXiv:2307.02486*, 2023.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Clement WH Lam. The search for a finite projective plane of order 10. *The American mathematical monthly*, 98(4):305–318, 1991.

Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Xuezhe Ma, Ion Stoica, Joseph E Gonzalez, and Hao Zhang. Distflashattn: Distributed memory-efficient attention for long-context llms training. *arXiv preprint arXiv:2310.03294*, 2023a.

Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2391–2404, 2023b.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *International Conference on Learning Representations*, volume 2024, pp. 3992–4008, 2024.

F Jessie MacWilliams, Neil JA Sloane, and John G Thompson. On the existence of a projective plane of order 10. *Journal of Combinatorial Theory, Series A*, 14(1):66–78, 1973.

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

Jiezhong Qiu, Hao Ma, Omer Levy, Wen-tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 2555–2565, 2020.

Markus N Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021.

James Singer. A theorem in finite projective geometry and some applications to number theory. *Transactions of the American Mathematical Society*, 43(3):377–385, 1938.

Douglas R Stinson. Combinatorial designs: constructions and analysis. *ACM SIGACT News*, 39(4): 17–21, 2008.

Jacobus Hendricus Van Lint and Richard Michael Wilson. *A course in combinatorics*. Cambridge university press, 2001.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

A NATIVE KERNEL IMPLEMENTATION

The preceding sections specify the decomposition, exact recomposition, and depth-selection policy. They do not by themselves yield a competitive kernel. A reference implementation that materializes the local score matrix (Appendix B) demonstrates correctness but runs roughly an order of magnitude slower than a production attention path.

The native kernel addresses three sources of slowdown. First, recomposition through unnormalized sums can overflow fp32 because it requires exponentiating log-sum-exp values. Second, every subproblem must apply the CQS mask cheaply enough that bookkeeping does not dominate. Third, decomposition multiplies kernel launches and host–device transfers, so the schedule must avoid waiting on the host once per subproblem.

A.1 OVERFLOW-FREE RECOMPOSITION

The direct recomposition uses an unnormalized numerator and denominator. For subproblem i with local scores R_i,

$$\mathrm{Num}_i = \exp(R_i)V_i \qquad \mathrm{Den}_i = \mathrm{row_sum}\ \exp(R_i) \qquad O = \Big(\textstyle\sum_i \mathrm{Num}_i\Big) \Big/ \Big(\textstyle\sum_i \mathrm{Den}_i\Big) \tag{7}$$

This is natural because many local kernels return normalized output $\texttt{out}_i$ and log-sum-exp $\texttt{lse}_i$, from which one can reconstruct $\mathrm{Den}_i = \exp(\texttt{lse}_i)$ and $\mathrm{Num}_i = \texttt{out}_i \odot \mathrm{Den}_i$. Equation equation 7 is algebraically correct but numerically unsafe. The exponential overflows fp32 when $\texttt{lse} > \ln(\text{FLT_MAX}) \approx 88.72$. Beyond that point Den_i becomes ∞, the ratio becomes ∞/∞, and the result can silently become zero or NaN. The backward substitution has the same problem through $\mathbf{d}\mathrm{Num} = \mathbf{d}O/\mathrm{Den}$ and $\mathbf{d}\mathrm{Den} = -\mathrm{row_sum}(\mathbf{d}O \odot \mathrm{Num})/\mathrm{Den}^2$. We avoid overflow in both passes.

Forward max-shifted merge. Each subproblem returns three statistics rather than a normalized output, namely a running row maximum m, a running sum ℓ of $\exp(\text{score} - m)$, and a running weighted sum acc of $\exp(\text{score} - m)V$. Merging a contribution $(\mathrm{acc}_i, \ell_i, m_i)$ re-bases both sides onto $m' = \max(m, m_i)$. Thus,

$$\mathrm{acc} \leftarrow \mathrm{acc}\, e^{m-m'} + \mathrm{acc}_i\, e^{m_i-m'} \qquad \ell \leftarrow \ell\, e^{m-m'} + \ell_i\, e^{m_i-m'} \qquad m \leftarrow m' \tag{8}$$

Since m' is the larger maximum, both exponents are at most zero, so no intermediate overflows. The final output $O = \mathrm{acc}/\ell$ and global log-sum-exp $\texttt{lse} = m + \log \ell$ are formed only after all subproblems are merged. No subproblem computes $\exp(\texttt{lse})$. The merge is order-independent, allowing subproblems to finish in scheduler order. Eq. equation 8 is the standard online softmax recurrence (Milakov & Gimelshein, 2018). The important point is that Stream-CQSA must carry the same three statistics across subproblems. A framework that collects only $\texttt{out}_i$ and $\texttt{lse}_i$ would still need the unsafe exponential in Eq. equation 7.

Backward global-lse reformulation. Substituting $\mathrm{Num} = \mathrm{Den} \odot O$ and folding Den into P gives

$$P[q,k] = \exp\big(s[q,k] - \texttt{lse}_{\text{global}}[q]\big) \le 1 \qquad \text{for all } q, k. \tag{9}$$

The forward already produces $\texttt{lse}_{\text{global}}$, so keeping it is free. Each subproblem receives this value and runs the standard softmax backward on the pairs it retains. Because CQS covers each query–key pair exactly once, subproblem gradients add directly. Also, $P \le 1$ by construction, so intermediates cannot overflow regardless of score range. Eq. equation 9 is algebraically identical to Algorithm 2. We state the implemented form separately because a literal implementation of the algorithm would reintroduce the overflow above.

Statistics in fp32. The triple (acc, ℓ, m) and the merge in Eq. equation 8 are computed in fp32 for all input dtypes, and the accumulator is written in fp32 instead of being rounded to 16 bits. This prevents merge error from compounding (Section 4.2) but makes the accumulator larger than Q, K, and V combined at 16-bit input precision. Its size also does not shrink with decomposition depth, driving the memory behavior in Section 4.4. One corner case remains. A subsequence may retain no keys for some query row. The sentinel $m = -\infty$ would make Eq. equation 8 compute $\exp(-\infty - (-\infty))$, producing NaN, so non-finite contributions are replaced by null contributions before merge.

A.2 MAKING THE CQS MASK CHEAP

A subsequence gathers whole chunks, but not every local query–key pair belongs to that subproblem. The CQS mask (Section 3.3) excludes the others. Each token carries a small bit vector recording its CQS groups, and a local pair (q, k) is retained exactly when $\texttt{bits}[q] \wedge \texttt{bits}[k] = 0$. The generic implementation tests this predicate per element and reads both bit vectors from global memory in the innermost loop. With an all-zero mask, which excludes nothing and performs no useful masking, this path made the forward $4.3\times$ slower and backward $9.1\times$ slower than the unmodified kernel. Three techniques recover that overhead.

Block summaries and an $O(1)$ tile verdict. An arbitrary mask requires pairwise tests, but the CQS mask is chunk-structured. Since each subsequence is a union of contiguous chunks, neighboring tokens share bits except at chunk boundaries. We exploit this with a summary test that classifies a whole score-matrix tile in constant time from precomputed block summaries. Here a *block* is a one-dimensional run of 64 tokens, distinct from a two-dimensional attention tile. For each block we store the bitwise OR and AND of its token group bits. The OR records bits present in any token, and the AND records bits present in every token. Summaries depend only on the decomposition shape and are computed once per subproblem with the mask.

A FlashAttention *tile* is a 128×128 score region. Its rows span two token blocks and its columns span two more, so the summary test reads four block summaries. OR-ing the row-block ORs gives $\mathrm{OR}(\text{rows})$, while AND-ing the row-block ANDs gives $\mathrm{AND}(\text{rows})$. Columns are combined similarly. Two bitwise conjunctions then classify the tile before any per-element work runs.

