OmniAttention: supports interleaved mask with comparable performance of FlexAttention

Mira Xiao(minxiao), Guanwei Wu(guanweiw)

Due date: December 8, 2025 at 11:59 PM

1 Summary

This project introduces **OmniAttention**, a kernel built on top of FlashAttention-2's Q-tiling strategy and extended to support interleaved (or arbitrarily flexible) mask patterns. We partition the sparse attention mask into fixed-size `BLOCK_SIZE` tiles, each annotated with a mask type: `FULL`, `CAUSAL`, or `PARTIAL`. The kernel is implemented in CUDA with MMA tensor-core instructions, and we apply several optimization techniques, including parallel small-tile computation, shared-memory reuse for K/V tiles, double-buffer prefetching, and layout swizzling to reduce bank conflicts. We evaluate performance using randomly generated inputs and block masks with sequence lengths of 512, 1024, and 2048. OmniAttention achieves performance comparable to FlexAttention(from PyTorch, support such arbitrary mask patterns) for long sequences and, in some cases, the best kernel delivers 1.04–2.97× speedups. The code of this project is available at https://github.com/Echo-minn/omni-attention

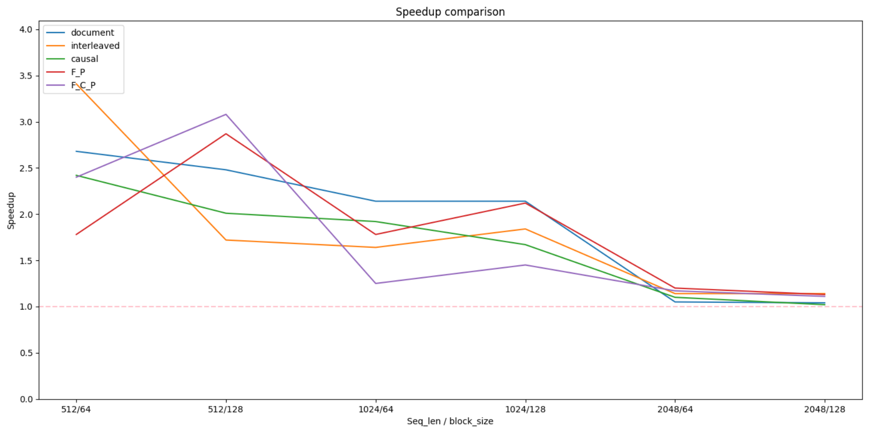


Figure 1: Swizzling kernel's speedup on different seq_len and mask patterns

2 Background

2.1 Vision Language Models

Modern language models are built on autoregressive Transformers, where each token predicts the next conditioned only on its past, enforced through a causal attention mask. Foundational works such as "Attention Is All You Need"[8] and later large-scale LLMs like "GPT-3: Language Models are Few-Shot Learners"[1] demonstrate that strict causal masking is essential for stable next-token prediction and prevents information leakage during training. In contrast, the vision components of multimodal models benefit from non-causal or fully bidirectional attention, as shown in Vit[5] and Pali[2], where dense spatial interactions are crucial for capturing global context and fine-grained image structure. As a result, interleaved text–image sequences used in state-of-the-art VLMs (e.g., "LLaVA: Large Language and Vision Assistant"[6], "DeepSeek-VL: Towards Real-World Vision-Language Agents")[7] inherently require heterogeneous masking—causal for language segments and full or block-dense for visual segments.

2.2 SOTA method

However, existing high-performance attention kernels are not designed for such mixed patterns. FlashAttention-2[3] achieves state-of-the-art throughput through fine-grained tiling, optimal warp partitioning, and memory-efficient IO-aware scheduling, but it assumes uniform masking across the sequence and therefore cannot directly support interleaved causal and non-causal regions.

FlexAttention, introduced in PyTorch ("FlexAttention: Fast, Flexible Attention for Transformers"[4]), provides a programmable attention interface where users specify `mask_mod` (operations modifying attention scores) and `mask_mod` (operations determining which positions are valid to attend). Instead of hard-coding causal or block masks, FlexAttention lowers these user-defined Python functions into small Triton subgraphs that are injected into pre-written attention kernel templates. The `mask_mod` function receives the query and key indices (e.g., `q_idx, k_idx`) and returns a boolean indicating whether the pair should be masked out; for example, causal masking is implemented as `k_idx > q_idx`, while vision blocks can be defined as allowing all positions within an image segment. During lowering, FlexAttention compiles this predicate into vectorized Triton code, enabling the kernel to dynamically decide at runtime which `<query, key>` pairs to invalidate by setting their attention scores to `-inf`. This mechanism offers great flexibility that supporting arbitrary structured masks, but introduces overhead because the masking logic is evaluated per tile rather than being baked into a specialized kernel.

2.3 Why OmniAttention

These limitations motivate the design of an attention variant that natively supports interleaved masking patterns—allowing causal text tokens and fully-attending vision tokens to coexist efficiently within a single sequence—while maintaining performance competitive with optimized frameworks such as FlashAttention-2 and FlexAttention. Such a kernel is critical for scaling cross-modal VLMs, improving training throughput, and minimizing memory cost without sacrificing modeling fidelity across modalities.

From a parallel computing perspective, implementing an attention kernel that supports interleaved masking provides a concrete opportunity to apply principles of efficient GPU tiling, memory hierarchy optimization, and warp-level parallelism. Unlike uniform-mask attention, interleaved masking introduces irregular data dependencies and heterogeneous compute patterns, requiring careful design of thread-block scheduling, shared-memory layout, and fused softmax–matmul pipelines to avoid performance regressions. By building this variant based on the FlashAttention-style IO-aware framework and will-designed fine-grained sparse mask block, the project bridges theoretical concepts in parallel algorithms with a real high-performance GPU workload, demonstrating how system-level optimization directly impacts the scalability of modern vision–language models.

3 Approach

3.1 Data Preparation

3.1.1 Data Generation Strategy

The debug data preparation process (`generate_debug_data.py`) employs a multi-stage pipeline that produces self-contained test artifacts. For each test pattern, we generate:

1. **Input Tensors**: Random Q, K, V tensors with fixed seeds for reproducibility, stored in both FP16 (kernel input) and FP32 (reference computation) formats.
2. **Mask Representations**: We construct masks at two levels of granularity:
   - Dense masks: Full `[B, H, seq_len, seq_len]` boolean tensors for reference computation
   - Block-sparse masks: Compressed `OmniBlockMask` structures encoding sparsity patterns
3. **Reference Outputs**: Ground-truth attention outputs computed via naive PyTorch implementation using dense masks, enabling numerical verification of kernel correctness.
4. **Metadata**: Configuration parameters (batch size, sequence length, block sizes, seed values) are embedded within each debug artifact to ensure test reproducibility.

