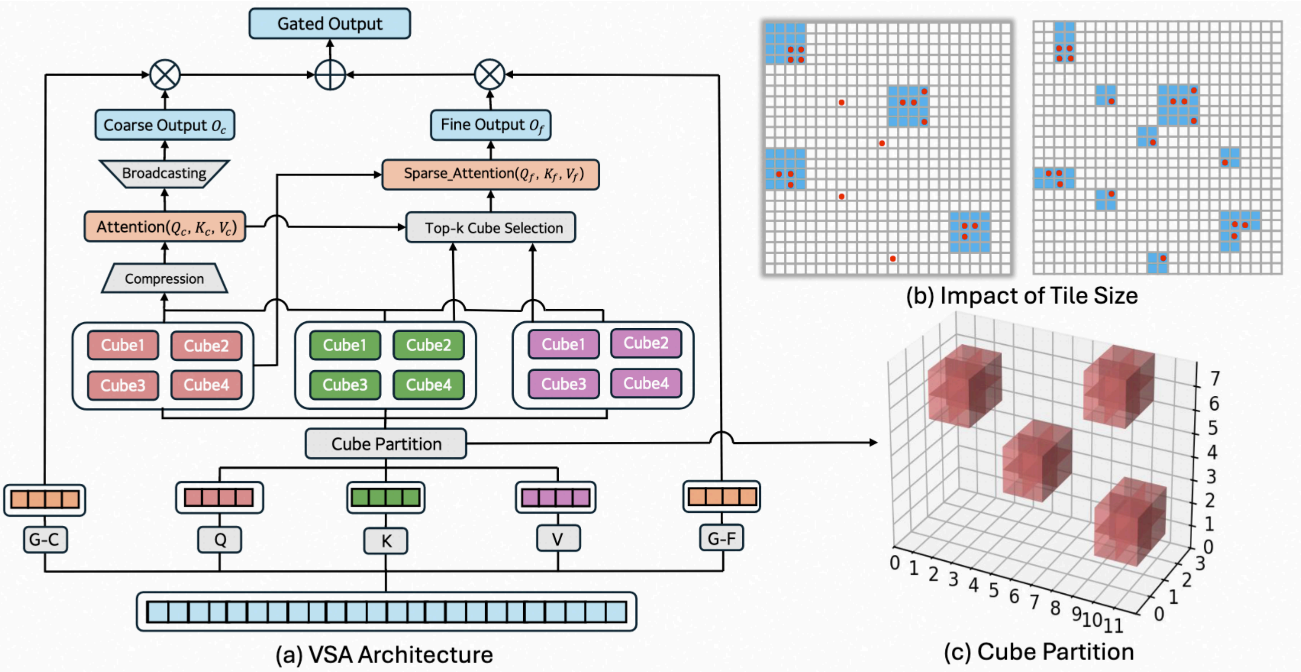
BlockVid2_kernel



(a) VSA Architecture

(b) Impact of Tile Size

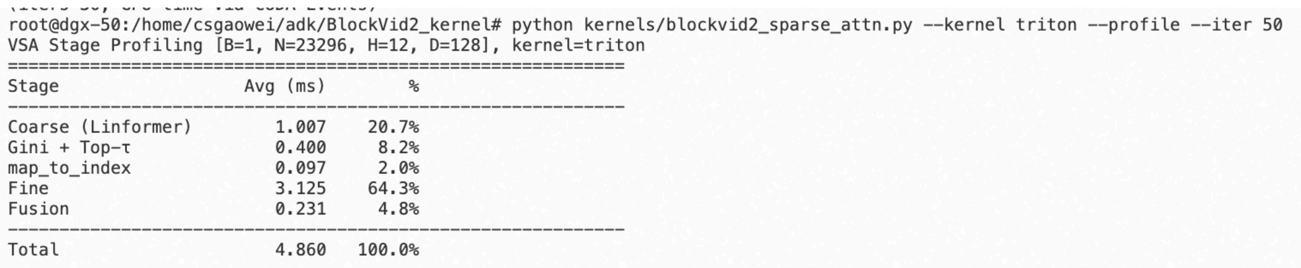
(c) Cube Partition

左侧替换为为Linformer的结构（transformer基础上加两个矩阵乘降低kv的seq_len），得到的attn_score用来选择稀疏的kv block。

BLOCK_SIZE=64 tokens，对每个 query block 取一组 KV blocks，并且每个 KV block 允许不同的有效长度的token（variable_block_sizes[kv_idx]），然后把这些 KV 片段 cat 成一段连续的 k_blk/v_blk 来做注意力。

v0

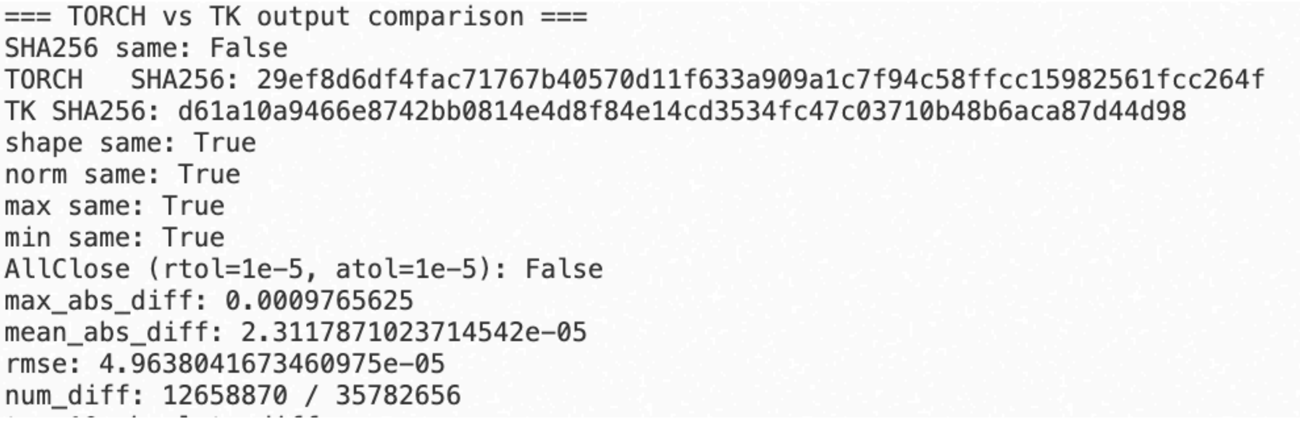
profile:



