A Head Group Fusion for Grouped-Query Attention

Grouped-Query Attention (GQA) (Ainslie et al., 2023) allows multiple query heads to share the same key-value (KV) heads. A straightforward implementation that assigns distinct GPU threadblocks to each query head leaves much of the potential KV-Cache reuse underutilized when the query length is short. To address this limitation, FlashInfer offers a *head-group fusion* strategy: different KV heads are mapped to individual threadblocks, while query heads are fused with the query length dimension. This fusion scheme is illustrated in Figure 11, which shows how the fused row index relates to the original row index and the head indices. By merging the query-head dimension with the row dimension in the threadblock mapping, a single shared-memory load of the KV-Cache suffices for all query heads in the group, leading to better memory reuse and improved throughput for GQA operations.

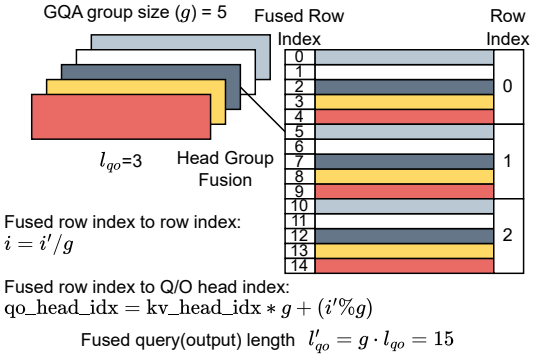

Figure 11. FlashInfer's head-group fusion of query heads with the query length dimension in GQA.

We prefer head-group fusion primarily for short query lengths. When the query length is sufficiently large, the query dimension itself yields enough workload to effectively utilize the KV-Cache, making head-group fusion less critical. Similar ideas have also been explored in other frameworks, such as XQA (NVIDIA, 2024b) in TensorRT-LLM (NVIDIA, 2023a).

B Overhead of Sparse Gathering

In Section 3.2.1, we detailed the design of FlashInfer's sparse loading module, which transfers sparse rows from global memory into contiguous shared memory. Here, we measure the performance overhead associated with sparse gathering in FlashInfer for both *decode* and *prefill* kernels.

Figure 12 compares achieved throughput in both prefill and

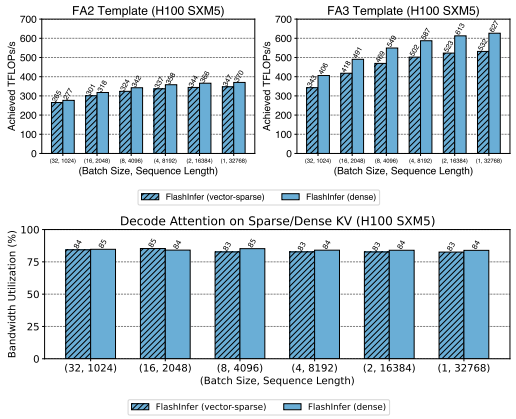

Figure 12. **Top:** Achieved TFLOPs/s for (causal) prefill attention kernels on FA2/FA3 templates with both dense/sparse KV-Cache. **Bottom:** Achieved bandwidth utilization for decode attention kernels for both dense/sparse KV-Cache. We use PageAttention with page size 1 (vector-sparse) for sparse KV-Cache. The x-axis shows various batch sizes and sequence lengths.

decode kernels for sparse and dense (contiguous) KV-Cache. For the prefill kernels, we measure the *causal* attention scenario, which is common in LLM serving. For contiguous KV-Cache, We use the variable-length RaggedTensor prefill attention API[8]. For sparse KV-Cache, we use the PagedKV-Cache prefill attention API[9].

The number of query heads and KV heads are both fixed at 32, head dimension is set to 128. We vary batch size and sequence length to measure the achieved throughput. For decode kernels, the performance gap between sparse and dense KV-Cache is negligible (within 1%). For prefill kernels, there is approximately a 10% performance gap.

Note that dense attention in the FA3 template uses TMA instructions (Tensor Memory Access) for key/value loading, which is unavailable for sparse gathering because Hopper Architecture's TMA only supports fixed-stride accesses, whereas sparse gathering requires arbitrary row indices. Consequently, sparse gathering on FA3 relies on Ampere-style asynchronous copy instructions and manual pointer arithmetic. This approach consumes more registers and necessitates smaller KV-tile size to avoid register spilling, leading to a slightly larger performance gap. By contrast, in the FA2 template (where both sparse and dense use Ampere's async-copy), the gap is smaller because the same tile size is used.