3.1.2 Mask Pattern Variants

Our method can generate different mask pattern which is combination of various mask block types:

1. **Causal patterns**: Standard autoregressive attention where each query attends only to preceding keys

2. **Full/Causal mixtures**: Random combinations of `FULL` and `CAUSAL` block types to simulate heterogeneous attention patterns
3. **Full/Causal/Partial mixtures**: Complex patterns including `PARTIAL` blocks that require per-token mask lookups

Each pattern variant is serialized to disk as a PyTorch checkpoint, enabling offline correctness testing and performance profiling without regenerating inputs. Here is one data sample and more details and examples are covered in the Appendix7.2:

```
Block mask pattern (head=0, F=FULL, C=CAUSAL, P=PARTIAL):
### Document pattern
  q_block=0: F F F F
  q_block=1: F F F F
  q_block=2: F F F F
  q_block=3: F F F F
  q_block=4:         F P
  q_block=5:         P P P P
  q_block=6:           P F F
  q_block=7:           P F F

### Interleaved pattern
  q_block=0: C
  q_block=1: F C
  q_block=2: F F P P P P P
  q_block=3: F F F F F F P
  q_block=4: F F F F F F P
  q_block=5: F F F F F F P
  q_block=6: F F F F F F P
  q_block=7: F F F F F F F C
```

3.2 Technology Stack and Framework

3.2.1 High-Level Architecture

The implementation follows a hybrid Python-CUDA architecture that separates interface logic from performance-critical computation:

1. **Python Layer** (`omni_attn_torch.py`):
   - PyTorch tensor management and shape validation
   - Block mask construction and conversion utilities
   - Reference implementations for correctness verification
   - Kernel dispatch logic
2. **CUDA Kernel Layer** (`mma/omni_attn_*.cu`):

- Hand-tuned attention kernels using CUDA C++
- Direct hardware-level optimizations (MMA instructions, shared memory management)
- Multiple kernel variants targeting different performance/feature tradeoffs

3. **Binding Layer (`setup.py`)**:
   - PyBind11 integration for seamless Python-CUDA interop
   - Automatic compilation and module loading
   - Type conversion and memory layout management

4 Algorithm

4.1 Mask Presentation and Design

4.1.1 Block-Sparse Mask Format

To address the computational overhead of irregular attention patterns, we adopt a block-sparse mask representation that aligns with GPU memory hierarchy and tiling strategies. The OmniBlockMask data structure encodes sparsity at block granularity rather than per-token, dramatically reducing metadata size and enabling efficient kernel-level skipping of masked regions.

The mask format consists of three core tensors:

1. `kv_num_blocks`: `[B, H, num_q_blocks]` - Number of active KV blocks per Q block
2. `kv_indices`: `[B, H, num_q_blocks, max_blocks]` - Indices of active KV blocks for each Q block
3. `block_mask_types`: `[B, H, num_q_blocks, max_blocks]` - Mask type per block (FULL, CAUSAL, PARTIAL, MASKED)

4.1.2 Mask Type Semantics

Each block is classified into one of four categories:

1. FULL: No masking applied within the block; all Q-K pairs are computed
2. CAUSAL(Fig.2a): Causal masking applied; only `q_idx >= kv_idx` pairs are valid
3. PARTIAL(Fig.2b): Per-token masking required; uses auxiliary dense bitmasks stored separately; the mask positions can be random
4. MASKED: Block entirely skipped; no computation or memory access

This classification enables kernel specialization: FULL blocks use optimized dense matrix multiplication paths, CAUSAL blocks apply efficient triangular masking, and MASKED blocks are completely elided from computation.

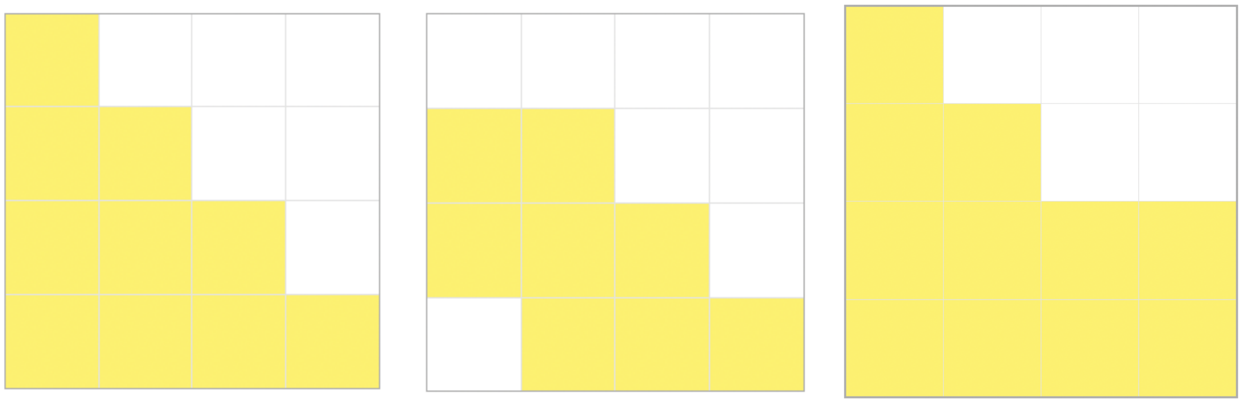

(a) CAUSAL block (b) PARTIAL block (c) PARTIAL block 2

Figure 2: Mask Type of an OmniMaskBlock

4.2 Overhead Reduction Techniques

1. **Block Alignment**: Sequences are padded to multiples of `Q_BLOCK_SIZE` and `KV_BLOCK_SIZE` (typically 64 or 128), ensuring that mask boundaries align with kernel tile boundaries. This eliminates the need for per-token boundary checks within the inner computation loop.

2. **Compressed Partial Masks**: For PARTIAL blocks, we employ a two-level indirection scheme. Rather than storing dense masks for all blocks, we maintain:

   - `partial_block_mask_indices`: Sparse index mapping from block coordinates to a compact mask pool
   - `partial_block_masks`: A deduplicated tensor of `num_partial_blocks`, `Q_BLOCK_SIZE` and `KV_BLOCK_SIZE` boolean masks

   This design reduces memory overhead when multiple blocks share identical mask patterns.