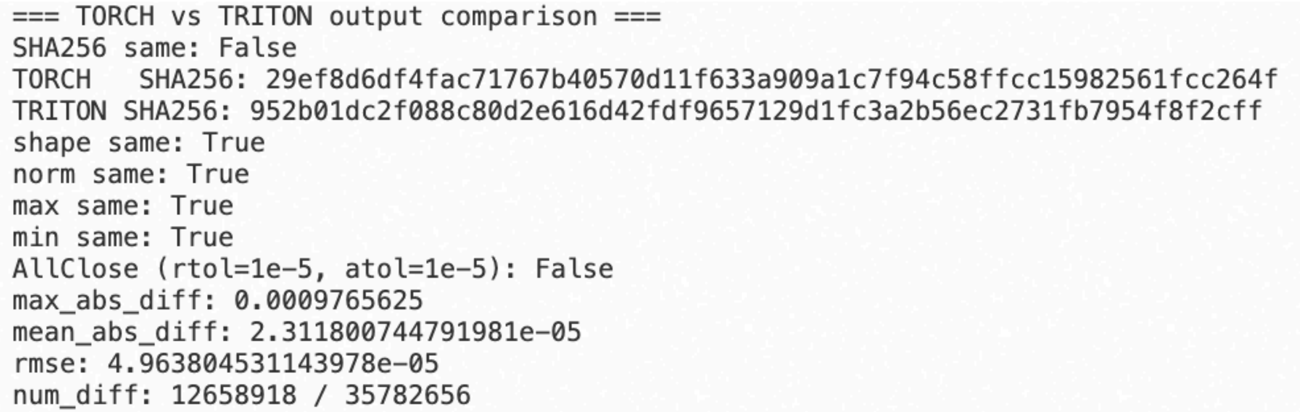
```
root@dgx-50:/home/csgaowei/adk/BlockVid2_kernel# python kernels/blockvid2_sparse_attn.py --kernel triton --profile --iter 50
VSA Stage Profiling [B=1, N=23296, H=12, D=128], kernel=triton
============================================================
Stage                    Avg (ms)         %
------------------------------------------------------------
Coarse (Linformer)          1.007     20.7%
Gini + Top-τ                0.400      8.2%
map_to_index                0.097      2.0%
Fine                        3.125     64.3%
Fusion                      0.231      4.8%
------------------------------------------------------------
Total                       4.860    100.0%
```

v1: 使用TK

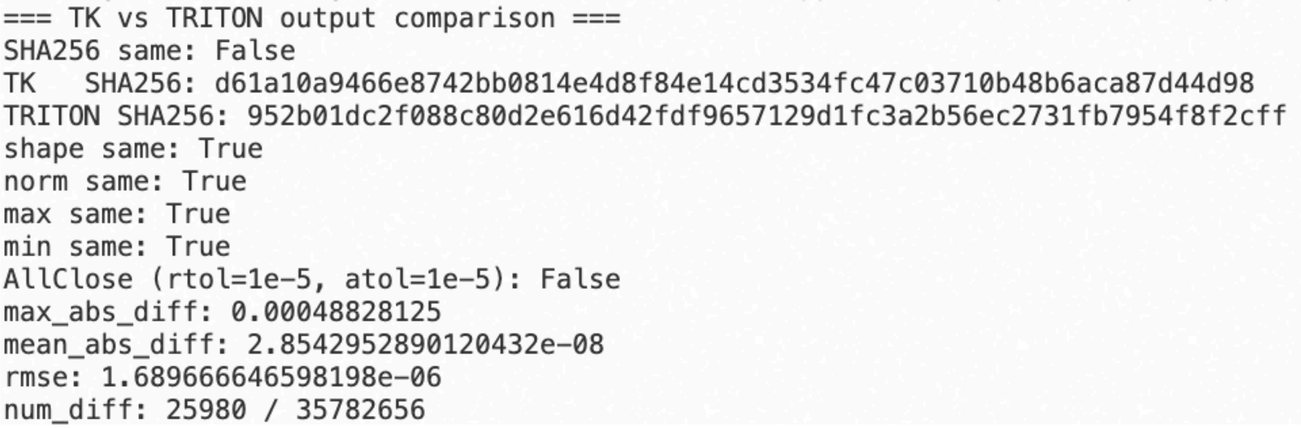
精度对比：



```
=== TORCH vs TK output comparison ===
SHA256 same: False
TORCH   SHA256: 29ef8d6df4fac71767b40570d11f633a909a1c7f94c58ffcc15982561fcc264f
TK SHA256: d61a10a9466e8742bb0814e4d8f84e14cd3534fc47c03710b48b6aca87d44d98
shape same: True
norm same: True
max same: True
min same: True
AllClose (rtol=1e-5, atol=1e-5): False
max_abs_diff: 0.0009765625
mean_abs_diff: 2.3117871023714542e-05
rmse: 4.9638041673460975e-05
num_diff: 12658870 / 35782656
```



```
=== TORCH vs TRITON output comparison ===
SHA256 same: False
TORCH   SHA256: 29ef8d6df4fac71767b40570d11f633a909a1c7f94c58ffcc15982561fcc264f
TRITON SHA256: 952b01dc2f088c80d2e616d42fdf9657129d1fc3a2b56ec2731fb7954f8f2cff
shape same: True
norm same: True
max same: True
min same: True
AllClose (rtol=1e-5, atol=1e-5): False
max_abs_diff: 0.0009765625
mean_abs_diff: 2.311800744791981e-05
rmse: 4.963804531143978e-05
num_diff: 12658918 / 35782656
```



```
=== TK vs TRITON output comparison ===
SHA256 same: False
TK   SHA256: d61a10a9466e8742bb0814e4d8f84e14cd3534fc47c03710b48b6aca87d44d98
TRITON SHA256: 952b01dc2f088c80d2e616d42fdf9657129d1fc3a2b56ec2731fb7954f8f2cff
shape same: True
norm same: True
max same: True
min same: True
AllClose (rtol=1e-5, atol=1e-5): False
max_abs_diff: 0.00048828125
mean_abs_diff: 2.8542952890120432e-08
rmse: 1.689666646598198e-06
num_diff: 25980 / 35782656
```

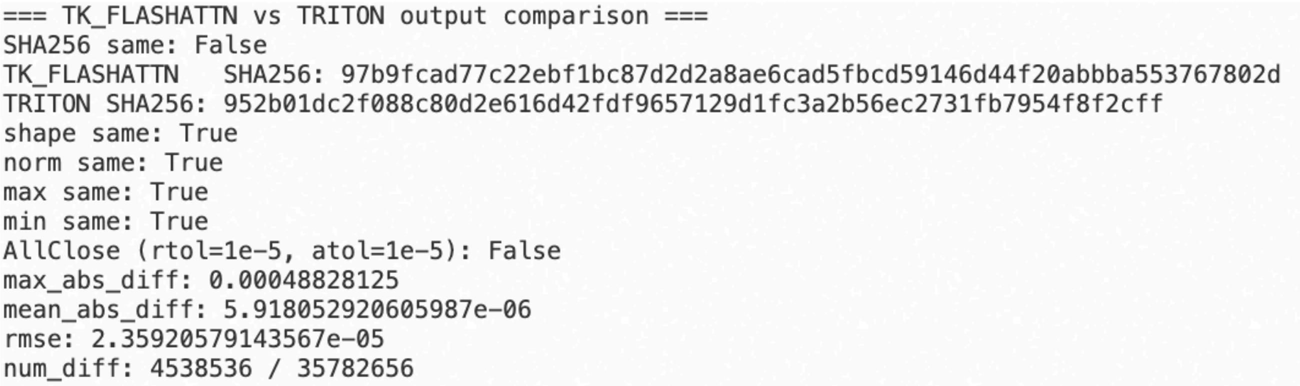
这里TK和triton的精度已经很接近了，tk/triton和torch原版都有精度差异

v2: linformer使用flashattention

flash_attention只能得到output，不能得到attn_scores的矩阵，所以两处coarse_attention变成了两个函数，gini处的coarse_attention无法使用flashattention。同时gini处可以只计算attn_scores，省去一个乘法计算