A tile is *clear* when $\mathrm{OR}(\text{rows}) \wedge \mathrm{OR}(\text{cols}) = 0$. No row bit appears in any column, so no pair is masked and the tile uses the unmodified FlashAttention path. It is *fully masked* when $\mathrm{AND}(\text{rows}) \wedge \mathrm{AND}(\text{cols}) \neq 0$. A shared bit appears in every row and column, so every pair is excluded and the tile contributes zero. These tests cannot create an incorrect keep/drop decision. Their only possible failure is conservative. A tile may be classified as mixed and sent to the per-element path even though a finer test could have resolved it.

In practice, almost every tile is resolved by the summary test. Figure 7a reports a pooled census at $N = 65\,536$ over causal-retained 128×128 tiles. At $\texttt{itr} = 1$, the seven subproblems yield 22.0% fully masked, 77.2% clear, and 0.82% mixed. At $\texttt{itr} = 2$, the forty-nine subproblems yield 38.8% fully masked, 58.9% clear, and 2.30% mixed. Thus 97–99% of tiles never reach the per-element path. The masked/clear split varies by subproblem, especially at $\texttt{itr} = 2$, but the mixed fraction is small and decreases with sequence length.

This follows from counting. Let $n_b = \lceil L/128 \rceil$ be the number of tiles along one side of a local score matrix, so the matrix has n_b^2 tiles. Tokens within a chunk share bits, so a tile can be mixed only if it straddles a chunk boundary. A subsequence with l chunks has at most l such boundaries, and each boundary affects at most one row and one column of tiles. Mixed tiles are therefore $O(l\, n_b)$ out of $O(n_b^2)$, giving a fraction on the order of l/n_b that vanishes with sequence length.

Measurements match this scaling and stay below the bound. At $\texttt{itr} = 2$, the mixed fractions are 2.30% at $N = 65\,536$ ($n_b = 95$), 0.58% at $N = 262\,144$ ($n_b = 377$), and 0.15% at $N = 1\,048\,576$ ($n_b = 1505$), versus predicted bounds of 3.16%, 0.80%, and 0.20%. The bound overshoots because it charges every boundary a full tile row and column, while boundaries on tile edges cost nothing. The scaling is what matters. At $N = 1\,048\,576$, only about one tile in 670 takes the general path, so optimizing that path further could recover at most 0.15% of runtime.

Register budget. The largest single effect was register pressure. Each CUDA thread has a compile-time register allocation capped at 255 registers. If more values are live than fit, the compiler spills to CUDA local memory, which resides in off-chip DRAM. A spill inside an innermost loop turns a register read into repeated memory traffic.

FlashAttention kernels already use nearly all available registers to keep score tiles, softmax statistics, accumulators, and input fragments out of memory. Our backward kernel compiled at 254–255 registers per thread. The mask added nine values, including pointers that cost two registers each. Storing them as members of a mask object kept them live for the whole kernel and caused spills. We moved the same values into the kernel parameter block, which resides in constant memory, and

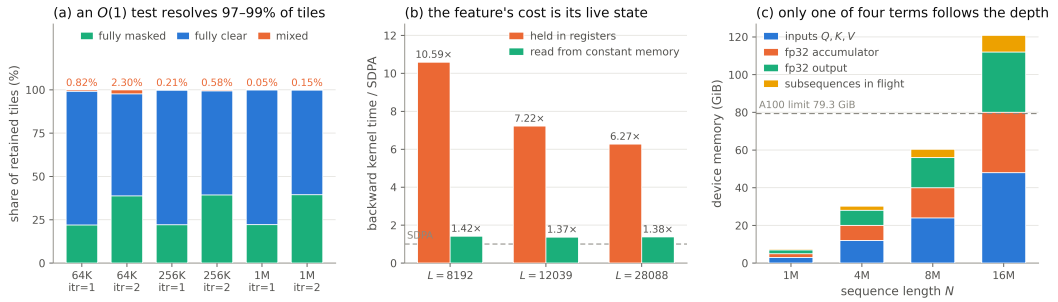


Figure 7: The three implementation decisions measured. **(a)** Census of the 128×128 tiles that causal masking retains. Because the CQS mask is block-structured, the $O(1)$ summary test resolves 97–99% of them, and the per-element path runs on the small remainder printed above each bar. That remainder falls as N grows, since a tile can be mixed only where it straddles a chunk boundary. **(b)** Backward kernel time relative to SDPA at identical shape, measured with an all-zero CQS mask so that what is isolated is the cost of carrying the feature rather than of any masking it performs. Holding the mask state as object members pushed a register-saturated kernel into local-memory spill. Reading the same state from constant memory restores near parity, with the masking logic unchanged. **(c)** The four $O(N)$ contributions to device memory. Only the in-flight subsequences shrink as `itr` grows, so raising the depth alone cannot avert an out-of-memory failure once the other three dominate, which is the observation that governs the recovery behavior in Section 4.4.

read them only at use sites. The masking logic did not change. At $L = 8\,192$, backward time fell from $10.59\times$ SDPA to $1.42\times$ (Figure 7b). At $L = 28\,088$, it fell from $6.27\times$ to $1.38\times$. In a register-saturated kernel, the cost of a feature can be dominated by live state rather than arithmetic.

Skipping fully-masked tiles. A fully masked tile contributes zero, making its five GEMMs redundant. In the backward pass, however, the m-block loop also performs stateful work that must still run. It accumulates and writes back `dq_accum`, swaps shared-memory double buffers, issues `cp.async` prefetches, and reaches synchronization barriers. We therefore guard the GEMMs, mask application, and exponential individually while leaving stateful steps active. This is safe because the tile verdict depends only on block indices and summaries, not thread indices, so every thread in the block takes the same branch and reaches the same barriers.

Correctness does not require every downstream guard to fire. Accumulators are cleared at the loop head. If the first GEMM is skipped, the score block S remains zero. If the exponential is skipped, the softmax weights P remain zero. The score gradient is $\mathbf{d}S = P \odot (\mathbf{d}P - D)$, so $\mathbf{d}S = 0$ regardless of $\mathbf{d}P$ or the per-row correction D. Any remaining GEMMs that use $\mathbf{d}S$ then add zero. This explains why, measured in isolation, a real CQS mask is $1.42\times$ faster than an all-zero mask at $N = 262\,144$ and `itr` $= 2$. The all-zero mask excludes nothing, so no tile is fully masked and no arithmetic can be skipped.

A.3 Scheduling, Occupancy and Residency

Subproblems are independent and run on separate CUDA streams. The useful number in flight is limited by memory and device saturation. The kernel assigns one thread block to each 128-query strip for every head and batch element, so a subsequence of length L issues about $\lceil L/128 \rceil \cdot B \cdot H$ thread blocks. Once that count exceeds the number of streaming multiprocessors, another stream cannot add arithmetic throughput, though it can still overlap gather and transfer with compute. At the relevant sizes the device is heavily oversubscribed. An A100 has 108 SMs, while one subproblem at $N = 32\,768$ and $H = 8$ issues 880 blocks. The scheduler therefore caps concurrency at two streams whenever one task alone issues at least twice the SM count, and allows more only for smaller tasks. We also gather chunks in ascending order so local token order matches global order. The unmodified FlashAttention causal mask is then correct and composes with the CQS mask by conjunction.

In this kernel, `n_cap` is chosen for memory rather than throughput because one subproblem already saturates the device. Calibration admits as many resident subsequences as the budget allows, which

is seven or eight at shorter measured lengths and one from $N = 2\,097\,152$ onward. In the recovery regime, `n_cap` $= 1$ and the guardrail reduces to the depth rule. The branch that decreases depth requires all subproblems to be resident with spare memory. That condition never occurs in our measurements, so reported depth is always the first successful depth.

Depth can be explicit or automatic. An explicit depth bypasses the guardrail. If it OOMs, the failure is reported. This is how fixed-depth rows in Section 4 are measured, so a row labeled `itr` $= 1$ reports that depth even where it does not fit. Automatic mode begins at `itr` $= 1$ and applies the guardrail, reaching the smallest depth at which no subproblem exhausts memory.