[8] https://docs.flashinfer.ai/api/prefill.html#flashinfer.prefill.BatchPrefillWithRaggedKVCacheWrapper

[9] https://docs.flashinfer.ai/api/prefill.html#flashinfer.prefill.BatchPrefillWithPagedKVCacheWrapper

When the block column size in a block-sparse matrix is large (e.g., 128 or greater), TMA can be used for sparse gathering since each TMA instruction operates within a single block with fixed stride. We leave this optimization for future work. However, increasing the block column size reduces the flexibility of the block-sparse format, which might not be suitable for all use cases.

C The Choice of Backend

For NVIDIA GPUs, we build FlashInfer on top of CUDA/-CUTLASS (Thakkar et al., 2023) instead of Triton (Tillet et al., 2019) for the following reasons:

1. **Advanced NVIDIA GPU Features.** CUTLASS supports specialized GPU capabilities such as warp-specialization (NVIDIA, 2024a) and TMA instructions (NVIDIA, 2022), which are experimental or unsupported in Triton at this moment.

2. **Fine-Grained Kernel Optimization.** While Triton provides tile-level abstractions, CUDA/CUTLASS affords finer control over thread-level registers. This flexibility simplifies incorporating low-level optimizations (e.g., PTX intrinsics) directly into our JIT templates, which is more challenging in Triton.

Our load-balancing scheduler design (Section 3.3.1) is largely backend-agnostic, allowing us to potentially integrate Triton in future versions of FlashInfer and to adapt our approach to other hardware platforms.

D Memory Management

FlashInfer manages a page-locked (pinned) host buffer and a device workspace buffer to store scheduler metadata and split-k partial outputs. We divide the device workspace buffer into *sections*, each corresponding to an array of either scheduler metadata or partial split-k outputs. For each plan call in the scheduler, we compute the scheduler metadata on the pinned host buffer and then issue a cudaMemcpyAsync to transfer this data into the corresponding sections of the device workspace buffer.

D.1 CUDAGraph-Compatible Workspace Layout

Once a kernel is captured by CUDA Graph, its arguments (pointers and scalars) become fixed, implying that each section of the device workspace buffer must maintain a consistent address for the entire captured graph's lifetime. Therefore, we allocate the workspace buffer to its maximum required capacity for each section, based on upper-bound estimations of scheduler metadata and partial outputs.

D.2 Split-K Writethrough Optimizations

In FlashInfer's load-balancing scheduler (Section 3.3.1), KV-splitting is only applied to requests that have large KV lengths. Requests with short KV lengths do not require splitting and hence have no reduction step from partial output. To save both computation and workspace memory, these small requests can write their partial outputs directly to the final output buffer (bypassing the device workspace buffer). This approach reduces both the required workspace size and the computational load within the contraction kernel.

D.3 Workspace Buffer Size Estimation

The workspace buffer size depends on two main factors: (1) the required space for scheduler metadata, and (2) the required space for storing partial split-k outputs.

Scheduler Metadata. The maximum size of each metadata section is derived from the largest possible number of concurrent requests and the maximum accumulated request length. Users must provide these upper bounds during the scheduler's first planning stage.

Partial Outputs. The size of partial outputs depends on both the problem dimensions (i.e., the number of heads and the head dimension) and the number of CTAs per kernel launch. In our load-balancing algorithm 3.3.1, only requests deemed "long" – those whose KV length exceeds the total KV length divided by the number of CTAs – are split. According to the Writethrough Optimizations in Section D.2, only these split requests produce outputs in the workspace buffer. Because the number of splits cannot exceed the total number of CTAs, and each split yields at most two tiles that must be merged, there are at most $2 \times \#\text{CTA}$ partial outputs. Each tile produces a partial output of size $T_q \cdot H_{qo} \cdot (D+1)$, where T_q is the query tile size, H_{qo} is the number of heads, and $D+1$ is the head dimension and LSE dimension. Therefore, the upper bound for the total partial output size is:

$$2\,\#\text{CTA} \times T_q \times H_{qo} \times (D+1).$$

By default, the total number of CTAs is set to $k \times \#\text{SM}$, where $\#\text{SM}$ denotes the number of streaming multiprocessors on the GPU and k is chosen to maximize CTA-level occupancy. For tensor-core based microkernels with high register usage, k typically does not exceed 2 on Ampere, and it is often 1 on Hopper (one CTA per SM, also referred to as a persistent kernel).

