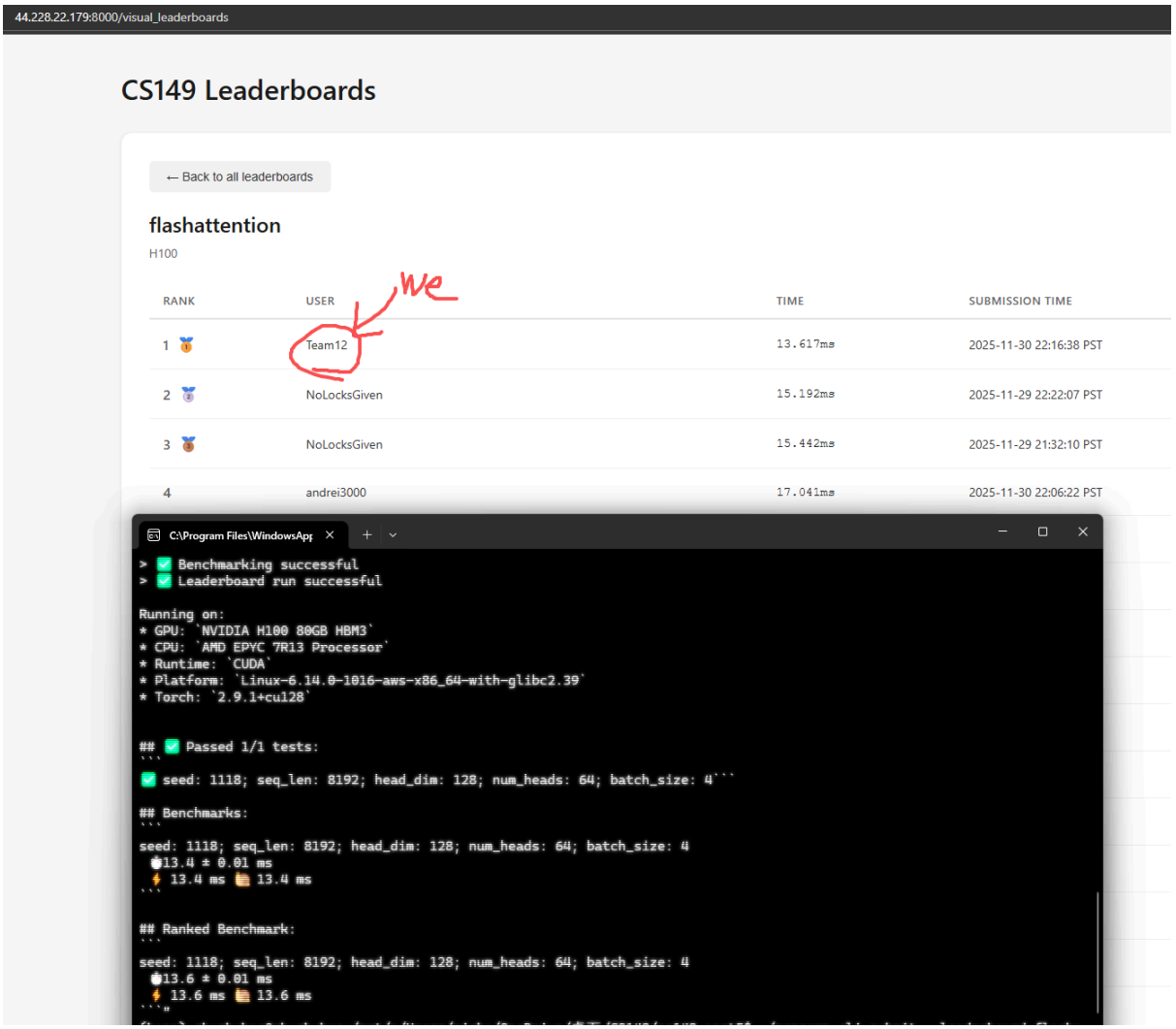
Partners: Chenhao Zhu (chenhzhu), Zhenyu Chen (zhenyuc)

Flash Attention



Work Log Part 1: The Steps You Took

How is the code structured in the current step*?* (We'd like you to submit the code, but also describe it at a high level of abstraction in the handout. e.g., "We were blocking the outermost loop and mapping blocks to CUDA thread blocks. And the innermost loop was parallelized over threads in the thread block").

AWS:

We created a TileLang version of Flash Attention. Firstly we defined the basic **config** for the parallelization:

```python
def get_configs():
    iter_params = dict(block_M=[128], block_N=[128], num_stages=[2], threads=[256])
    return [dict(zip(iter_params, values)) for values in itertools.product(*iter_params.values())]
```

As you can see, we defined the **block**/**chunk** size to be 128 with **metapipeline** stages number to be 2 and using 256 **threads**, which we tuned for optimized performance. With these configurations, in the *main()* function, we **chunk** the *Q* sequence *seq_q* into chunks with size of *block_M* . Each thread block will process a *block_M* × *dim Q* chunk.

```python
@T.prim_func
 def main(
        Q: T.Tensor(q_shape, dtype),
        K: T.Tensor(kv_shape, dtype),
        V: T.Tensor(kv_shape, dtype),
        Output: T.Tensor(q_shape, dtype),
 ):
     with T.Kernel(T.ceildiv(seq_q, block_M), heads, batch, threads=threads) as (bx, by, bz):
     ...
```

For the inner loop, we **chunk** the *KV* sequence *seq_kv* into chunks with size of *block_N*. Each iteration we process a *block_N* × *dim* size *KV* chunk:

```python
        loop_range = (
          T.min(
            T.ceildiv(seq_kv, block_N), T.ceildiv(
            (bx + 1) * block_M +
            past_len, block_N)) if is_causal else T.ceildiv(seq_kv, block_N))

        for k in T.Pipelined(
                loop_range,
        ...)
```

After chunking, we set up twostage **Metapipelining** to speed up the computation process:

```python
for k in T.Pipelined(
        loop_range,
        num_stages=num_stages,
        order=[1, 0, 3, 1, 1, 2],
        stage=[1, 0, 0, 1, 1, 1],
        group=[[0], [1, 2], [3, 4, 5, 6, 7, 8, 9, 10, 11], [12], [13], [14]]):
    MMA0(K, Q_shared, K_shared, acc_s, k, bx, by, bz)
    Softmax(acc_s, acc_s_cast, scores_max, scores_max_prev, scores_scale, scores_sum,logsum)
    Rescale(acc_o, scores_scale)
    MMA1(V, V_shared, acc_s_cast, acc_o, k, by, bz)
```

In the metapipeline stages, we further reduce the computation process into finer operations, which is the 15 indexes in the *group* parameter. The optimized *QK* matmul is processed in *MMA0* and *MMA1* deal with *V* matmul. With this ordering and grouping in metapipeline stages, **the program can load the next KV chunk while computing the current KV chunk**, which hides the memory latency and improves the GPU performance. For each component (i.e *MMA0 MMA1 Softmax Rescale*), we enable parallel execution to mathematics operations or GEMM to accelerate the computation.