The other axis is residency, meaning where tensors are kept. Residency does not change the computation, only device memory and therefore maximum sequence length. Four $O(N)$ quantities can occupy device memory (Figure 7c), including inputs Q, K, V, the fp32 accumulator, the returned output, and in-flight subsequences. Only in-flight subsequences shrink with depth. The other terms shrink only when moved to host memory. These terms set a device-memory floor that no decomposition depth can cross.

The device-resident boundary depends on memory budget, not a fixed N. At $B = 1$, $H = 8$, and $D = 64$, forward–backward at `itr` $= 1$ uses about 13.6 GiB per million tokens, placing the boundary near 5.8M on the 79.3 GiB available here. This matches the observed pattern. 4M completes and 8M does not. The same estimate gives roughly 2.9M on a 40 GiB card and 10M on a 140 GiB card. Forward–backward sets the boundary because a training step needs both passes.

Below that boundary, the device-resident configuration is faster. Above it, host residency becomes necessary, and its time cost is already small. Moving the accumulator to host reduces peak device memory by $2.50\times$ in the forward pass and $1.79\times$ in forward–backward at every measured length. Its runtime penalty shrinks with N, from $1.49\times$ at $N = 1$M to $1.05\times$ at $N = 8$M forward, and from $1.17\times$ to $1.04\times$ at $N = 4$M forward–backward. Transfer is linear while attention is quadratic, so the penalty is largest where a monolithic call would already work and only a few percent in the intended regime. We therefore use host-resident accumulators by default.

Forward–backward runs out first because it must accumulate three full-sequence fp32 tensors, one each for $\mathbf{d}Q$, $\mathbf{d}K$, and $\mathbf{d}V$. The forward pass accumulates one fp32 $[B, N, H, D]$ tensor plus row statistics. The backward pass needs three such tensors, or 12 bytes per element of $[B, N, H, D]$ versus the forward's 4. Deeper decomposition cannot shrink these full-sequence tensors, so the backward boundary arrives while forward still has room.

B Additional reference-kernel experiments

This appendix tests whether recovery is a framework property rather than an artifact of the native kernel. We convert two reference PyTorch kernels to the Stream-CQSA interface, an exact dense kernel and an approximate LongNet-style kernel (Ding et al., 2023). We write the unwrapped kernels as *dense* and *LongNet*, and the wrapped paths as $\mathcal{S}(\text{dense})$ and $\mathcal{S}(\text{LongNet})$. These implementations are not optimized. They establish compatibility requirements and recovery behavior. Each local subproblem must accept the combined causal and CQS mask, and the inner kernel must expose local numerator/denominator statistics for recomposition. They intentionally follow the algorithms directly, without the overflow-avoidance mechanisms of Section A.1, so the reported speed, memory, and numerical behavior should be read as properties of these reference paths rather than best-performance claims.

Unless otherwise stated, these experiments use causal attention with $B = 1$, $H = 8$, $D = 64$, bf16 precision, and one NVIDIA A100 80 GiB GPU. Synthetic Q/K/V tensors use fixed random seeds, are drawn from a zero-mean Gaussian scaled by 0.05, and are cast to the experiment precision. All Stream-CQSA paths use $c = 7$, $\mathcal{I} = (0, 1, 3)$, and `itr` $= 1$.

Output preservation Making a kernel Stream-CQSA compatible should not change its attention result. We compare each wrapped path with its unwrapped counterpart for forward outputs and backward gradients, using $N = 4096$ and 20 random Q/K/V draws per configuration. Forward errors cover the full output tensor $[B, N, H, D]$. Backward errors are computed jointly over $\mathbf{d}Q$, $\mathbf{d}K$, and $\mathbf{d}V$.

Table 3 shows that both reference kernels are preserved within numerical precision after decomposition and recomposition. Table 4 breaks backward errors down by gradient tensor. The largest low-precision maximum errors come from $\mathbf{d}V$, especially in bf16, because $\mathbf{d}V = P^\top \mathbf{d}O$ accumulates unit-scale upstream gradients over many query positions. fp16 is more vulnerable to overflow in the direct exponential, while bf16 has wider exponent range but coarser mantissa precision. Mean absolute errors remain small ($\approx 10^{-5}$ for bf16 and $\approx 10^{-6}$ for fp16), and fp32 errors are near numerical noise. These runs deliberately omit the max-shifted merge of Section A.1. The visible low-precision errors motivate that implementation requirement.

Table 3: Forward-output and backward-gradient preservation across precision settings.

Pair	Pass	Precision	Max abs. error	Mean abs. error
dense vs $\mathcal{S}$(dense)	fwd	fp32	$(1.56 \pm 0.20) \times 10^{-8}$	$(7.99 \pm 0.16) \times 10^{-10}$
		bf16	$(1.56 \pm 0.19) \times 10^{-8}$	$(7.98 \pm 0.14) \times 10^{-10}$
		fp16	$(1.51 \pm 0.10) \times 10^{-8}$	$(7.99 \pm 0.13) \times 10^{-10}$
	bwd	fp32	$(9.26 \pm 1.60) \times 10^{-6}$	$(6.11 \pm 0.08) \times 10^{-9}$
		bf16	$(1.88 \pm 0.64) \times 10^{-2}$	$(1.08 \pm 0.01) \times 10^{-5}$
		fp16	$(2.25 \pm 0.72) \times 10^{-3}$	$(1.35 \pm 0.02) \times 10^{-6}$
LongNet vs $\mathcal{S}$(LongNet)	fwd	fp32	$(1.17 \pm 0.12) \times 10^{-8}$	$(5.66 \pm 0.04) \times 10^{-10}$
		bf16	$(1.18 \pm 0.15) \times 10^{-8}$	$(5.66 \pm 0.04) \times 10^{-10}$
		fp16	$(1.14 \pm 0.08) \times 10^{-8}$	$(5.67 \pm 0.03) \times 10^{-10}$
	bwd	fp32	$(3.23 \pm 0.69) \times 10^{-6}$	$(4.25 \pm 0.03) \times 10^{-9}$
		bf16	$(1.72 \pm 0.48) \times 10^{-2}$	$(1.86 \pm 0.01) \times 10^{-5}$
		fp16	$(2.25 \pm 0.72) \times 10^{-3}$	$(2.33 \pm 0.02) \times 10^{-6}$

Table 4: Backward-gradient preservation decomposed by gradient tensor. Errors are computed after converting both tensors to fp32.

Pair	Precision	$\mathbf{d}Q$ max abs. error	$\mathbf{d}K$ max abs. error	$\mathbf{d}V$ max abs. error
dense vs $\mathcal{S}$(dense)	fp32	$(2.43 \pm 0.63) \times 10^{-9}$	$(1.70 \pm 0.46) \times 10^{-8}$	$(9.26 \pm 1.60) \times 10^{-6}$
	bf16	$(3.15 \pm 2.08) \times 10^{-6}$	$(3.36 \pm 0.94) \times 10^{-5}$	$(1.88 \pm 0.64) \times 10^{-2}$
	fp16	$(7.87 \pm 5.36) \times 10^{-7}$	$(3.91 \pm 0.97) \times 10^{-6}$	$(2.25 \pm 0.72) \times 10^{-3}$
LongNet vs $\mathcal{S}$(LongNet)	fp32	$(1.30 \pm 0.40) \times 10^{-9}$	$(5.41 \pm 1.31) \times 10^{-9}$	$(3.23 \pm 0.69) \times 10^{-6}$
	bf16	$(4.01 \pm 0.85) \times 10^{-6}$	$(3.20 \pm 0.68) \times 10^{-5}$	$(1.72 \pm 0.48) \times 10^{-2}$
	fp16	$(5.25 \pm 1.47) \times 10^{-7}$	$(4.01 \pm 0.85) \times 10^{-6}$	$(2.25 \pm 0.72) \times 10^{-3}$