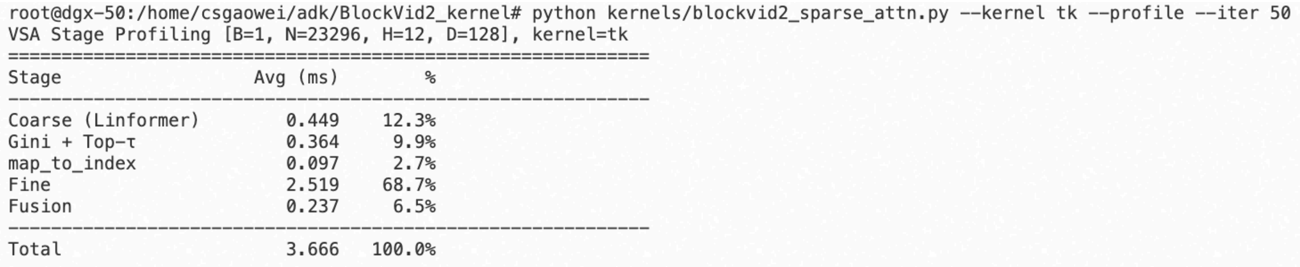
省去一些transpose。

精度对比:



```
=== TK_FLASHATTN vs TRITON output comparison ===
SHA256 same: False
TK_FLASHATTN   SHA256: 97b9fcad77c22ebf1bc87d2d2a8ae6cad5fbcd59146d44f20abbba553767802d
TRITON SHA256: 952b01dc2f088c80d2e616d42fdf9657129d1fc3a2b56ec2731fb7954f8f2cff
shape same: True
norm same: True
max same: True
min same: True
AllClose (rtol=1e-5, atol=1e-5): False
max_abs_diff: 0.00048828125
mean_abs_diff: 5.918052920605987e-06
rmse: 2.35920579143567e-05
num_diff: 4538536 / 35782656
```

时间:

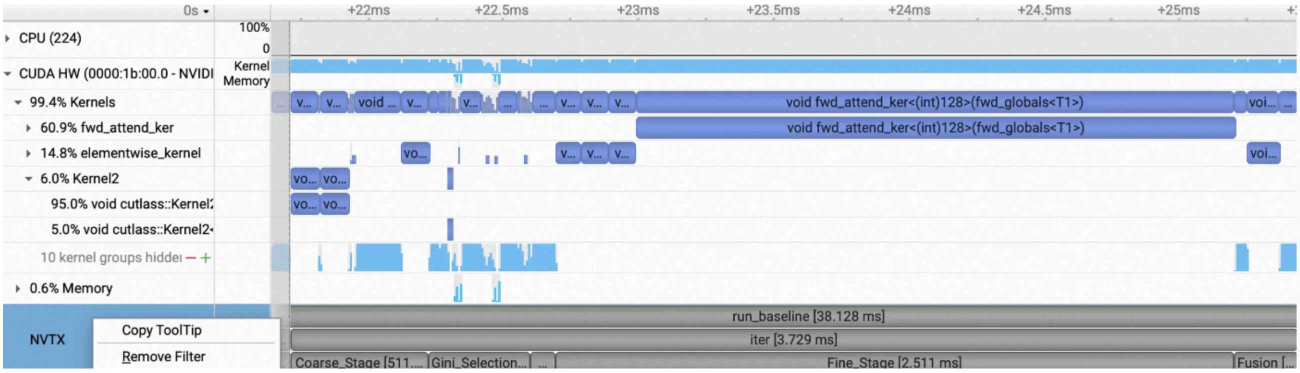


```
root@dgx-50:/home/csgaowei/adk/BlockVid2_kernel# python kernels/blockvid2_sparse_attn.py --kernel tk --profile --iter 50
VSA Stage Profiling [B=1, N=23296, H=12, D=128], kernel=tk
============================================================
Stage                  Avg (ms)        %
------------------------------------------------------------
Coarse (Linformer)        0.449    12.3%
Gini + Top-τ              0.364     9.9%
map_to_index              0.097     2.7%
Fine                      2.519    68.7%
Fusion                    0.237     6.5%
------------------------------------------------------------
Total                     3.666   100.0%
```

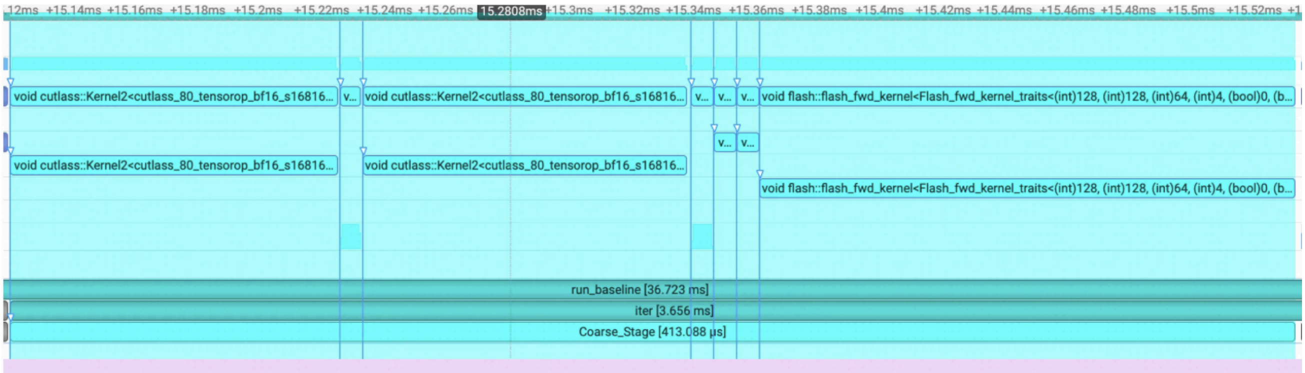
	triton	+tk	+flash_attn +减少不必要的计算
Coarse (Linformer)	1.007		**0.449**
Gini+top-t	0.400		**0.364**
map_to_index	0.097		**0.097**
Fine	3.125	**2.503**	**2.519**
Fusion	0.231		**0.237**
TOTAL	4.860	**4.260 (14.1%)**	**3.666 (32.6%)**