Python

```python
@T.macro
    def MMA0(
        ...
    ):
        T.copy(K[bz, by, k * block_N:(k + 1) * block_N, :], K_shared)
        if is_causal:
            for i, j in T.Parallel(block_M, block_N):
                q_idx = bx * block_M + i + past_len
                k_idx = k * block_N + j
                acc_s[i, j] = T.if_then_else(q_idx >= k_idx, 0, T.infinity(acc_s.dtype))
        else:
            for i, j in T.Parallel(block_M, block_N):
                acc_s[i, j] = T.if_then_else(k*block_N + j >= seq_kv,T.infinity(acc_s.dtype), 0)
        T.gemm(Q_shared, K_shared, acc_s, transpose_B=True, policy=T.GemmWarpPolicy.FullRow)

    @T.macro
    def MMA1(
        ...
    ):
        T.copy(V[bz, by, k * block_N:(k + 1) * block_N, :], V_shared)
        T.gemm(acc_s_cast, V_shared, acc_o, policy=T.GemmWarpPolicy.FullRow)

    @T.macro
    def Softmax(
        ...
    ):
        T.copy(scores_max, scores_max_prev)
        T.fill(scores_max, T.infinity(accum_dtype))
        T.reduce_max(acc_s, scores_max, dim=1, clear=False)
        for i in T.Parallel(block_M):
            scores_max[i] = T.max(scores_max[i], scores_max_prev[i])

        for i in T.Parallel(block_M):
            scores_scale[i] = T.exp2(scores_max_prev[i] * scale  scores_max[i] * scale)

        for i, j in T.Parallel(block_M, block_N):
            acc_s[i, j] = T.exp2(acc_s[i, j] * scale  scores_max[i] * scale)
        T.reduce_sum(acc_s, scores_sum, dim=1)
        for i in T.Parallel(block_M):
            logsum[i] = logsum[i] * scores_scale[i] + scores_sum[i]
        T.copy(acc_s, acc_s_cast)

    @T.macro
    def Rescale(
        ...
    ):
        for i, j in T.Parallel(block_M, dim):
            acc_o[i, j] *= scores_scale[i]
```

The *Softmax* part is online, doing rowwise maximum and cumulative sum, as described in Github repo..

The parameter of the TileLang Pipeline can be understood in this way:

Shell

```
1. order=[1, 0, 3, 1, 1, 2, ...] (15 elements)
```

```
Meaning: Defines the execution order of operations
1: Does not participate in reordering, executes in original order
Non1 values: Operations are sorted by this value (smaller values execute first)
Example:
order[0] = 1: Operation 0 is not reordered
order[1] = 0: Operation 1 should execute first
order[2] = 3: Operation 2 executes at position 4
order[3] = 1: Operation 3 executes at position 2
2. stage=[1, 0, 0, 1, 1, 1, ...] (15 elements)
Meaning: Assigns operations to different pipeline stages
1: Does not participate in pipelining (synchronization point, must wait for completion)
0: Stage 0 (first pipeline stage)
1: Stage 1 (second pipeline stage)
Example:
stage[0] = 1: Operation 0 is a synchronization point
stage[1] = 0: Operation 1 is in Stage 0
stage[2] = 0: Operation 2 is in Stage 0
stage[3] = 1: Operation 3 is in Stage 1
3. group=[[0], [1, 2], [3, 4, 5, 6, 7, 8, 9, 10, 11], [12], [13], [14]]
Meaning: Defines groups of operations that can execute in parallel
Operations within the same group can execute in parallel
Different groups execute sequentially
Example:
[0]: Operation 0 executes independently
[1, 2]: Operations 1 and 2 can execute in parallel
[3, 4, 5, 6, 7, 8, 9, 10, 11]: Operations 311 can execute in parallel (Softmax internal
operations)
[12]: Operation 12 executes independently
[13]: Operation 13 executes independently
[14]: Operation 14 executes independently
```

And the detailed metapipeline progress is shown below:

Shell

```
Operation 0:  T.copy(K[...], K_shared) in MMA0  Load K block
Operations 12: Mask filling operations in MMA0 (parallel loops)
Operations 311: All operations inside Softmax (copy, fill, reduce_max, parallel loops, reduce_sum,
etc.)
Operation 12: Rescale operation
Operation 13: T.copy(V[...], V_shared) in MMA1  Load V block
Operation 14: T.gemm(...) in MMA1  Matrix multiplication


Iteration k=0 (First Block)
Stage 0:
Operation 0 (sync point): T.copy(K[block0], K_shared)  Load K block 0 from HBM to shared memory
Operations 12 (parallel group): Mask filling operations
Operations 311 (parallel group): All Softmax operations
Operation 12: Rescale operation
Stage 1:
Operation 13: T.copy(V[block0], V_shared)  Load V block 0 from HBM to shared memory
Operation 14: T.gemm(...)  Compute P·V
Iteration k=1 (Second Block, Overlapped with k=0)
Stage 0 (overlapped with k=0's Stage 1):
Operation 0 (sync point): T.copy(K[block1], K_shared)  Load K block 1 (parallel with k=0's MMA1)
Operations 12: Mask filling
Operations 311: Softmax
```

```
Operation 12: Rescale
Stage 1:
Operation 13: T.copy(V[block1], V_shared)  Load V block 1
Operation 14: T.gemm(...)  Compute P·V
```

In the end, we normalize and write back the output of this chunk of Q into global memory (*Output*). Then loop to next Q chunk:

Python
```
with T.Kernel(T.ceildiv(seq_q, block_M), heads, batch, threads=threads) as (bx, by, bz):
        ...
        ...
      for i, j in T.Parallel(block_M, dim):
          acc_o[i, j] /= logsum[i]
      T.copy(acc_o, O_shared)
      T.copy(O_shared, Output[bz, by, bx * block_M:(bx + 1) * block_M, :])
```

What is the performance of the code (give the runtime explicitly in the handin)?