E Overlap of Attention with Other Operations

Nanoflow (Zhu et al., 2024a) overlaps GEMM, attention, and inter-device communication in separate CUDA streams,

assigning a fixed number of SMs to each operation. In FlashInfer, this SM number can be provided by the user through the plan functions, and the FlashInfer load-balancing scheduler will allocate tiles accordingly.

F FP8–FP16 MIXED-PRECISION ATTENTION

Recent LLMs frequently adopt `fp8` KV-Cache to reduce memory bandwidth and storage costs (Micikevicius et al., 2022). In FlashInfer, we implement *mixed-precision* attention kernels wherein the query and output remain in `fp16`, while the KV-Cache is stored in `fp8`. We leverage the fast numerical array converter and fragment shuffler proposed by Gupta (2024) to accelerate dequantization and handle bitwidth mismatches efficiently. This design allows for reduced memory footprints and higher bandwidth utilization without significantly compromising numerical accuracy.

G ADDITIONAL EVALUATION

In this section, we present additional evaluation results to further validate the performance, scalability, and robustness of FlashInfer across diverse experimental conditions.

G.1 Comparison with FlexAttention

We compare FlashInfer and FlexAttention (He et al., 2024) on different attention variants using the AttentionGym (PyTorch-Labs, 2024) benchmark on NVIDIA H100 80GB SXM. We evaluated with batch size 16, number of heads 16 and head dim 128, the CUDA version and the Triton version were fixed to 12.4 and 3.2, respectivelyrespectively. Tables 1 to 4 show the performance of FlashInfer and FlexAttention in TFLOPS/s, where higher numbers mean better performance. Across all four scenarios and a range of sequence lengths, FlashInfer consistently outperforms FlexAttention, with especially large gains at longer sequence lengths. The better performance is mainly due to the usage of Hopper microarchitecture's advanced features (such as warp specialization and TMA), and CUTLASS's fine-grained resource control (at register-level rather than tile-level) over Triton. Note that these gaps will be alleviated once Triton fully supports these features.

Table 1. Causal Attention

Seq Length	FlexAttention	FlashInfer
512	209.11	**250.454**
1024	294.53	**406.554**
2048	376.90	**487.236**
4096	421.00	**548.388**
8192	441.26	**587.903**
16384	453.57	**612.259**

Table 2. Attention with Logits SoftCap

Seq Length	FlexAttention	FlashInfer
512	241.51	**336.487**
1024	327.50	**409.534**
2048	379.57	**468.769**
4096	403.39	**489.667**
8192	407.82	**515.573**
16384	409.89	**520.935**

G.2 Evaluation of Shared-Prefix Attention Kernels

We measure shared-prefix attention kernels with suffix length 128. Table 5 shows the kernel latency under different shared prefix lengths, scenarios and batch sizes, where numbers are in microseconds (us), and "composable" means composable format while "single" means single format. The composable format benefits long prefixes (e.g., 32k) and large batch sizes (e.g., 64). However, these speedups do not always yield proportional end-to-end gains because real-world shared prefix sizes tend to be smaller.

G.3 Ablation Study on Variable Sequence Length and load-balancing scheduler

We conduct ablations on the effect of load-balancing scheduler (Section 3.3.1). Table 6 and 7 show the results for Llama 3.1-8B-Instruct running on an NVIDIA H100 SXM5 GPU with SGLang (Zheng et al., 2023b) + FlashInfer (with and without load-balancing scheduler). We evaluate the inter-token latency (ITL, ms) and time-to-first-token (TTFT, ms) with three datasets: ShareGPT, variable sequence length with input lengths sampled from $U(512, 2048)$ and output fixed at 256, and variable sequence length with input lengths sampled from $U(4096, 16384)$ and output fixed at 256. "RR" in the tables means request rate.