- context length 更长测试

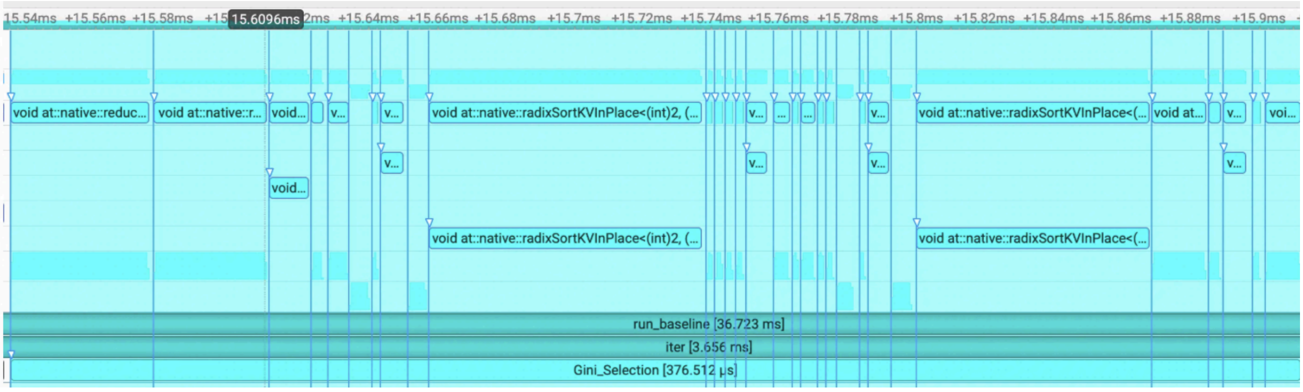
目前的profile



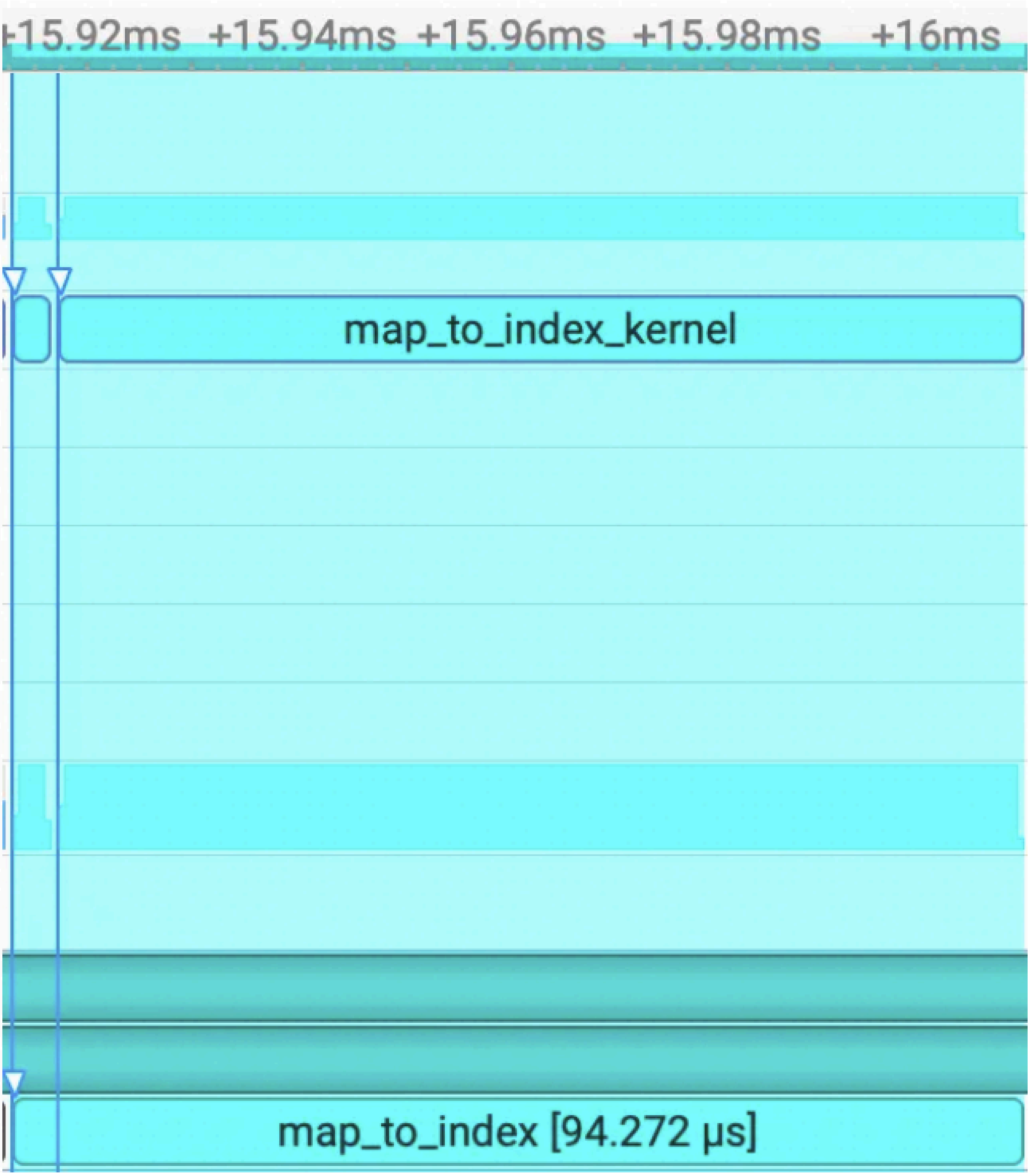
coarse stage:


+15.14ms
+15.16ms
+15.18ms
+15.2ms
+15.22ms
+15.24ms
+15.26ms
15.2808ms
+15.3ms
+15.32ms
+15.34ms
+15.36ms
+15.38ms
+15.4ms
+15.42ms
+15.44ms
+15.46ms
+15.48ms
+15.5ms
+15.52ms
void cutlass::Kernel2<cutlass_80_tensorop_bf16_s16816...
void cutlass::Kernel2<cutlass_80_tensorop_bf16_s16816...
void flash::flash_fwd_kernel<Flash_fwd_kernel_traits<(int)128, (int)128, (int)64, (int)4, (bool)0, (b...
void cutlass::Kernel2<cutlass_80_tensorop_bf16_s16816...
void cutlass::Kernel2<cutlass_80_tensorop_bf16_s16816...
void flash::flash_fwd_kernel<Flash_fwd_kernel_traits<(int)128, (int)128, (int)64, (int)4, (bool)0, (b...
run_baseline [36.723 ms]
iter [3.656 ms]
Coarse_Stage [413.088 μs]