Shell
```
## Benchmarks:
```
seed: 1118; seq_len: 8192; head_dim: 128; num_heads: 64; batch_size: 4
 ⏱ 13.4 ± 0.01 ms
 ⚡ 13.4 ms 🐌 13.4 ms
```

## Ranked Benchmark:
```
seed: 1118; seq_len: 8192; head_dim: 128; num_heads: 64; batch_size: 4
 ⏱ 13.6 ± 0.01 ms
 ⚡ 13.6 ms 🐌 13.6 ms
```"
```

What other statistics did you measure and look at for the current code?
We ran profile to see cache hit rate, throughput and traffic:

Shell
```
 Kernel 2: distribution_elementwise_grid_stride_kernel
    Duration            : 755.84 us
    Compute_Throughput  : 77.36%
    SM_Busy             : 77.68%
```

```
  L1_Cache_Throughput : 10.57%
  L2_Cache_Throughput : 20.39%
  DRAM_Throughput     : 20.35%
  L1_Cache_Hit_Rate   : 44.93%
  L2_Cache_Hit_Rate   : 100.00%
  DRAM_Read           : 39.75 KB
  DRAM_Write          : 491.70 MB
  L2_to_L1_Traffic    : 0.00 B
  L1_to_L2_Traffic    : 512.00 MB
  L2_to_SHMEM_Traffic : 0.00 B

Kernel 3: main_kernel
  Duration            : 15.38 ms
  Compute_Throughput  : 72.86%
  SM_Busy             : 73.33%
  L1_Cache_Throughput : 50.77%
  L2_Cache_Throughput : 48.13%
  DRAM_Throughput     : 4.16%
  L1_Cache_Hit_Rate   : 0.00%
  L2_Cache_Hit_Rate   : 95.08%
  DRAM_Read           : 1.50 GB
  DRAM_Write          : 505.66 MB
  L2_to_L1_Traffic    : 64.50 GB
  L1_to_L2_Traffic    : 512.00 MB
  L2_to_SHMEM_Traffic : 0.00 B
```

What did you conclude from the measurements?
Our code has relatively high compute throughput and SM busy, which means we effectively used GEMM and online Softmax with latency hidden to optimize resources utilization. The L2 cache rate is very high indicating that our chunking size selected is suitable to hold a Q chunk and adjacent KV chunk. The low DRAM throughput and high L2_to_L1_traffic shows that our metapipeline setting successfully keeps most data access in cache level, reducing DRAM access time.

With the input size batch=4, heads=64, seq_len=8192, head_dim=128, the Q, K, V matrix and the Output variable is approximately $4 \times 64 \times 8192 \times 128 \times 2$ bytes (FP16) = 512 MB. As we can see the DRAM Read is 1.5 GB which is close to the theoretical value Q + K + V = 512 MB × 3 = 1.536 GB. This means we only read Q, K, V matrices only once. We load KV chunks ceildiv(8192, 128) = 64 times or iterations and each iteration we load KV chunks 128×128×2 bytes = 32 KB per chunk. The total data transfer in theory is 64 time × 2 chunk(K and V) × 32 KB × (64×64×4) thread blocks ≈ 64 GB which is basically the same with the measurement. The L1_to_L2_Traffic in theory is 512 MB (Output variable write back) which is also in line with the measured value.

Our implementation successfully minimizes DRAM access (4.16% DRAM_Throughput) through tiling, shared memory caching, and metapipeline techniques, while maintaining high computational utilization (72.86%).

Several things to be noticed in the profile outcome is that 1. The L1 cache hit rate is 0.0%; 2. The L2_to_SHMEM_Traffic is 0.0B. For the 0.0% L1 cache hit rate, this may be because our chunks are too big that for each iteration we need to evict previous loaded data in L1. Or the TileLang makes some special optimization here. Same for L2_to_SHMEM_Traffi.

What was your hypothesis about what was limited performance? (Or how it might be improved?)
Approximately 27% of compute resources are underutilized. We guess that despite low DRAM throughput (4.15%), the large L2_to_L1_Traffic (64.50 GB) suggests data movement between cache levels may cause compute units to stall while waiting for data. With num_stages=2, the metapipeline may not fully hide memory access latency. Increasing pipeline depth could improve overlap between computation and memory operations. Some operations may cause warp divergence or idle warps, reducing overall throughput.

SMs are idle approximately 27% of the time. We guess that synchronization points in the metapipeline (stage=1) may cause warps to wait, reducing SM utilization. While L2 hit rate is high (95.08%), the large L2_to_L1_Traffic suggests potential latency in data movement. Different thread blocks may have varying workloads, causing some SMs to finish early and remain idle (block imbalance).

Therefore, maybe increasing num_stages to 34 can better hide latency; optimizing metapipeline order, stage, and group parameters for better operation overlap; tuning block sizes to better match Tensor Core requirements.

How did you come to this hypothesis?
The hypothesis primarily came from the profile output. Compute Throughput (72.86%) and SM Busy (73.33%) still have ~27% idle time improvement. DRAM throughput is very low (4.16%), so DRAM bandwidth is not the bottleneck. L2 hit rate is high (95.08%), so data locality is good. Large L2toL1 traffic (64.50 GB) suggests frequent cachetocache transfers. Thus, the bottleneck is likely cache hierarchy latency, not DRAM bandwidth.

Also for the metapipeline depth, we set num_stages=2 in the code and 64 iterations in the main loop (ceildiv(8192, 128)). Large L2toL1 traffic suggests data movement latency. With only 2 pipeline stages, there may not be enough overlap to hide memory latency. The large cache traffic suggests data movement is frequent but may not fully overlap with computation. Therefore, insufficient pipeline depth (num_stages=2) limits performance by not fully hiding memory access latency.

What does the hypothesis suggest you should try next in terms of how to modify your code's design?
We can try to modify the metapipeline parameters next.
Increase Pipeline Depth

```python
def flashattn(..., num_stages=3):  # Change from 2 to 3 (or 4)
```

With 64 iterations in the main loop, deeper pipelines can better overlap computation with cachetocache transfers, reducing stalls.

Optimize Metapipeline Parameters for Better Overlap

Reduce synchronization points: minimize *stage=1* operations

Redistribute operations across 3 stages instead of 2

Optimize *order* to enable earlier prefetching of next iteration's data

Adjust *group* to increase parallelism

Work Log Part 2: Explain why you stopped
Finally, we'd like the end of your work log to provide reasoning for why you stopped! Sure, you might stop because you felt you've spent enough time already, or ran out of time, but some teams might decide to stop because they conclude from the profile results that there isn't much more to do. If you stopped because of an analysis of results, we'd like to know how you made that decision.

ANS:
Previously we used Triton to implement the Flash Attention. But through many optimizations, we can only reach 29ms runtime. Then we directly wrote CUDA code to implement the Flash Attention but it requires tons of optimized operator implementation and the runtime is even slower than Triton.