3. **Early Block Skipping**: The kernel iterates only over active blocks as specified by `kv_indices`, completely bypassing masked regions. This reduces both computation and memory bandwidth, with effectiveness proportional to mask sparsity.

4. **Shared KV Buffers**: For Q blocks that attend to overlapping KV regions, we reuse shared memory buffers across multiple Q tiles, since we need data from K and V tiles in different computation stages, we can load data data in order and overwrite the previous one, thus reducing redundant global memory loads. This optimization is particularly effective for causal attention patterns where adjacent Q blocks share significant KV overlap.

4.3 MMA Macro

At those macros we wrap the Tensor Core MMA instruction for a $16 \times 8 \times 16$ tile. `HMMA16816` multiplies a 16×16 fragment of A (4 half registers `RA0-RA3`) by a 16×8 fragment of B (2 half registers `RB0-RB1`) and accumulates into two half registers `RC0/RC1`, producing two half registers `RD0/RD1`. It is FP16 multiply with FP16 accumulation. `HMMA16816F32` uses the same FP16 inputs but accumulates in FP32: `RC0-RC3` hold FP32 accumulators, and `RD0-RD3` receive the FP32 results, we use it to avoid precision loss in the online-softmax process. Functionally the K dimension is 16, M is 16, N is 8; the macro maps register operands to the `mma.sync.aligned.m16n8k16` instruction variant with either FP16 or FP32 accumulation.

```
// mma m16n8k16
#define HMMA16816(RD0, RD1, RA0, RA1, RA2, RA3, RB0, RB1, RC0, RC1)           \
  asm volatile(                                                               \
      "mma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16 {%0, %1}, {%2, %3, " \
      "%4, %5}, {%6, %7}, {%8, %9};\n"                                        \
      : "=r"(RD0), "=r"(RD1)                                                  \
      : "r"(RA0), "r"(RA1), "r"(RA2), "r"(RA3), "r"(RB0), "r"(RB1), "r"(RC0), \
        "r"(RC1))
#define HMMA16816F32(RD0, RD1, RD2, RD3, RA0, RA1, RA2, RA3, RB0, RB1, RC0,   \
                     RC1, RC2, RC3)                                           \
  asm volatile(                                                               \
      "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 {%0,  %1,  %2,  "    \
      "%3}, {%4, %5, %6, %7}, {%8, %9}, {%10, %11, %12, %13};\n"              \
      : "=r"(RD0), "=r"(RD1), "=r"(RD2), "=r"(RD3)                            \
      : "r"(RA0), "r"(RA1), "r"(RA2), "r"(RA3), "r"(RB0), "r"(RB1), "r"(RC0), \
        "r"(RC1), "r"(RC2), "r"(RC3))
```

4.4 SharedKV: Algorithm

This kernel implements a FlashAttention-style, IO-aware attention variant that operates on block tiles of size $B_r \times B_c$ and supports three block mask types (causal, full, partial) plus fully masked blocks. Each CUDA thread block is mapped to a single $(\text{batch}, \text{head}, q_{\text{block}})$ tile; it loads the corresponding Q rows into shared memory once, then iterates over all active KV blocks for that tile. For each KV block, it (1) loads K into shared memory, (2) computes $S = QK^\top$ using tensor core MMA instructions in F16 with F32 accumulation for high precision preservation, (3) applies the appropriate mask—either purely causal based on absolute positions or a dense boolean partial mask in the local $B_r \times B_c$ block, (4) performs online softmax across blocks using running max and sum in F32, and (5) multiplies the normalized probabilities by V (also loaded from shared memory) to update an output accumulator. Final normalization by the accumulated softmax denominator is done before writing O back to global memory.

From a parallel computing perspective, the design showcases several key ideas: hierarchical tiling and shared-memory reuse (Q is loaded once per block, K/V are streamed blockwise), warp-level collectives for reductions (row-wise max and sum), and streaming (online) softmax to avoid storing full score matrices while still supporting arbitrarily many KV blocks. The block-wise mask types let us exploit structured sparsity that matches interleaved text–image layouts, while still preserving coalesced global memory access and high tensor-core utilization. Overall, this maps a complex, irregular masking pattern onto a regular, high-throughput GPU execution pattern—exactly the kind of systems-level reasoning emphasized in a parallel computing course.

Algorithm 1 Omni-Attention: Block-Tiled computation

```
Require: $Q, K, V \in \mathbb{R}^{B\times H\times L\times d}$, Q-block size $B_r$, KV-block size $B_c$
Require: kv_num_blocks, kv_indices, block_mask_types
Require: partial_block_mask_indices, partial_block_masks
Ensure: $O \in \mathbb{R}^{B\times H\times L\times d}$
 1: for $b = 0$ to $B-1$, $h = 0$ to $H-1$ do                ▷ batch and head
 2:     for each Q-block index $q$ do
 3:         Q-rows ← indices of Q-block $q$ (length $B_r$)
 4:         Load $Q[b, h, \text{Q-rows}, :]$ into shared memory
 5:         $m \leftarrow -\infty,\ \ell \leftarrow 0$                ▷ online softmax stats
 6:         $O_{\text{acc}}[\text{Q-rows}, :] \leftarrow 0$
 7:         nb ← kv_num_blocks$[b, h, q]$
 8:         for $k = 0$ to nb $- 1$ do
 9:             kv_block ← kv_indices$[b, h, q, k]$
10:             mask_type ← block_mask_types$[b, h, q, k]$
11:             p_id ← partial_block_mask_indices$[b, h, q, k]$
12:             ProcessKVBlock
13:         end for
14:         for each row $i$ in Q-rows do
15:             if $\ell \approx 0$ then
16:                 $O[b, h, i, :] \leftarrow 0$
17:             else
18:                 $O[b, h, i, :] \leftarrow O_{\text{acc}}[i, :]/\ell$
19:             end if
20:         end for
21:     end for
```