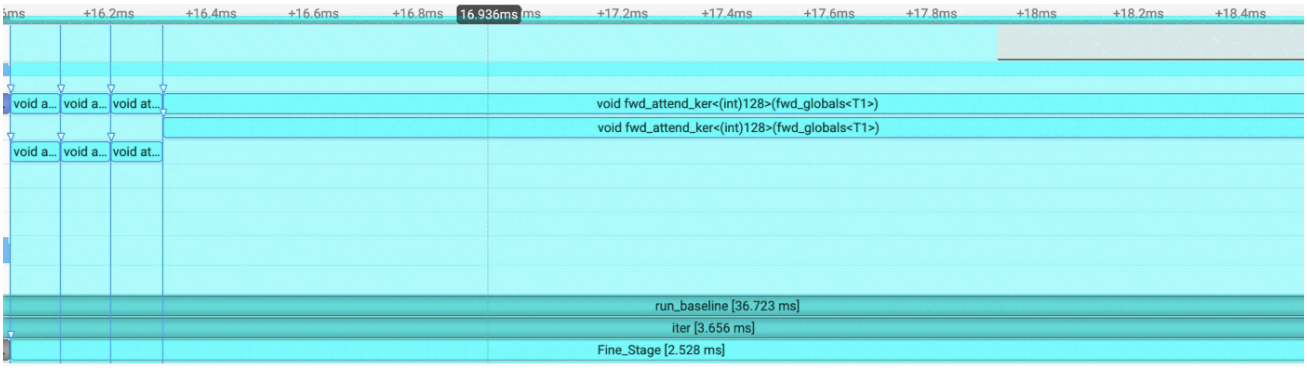
gini selection:


+15.54ms
+15.56ms
+15.58ms
15.6096ms
+15.64ms
+15.66ms
+15.68ms
+15.7ms
+15.72ms
+15.74ms
+15.76ms
+15.78ms
+15.8ms
+15.82ms
+15.84ms
+15.86ms
+15.88ms
+15.9ms
void at::native::reduc...
void at::native::r...
void at::native::radixSortKVInPlace<(int)2, (...
void at::native::radixSortKVInPlace<(...
void at...
void at::native::radixSortKVInPlace<(int)2, (...
void at::native::radixSortKVInPlace<(...
run_baseline [36.723 ms]
iter [3.656 ms]
Gini_Selection [376.512 μs]

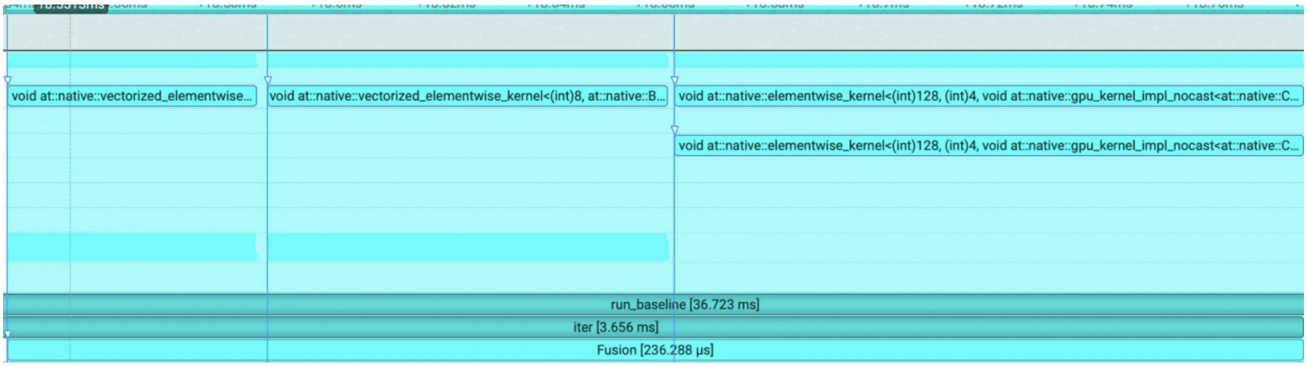
map to index:



fine stage:



fusion:

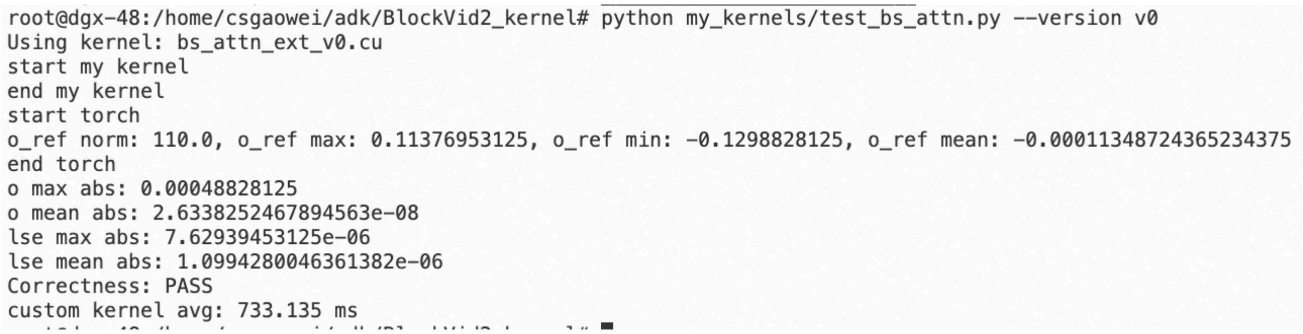

void at::native::vectorized_elementwise...
void at::native::vectorized_elementwise_kernel<(int)8, at::native::B...
void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_kernel_impl_nocast<at::native::C...
void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_kernel_impl_nocast<at::native::C...
run_baseline [36.723 ms]
iter [3.656 ms]
Fusion [236.288 μs]

Triton进一步优化

写的代码差不多，还没调出优化

my_kernel

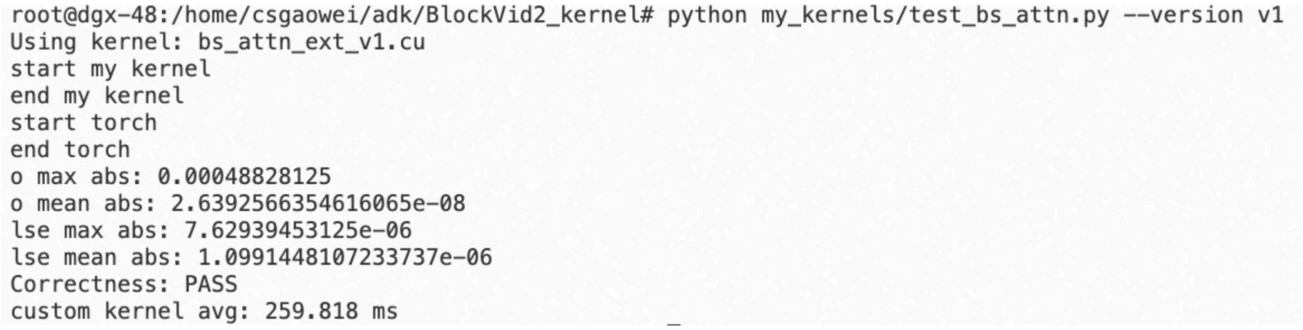
v0



```
root@dgx-48:/home/csgaowei/adk/BlockVid2_kernel# python my_kernels/test_bs_attn.py --version v0
Using kernel: bs_attn_ext_v0.cu
start my kernel
end my kernel
start torch
o_ref norm: 110.0, o_ref max: 0.11376953125, o_ref min: -0.1298828125, o_ref mean: -0.00011348724365234375
end torch
o max abs: 0.00048828125
o mean abs: 2.6338252467894563e-08
lse max abs: 7.62939453125e-06
lse mean abs: 1.0994280046361382e-06
Correctness: PASS
custom kernel avg: 733.135 ms
```