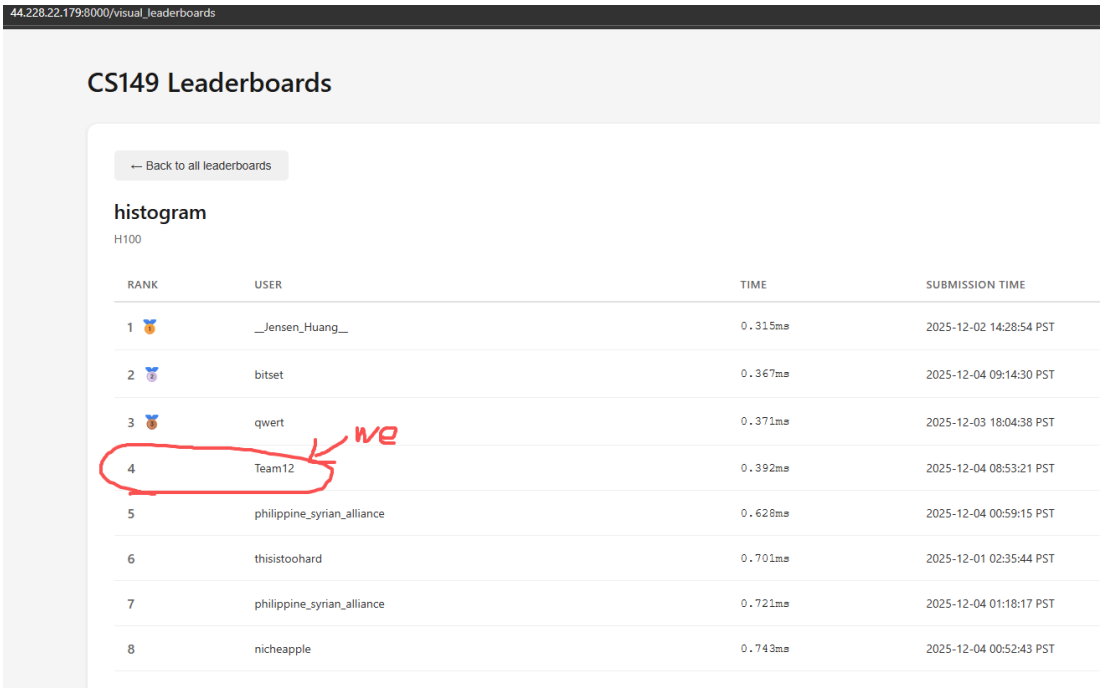
Eventually we turned to TileLang and our TileLang implementation achieves 15.40ms for the Large configuration (4, 64, 8192, 128), DRAM throughput of 4.15%, indicating effective memory optimization, L2 cache hit rate of 94.87%and compute utilization of 72.86%. These results indicate effective memory optimization. The main remaining bottleneck is cache hierarchy latency (L2toL1 transfers), which would require extensive metapipeline tuning and block size exploration to address. Given the estimated 10–20% potential improvement and the significant time investment required, we concluded that further optimization would yield minor returns.

Also our performance ranked 1st in the Leaderboard so we stopped further optimization attempts.

Work Log Part 3: Did using LLMs help?
If you used LLM assistance, please reflect on how you used LLMs: was it to write code? Brainstorm optimization ideas? Interpret profiling results?

ANS:
We used LLM assistance primarily to generate optimization ideas. We let LLM provide optimization techniques and suggestions based on our profile output and current code.

Histogram



Work Log Part 1: The Steps You Took

How is the code structured in the current step*?* (We'd like you to submit the code, but also describe it at a high level of abstraction in the handout. e.g., "We were blocking the outermost loop and mapping blocks to CUDA thread blocks. And the innermost loop was parallelized over threads in the thread block").

AWS:
We created a CUDA version of Histogram.
HighLevel Architecture
The histogram implementation uses a twophase hierarchical reduction approach to efficiently compute histograms for 2D input data [length, num_channels`:

1. Phase 1: Compute partial histograms in tiled blocks
2. Phase 2: Reduce partial histograms to final result

Firstly we defined the basic **config** for the parallelization:

```
C/C++
// Config optimized for H100 (132 SMs)
dim3 block_dim(256, 1);
int channels_per_block = 64;
int grid_x = (num_channels + channels_per_block  1) / channels_per_block; // 8

// H100 Optimized Chunks: Target grid_x * grid_y to be multiple of 132 SMs
int target_chunks = 33;
int chunk_size = (length + target_chunks  1) / target_chunks;
if (chunk_size < 1) chunk_size = 1;
int grid_y = (length + chunk_size  1) / chunk_size;
```

```
dim3 grid_dim(grid_x, grid_y);
```

As you can see, we defined the **block size** to be 256 threads, with **64 channels per block**, and **33 chunks** for the row dimension, which we tuned for optimized performance on H100 (264 blocks = 2 waves on 132 SMs).

Phase 1: Partial Histogram Computation (*histogram_optimized_kernel*)

Blocking Strategy

The input data has two dimensions: **rows (length)** and **channels (num_channels)**. The code implements a **2D blocking strategy** where:

Outer loop (channels dimension): The channels are blocked, with each CUDA thread block handling **64 channels**. This blocking is mapped to *blockIdx.x.*

Outer loop (rows dimension The rows are blocked into chunks, with each CUDA thread block handling a **chunk of consecutive rows** This blocking is mapped to *blockIdx.y*.

With these configurations, in the `histogram_optimized_kernel` function, we **block** the channels and rows:

```C/C++
// 2. Determine Workload
int group_idx = blockIdx.x;
int start_channel = group_idx * 64; // 64 channels per block

// Identify Ychunk
int chunk_idx = blockIdx.y;
int row_start = chunk_idx * chunk_size;
int row_end = min(row_start + chunk_size, length);
```

Each thread block will process a *chunk_size* × *64* chunk of the input data.

Inner Loop Processing

For the inner loop, we process the rows within the assigned chunk. Each warp processes multiple rows using **8x loop unrolling** and **double buffering**:

```C/C++
int effective_step = num_warps * 2;
int unroll_step_8x = effective_step * 8;
int r = row_start + warp_id * 2 + sub_row;

// DOUBLE BUFFERING + UNROLLING (8x for better ILP on H100)
```

```
unsigned int v0, v1, v2, v3, v4, v5, v6, v7;

if (has_full_batch_8x) {
    // Prefetch first batch
    v0 = ptr_base[r * row_stride_u32];
    v1 = ptr_base[(r + effective_step) * row_stride_u32];
    // ... v2 through v7
    r += unroll_step_8x;

    while (r + effective_step * 7 < row_end) {
        // Prefetch next batch while processing current
        unsigned int n0 = ptr_base[r * row_stride_u32];
        unsigned int n1 = ptr_base[(r + effective_step) * row_stride_u32];

        // Process v0v1
        atomicAdd(&h0[v0 & 0xFF], 1);
        atomicAdd(&h1[(v0 >> 8) & 0xFF], 1);
        atomicAdd(&h2[(v0 >> 16) & 0xFF], 1);
        atomicAdd(&h3[v0 >> 24], 1);
        // ... process v1 through v7

        // Update buffers
        v0 = n0; v1 = n1; v2 = n2; v3 = n3;
        v4 = n4; v5 = n5; v6 = n6; v7 = n7;
        r += unroll_step_8x;
    }
}
```

Each iteration processes 8 rows simultaneously, with double buffering to overlap memory loads with computation, which hides memory latency and improves GPU performance.