Algorithm 2 ProcessKVBlock (run inside one CUDA thread block)

```
Require: $b, h, q, \text{kv\_block}$
Require: $\text{mask\_type}, \text{p\_id}, Q, K, V,$ partial_block_masks
Require: $B_r, B_c, m, \ell, O_{\text{acc}}$
 1: procedure ProcessKVBlock
 2:     Q-rows ← indices of Q-block $q$ (length $B_r$)
 3:     K-cols ← indices of KV-block kv_block (length $B_c$)
 4:     if mask_type is Masked then
 5:         return
 6:     end if
 7:     Load $K[b, h, \text{K-cols}, :]$ into shared memory
 8:     $S \leftarrow Q_{\text{tile}} K_{\text{tile}}^{\mathsf{T}} / \sqrt{d}$                        ▷ MMA, F16→F32
 9:     for each $(i, j)$ in $S$ in parallel do
10:         $q\_idx \leftarrow \text{Q-rows}[i], \quad k\_idx \leftarrow \text{K-cols}[j]$
11:         if mask_type is Causal and $q\_idx < k\_idx$ then
12:             $S_{ij} \leftarrow -\infty$
13:         else if mask_type is Partial then
14:             map $(i, j)$ to local $(i', j')$ in $[0, B_r) \times [0, B_c)$
15:             if partial_block_masks$[\text{p\_id}, i', j'] = \text{false}$ then
16:                 $S_{ij} \leftarrow -\infty$
17:             end if
18:         end if                                                  ▷ Full: no masking
19:     end for
20:     $m_{\text{block}} \leftarrow \max S_{ij}, \quad m_{\text{new}} \leftarrow \max(m, m_{\text{block}})$
21:     $S_{ij} \leftarrow \exp(S_{ij} - m_{\text{new}})$ for all $(i, j)$
22:     $\ell_{\text{block}} \leftarrow \sum S_{ij}$
23:     $\alpha \leftarrow \exp(m - m_{\text{new}})$
24:     $\ell \leftarrow \ell_{\text{block}} + \alpha \cdot \ell, \quad m \leftarrow m_{\text{new}}$
25:     Load $V[b, h, \text{K-cols}, :]$ into shared memory
26:     $P \leftarrow S$ cast to F16
27:     $O_{\text{block}} \leftarrow P V_{\text{tile}}$                                 ▷ MMA, F16→F32
28:     $O_{\text{acc}} \leftarrow \alpha \cdot O_{\text{acc}} + O_{\text{block}}$
29: end procedure=0
```

4.5 PrefetchKV: Algorithm

To mitigate the bottleneck caused by High Bandwidth Memory (HBM) latency, we implemented a software pipelining strategy utilizing CUDA's asynchronous copy instructions (`cp.async`). In a standard attention implementation, the kernel strictly serializes memory loads and matrix multiplications (MMA). Our prefetch strategy overlaps these operations to maximize instruction-level parallelism.

We employ a double-buffering scheme within the shared memory:

- **Buffer Allocation**: We allocate separate shared memory regions for K and V, allowing concurrent memory operations.

- **Asynchronous V Loading**: At the start of each iteration, we immediately issue asynchronous load instructions for the V block.
- **Overlap Strategy - Q@K^T with V Load**: The warp schedulers proceed to compute $S = QK^T$ immediately after issuing the V load. This allows the substantial latency of loading V from global memory to be hidden behind the arithmetic intensity of the matrix multiplication.
- **Speculative K Prefetching**: After computing QK^T, we speculatively issue the load for the *next* iteration's K block (K_{next}) before the synchronization point.
- **Synchronization**: We enforce a synchronization barrier to ensure both V_{curr} and K_{next} are fully loaded before proceeding to the softmax and PV stages. This conservative approach guarantees data validity for the complex masking logic while still maintaining the primary compute-memory overlap.

Algorithm 3 Omni-Attention: Prefetch and Pipelining

```
Require: $Q$ loaded in Shared Memory
Require: First $K$ block ($K_{curr}$) loaded in Shared Memory
 1: procedure PipelinedLoop
 2:     for each active KV block $k = 0 \dots N-1$ do
 3:                                                     ▷ Stage 1: Prefetch V
 4:         Issue cp.async for $V_{curr} \to$ Smem Buffer 1
 5:         cp.async.commit_group()
 6:                                   ▷ Stage 2: Compute $QK^T$ (Hides V Load)
 7:         $S \leftarrow \text{MMA}(Q, K_{curr}^T)$
 8:                                                 ▷ Stage 3: Prefetch Next K
 9:         if $k < N-1$ then
10:             Issue cp.async for $K_{next} \to$ Smem Buffer 0
11:             cp.async.commit_group()
12:         end if
13:                                               ▷ Stage 4: Masking & Softmax
14:         Apply Masks (Causal/Partial) to $S$
15:                                                 ▷ Stage 5: Synchronization
16:         cp.async.wait_group()                        ▷ Ensure $V_{curr}$ is ready
17:         __syncthreads()
18:         Compute Softmax statistics $(m, \ell)$
19:         $P \leftarrow \exp(S - m)$
20:                             ▷ Stage 6: Compute PV (Hides Next K Load)
21:         $O \leftarrow \text{MMA}(P, V_{curr})$
22:                                                ▷ Prepare for next iteration
23:         $K_{curr} \leftarrow K_{next}$ (Pointer swap / Double buffer)
24:     end for
25: end procedure
```

4.6 Layout Swizzling: Algorithm

When loading Q/K/V into shared memory, each 16-byte (8 half) vector uses this swizzled start offset before writing into the shared tile. And thus threads access different banks rather than colliding on the same bank lines. Instead of padding every row with extra columns to avoid conflicts, the kernel permutes row/column strides to spread consecutive lanes across distinct banks, keeping the tiles contiguous for Tensor Core MMA while avoiding the wasted space of heavy padding.

To be more specific, the `swizzle_permuted()` permutes the starting offset of each 8-element slice within a 16-wide chunk: it XORs the row group with the 8-wide column group, then swaps the two 8-wide halves when the row group is odd. Consider rows 0–7 and columns 0–15 is one chunk. After swizzling, rows 0–3 store [0..7][8..15] as usual and rows 4–7 now store [8..15][0..7].

```
// i: row index; j: col index.
template <const int kColStride = 16, const int kStep = 8>
static __device__ __forceinline__ int swizzle_permuted(int i, int j) {
  static_assert(kStep == 4 || kStep == 8);
  static_assert(kColStride % kStep == 0);
  if constexpr (kStep == 8) {
    return (((j >> 3) ^ (i >> 2)) % (kColStride >> 3)) << 3;
  } else {
    static_assert(kStep == 4);
    return (((j >> 2) ^ (i >> 2)) % (kColStride >> 2)) << 2;
  }
}
```

5 Challenges

5.1 Precision loss

A significant challenge encountered during development was handling the precision discrepancies between the CUDA kernel and the PyTorch reference implementation. Our kernel utilizes Tensor Cores via `HMMA` instructions, which inherently require FP16 inputs while accumulating results in FP32. This mixed-precision pipeline introduces two primary sources of numerical deviation:

1. **Softmax Sensitivity**: The attention mechanism involves exponential operations ($\exp(S-m)$). Even minor rounding errors in the computation of $S = QK^T$ (due to FP16 inputs) are magnified exponentially by the Softmax function. This can lead to noticeable divergence in the probability matrix P, especially when attention scores are large.