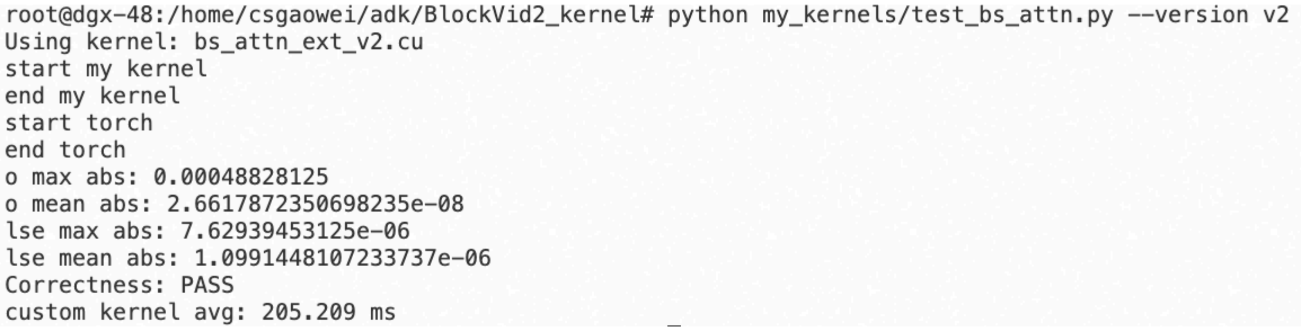
v1

改成 warp-per-query（32 threads）+ 向量化加载



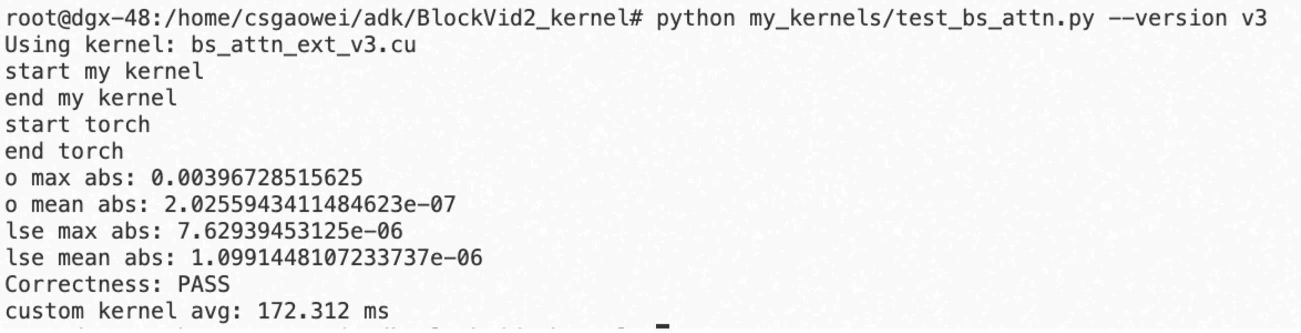
```
root@dgx-48:/home/csgaowei/adk/BlockVid2_kernel# python my_kernels/test_bs_attn.py --version v1
Using kernel: bs_attn_ext_v1.cu
start my kernel
end my kernel
start torch
end torch
o max abs: 0.00048828125
o mean abs: 2.6392566354616065e-08
lse max abs: 7.62939453125e-06
lse mean abs: 1.0991448107233737e-06
Correctness: PASS
custom kernel avg: 259.818 ms
```

v2



```
root@dgx-48:/home/csgaowei/adk/BlockVid2_kernel# python my_kernels/test_bs_attn.py --version v2
Using kernel: bs_attn_ext_v2.cu
start my kernel
end my kernel
start torch
end torch
o max abs: 0.00048828125
o mean abs: 2.6617872350698235e-08
lse max abs: 7.62939453125e-06
lse mean abs: 1.0991448107233737e-06
Correctness: PASS
custom kernel avg: 205.209 ms
```

v3

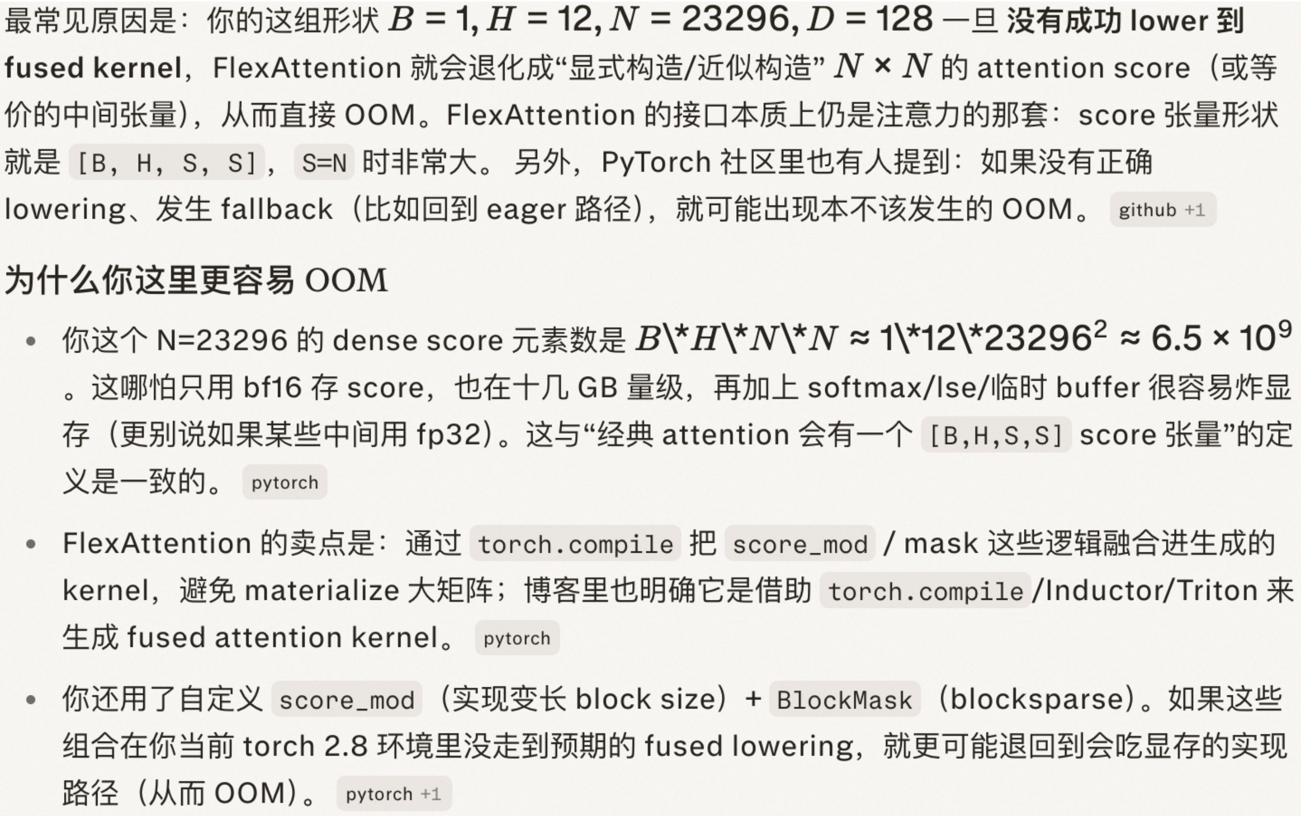


```
root@dgx-48:/home/csgaowei/adk/BlockVid2_kernel# python my_kernels/test_bs_attn.py --version v3
Using kernel: bs_attn_ext_v3.cu
start my kernel
end my kernel
start torch
end torch
o max abs: 0.00396728515625
o mean abs: 2.0255943411484623e-07
lse max abs: 7.62939453125e-06
lse mean abs: 1.0991448107233737e-06
Correctness: PASS
custom kernel avg: 172.312 ms
```