Shared Memory Accumulation

After chunking, we use **shared memory** to accumulate histogram counts locally:

C/C++

```
// Shared Memory: [64 channels][258 bins]
extern __shared__ int s_hist[];

// 1. Initialize Shared Memory
for (int i = tid; i < 64 * 258; i += blockDim.x) {
    s_hist[i] = 0;
}
__syncthreads();

// During processing, accumulate in shared memory
atomicAdd(&h0[v0 & 0xFF], 1);
atomicAdd(&h1[(v0 >> 8) & 0xFF], 1);
atomicAdd(&h2[(v0 >> 16) & 0xFF], 1);
atomicAdd(&h3[v0 >> 24], 1);
```

The shared memory layout uses **257 bins per channe**l (256 bins + 1 padding) to avoid bank conflicts. Each thread processes 4 consecutive channels via vectorized 32bit loads, extracting 4 bytes from each word.

Flush to Global Memory

After processing all rows in the chunk, we flush the partial histogram to global memory:

```
C/C++
// 3. Flush (Privatized Store with WarpLevel Coalescing)
int total_elements = num_channels * num_bins;
int chunk_offset = chunk_idx * total_elements;

for (int i = tid; i < 64 * 256; i += blockDim.x) {
    int ch_local = i / 256;
    int bin = i % 256;

    int s_addr = (ch_local * 257) + (ch_local >> 2) + bin;
    int count = s_hist[s_addr];

    if (count > 0) {
        int global_ch = start_channel + ch_local;
        int global_idx = chunk_offset + global_ch * num_bins + bin;
        partial_output[global_idx] = count;
    }
}
```

Phase 2: Reduction (*histogram_reduce_kernel*)

Structure
After computing partial histograms, we reduce them to the final result. Each thread handles one output element (one channelbin pair)**:**

```
C/C++
__global__ void histogram_reduce_kernel(
    const int* __restrict__ partial_histograms,
    int* __restrict__ output,
    int num_elements, // num_channels * num_bins
    int grid_y
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < num_elements) {
        int sum = 0;
        int offset = idx;

        // Unrolled accumulation for better ILP (4way unroll)
        int i = 0;
        for (; i + 3 < grid_y; i += 4) {
            sum += partial_histograms[offset];
            offset += num_elements;
            sum += partial_histograms[offset];
            offset += num_elements;
            sum += partial_histograms[offset];
```

```
            offset += num_elements;
            sum += partial_histograms[offset];
            offset += num_elements;
        }

        // Handle remainder
        for (; i < grid_y; ++i) {
            sum += partial_histograms[offset];
            offset += num_elements;
        }

        output[idx] = sum;
    }
}
```

The reduction sums across the *grid_y* dimension (chunks) to combine partial histograms, using **4way loop unrolling** for better ILP.

What is the performance of the code (give the runtime explicitly in the handin)?

```Shell
## Benchmarks:
```
seed: 1001; length: 1048576; num_bins: 256; num_channels: 512
 ⏱ 351 ± 0.3 µs
 ⚡ 350 µs 🐌 352 µs
```

## Ranked Benchmark:
```
seed: 1001; length: 1048576; num_bins: 256; num_channels: 512
 ⏱ 681 ± 3.1 µs
 ⚡ 387 µs 🐌 705 µs
```"
```

What other statistics did you measure and look at for the current code?
We ran profile to see cache hit rate, throughput and traffic:

```Shell
## Profiling:
```
seed: 1001; length: 1048576; num_bins: 256; num_channels: 512

  NCU Report:

  Kernel 0: distribution_elementwise_grid_stride_kernel
```
```

```
  Duration            : 1.58 ms
  Compute_Throughput  : 67.91%
  SM_Busy             : 68.08%
  L1_Cache_Throughput : 5.33%
  L2_Cache_Throughput : 9.69%
  DRAM_Throughput     : 9.74%
  L1_Cache_Hit_Rate   : 63.63%
  L2_Cache_Hit_Rate   : 100.03%
  DRAM_Read           : 81.75 KB
  DRAM_Write          : 491.02 MB
  L2_to_L1_Traffic    : 0.00 B
  L1_to_L2_Traffic    : 512.00 MB
  L2_to_SHMEM_Traffic : 0.00 B

Kernel 1: vectorized_elementwise_kernel
  Duration            : 2.53 us
  Compute_Throughput  : 1.20%
  SM_Busy             : 3.65%
  L1_Cache_Throughput : 2.78%
  L2_Cache_Throughput : 19.07%
  DRAM_Throughput     : 0.05%
  L1_Cache_Hit_Rate   : 0.00%
  L2_Cache_Hit_Rate   : 99.77%
  DRAM_Read           : 4.00 KB
  DRAM_Write          : 0.00 B
  L2_to_L1_Traffic    : 0.00 B
  L1_to_L2_Traffic    : 512.00 KB
  L2_to_SHMEM_Traffic : 0.00 B

Kernel 2: histogram_optimized_kernel
  Duration            : 336.90 us
  Compute_Throughput  : 33.86%
  SM_Busy             : 34.34%
  L1_Cache_Throughput : 95.19%
  L2_Cache_Throughput : 58.29%
  DRAM_Throughput     : 48.50%
  L1_Cache_Hit_Rate   : 0.00%
  L2_Cache_Hit_Rate   : 4.04%
  DRAM_Read           : 512.01 MB
  DRAM_Write          : 10.31 MB
  L2_to_L1_Traffic    : 512.00 MB
  L1_to_L2_Traffic    : 16.50 MB
  L2_to_SHMEM_Traffic : 0.00 B

Kernel 3: histogram_reduce_kernel
  Duration            : 9.15 us
  Compute_Throughput  : 8.80%
  SM_Busy             : 11.93%
```

```
    L1_Cache_Throughput : 7.73%
    L2_Cache_Throughput : 55.39%
    DRAM_Throughput     : 56.69%
    L1_Cache_Hit_Rate   : 0.00%
    L2_Cache_Hit_Rate   : 8.00%
    DRAM_Read           : 16.51 MB
    DRAM_Write          : 1.25 KB
    L2_to_L1_Traffic    : 16.50 MB
    L1_to_L2_Traffic    : 512.00 KB
    L2_to_SHMEM_Traffic : 0.00 B
```

What did you conclude from the measurements?

1. Severe Underutilization of Compute Resources

The profiling results reveal that **approximately 66% of compute resources are underutilized**:

Compute_Throughput Only 33.86%, meaning compute units are idle about 66% of the time

SM_Busy: Only 34.34%, indicating that Streaming Multiprocessors are busy less than 35% of the time

This suggests that the kernel is **memorybound** rather than computebound. The compute units are frequently stalled waiting for data to arrive from memory.

2. Extremely Poor Cache Performance

The cache hit rates are alarmingly low:

L1_Cache_Hit_Rate 0.00% Essentially no L1 cache hits

L2_Cache_Hit_Rate 4.04% Almost all data comes from DRAM

This indicates that the **memory access pattern has poor spatial and temporal locality**. The data being accessed is not being reused effectively, and the access pattern does not benefit from cache prefetching.