Pair	Precision	$\mathbf{d}Q$ mean abs. error	$\mathbf{d}K$ mean abs. error	$\mathbf{d}V$ mean abs. error
dense vs $\mathcal{S}$(dense)	fp32	$(4.96 \pm 0.03) \times 10^{-11}$	$(5.48 \pm 0.02) \times 10^{-11}$	$(1.82 \pm 0.02) \times 10^{-8}$
	bf16	$(7.56 \pm 0.03) \times 10^{-8}$	$(7.91 \pm 0.03) \times 10^{-8}$	$(3.21 \pm 0.03) \times 10^{-5}$
	fp16	$(1.71 \pm 0.02) \times 10^{-8}$	$(1.57 \pm 0.03) \times 10^{-8}$	$(4.01 \pm 0.06) \times 10^{-6}$
LongNet vs $\mathcal{S}$(LongNet)	fp32	$(3.06 \pm 0.003) \times 10^{-11}$	$(3.31 \pm 0.005) \times 10^{-11}$	$(1.27 \pm 0.01) \times 10^{-8}$
	bf16	$(1.46 \pm 0.002) \times 10^{-7}$	$(1.37 \pm 0.004) \times 10^{-7}$	$(5.56 \pm 0.02) \times 10^{-5}$
	fp16	$(2.20 \pm 0.003) \times 10^{-8}$	$(2.06 \pm 0.005) \times 10^{-8}$	$(6.94 \pm 0.05) \times 10^{-6}$

OOM boundary and recovery cost We compare each unwrapped kernel with its Stream-CQSA path as sequence length N increases. The wrapped paths should be slower while using less GPU memory. The question is whether they continue after the unwrapped implementation reaches its OOM boundary.

Figure 8 and Table 5 compare the dense and LongNet kernels before and after Stream-CQSA wrapping. Peak memory is the increase in PyTorch-allocated GPU memory measured by `torch.cuda.max_memory_allocated()` during attention. This is a reproducible implementation metric, not a theoretical bound or total device-pressure measurement.

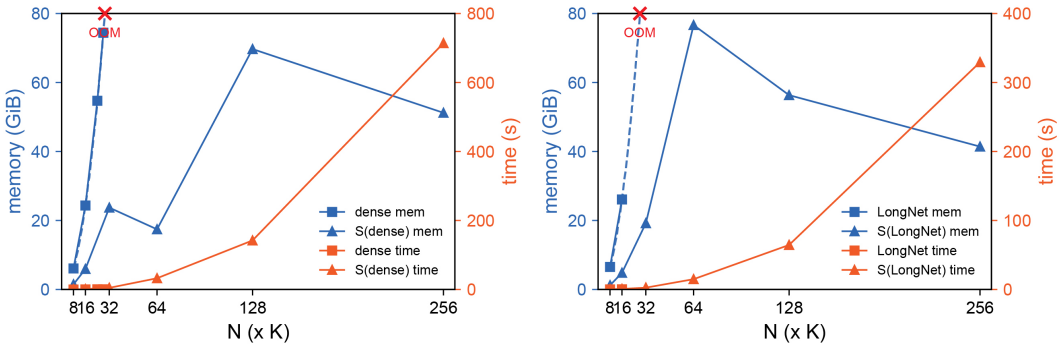


Figure 8: Peak allocated GPU memory and wall-clock time for the unwrapped reference kernels and their Stream-CQSA-compatible paths. A memory drop in a Stream-CQSA path indicates an additional decomposition level.

The exact dense reference succeeds through $N = 28672$ and OOMs at $N = 30720$, whereas $\mathcal{S}(\text{dense})$ completes the tested range through $N = 262144$ by increasing depth. LongNet succeeds through $N = 16384$ and OOMs at $N = 32768$, whereas $\mathcal{S}(\text{LongNet})$ also completes through $N = 262144$. The pattern matches the native-kernel results. Wrapping adds overhead while the original call fits and provides an executable path once it does not. Because the same recovery behavior appears for native, dense, and LongNet kernels, it belongs to the decomposition interface rather than a single implementation. Approximate attention benefits on the same terms as exact attention. It moves the OOM boundary outward but does not remove it.

Table 5: Recovery-overhead summary for the reference implementations. Peak allocated memory is the increase in PyTorch-allocated GPU memory during the measured attention computation. For Stream-CQSA, peak memory and final time correspond to the final successful decomposition attempt, with full Q/K/V kept on CPU. Adaptation overhead is total adaptive time minus final successful time.

Path	N	`itr`	Peak allocated mem. (GiB)	Final time (s)	Adaptation overhead (s)
dense	8192	–	6.11	0.045	–
	16384	–	24.34	0.179	–
	24576	–	54.70	0.400	–
	28672	–	74.43	0.546	–
	30720	–	OOM	OOM	–
$\mathcal{S}(\text{dense})$	8192	1	1.49	0.507	0.110
	16384	1	5.95	1.043	0.112
	32768	1	23.74	4.766	0.113
	65536	2	17.45	32.597	17.105
	131072	2	69.70	142.132	3.883
	262144	3	51.23	714.132	6.764
LongNet	8192	–	6.55	0.077	–
	16384	–	26.09	0.292	–
	32768	–	OOM	OOM	–
	65536	–	OOM	OOM	–
	131072	–	OOM	OOM	–
	262144	–	OOM	OOM	–
$\mathcal{S}(\text{LongNet})$	8192	1	1.22	0.331	0.103
	16384	1	4.84	0.780	0.113
	32768	1	19.22	2.680	0.137
	65536	1	76.66	14.978	0.105
	131072	2	56.35	64.565	5.397
	262144	3	41.43	329.482	8.519

Both wrapped reference paths keep full Q/K/V tensors in host memory and transfer one current subsequence to the GPU at a time. Local numerator/denominator statistics return to host memory

Table 6: Forward and backward profile at $N = 65536$ for the reference Stream-CQSA paths.

Path	Pass	Time (s)	Peak mem. (GiB)
$\mathcal{S}$(dense)	fwd	3.70 ± 0.11	58.74
	bwd	1.00 ± 0.03	57.24
$\mathcal{S}$(LongNet)	fwd	2.04 ± 0.37	55.54
	bwd	0.69 ± 0.06	51.93

for recomposition in global token coordinates. This conservative policy does not reuse overlapping tensors or pipeline transfers with compute, so it exposes near worst-case overhead for the reference framework implementation.

Forward vs. backward We also compare forward and backward phases for the two reference Stream-CQSA paths. The forward phase uses `requires_grad=True`, so the autograd graph required by backward is constructed. The backward phase is measured as `loss.backward()` for `loss` $= (O$`.float()` $\odot \mathbf{d}O)$`.sum()`. This experiment uses fp16, $B = 1$, $H = 1$, $D = 128$, causal masking, $c = 7$, $\mathcal{I} = (0, 1, 3)$, and `itr` $= 1$.

Table 6 reports $N = 65536$, the largest length in this profiling run. In these reference implementations, backward is not the dominant runtime cost. It takes $0.27\times$ the forward time for $\mathcal{S}$(dense) and $0.34\times$ for $\mathcal{S}$(LongNet). Peak memory is closer, at $0.97\times$ and $0.94\times$ of forward peak memory, respectively. These ratios are implementation-specific and depend on how the inner kernel saves or recomputes forward state and executes backward.

C CQS INTEREST SET

An interest set $\mathcal{I}$ ensures full coverage of pairwise interactions. Table 7 lists examples for several l values, and Figure 9 validates full coverage for $c = 13$ with $\mathcal{I} = (0, 1, 3, 9)$.