不太熟悉tma的写法，优化不动。。。

flexattention

OOM



最常见原因是：你的这组形状 $B = 1, H = 12, N = 23296, D = 128$ 一旦 **没有成功 lower 到 fused kernel**，FlexAttention 就会退化成“显式构造/近似构造” $N \times N$ 的 attention score（或等价的中间张量），从而直接 OOM。FlexAttention 的接口本质上仍是注意力的那套：score 张量形状就是 `[B, H, S, S]`，`S=N` 时非常大。另外，PyTorch 社区里也有人提到：如果没有正确 lowering、发生 fallback（比如回到 eager 路径），就可能出现本不该发生的 OOM。 github +1

为什么你这里更容易 OOM

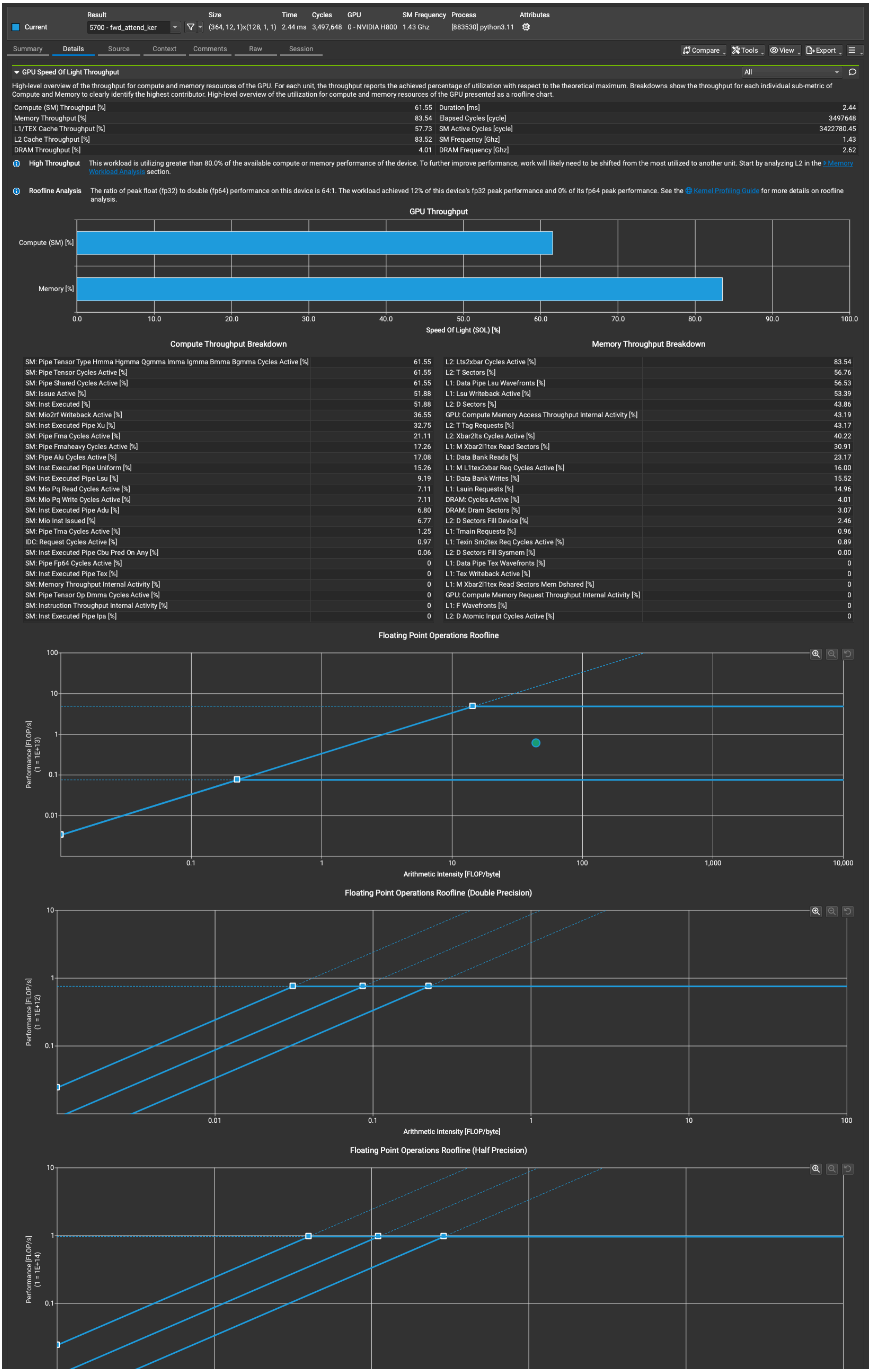
- 你这个 N=23296 的 dense score 元素数是 B*H*N*N ≈ 1*12*23296^2 ≈ 6.5×10^9。这哪怕只用 bf16 存 score，也在十几 GB 量级，再加上 softmax/lse/临时 buffer 很容易炸显存（更别说如果某些中间用 fp32）。这与“经典 attention 会有一个 `[B,H,S,S]` score 张量”的定义是一致的。 pytorch
- FlexAttention 的卖点是：通过 `torch.compile` 把 `score_mod` / mask 这些逻辑融合进生成的 kernel，避免 materialize 大矩阵；博客里也明确它是借助 `torch.compile`/Inductor/Triton 来生成 fused attention kernel。 pytorch
- 你还用了自定义 `score_mod`（实现变长 block size）+ `BlockMask`（blocksparse）。如果这些组合在你当前 torch 2.8 环境里没走到预期的 fused lowering，就更可能退回到会吃显存的实现路径（从而 OOM）。 pytorch +1

Block-Sparse-Attention(Hansong)