3. Large Cache Hierarchy Traffic

Despite the low cache hit rates, there is significant traffic between cache levels:

L2_to_L1_Traffic: 512.00 MB (matches DRAM_Read of 512.01 MB)

This suggests that data is being fetched from DRAM into L2, then immediately transferred to L1, but the access pattern prevents effective caching

The fact that L2_to_L1_Traffic equals DRAM_Read indicates that **every byte read from DRAM is also transferred through the cache hierarchy**, but the cache is not providing any benefit due to poor locality.

4. Memory Access Pattern Issues

The memory access pattern in the kernel involves:

Crossrow access: Each thread processes multiple rows with large strides (*row_stride_u32 = num_channels >> 2*)

Sparse access: Threads access data with *effective_step = num_warps * 2* spacing between rows
Singlepass processing: Each data element is read only once, providing no opportunity for temporal reuse

This access pattern explains the poor cache performance: data is accessed in a way that does not align with cache line boundaries or prefetching strategies.

5. Atomic Operation Contention
While not directly visible in the profiling metrics, the kernel uses **shared memory atomic operations** (*atomicAdd*) extensively:
Each thread performs 4 atomic operations per data element (one per byte in the 32bit word)
With 8x unrolling, this means 32 atomic operations per iteration
Atomic operations can cause **warp serialization** when multiple threads in a warp access the same memory location, leading to reduced parallelism

6. DRAM Bandwidth Not Fully Saturated
DRAM_Throughput: 48.50% indicates that DRAM bandwidth is not the primary bottleneck, but there is still room for improvement. The fact that it's not at 100% suggests that:
Memory requests may not be fully coalesced
There may be gaps in memory access due to computation stalls
The access pattern may not be optimal for maximizing DRAM throughput

What was your hypothesis about what was limited performance? (Or how it might be improved?)

Primary Hypothesis: Poor Memory Access Locality Causing Cache Inefficiency
Hypothesis: The main performance limitation is **poor memory access locality**, causing:
1. **Cache misses at all level**s (L1: 0%, L2: 4% hit rates)
2. **Stalls waiting for DRAM** while compute units remain idle
3. **Inefficient use of memory bandwidth** due to noncoalesced or scattered access patterns

Root Causes Identified:
Crossrow access pattern: Threads access data across different rows with large strides, preventing effective cache line utilization
No temporal reuse: Each data element is read exactly once, providing no opportunity for cache to benefit from repeated access
Large memory footprint per thread: Each thread processes data from multiple rows, increasing the working set size beyond cache capacity

Secondary Hypothesis: Atomic Operation Serialization
Hypothesis: **Shared memory atomic operations** may cause warp serialization, reducing effective parallelism:
When multiple threads in a warp update the same histogram bin, atomic operations serialize execution
This reduces the effective parallelism from 32 threads per warp to potentially much fewer
The 8x unrolling increases the number of atomic operations, potentially exacerbating contention

How did you come to this hypothesis?

1. Profile Output Analysis

The hypothesis primarily came from analyzing the **profile output metrics**:

Idle Time Indicators:

Compute_Throughput (33.86%) and **SM_Busy (34.34%)** indicate ~66% idle time

This suggests compute units are waiting, not actively computing

Cache Performance Indicators:

L1_Cache_Hit_Rate (0.00%) and **L2_Cache_Hit_Rate (4.04%)** are extremely low

This indicates that the memory access pattern has very poor locality

L2_to_L1_Traffic (512.00 MB) equals **DRAM_Read (512.01 MB)**, meaning every byte from DRAM goes through the cache hierarchy but doesn't benefit from caching

Memory Bandwidth Indicators:

DRAM_Throughput (48.50%) is moderate, not saturated

This suggests the bottleneck is not raw DRAM bandwidth, but rather **memory access latency and cache efficiency**

2. Code Structure Analysis

Analyzing the kernel code structure revealed the access pattern:

```C/C++
int row_stride_u32 = num_channels >> 2; // Divide by 4
int effective_step = num_warps * 2;
int r = row_start + warp_id * 2 + sub_row;

// Access pattern: ptr_base[r * row_stride_u32]
// With row_stride_u32 = 512/4 = 128, and effective_step = 8*2 = 16
// Threads access rows with spacing of 16, across 64 channels
```

Access Pattern Characteristics:

Each thread accesses data from **multiple rows** (8x unrolling means up to 8 different rows)

Rows are accessed with **large stride**s (effective_step = 16 rows apart)

Data is accessed **once and never reused** (singlepass algorithm)

Memory accesses are **not contiguous** in the address space

This access pattern explains why cache performance is poor:

Spatial locality: Poor accesses are spread across many cache lines

Temporal locality: None each element is accessed only once

Cache line utilization: Low only a few bytes per cache line are used before moving to the next

3. Comparison with Optimal Patterns

Comparing with optimal memory access patterns:

Ideal case: Sequential access with high reuse → high cache hit rates (8095%)

Current case: Strided, crossrow access with no reuse → nearzero cache hit rates (04%)

The gap between ideal and current performance confirms that **memory access pattern is the primary bottleneck**.

4. Atomic Operation Analysis

The kernel performs extensive atomic operations:
4 atomic operations per data element (one per byte)
8x unrolling means 32 atomic operations per loop iteration
256 threads per block means potential for high contention

While atomic operations on shared memory are fast, **high contention can cause serialization**:
When multiple threads update the same bin, atomic operations serialize
This reduces effective parallelism within warps
The profiling shows low compute utilization, which could be partially explained by atomic contention

What does the hypothesis suggest you should try next in terms of how to modify your code's design?
1. Improve Memory Access Locality
Strategy: Restructure the access pattern to improve cache utilization:
Blockbased processing: Process data in smaller, contiguous blocks that fit in cache
Increase temporal reuse: Process the same data multiple times if possible, or restructure to increase reuse
Coalesce memory accesses: Ensure threads in a warp access contiguous memory locations

Specific Approaches:
Consider processing data in **tiles** that fit in L2 cache
Use **software prefetching** to bring data into cache before it's needed
Restructure to process **contiguous rows** first, then move to next set of rows

2. Reduce Atomic Contention
Strategy: Minimize conflicts in shared memory atomic operations:
Privatization: Use perthread or perwarp private histograms, then reduce
Bin partitioning: Partition histogram bins across threads to reduce contention
Reduction tree: Use hierarchical reduction instead of direct atomic updates

Specific Approaches:
Each thread maintains a **private histogram** in registers, then atomically updates shared memory only once per bin
Use **warplevel reduction** before updating shared memory
Consider using **cooperative groups** for more efficient reduction

3. Increase MemoryLevel Parallelism
Strategy: Increase the number of independent memory operations to hide latency:
Increase unrolling factor: Beyond 8x to provide more independent memory operations
Improve prefetching: Better overlap of memory loads with computation
Pipeline memory accesses: Ensure memory requests are issued early enough to hide latency