Interest sets are not unique and occur in pairs (Bian & Somani, 2021). Given $(0, 1, a_2, a_3, \ldots, a_{l-1})$, the paired interest set is $(0, 1, c + 1 - a_{l-1}, c + 1 - a_{l-2}, \ldots, c + 1 - a_2)$. For $c = 7$, the paired interest set of $(0, 1, 3)$ is $(0, 1, 5)$.

Interest sets do not exist for all l values, including $l = 7$ and $l = 11$ in Table 7. Formally, an interest set corresponds to a (v, k, λ)-*difference set* with $\lambda = 1$. This is a k-subset $D \subseteq G$ of an abelian group G of order v such that each nonzero $g \in G$ appears exactly λ times in the multiset of differences $(x - y \mid x, y \in D)$ (Van Lint & Wilson, 2001). In our notation, $k = l$ and $v = c$.

Let $q = l - 1$. Existence of a cyclic $(q^2 + q + 1, q + 1, 1)$-*difference set* is equivalent to a projective plane of order q with a cyclic automorphism containing one cycle of length $q + 1$ (Van Lint & Wilson, 2001). Such difference sets exist for prime-power q by the Singer construction (Singer, 1938). For non-prime-power q, existence remains open in many cases and impossible in some.

Lam's problem asks whether a finite projective plane of order 10 exists. By earlier structural results (MacWilliams et al., 1973), this is equivalent to asking whether an interest set of size $l = 11$ exists for $c = 111$. Lam (1991) answered no by exhaustive computational search.

D CHAIN-RULE DEDUCTIONS IN BACKWARD PASS

Given upstream gradient $\mathbf{d}O = \frac{\partial \mathcal{L}}{\partial O}$ for loss $\mathcal{L}$, the goal is to compute $\mathbf{d}Q$, $\mathbf{d}K$, and $\mathbf{d}V$.

For subsequence i, the forward pass generates

$$R_i = \alpha(Q_i K_i^\top) \quad P_i = \exp(R_i) \odot M_i \quad \mathrm{Num}_i = P_i V_i \quad \mathrm{Den}_i = \mathrm{row_sum}(P_i) \tag{10}$$

where M_i is the CQS mask.

The global merge produces

$$\mathrm{Num} = \sum_{i \in S} \mathrm{scatter}(\mathrm{Num}_i) \quad \mathrm{Den} = \sum_{i \in S} \mathrm{scatter}(\mathrm{Den}_i) \quad O = \frac{\mathrm{Num}}{\mathrm{Den}} \tag{11}$$

Table 7: Example interest sets for several l values.

c	l	q	prime power	$\mathcal{I}$
7	3	2	2^1	$(0, 1, 3)$
13	4	3	3^1	$(0, 1, 3, 9)$
21	5	4	2^2	$(0, 1, 4, 14, 16)$
31	6	5	5^1	$(0, 1, 3, 8, 12, 18)$
43	7	6	NA	None
57	8	7	7^1	$(0, 1, 3, 13, 32, 36, 43, 52)$
73	9	8	2^3	$(0, 1, 3, 7, 15, 31, 36, 54, 63)$
91	10	9	3^2	$(0, 1, 3, 9, 27, 49, 56, 61, 77, 81)$
111	11	10	NA	None
133	12	11	11^1	$(0, 1, 3, 12, 20, 34, 38, 81, 88, 94, 104, 109)$

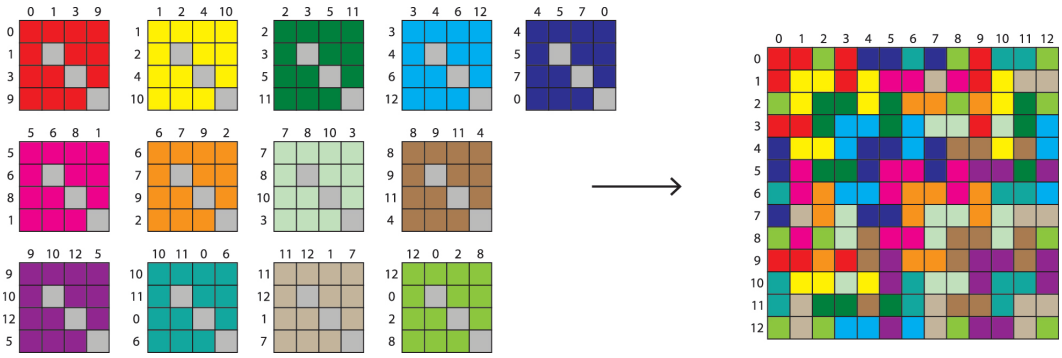


Figure 9: Full coverage validation for $c = 13$ and $\mathcal{I} = (0, 1, 3, 9)$.

The partial derivatives are

$$\frac{\partial O}{\partial \text{Num}} = \frac{1}{\text{Den}} \qquad \frac{\partial O}{\partial \text{Den}} = -\frac{\text{Num}}{\text{Den}^2} \tag{12}$$

Below, batch and head dimensions are omitted for simplicity. From the global view,

$$\mathbf{d}O[n, d] = \frac{\partial \mathcal{L}}{\partial O[n, d]} \tag{13}$$

where $\mathbf{d}O[n, d]$ is a scalar from $\mathbf{d}O \in \mathbb{R}^{N \times D}$. Then

$$\mathbf{d}\text{Num}[n, d] = \frac{\partial \mathcal{L}}{\partial \text{Num}[n]} = \frac{\partial \mathcal{L}}{\partial O[n, d]} \times \frac{\partial O[n, d]}{\partial \text{Num}[n, d]} = \frac{\mathbf{d}O[n, d]}{\text{Den}[n]} \tag{14}$$

$$\begin{aligned} \mathbf{d}\text{Den}[n] &= \frac{\partial \mathcal{L}}{\partial \text{Den}[n]} = \frac{\partial \mathcal{L}}{\partial O[n, d]} \times \frac{\partial O[n, d]}{\partial \text{Den}[n]} \\ &= \sum_d \mathbf{d}O[n, d](-\frac{\text{Num}[n, d]}{\text{Den}[n]^2}) = -\frac{\sum_d \mathbf{d}O[n, d]\text{Num}[n, d]}{\text{Den}[n]^2} \end{aligned} \tag{15}$$

A compact version of Eqs. 14 and 15 is

$$\mathbf{d}\text{Num} = \frac{\mathbf{d}O}{\text{Den}} \qquad \mathbf{d}\text{Den} = -\frac{\sum_d \mathbf{d}O_d \cdot \text{Num}_d}{\text{Den}^2} \tag{16}$$

For subsequence i of length L, gathering from global $\mathbf{d}\text{Num} \in \mathbb{R}^{N \times D}$ and $\mathbf{d}\text{Den} \in \mathbb{R}^N$ gives $\mathbf{d}\text{Num}_i \in \mathbb{R}^{L \times D}$ and $\mathbf{d}\text{Den}_i \in \mathbb{R}^L$. The local computations are

$$\mathbf{d}V_i[m, d] = \sum_l P_i[l, m]\mathbf{d}\text{Num}_i[l, d] \tag{17}$$

$$\mathbf{d}P_i[l, m] = \sum_d \mathbf{d}\text{Num}_i[l, d]V_i[m, d] + \mathbf{d}\text{Den}_i[l] \tag{18}$$

Table 8: Backward error by gradient tensor ($N = 8192$, mean over 10 draws).