2. **Type Casting for MMA**: To utilize Tensor Cores for the second matrix multiplication ($O = PV$), the probability matrix P, which is computed in FP32 for stability during Softmax, must be cast back to FP16. This downcasting (`__float2half`) truncates precision. In contrast, the standard PyTorch reference implementation typically

maintains FP32 or BF16 precision throughout the computation graph without these intermediate truncation steps.

3. **Accumulation Order**: Floating-point addition is non-associative. The tiled execution order of the CUDA kernel differs from the global reduction performed by PyTorch, leading to bit-level differences that can accumulate over long sequences.

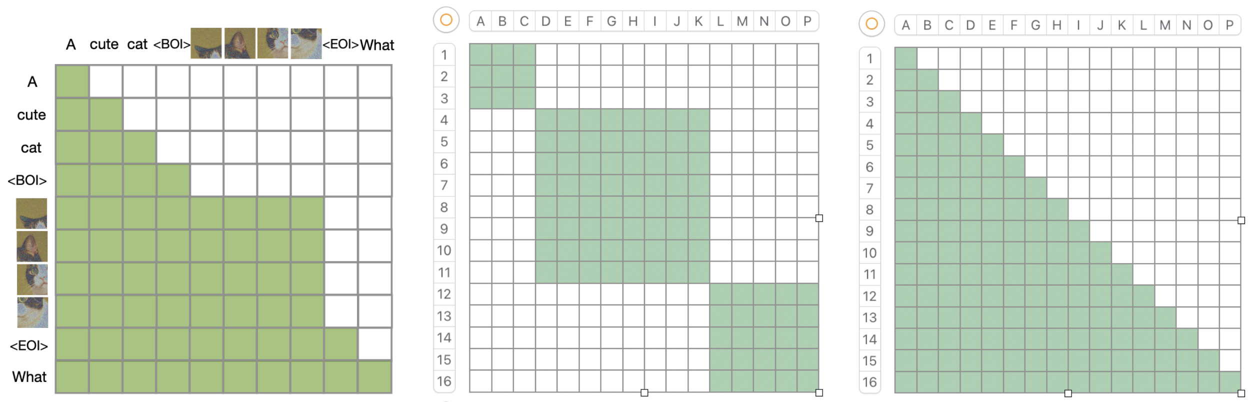
These factors occasionally caused the kernel output to exceed the strict Mean Squared Error (MSE) thresholds ($< 1e-5$) used in correctness tests, particularly for random inputs with high variance. To mitigate this, we enforced FP32 accumulation for all intermediate register operations and adjusted the test tolerance to accommodate the inherent limitations of FP16 Tensor Core arithmetic.

6 Results

The experiments run on:

1. different mask patterns that are widely used in LLMs and VLMs.
2. arbitrary combination of mask type, each block may have random generated value and type.

Also the shape the data we use small sequence length 512, medium sequence length 1024, and long context length 2048. To explore the performance sensitivity to the block size, we partition the data into fixed 64 and 128-len chunks. The speedup is compared to the FlexAttention(same shape, same value, but different block size due it supports block size ≥ 128.



(a) cross-modal data [10] (b) document mask data (c) causal mask data

Figure 3: Different mask pattern. Actual representation details see Appendix[7.2]

6.1 Block size sensitivity

We can tell from the figures 4, 5, 6, 7, 8 that with same sequence, the performance is not sensitive to the block size(speedup plateau within the same sequence length). For a fixed sequence length, changing the block size (e.g., 64 → 128) alters the tiling granularity but does not significantly change the total amount of work, memory traffic, thus both 64- and 128-token blocks achieve similar arithmetic intensity, reuse patterns, and GPU occupancy, the kernel remains compute-bound and thus performance plateaus across block sizes for the same sequence length.

6.2 Results of different Mask Pattern

Form the figures, which show the speedup and TFLOPs of each kernel, the `swizzle` kernel optimization achieves the highest speedup, consistently outperforming both `shared-KV` and `prefetch` kernels. It's built on the top of shared-kv kernel, we got significant and consistent benefit from the layout swizzling and reduced bank conflicts across all sequence length. On the other hand, as the sequence length grows, the TFLOPs increase and speedups compared to FlexAttention decrease.