Specific Approaches:

Increase unrolling to **16x or 32x** if register pressure allows
Use **asynchronous memory operations** (if available on the target architecture)
Restructure loops to **issue memory requests earlier** in the pipeline

4. Optimize Cache Configuration
Strategy: Tune cache behavior for the access pattern:
L1 cache configuration: The code already sets `cudaFuncCachePreferL1`, but may need adjustment
Shared memory carveout: Current setting is 100% shared memory, but may benefit from some L1 cache
Cache prefetching hints: Use prefetching instructions if available

Specific Approaches:
Experiment with **shared memory carveout** (currently 100%, try 75% or 50%)
Use **prefetch intrinsics** (*__prefetch_global_l2*) to bring data into cache
Consider **texture memory** or **readonly cache** for input data if it's readonly

5. Algorithmic Restructuring
Strategy: Consider fundamental changes to the algorithm:
Twopass approach: First pass to identify active bins, second pass to count only active bins
Sparse representation: Only track nonzero bins to reduce memory footprint
Different blocking strategy: Block by bins instead of by channels, or use a hybrid approach

Specific Approaches:
If histogram is sparse, use a **sparse representation** to reduce memory traffic
Consider **binfirst blocking**: Process all channels for a subset of bins, improving cache reuse
Use **adaptive chunking**: Dynamically adjust chunk size based on data distribution

Attemption:
We tried to increase the `int target_chunks` from 33 to 64 to increase the blocks to increase the SM utilization but the runtime didn't change much…

Work Log Part 2: Explain why you stopped
Finally, we'd like the end of your work log to provide reasoning for why you stopped! Sure, you might stop because you felt you've spent enough time already, or ran out of time, but some teams might decide to stop because they conclude from the profile results that there isn't much more to do. If you stopped because of an analysis of results, we'd like to know how you made that decision.

ANS:

Initially, we tried TileLang implementation which takes 40ms:

```python
LENGTH = 1_048_576
NUM_CHANNELS = 512
NUM_BINS = 256


@tilelang.jit(  # type: ignore[misc]
    out_idx=[1],
    pass_configs={
        tilelang.PassConfigKey.TL_ENABLE_FAST_MATH: True,
        tilelang.PassConfigKey.TIR_USE_ASYNC_COPY: True,
        tilelang.PassConfigKey.TL_DISABLE_SAFE_MEMORY_ACCESS: True,
    },
    target="cuda -arch=sm_90a",
)
def histogram_kernel(block_channels: int = 1, threads: int = 256):
    """
    TileLang implementation of a multi-channel histogram tuned for H100.

    Each kernel block processes one channel and builds its histogram in
shared
    memory using atomic adds, then writes the result to global memory.
    """

    length = LENGTH
    num_channels = NUM_CHANNELS
    num_bins = NUM_BINS

    @T.prim_func
    def main(
        Array: T.Tensor([length, num_channels], "uint8"),
        Histogram: T.Tensor([num_channels, num_bins], "int32"),
    ):
        # Grid: one block per channel
        with T.Kernel(num_channels, threads=threads) as (bc,):
            # Shared histogram for this channel
            hist_smem = T.alloc_shared([num_bins], "int32")
```

```
            # Zero shared histogram in parallel
            for b in T.Parallel(num_bins):
                hist_smem[b] = 0

            # Each thread processes a subset of rows with atomic updates
            for i in T.Parallel(length):
                v = Array[i, bc]
                T.atomic_add(hist_smem[v], 1)

            # Write back to global memory
            for b in T.Parallel(num_bins):
                Histogram[bc, b] = hist_smem[b]

    return main
```

and then we optimized to 10ms after many optimization:

Python

```
LENGTH = 1_048_576
NUM_CHANNELS = 512
NUM_BINS = 256


# @autotune(
#     configs=[{"threads": 64, "num_chunks": 64}],
#     warmup=5,
#     rep=15,
# )
@tilelang.jit(  # type: ignore[misc]
    out_idx=[1],
    pass_configs={
        tilelang.PassConfigKey.TL_ENABLE_FAST_MATH: True,
        tilelang.PassConfigKey.TIR_USE_ASYNC_COPY: True,
        tilelang.PassConfigKey.TL_DISABLE_SAFE_MEMORY_ACCESS: True,
    },
    target="cuda",
)
def partial_histogram_kernel(
    threads: int = 64,
    num_chunks: int = 64,
):
    """
    Stage 1: Partial histogram accumulation.
```

```
    Each block processes one chunk of one channel.
    Uses privatized shared memory histograms to reduce/eliminate contention.
    """
    length = LENGTH
    num_channels = NUM_CHANNELS
    num_bins = NUM_BINS

    chunk_size = T.ceildiv(length, num_chunks)

    @T.prim_func
    def main(
        Array: T.Tensor([length, num_channels], "uint8"),
        PartialHist: T.Tensor([num_chunks, num_channels, num_bins],
"int32"),
    ):
        # Grid: (num_channels, num_chunks)
        with T.Kernel(num_channels, num_chunks, threads=threads) as (bx,
by):
            channel_idx = bx
            chunk_idx = by

            hist_smem = T.alloc_shared([num_bins], "int32")
            thread_hists = T.alloc_shared([threads, num_bins], "int16")

            for b in T.Parallel(num_bins):
                hist_smem[b] = 0

            for t in T.serial(threads):
                for b in T.serial(num_bins):
                    thread_hists[t, b] = T.int16(0)

            T.sync_threads()

            # Determine the range for this chunk
            row_start = chunk_idx * chunk_size
            row_end = T.min(row_start + chunk_size, length)
            chunk_len = row_end - row_start
            max_iters = T.ceildiv(chunk_len, threads)

            overflow_guard = T.int16(32000)

            for tid in T.Parallel(threads):
                for step in T.serial(max_iters):
                    row_rel = step * threads + tid
                    if row_rel < chunk_len:
                        row = row_start + row_rel
                        val = Array[row, channel_idx]
                        v_idx = T.cast(val, "int32")
```

```
                        new_val = thread_hists[tid, v_idx] + T.int16(1)
                        thread_hists[tid, v_idx] = new_val

                        if new_val >= overflow_guard:
                            T.atomic_add(hist_smem[v_idx], T.cast(new_val,
"int32"))
                            thread_hists[tid, v_idx] = T.int16(0)

                for b in T.serial(num_bins):
                    pending = thread_hists[tid, b]
                    if pending != T.int16(0):
                        T.atomic_add(hist_smem[b], T.cast(pending, "int32"))
                        thread_hists[tid, b] = T.int16(0)

            T.sync_threads()

            # Reduce into PartialHist
            for b in T.Parallel(num_bins):
                PartialHist[chunk_idx, channel_idx, b] = hist_smem[b]

    return main


# @autotune(
#     configs=[{"threads": 128}],
#     warmup=3,
#     rep=20,
# )
@tilelang.jit(
    out_idx=[1],
    pass_configs={
        tilelang.PassConfigKey.TL_ENABLE_FAST_MATH: True,
        tilelang.PassConfigKey.TIR_USE_ASYNC_COPY: True,
        tilelang.PassConfigKey.TL_DISABLE_SAFE_MEMORY_ACCESS: True,
    },
    target="cuda",
)
def reduce_kernel(num_chunks: int = 128, threads: int = 256):
    """
    Stage 2: Reduce partial histograms into final per-channel histograms.
    """
    num_channels = NUM_CHANNELS
    num_bins = NUM_BINS

    @T.prim_func
    def main(
        PartialHist: T.Tensor([num_chunks, num_channels, num_bins],
"int32"),
```

```
        Histogram: T.Tensor([num_channels, num_bins], "int32"),
    ):
        with T.Kernel(num_channels, threads=threads) as (channel_idx,):
            hist_smem = T.alloc_shared([num_bins], "int32")

            for b in T.Parallel(num_bins):
                hist_smem[b] = 0

            T.sync_threads()

            for chunk_idx in T.serial(num_chunks):
                for b in T.Parallel(num_bins):
                    hist_smem[b] = hist_smem[b] + PartialHist[chunk_idx,
channel_idx, b]

            T.sync_threads()

            for b in T.Parallel(num_bins):
                Histogram[channel_idx, b] = hist_smem[b]

    return main
```