Method	fp16			bf16		
	$\mathbf{d}Q$	$\mathbf{d}K$	$\mathbf{d}V$	$\mathbf{d}Q$	$\mathbf{d}K$	$\mathbf{d}V$
SDPA	3.075×10^{-4}	3.029×10^{-4}	2.893×10^{-4}	2.458×10^{-3}	2.421×10^{-3}	2.312×10^{-3}
SDPA (mem-eff.)	3.739×10^{-4}	3.679×10^{-4}	2.900×10^{-4}	2.980×10^{-3}	2.929×10^{-3}	2.313×10^{-3}
FlashAttention-2	3.075×10^{-4}	3.028×10^{-4}	2.893×10^{-4}	2.458×10^{-3}	2.421×10^{-3}	2.312×10^{-3}
$\mathcal{S}$ ITR=0 (undecomposed)	3.075×10^{-4}	3.028×10^{-4}	2.893×10^{-4}	2.458×10^{-3}	2.421×10^{-3}	2.312×10^{-3}
$\mathcal{S}$ ITR=1	3.080×10^{-4}	3.024×10^{-4}	2.891×10^{-4}	2.462×10^{-3}	2.424×10^{-3}	2.311×10^{-3}
$\mathcal{S}$ ITR=2	3.085×10^{-4}	3.023×10^{-4}	2.884×10^{-4}	2.465×10^{-3}	2.419×10^{-3}	2.311×10^{-3}

$$\mathbf{d}R_i[l,m] = \mathbf{d}P_i[l,m] \cdot P_i[l,m] \tag{19}$$

$$\mathbf{d}Q_i[l,d] = \alpha \sum_m \mathbf{d}R_i[l,m] K_i[m,d] \tag{20}$$

$$\mathbf{d}K_i[m,d] = \alpha \sum_l \mathbf{d}R_i[l,m] Q_i[l,d] \tag{21}$$

A compact version of Eqs. 17–21 is

$$\mathbf{d}V_i = P_i^\top \mathbf{d}\mathrm{Num}_i \tag{22}$$

$$\mathbf{d}P_i = \mathbf{d}\mathrm{Num}_i V_i^\top + \mathbf{d}\mathrm{Den}_i \mathbf{1}^\top \tag{23}$$

$$\mathbf{d}R_i = \mathbf{d}P_i \odot P_i \tag{24}$$

$$\mathbf{d}Q_i = \alpha \mathbf{d}R_i K_i \tag{25}$$

$$\mathbf{d}K_i = \alpha \mathbf{d}R_i^\top Q_i \tag{26}$$

The last step scatters all $\mathbf{d}Q_i, \mathbf{d}K_i, \mathbf{d}V_i \in \mathbb{R}^{L\times D}$ into $\mathbf{d}Q, \mathbf{d}K, \mathbf{d}V \in \mathbb{R}^{N\times D}$.

E SUPPLEMENTARY MATERIALS

Algorithm 1 CQSA forward pass

Require: $Q, K, V \in \mathbb{R}^{B\times H\times N\times D}$, chunk count c, decomposition depth `itr`, interest set $\mathcal{I}$, scale α

```
subseq_entries ← BUILDSUBSEQ(N, c, itr, I)
Num ← 0_{B×H×N×D}
Den ← 0_{B×H×N}
for each subsequence i ∈ subseq_entries do
    idx ← token_ids[i]
    Q_i ← GATHER(Q, idx), K_i ← GATHER(K, idx), V_i ← GATHER(V, idx)
    M_i ← mask[i]
    R_i ← α(Q_i K_i^⊤)
    P_i ← exp(R_i) ⊙ M_i
    Num_i ← P_i V_i
    Den_i ← ROWSUM(P_i)
    Num.INDEXADD(2, idx, Num_i)
    Den.INDEXADD(2, idx, Den_i)
end for
O ← Num/Den.UNSQUEEZE(−1)
return O
```

Algorithm 2 CQSA backward pass. The local attention matrix P_i is recomputed from Q_i, K_i.

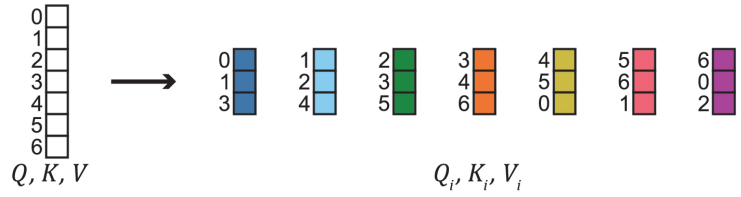
Require: $Q, K, V \in \mathbb{R}^{B\times H\times N\times D}$, upstream $\mathbf{d}O$, `subseq_entries`, Num, Den, scale α

$\mathbf{d}Q \leftarrow \mathbf{0}$, $\mathbf{d}K \leftarrow \mathbf{0}$, $\mathbf{d}V \leftarrow \mathbf{0}$
$\mathbf{d}\mathrm{Num} \leftarrow \mathbf{d}O/\mathrm{Den}.\textsc{Unsqueeze}(-1)$
$\mathbf{d}\mathrm{Den} \leftarrow -\textsc{RowSum}_D(\mathbf{d}O \odot \mathrm{Num})/(\mathrm{Den} \odot \mathrm{Den})$
for each subsequence $i \in$ `subseq_entries` **do**
  `idx` $\leftarrow$ `token_ids`$[i]$
  $Q_i \leftarrow \textsc{Gather}(Q, \mathtt{idx})$, $K_i \leftarrow \textsc{Gather}(K, \mathtt{idx})$, $V_i \leftarrow \textsc{Gather}(V, \mathtt{idx})$
  $M_i \leftarrow$ `mask`$[i]$
  $R_i \leftarrow \alpha(Q_i K_i^\top)$
  $P_i \leftarrow \exp(R_i) \odot M_i$
  $\mathbf{d}\mathrm{Num}_i \leftarrow \textsc{Gather}(\mathbf{d}\mathrm{Num}, \mathtt{idx})$, $\mathbf{d}\mathrm{Den}_i \leftarrow \textsc{Gather}(\mathbf{d}\mathrm{Den}, \mathtt{idx})$
  $\mathbf{d}V_i \leftarrow P_i^\top \mathbf{d}\mathrm{Num}_i$
  $\mathbf{d}P_i \leftarrow \mathbf{d}\mathrm{Num}_i V_i^\top + \mathbf{d}\mathrm{Den}_i \mathbf{1}^\top$
  $\mathbf{d}R_i \leftarrow \mathbf{d}P_i \odot P_i$
  $\mathbf{d}Q_i \leftarrow \alpha(\mathbf{d}R_i K_i)$
  $\mathbf{d}K_i \leftarrow \alpha(\mathbf{d}R_i^\top Q_i)$
  $\mathbf{d}Q.\textsc{IndexAdd}(2, \mathtt{idx}, \mathbf{d}Q_i)$
  $\mathbf{d}K.\textsc{IndexAdd}(2, \mathtt{idx}, \mathbf{d}K_i)$
  $\mathbf{d}V.\textsc{IndexAdd}(2, \mathtt{idx}, \mathbf{d}V_i)$
end for
return $\mathbf{d}Q, \mathbf{d}K, \mathbf{d}V$

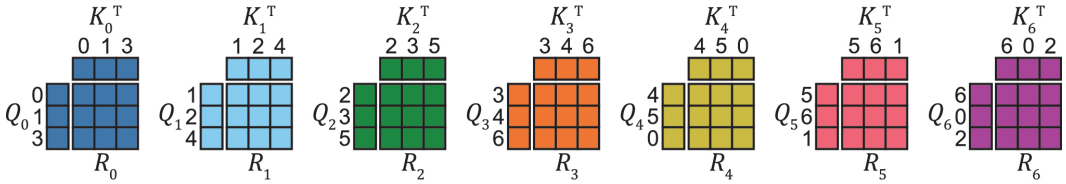
Table 9: Forward pass, fp16, on one 79.3 GiB A100. Wall-clock time (s) and peak device memory (GiB, `max_memory_allocated`) are shown from 8192 to 8 388 608, the longest forward length FlashAttention-2 completes. Parentheses give ratios to FlashAttention-2. These measurements underlie Figure 4.