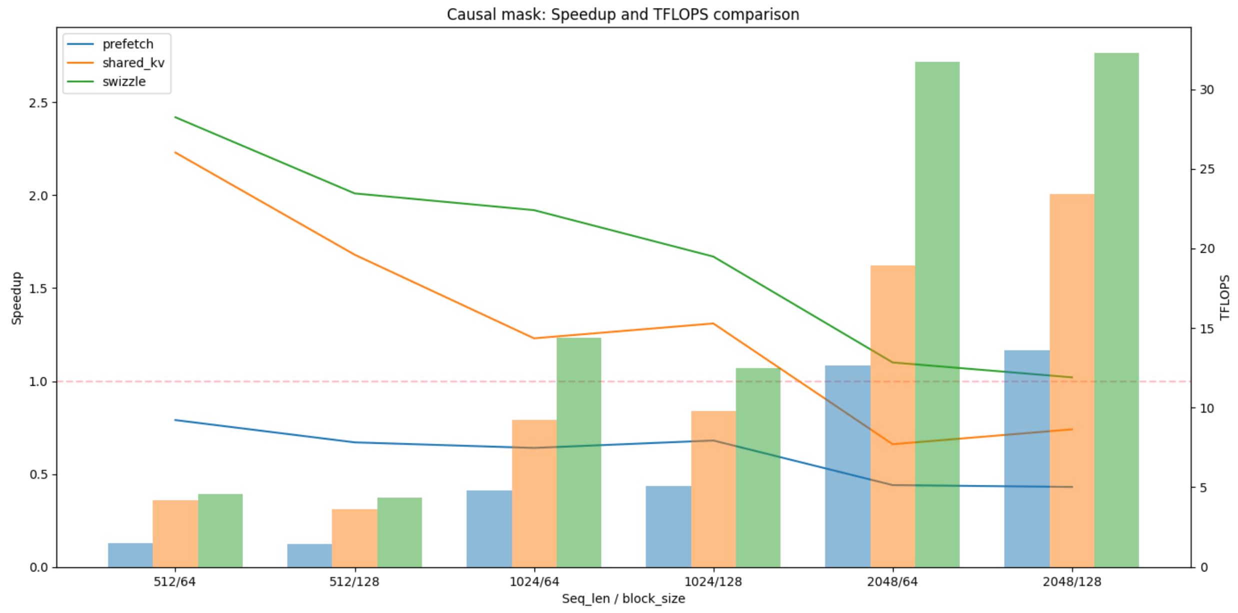


Figure 4: Causal mask: Speedup and TFLOPS comparison

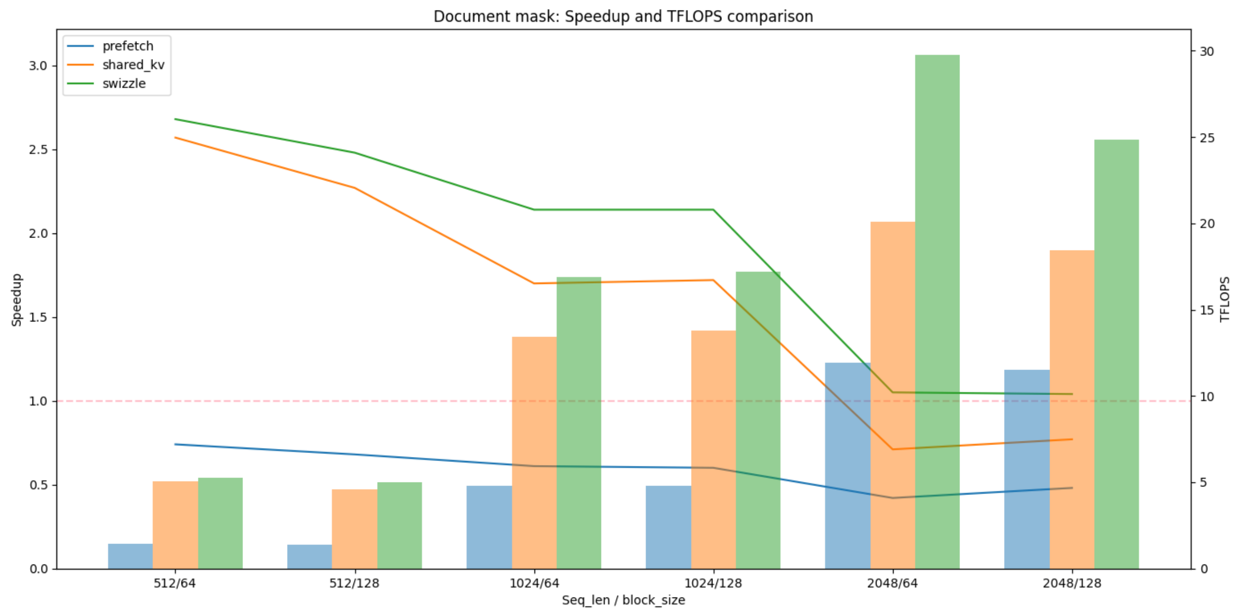


Figure 5: Document mask: Speedup and TFLOPS comparison

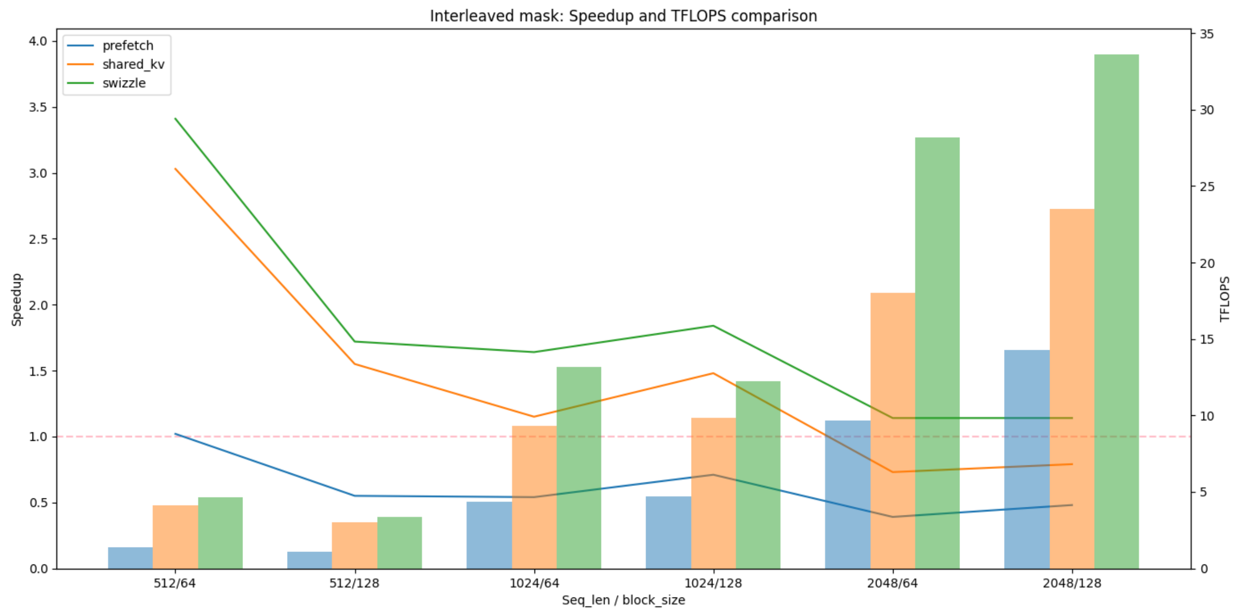


Figure 6: Interleaved mask: Speedup and TFLOPS comparison

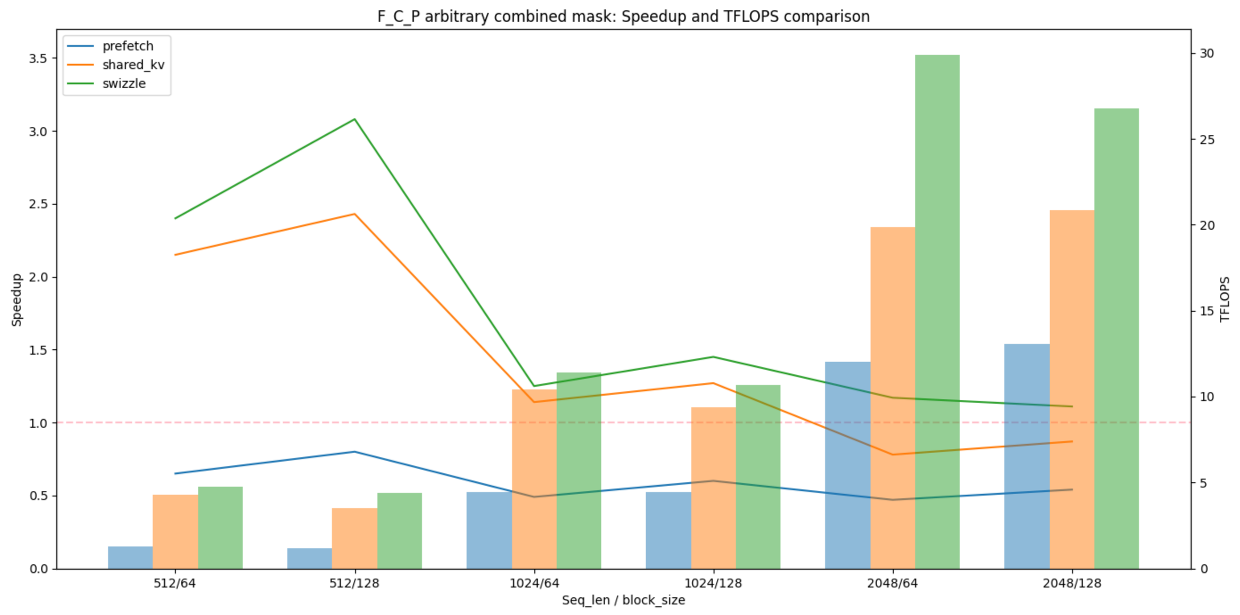


Figure 7: Random FULL/CAUSAL/PARTIAL mask combination: Speedup and TFLOPS comparison

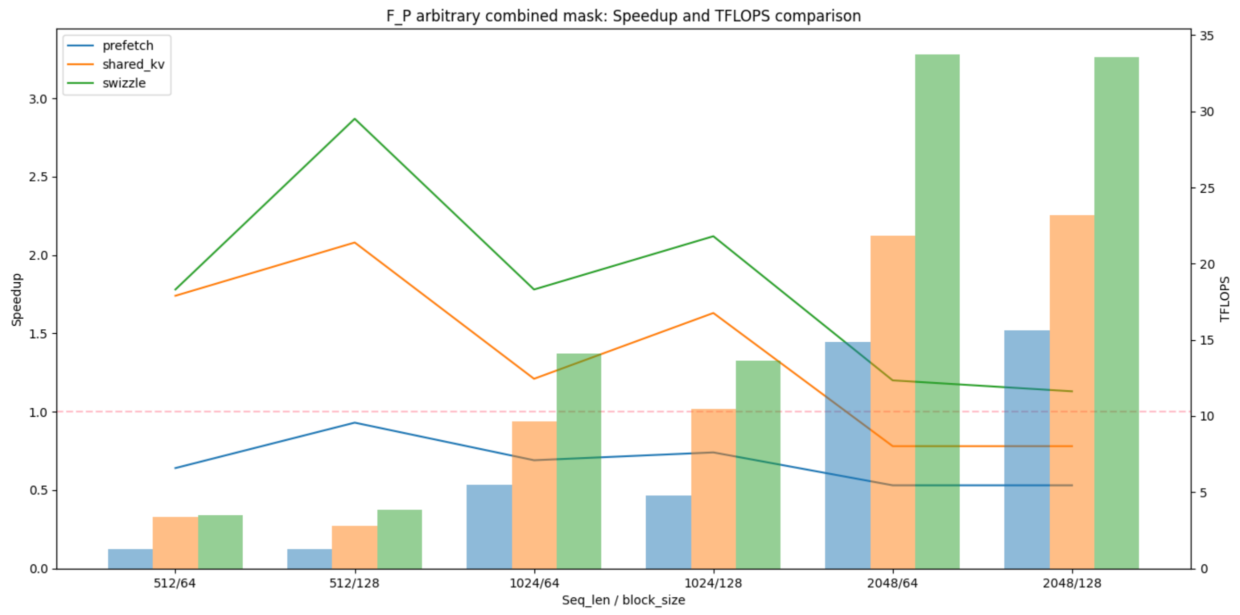


Figure 8: Random FULL/PARTIAL mask combination: Speedup and TFLOPS comparison

7 Further Improvements

7.1 Deep Pipelining with Granular Synchronization

Our current implementation uses a conservative synchronization strategy (`cp.async.wait_group(0)`) which waits for both the current V tile and the next K tile to complete before proceeding to Softmax. A future optimization is to implement finer-grained synchronization using `cp.async.wait_group(1)`. By organizing the pipeline stages to keep K_{next} in flight (pending group) while consuming only V_{curr} (completed group), we can overlap the memory latency of loading the next K block with the computation of Softmax and PV. This "second overlap" would further improve occupancy and throughput, particularly for memory-bound configurations.

7.2 Integrate it into a real VLM training framework