G.4 vLLM Integration Evaluation

We integrate FlashInfer to vLLM and compare with the default backend with a fixed request rate of 16, reporting throughput (tokens/s), inter-token latency (ITL, ms), and time-to-first-token (TTFT, ms) in Table 8. FlashInfer reduces ITL by aroudn 13% using fp8 KV-cache, but heavy Python overhead in our vLLM integration (e.g. array operations) at host side causes minor regressions with bf16. Our future optimizations will address these in C++ and move the scheduler to device.

G.5 Fine-Grained Block-Sparsity Evaluation

FlashInfer supports fine-grained block-sparse matrices, which is useful in many KV-Cache pruning algorithms. We

Table 3. ALiBi Bias (Press et al., 2022)

Seq Length	FlexAttention	FlashInfer
512	253.22	**403.899**
1024	344.70	**500.220**
2048	406.14	**535.498**
4096	426.13	**561.324**
8192	436.35	**573.493**
16384	434.86	**578.005**

Table 4. Sliding Window (window size = 1024)

Seq Length	FlexAttention	FlashInfer
512	206.51	**236.363**
1024	292.25	**374.108**
2048	350.91	**381.464**
4096	368.45	**384.998**
8192	373.25	**384.514**
16384	367.91	**380.506**

measure the kernel performance on Quest (Tang et al., 2024), a state-of-the-art long-context modeling algorithm, which uses fine-grained sparsity in KV-Cache. We compared the batch decoding attention kernel in Quest using FlashInfer and compared its performance to PyTorch SDPA and FlexAttention on an NVIDIA H100 SXM5 GPU with the configuration (block size 16, num_qo_heads 32, num_kv_heads 32, head_dim 128). All latency values reported are in microseconds (us).

As shown in Table 9 to 11, FlashInfer demonstrates a considerable performance advantage, achieving up to a 20x speedup for long sequence lengths. Currently FlexAttention relies on large block templates, while FlashInfer employs a sparse-row gathering strategy to leverage dense tensor cores for small block sizes. This design choice supports fine-grained KV-cache pruning.

Table 5. Latency of Shared-Prefix Attention Kernels

Shared Prefix Length	Composable (BS=16)	Single (BS=16)	Composable (BS=64)	Single (BS=64)
1024	**45.17**	46.52	87.86	130.49
8192	**88.67**	226.57	125.76	931.75
32768	**217.42**	945.67	254.54	4090

Table 6. Load-balancing Scheduler Ablation Study (ITL)

Scenario	w/ Load-Balancing	w/o Load-Balancing	Triton
ShareGPT (RR=16)	**8.96**	9.16	9.36
$U(512, 2048)$ (RR=8)	**8.21**	8.42	8.49
$U(4096, 16384)$ (RR=1)	**8.63**	13.89	11.08

Table 7. Load-balancing Scheduler Ablation Study (TTFT)

Scenario	w/ Load-Balancing	w/o Load-Balancing	Triton
ShareGPT (RR=16)	**39.05**	39.42	52.92
$U(512, 2048)$ (RR=8)	**66.78**	67.38	68.48
$U(4096, 16384)$ (RR=1)	**411.02**	421.60	566.30

Table 8. vLLM Integration Evaluation

Backend	Throughput	Median ITL	Median TTFT
Default (bf16)	6062.89	10.42	35.85
FlashInfer (bf16)	6065.41	10.63	36.60
Default (e4m3)	6015.86	12.56	39.74
FlashInfer (e4m3)	6020.32	10.92	37.93

Table 9. FlashInfer Fine-Grained Sparsity Latency (us)

	page_budget			
seq_len	64	128	256	512
4096	20.299	30.361	44.383	44.430
8192	22.273	28.603	44.928	68.194
16384	20.485	28.678	44.677	68.700
32768	22.371	28.700	44.988	68.478

Table 10. PyTorch SDPA Fine-Grained Sparsity Latency (us)

	page_budget			
seq_len	64	128	256	512
4096	287.684	288.904	287.715	287.807
8192	474.631	474.508	474.683	473.070
16384	857.319	857.570	857.094	857.728
32768	1711.955	1711.621	1713.093	1711.709

Table 11. FlexAttention Fine-Grained Sparsity Latency (us)

	page_budget			
seq_len	64	128	256	512
4096	1100.349	1097.356	1073.753	1071.797
8192	1092.695	1099.100	1078.081	1074.886
16384	1109.817	1101.535	1077.639	1076.859
32768	1169.109	1187.395	1176.332	1174.502