	Wall-clock time (s)				Peak memory (GiB)			
		Stream-CQSA				Stream-CQSA		
N	Flash-Attention-2	ITR=1 ACC=GPU	ITR=2 ACC=GPU	ITR=AUTO* ACC=CPU	Flash-Attention-2	ITR=1 ACC=GPU	ITR=2 ACC=GPU	ITR=AUTO* ACC=CPU
8K	0.001	0.034 (34.0)	0.121 (121)	0.084 (84.0)	0.055	0.051 (0.93)	0.040 (0.73)	0.020 (0.36)
16K	0.002	0.051 (25.5)	0.161 (80.5)	0.163 (81.5)	0.102	0.109 (1.07)	0.073 (0.72)	0.048 (0.47)
32K	0.006	0.093 (15.5)	0.244 (40.7)	0.311 (51.8)	0.196	0.210 (1.07)	0.138 (0.70)	0.089 (0.45)
64K	0.027	0.169 (6.26)	0.414 (15.3)	0.669 (24.8)	0.385	0.413 (1.07)	0.268 (0.70)	0.170 (0.44)
128K	0.108	0.335 (3.10)	0.730 (6.76)	1.34 (12.4)	0.762	0.818 (1.07)	0.528 (0.69)	0.332 (0.44)
256K	0.454	0.937 (2.06)	1.34 (2.95)	3.21 (7.08)	1.52	1.63 (1.07)	1.05 (0.69)	0.656 (0.43)
512K	1.86	3.42 (1.84)	3.71 (2.00)	7.63 (4.10)	2.52	3.25 (1.29)	2.09 (0.83)	1.30 (0.52)
1M	7.89	14.2 (1.80)	14.8 (1.88)	21.2 (2.68)	5.04	6.49 (1.29)	4.17 (0.83)	2.60 (0.52)
2M	32.3	52.5 (1.62)	65.2 (2.02)	66.4 (2.06)	10.1	13.0 (1.29)	8.34 (0.83)	5.19 (0.52)
4M	130	203 (1.56)	240 (1.85)	228 (1.76)	20.1	25.9 (1.29)	16.7 (0.83)	10.4 (0.52)
8M	532	802 (1.51)	914 (1.72)	846 (1.59)	40.3	51.8 (1.29)	33.3 (0.83)	20.7 (0.52)

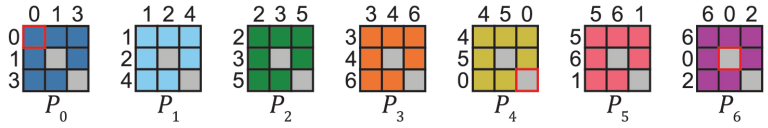
Step 1. Construct 7 subsequences and gather chunks for Q_i, K_i and V_i accordingly



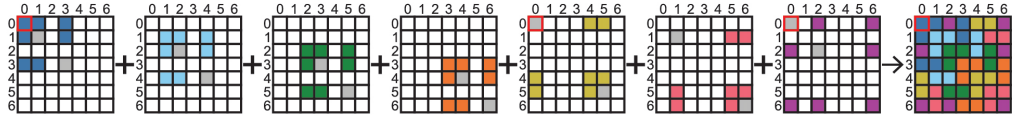
Step 2. Compute $R_i = Q_i K_i^{\mathrm{T}}$



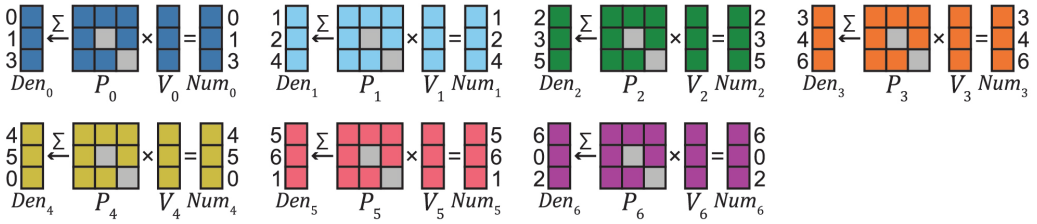
Step 3. Compute the exponential of R_i and apply the CQS mask $P_i = \exp(R_i) \odot M_i$



Validation of full coverage from the global view



Step 4. Compute row-wise sum $Den_i = \sum_j (P_i)_{:,j}$ and $Num_i = P_i V_i$



Step 5. Chunk-wise accumulation of Num_i and Den_i, and compute $O = Num \,/\, Den$

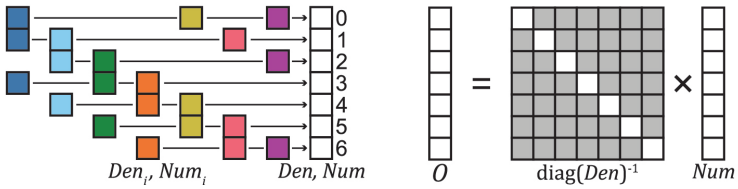
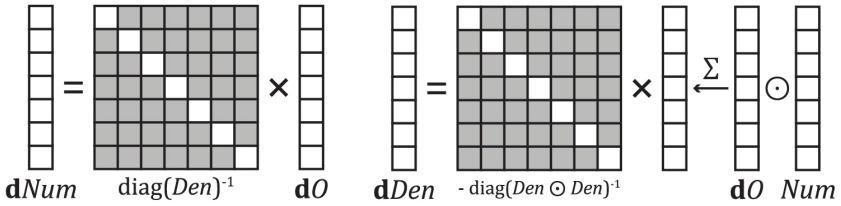


Figure 10: CQSA forward pass with $c = 7$ and $\mathcal{I} = (0, 1, 3)$. Step 3 highlights chunk pair $(0, 0)$, which appears in subsequences 0, 4, and 6. The CQS mask M_i keeps exactly one contribution and masks the rest. The global attention view confirms that every chunk pair is covered once. In step 5, Num and Den are accumulated the same way but have dimensions $N \times D$ and $N \times 1$, respectively.

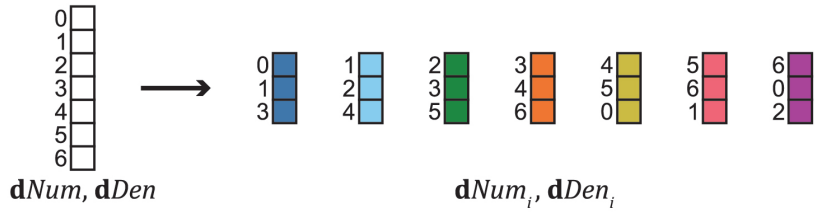
Step 1. Reuse *Num, Den* in the forward pass

Step 2. Compute $\mathbf{d}Num$, $\mathbf{d}Den$

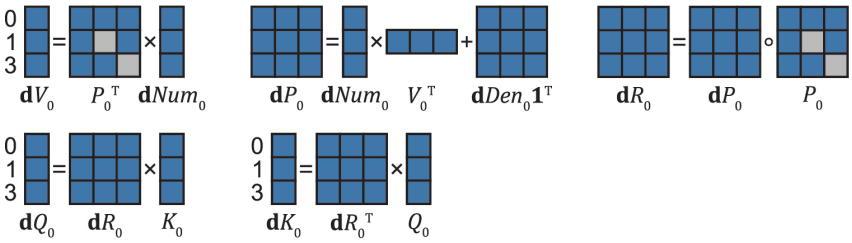


Step 3. Recompute $R_i = Q_i K_i^{\mathrm{T}}$, $P_i = \exp(R_i) \odot M_i$, $Den_i = \sum_j (P_i)_{:,j}$, $Num_i = P_i V_i$

Step 4. Gather chunks to construct $\mathbf{d}Den_i$, $\mathbf{d}Num_i$



Step 5. Compute $\mathbf{d}V_i$, $\mathbf{d}P_i$, $\mathbf{d}R_i$, $\mathbf{d}Q_i$, $\mathbf{d}K_i$



Step 6. Chunk-wise accumulation of $\mathbf{d}Q_i$, $\mathbf{d}K_i$, $\mathbf{d}V_i$

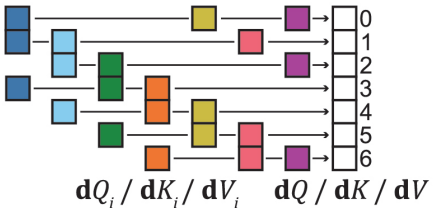


Figure 11: CQSA backward pass. Step 5 shows only Seq_0 because all subsequences follow the same pattern. $\mathbf{d}\mathrm{Num} \in \mathbb{R}^{N \times D}$ and $\mathbf{d}\mathrm{Den} \in \mathbb{R}^{N \times 1}$.