Currently, we are using randomly generated data with random mask type combinations. However, in the real training process of vision language models, due to the difference of embedding layers, the data shape, cross-modality splits and matrix sparsity can be various for specific task. For the next step, we're planing to integrate it into VLM training frameworks like Transfusion[10] and Show-o[9].

References

[1] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[2] Xi Chen et al. “Pali: A jointly-scaled multilingual language-image model”. In: *arXiv preprint arXiv:2209.06794* (2022).

[3] Tri Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. 2023. arXiv: 2307.08691 [cs.LG]. URL: https://arxiv.org/abs/2307.08691.

[4] Juechu Dong et al. *Flex Attention: A Programming Model for Generating Optimized Attention Kernels*. 2024. arXiv: 2412.05496 [cs.LG]. URL: https://arxiv.org/abs/2412.05496.

[5] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *CoRR* abs/2010.11929 (2020). arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

[6] Haotian Liu et al. *Visual Instruction Tuning*. 2023. arXiv: 2304.08485 [cs.CV]. URL: https://arxiv.org/abs/2304.08485.

[7] Haoyu Lu et al. *DeepSeek-VL: Towards Real-World Vision-Language Understanding*. 2024. arXiv: 2403.05525 [cs.AI]. URL: https://arxiv.org/abs/2403.05525.

[8] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[9] Jinheng Xie et al. *Show-o: One Single Transformer to Unify Multimodal Understanding and Generation*. 2025. arXiv: 2408.12528 [cs.CV]. URL: https://arxiv.org/abs/2408.12528.