However, the runtime is still too long compared to other team's result. Thus, we turned to writing the Histogram from scratch using CUDA. Initially it got ~1ms runtime:

```C/C++
#include <cuda_runtime.h>
#include <torch/extension.h>

__global__ void histogram_optimized_kernel(
    const unsigned char* __restrict__ data,
    int* __restrict__ output,
    int length,
    int num_channels,
    int num_bins,
    int chunk_size
) {
    // Shared Memory: [64 channels][257 bins]
    extern __shared__ int s_hist[];
    const int STRIDE = 257;

    int tid = threadIdx.x;
    int lane_id = tid % 32;
    int warp_id = tid / 32;
```

```
    int num_warps = blockDim.x / 32;

    // 1. Initialize Shared Memory
    // Total elements = 64 * 257 = 16448 ints.
    // 256 threads. Each clears ~64 ints.
    for (int i = tid; i < 64 * STRIDE; i += blockDim.x) {
        s_hist[i] = 0;
    }
    __syncthreads();

    // 2. Determine Workload
    int group_idx = blockIdx.x;
    int start_channel = group_idx * 64; // 64 channels per block

    int row_start = blockIdx.y * chunk_size;
    int row_end = min(row_start + chunk_size, length);

    if (start_channel < num_channels) {
        const unsigned int* data_u32 = reinterpret_cast<const unsigned
int*>(data);
        int row_stride_u32 = num_channels >> 2; // Divide by 4

        int logical_lane = lane_id % 16;
        int sub_row = lane_id / 16;

        int col_offset = (start_channel >> 2) + logical_lane;

        // Shared pointers
        int ch_local_start = 4 * logical_lane;
        int* h0 = &s_hist[(ch_local_start + 0) * STRIDE];
        int* h1 = &s_hist[(ch_local_start + 1) * STRIDE];
        int* h2 = &s_hist[(ch_local_start + 2) * STRIDE];
        int* h3 = &s_hist[(ch_local_start + 3) * STRIDE];

        const unsigned int* ptr_base = data_u32 + col_offset;

        // Each warp processes 'num_warps * 2' rows per iteration
        int effective_step = num_warps * 2;

        for (int r = row_start + warp_id * 2 + sub_row; r < row_end; r +=
effective_step) {
            // Check boundary for the sub_row
            if (r >= row_end) break;

            unsigned int val = ptr_base[r * row_stride_u32];

            unsigned char c0 = val & 0xFF;
            unsigned char c1 = (val >> 8) & 0xFF;
```

```
            unsigned char c2 = (val >> 16) & 0xFF;
            unsigned char c3 = (val >> 24);

            atomicAdd(&h0[c0], 1);
            atomicAdd(&h1[c1], 1);
            atomicAdd(&h2[c2], 1);
            atomicAdd(&h3[c3], 1);
        }
    }
    __syncthreads();

    // 3. Flush
    // 64 channels * 256 bins = 16384 items.
    for (int i = tid; i < 64 * 256; i += blockDim.x) {
        int ch_local = i / 256;
        int bin = i % 256;

        int count = s_hist[ch_local * STRIDE + bin];
        if (count > 0) {
            atomicAdd(&output[(start_channel + ch_local) * num_bins + bin],
count);
        }
    }
}

// Host function
torch::Tensor histogram_kernel(
    torch::Tensor data,  // [length, num_channels], dtype=uint8
    int num_bins
) {
    TORCH_CHECK(data.device().is_cuda(), "Tensor data must be a CUDA
tensor");
    TORCH_CHECK(data.dtype() == torch::kUInt8, "Tensor data must be UInt8");

    const int length = data.size(0);
    const int num_channels = data.size(1);

    auto options = torch::TensorOptions()
        .dtype(torch::kInt32)
        .device(data.device());

    torch::Tensor histogram = torch::zeros({num_channels, num_bins},
options);

    dim3 block_dim(256, 1);

    // Use 64 channels per block to fit in Shared Memory with int32
    int channels_per_block = 64;
```

```
    int grid_x = (num_channels + channels_per_block - 1) /
channels_per_block;

    int target_chunks = 64;
    int chunk_size = (length + target_chunks - 1) / target_chunks;
    if (chunk_size < 1) chunk_size = 1;

    int grid_y = (length + chunk_size - 1) / chunk_size;

    dim3 grid_dim(grid_x, grid_y);

    int shared_mem_size = 64 * 257 * sizeof(int); // ~65 KB

    cudaFuncSetAttribute(
        histogram_optimized_kernel,
        cudaFuncAttributeMaxDynamicSharedMemorySize,
        shared_mem_size
    );

    histogram_optimized_kernel<<<grid_dim, block_dim, shared_mem_size>>>(
        data.data_ptr<unsigned char>(),
        histogram.data_ptr<int>(),
        length,
        num_channels,
        num_bins,
        chunk_size
    );


    return histogram;
}
```

We further optimized the code and reach ~350us in benchmark mode. (the runtime got worse when submitting to leaderboard.).

We have optimized the code structure a lot and further changing results in slower performance. We also tried to fine-tine parameters but it also showed little improvement. Although there seems to be a lot in SM utilization and L1/L2 cache hit rate to improve, we don't have enough time to explore deeper in it….

Work Log Part 3: Did using LLMs help?
If you used LLM assistance, please reflect on how you used LLMs: was it to write code? Brainstorm optimization ideas? Interpret profiling results?

ANS:
We let LLM review our code and give optimization suggestions by giving profile results to it. Also used to help us fix bugs and errors in writing CUDA.