Algorithm 3 BUILDSUBSEQ$(N, c, \mathrm{itr}, \mathcal{I})$

Require: sequence length N, chunk count c, decomposition depth itr, interest set $\mathcal{I}$
Ensure: `subseq_entries`, where each entry has `token_ids` and local mask M_i

`subseq_entries` $\leftarrow [\,]$
for each quorum tuple $i = (q_1, \ldots, q_{\mathrm{itr}}) \in \{0, \ldots, c-1\}^{\mathrm{itr}}$ **do**
  `token_ids` $\leftarrow [0, 1, \ldots, N-1]$
  `label_history` $\leftarrow [\,]$, `chunks_history` $\leftarrow [\,]$
  for $t = 1$ to itr **do**
    $L \leftarrow |$`token_ids`$|$
    (`starts`, `ends`) $\leftarrow$ BALANCEDCHUNKLAYOUT(L, c)
    `chunks` $\leftarrow \{(q_t + o) \bmod c \mid o \in \mathcal{I}\}$ (ordered by $\mathcal{I}$)
    `labels`$[\ell] \leftarrow$ chunk id of local index $\ell \in [0, L)$
    `gather_idx` $\leftarrow$ Concat$\big([\,$`starts`$[u] \ldots$ `ends`$[u)\ \forall u \in$ `chunks`$\,]\big)$
    for $s = 1$ to $|$`label_history`$|$ **do**
      `label_history`$[s] \leftarrow$ `label_history`$[s]$[`gather_idx`]
    end for
    append `labels`[`gather_idx`] to `label_history`
    `token_ids` $\leftarrow$ `token_ids`[`gather_idx`]
    append $(q_t,$ `chunks`$)$ to `chunks_history`
  end for
  `group_runs` $\leftarrow [\,]$
  for $t = 1$ to itr **do**
    (`owner`, `chunks`) $\leftarrow$ `chunks_history`$[t]$
    `labels` $\leftarrow$ `label_history`$[t]$
    for each `chunk` $\in$ `chunks` with `chunk` $\neq$ `owner` **do**
      `idx` $\leftarrow \{\ell \mid$ `labels`$[\ell] =$ `chunk`$\}$
      `runs` $\leftarrow$ INDICESTORUNS(`idx`)
      if `runs` $\neq \emptyset$ **then** append `runs` to `group_runs`
      end if
    end for
  end for
  `group_runs` $\leftarrow$ UNIQUE(`group_runs`)
  $M_i \leftarrow$ LOCALMASKFROMGROUPRUNS($|$`token_ids`$|$, `group_runs`)
  append $\{$`quorum_idx` $= i$, `token_ids`, `mask` $= M_i\}$ to `subseq_entries`
end for
return `subseq_entries`

Table 10: Corresponding forward–backward statistics. As defined in the Metrics paragraph, these measurements time the forward pass followed by gradient computation and therefore represent a full training step.

	Wall-clock time (s)				Peak memory (GiB)			
		Stream-CQSA				Stream-CQSA		
N	Flash-Attention-2	ITR=1 ACC=GPU	ITR=2 ACC=GPU	ITR=AUTO* ACC=CPU	Flash-Attention-2	ITR=1 ACC=GPU	ITR=2 ACC=GPU	ITR=AUTO* ACC=CPU
8K	0.002	0.063 (31.5)	0.245 (122)	0.121 (60.5)	0.110	0.107 (0.97)	0.082 (0.75)	0.060 (0.55)
16K	0.006	0.088 (14.7)	0.309 (51.5)	0.219 (36.5)	0.212	0.213 (1.00)	0.163 (0.77)	0.119 (0.56)
32K	0.024	0.154 (6.42)	0.429 (17.9)	0.406 (16.9)	0.416	0.426 (1.02)	0.326 (0.78)	0.238 (0.57)
64K	0.095	0.349 (3.67)	0.696 (7.33)	0.883 (9.29)	0.824	0.852 (1.03)	0.652 (0.79)	0.475 (0.58)
128K	0.380	1.03 (2.71)	1.39 (3.65)	3.08 (8.11)	1.64	1.71 (1.04)	1.30 (0.79)	0.950 (0.58)
256K	1.57	4.56 (2.91)	5.00 (3.19)	7.36 (4.70)	3.27	3.41 (1.04)	2.61 (0.80)	1.90 (0.58)
512K	6.20	14.8 (2.39)	16.4 (2.64)	20.6 (3.32)	5.04	6.82 (1.35)	5.22 (1.04)	3.80 (0.75)
1M	25.3	56.2 (2.22)	65.5 (2.59)	66.0 (2.61)	10.1	13.6 (1.35)	10.4 (1.04)	7.60 (0.75)
2M	101	216 (2.13)	256 (2.52)	235 (2.31)	20.1	27.3 (1.35)	20.9 (1.04)	15.2 (0.76)
4M	407	847 (2.08)	983 (2.41)	885 (2.17)	40.3	54.5 (1.35)	41.7 (1.04)	30.4 (0.76)

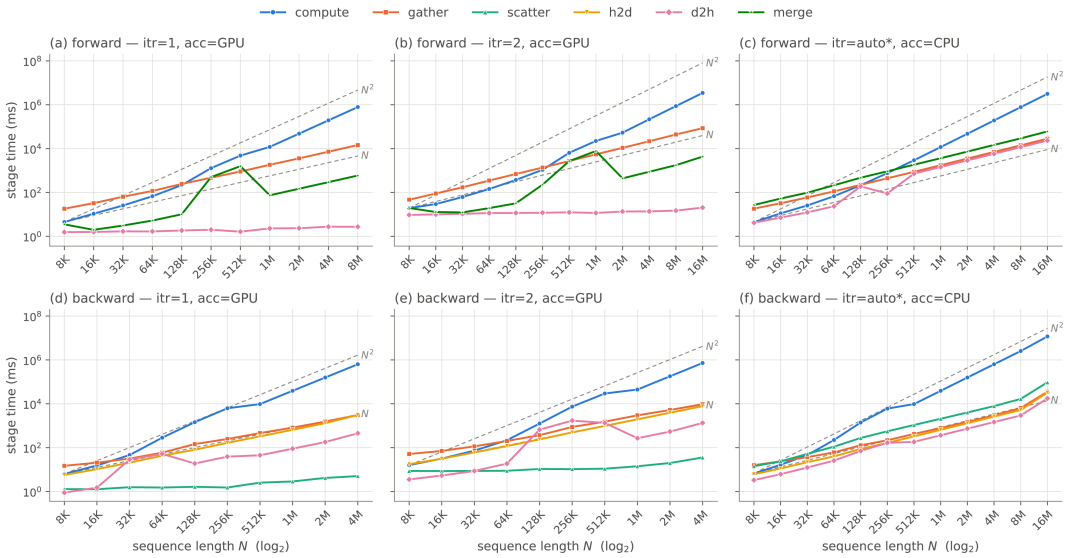


Figure 12: The measurements of Figure 6 in absolute time on log-log axes. Grey guides of slope one and two are anchored to the local kernel at the shortest length to mark linear and quadratic growth in N. They are references, not fits. Time queued behind another stream is excluded because it runs concurrently with the charged stage. As in Figure 6, backward panels cover the backward pass alone.