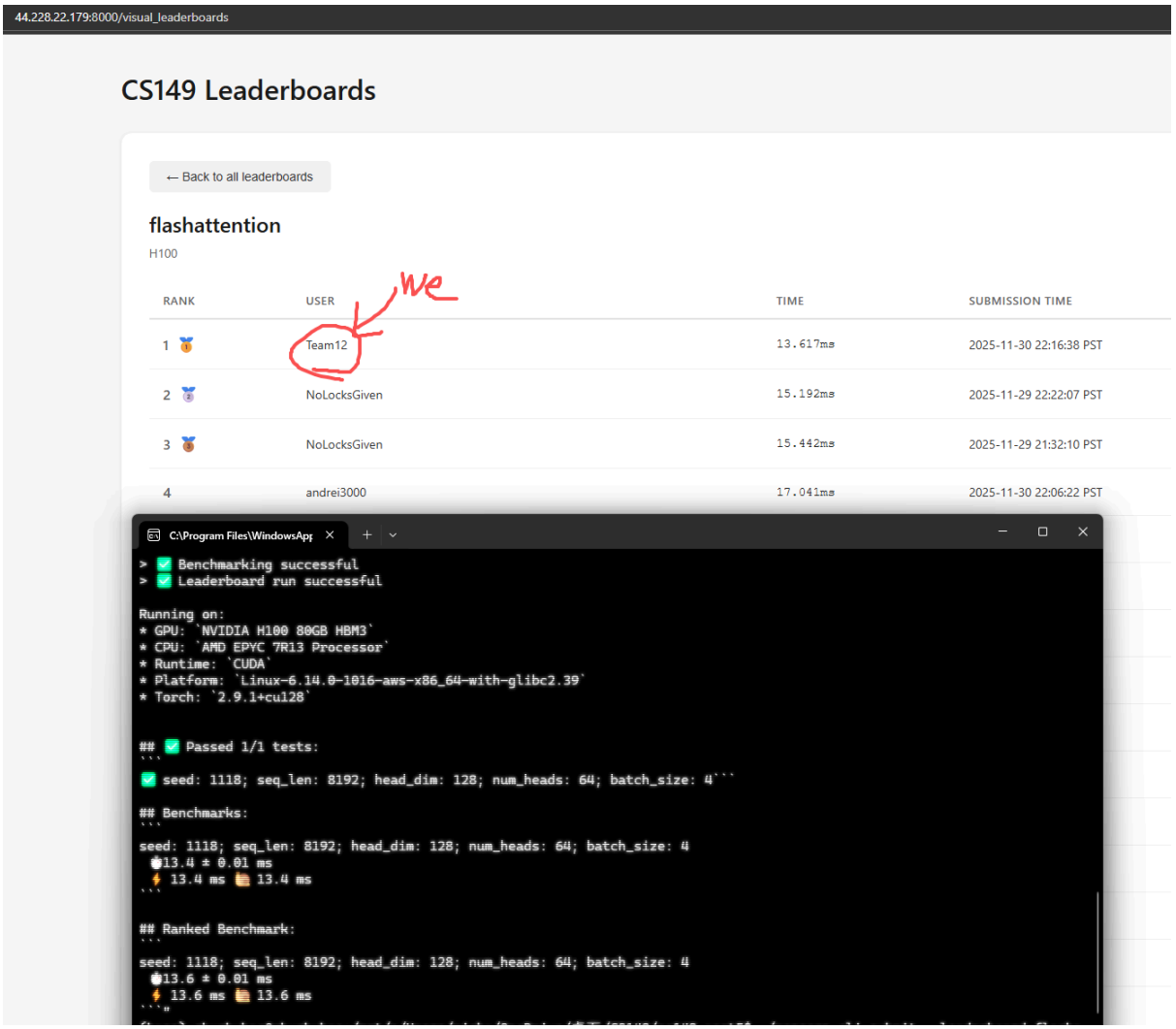
Partners: Chenhao Zhu (chenhzhu), Zhenyu Chen (zhenyuc)

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 hand-out. 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 two-stage **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 row-wise 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
Non--1 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 3-11 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 1-2: Mask filling operations in MMA0 (parallel loops)
Operations 3-11: 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 1-2 (parallel group): Mask filling operations
Operations 3-11 (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 1-2: Mask filling
Operations 3-11: 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 3-4 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 L2-to-L1 traffic (64.50 GB) suggests frequent cache-to-cache 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 L2-to-L1 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 cache-to-cache 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 (L2-to-L1 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.