[10] Chunting Zhou et al. *Transfusion: Predict the Next Token and Diffuse Images with One Multi-Modal Model*. 2024. arXiv: 2408.11039 [cs.AI]. URL: https://arxiv.org/abs/2408.11039.

Appendix

Attention data mask pattern detail

```
### Document pattern

B: 1, H: 8, seq_len: 512, head_dim: 64, Q_BLOCK_SIZE: 64, KV_BLOCK_SIZE: 64
Document segments: 256, 68, 188

Saving to data/attn_data/512_document_64_64.pt...
Sparsity: 0.5975 (59.75% sparse)
Block mask pattern (head=0, F=FULL, C=CAUSAL, P=PARTIAL):
  q_block=0: F F F F
  q_block=1: F F F F
  q_block=2: F F F F
  q_block=3: F F F F
  q_block=4:         F P
  q_block=5:         P P P P
  q_block=6:           P F F
  q_block=7:           P F F

### Interleaved pattern

B: 1, H: 8, seq_len: 512, head_dim: 64, Q_BLOCK_SIZE: 64, KV_BLOCK_SIZE: 64

Segment lengths: 133, 309, 70

Saving to data/attn_data/512_interleaved_64_64.pt...
 Saved. File size: 16.29 MB
Sparsity: 0.3175 (31.75% sparse)
Block mask pattern (head=0, F=FULL, C=CAUSAL, P=PARTIAL):
  q_block=0: C
  q_block=1: F C
  q_block=2: F F P P P P P
  q_block=3: F F F F F F P
  q_block=4: F F F F F F P
  q_block=5: F F F F F F P
  q_block=6: F F F F F F P
  q_block=7: F F F F F F F C

### Causal pattern
B: 1, H: 8, seq_len: 512, head_dim: 64, Q_BLOCK_SIZE: 64, KV_BLOCK_SIZE: 64
Generating fixed debug data...

Saving to data/attn_data/512_causal_64_64.pt...
 Saved. File size: 16.01 MB
Sparsity: 0.4990 (49.90% sparse)
Block mask pattern (head=0, F=FULL, C=CAUSAL, P=PARTIAL):
  q_block=0: C
  q_block=1: F C
  q_block=2: F F C
  q_block=3: F F F C
  q_block=4: F F F F C
  q_block=5: F F F F F C
  q_block=6: F F F F F F C
  q_block=7: F F F F F F F C
```

Arbitrary mask combination data

```
### FULL and PARTIAL random combination

Generating random FULL/CAUSAL debug data:
B=1, H=8, seq_len=1024, head_dim=64, Q_BLOCK_SIZE=128, KV_BLOCK_SIZE=128

Saving to data/1024/128_128_F_P.pt...
 Saved. File size: 52.01 MB
Sparsity: 0.4375 (43.75% sparse)
Block mask pattern (head=0, F=FULL, C=CAUSAL, P=PARTIAL):
  q_block=0: F
  q_block=1: P F
  q_block=2: F P F
  q_block=3: F F F F
  q_block=4: P F P P F
  q_block=5: F P F F F F
  q_block=6: F P P F F F F
  q_block=7: F P F P F F F F

### Random FULL/CAUSAL/PARTIAL combination
Generating random FULL/CAUSAL/PARTIAL debug data:
B=1, H=8, seq_len=1024, head_dim=64, Q_BLOCK_SIZE=128, KV_BLOCK_SIZE=128

Saving to data/1024/128_128_F_C_P.pt...
 Saved. File size: 53.63 MB; Sparsity: 0.4937 (49.37% sparse)
Block mask pattern (head=0, F=FULL, C=CAUSAL, P=PARTIAL):
  q_block=0: C
  q_block=1: P C
  q_block=2: P P C
  q_block=3: C F F C
  q_block=4: P F P P P
  q_block=5: C P F P F C
  q_block=6: F P P C F C C
  q_block=7: F P C P F C C C
```

Some test result logs

Some FAILEs come from precision loss.

```
============================================================
Testing with data: data/2048/debug_data_F_C.pt ...
============================================================
  Simple PASSED, time: 66.68ms, TFLOPS: 0.13, speedup: 0.00x
  Prefetch PASSED, time: 0.75ms, TFLOPS: 11.67, speedup: 0.43x
  Shared_kv PASSED, time: 0.48ms, TFLOPS: 18.39, speedup: 0.68x
  Swizzle PASSED, time: 0.31ms, TFLOPS: 28.10, speedup: 1.03x
  Flex PASSED, time: 0.32ms, TFLOPS: 27.23

============================================================
Testing with data: data/2048/debug_data_F_P.pt ...
============================================================
  Simple FAILED, time: 35.53ms, TFLOPS: 0.25, speedup: 0.01x
  Prefetch FAILED, time: 0.50ms, TFLOPS: 17.42, speedup: 0.64x
  Shared_kv PASSED, time: 0.48ms, TFLOPS: 18.24, speedup: 0.67x
  Swizzle PASSED, time: 0.31ms, TFLOPS: 28.06, speedup: 1.04x
  Flex PASSED, time: 0.32ms, TFLOPS: 27.07

============================================================
Testing with data: data/1024/debug_data_F_C.pt ...
============================================================
  Simple PASSED, time: 22.43ms, TFLOPS: 0.10, speedup: 0.01x
  Prefetch PASSED, time: 0.50ms, TFLOPS: 4.35, speedup: 0.52x
  Shared_kv PASSED, time: 0.25ms, TFLOPS: 8.73, speedup: 1.05x
  Swizzle PASSED, time: 0.19ms, TFLOPS: 11.64, speedup: 1.40x
  Flex PASSED, time: 0.26ms, TFLOPS: 8.34

============================================================
Testing with data: data/1024/debug_data_F_P.pt ...
============================================================
  Simple PASSED, time: 10.83ms, TFLOPS: 0.05, speedup: 0.02x
  Prefetch PASSED, time: 0.47ms, TFLOPS: 1.17, speedup: 0.56x
  Shared_kv PASSED, time: 0.17ms, TFLOPS: 3.20, speedup: 1.55x
  Swizzle PASSED, time: 0.14ms, TFLOPS: 3.99, speedup: 1.93x
  Flex PASSED, time: 0.26ms, TFLOPS: 2.07
```

8 Credit

8.1 Mira Xiao

1. Environment setup
2. Data generation and test pipeline setup
3. Shared kv and Swizzling kernels

8.2 Guanwei Wu

1. Naive kernel
2. Prefetch kernel