不支持每个block内部的variable_block_sizes，且blocksize只支持128。

继续优化tk

tk profile



	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	5700 - fwd_attend_ker	(364, 12, 1)x(128, 1, 1)	2.44 ms	3,497,648	0 - NVIDIA H800	1.43 Ghz	[883530] python3.11	

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	61.55	Duration [ms]	2.44
Memory Throughput [%]	83.54	Elapsed Cycles [cycle]	3497648
L1/TEX Cache Throughput [%]	57.73	SM Active Cycles [cycle]	3422780.45
L2 Cache Throughput [%]	83.52	SM Frequency [Ghz]	1.43
DRAM Throughput [%]	4.01	DRAM Frequency [Ghz]	2.62

High Throughput This workload is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing L2 in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The workload achieved 12% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

GPU Throughput

Compute Throughput Breakdown

SM: Pipe Tensor Type Hmma Hgmma Qgmma Imma Igmma Bmma Bgmma Cycles Active [%]	61.55
SM: Pipe Tensor Cycles Active [%]	61.55
SM: Pipe Shared Cycles Active [%]	61.55
SM: Issue Active [%]	51.88
SM: Inst Executed [%]	51.88
SM: Mio2rf Writeback Active [%]	36.55
SM: Inst Executed Pipe Xu [%]	32.75
SM: Pipe Fma Cycles Active [%]	21.11
SM: Pipe Fmaheavy Cycles Active [%]	17.26
SM: Pipe Alu Cycles Active [%]	17.08
SM: Inst Executed Pipe Uniform [%]	15.26
SM: Inst Executed Pipe Lsu [%]	9.19
SM: Mio Pq Read Cycles Active [%]	7.11
SM: Mio Pq Write Cycles Active [%]	7.11
SM: Inst Executed Pipe Adu [%]	6.80
SM: Mio Inst Issued [%]	6.77
SM: Pipe Tma Cycles Active [%]	1.25
IDC: Request Cycles Active [%]	0.97
SM: Inst Executed Pipe Cbu Pred On Any [%]	0.06
SM: Pipe Fp64 Cycles Active [%]	0
SM: Inst Executed Pipe Tex [%]	0
SM: Memory Throughput Internal Activity [%]	0
SM: Pipe Tensor Op Dmma Cycles Active [%]	0
SM: Instruction Throughput Internal Activity [%]	0
SM: Inst Executed Pipe Ipa [%]	0

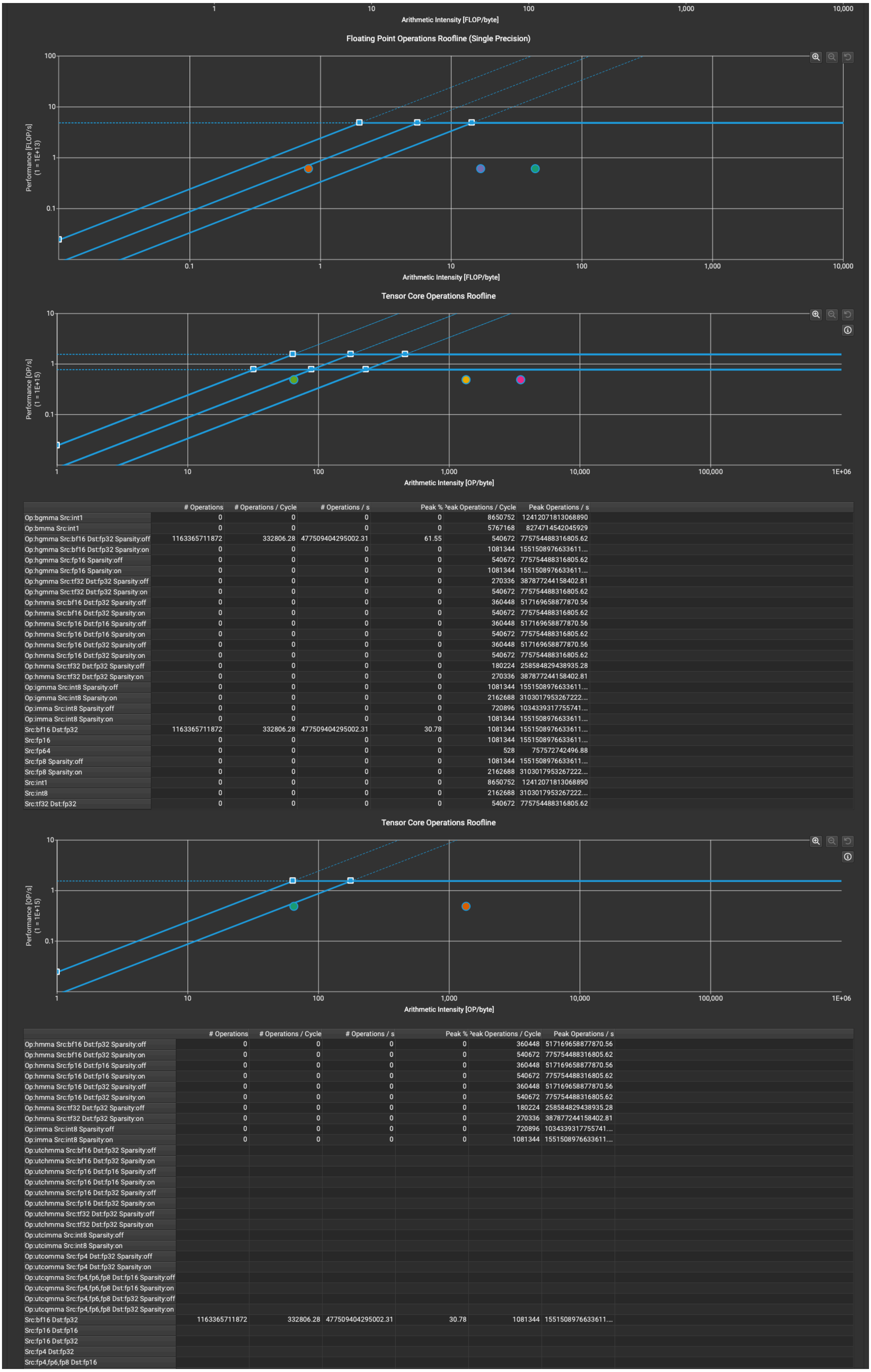
Memory Throughput Breakdown

L2: Lts2xbar Cycles Active [%]	83.54
L2: T Sectors [%]	56.76
L1: Data Pipe Lsu Wavefronts [%]	56.53
L1: Lsu Writeback Active [%]	53.39
L2: D Sectors [%]	43.86
GPU: Compute Memory Access Throughput Internal Activity [%]	43.19
L2: T Tag Requests [%]	43.17
L2: Xbar2lts Cycles Active [%]	40.22
L1: M Xbar2l1tex Read Sectors [%]	30.91
L1: Data Bank Reads [%]	23.17
L1: M L1tex2xbar Req Cycles Active [%]	16.00
L1: Data Bank Writes [%]	15.52
L1: Lsuin Requests [%]	14.96
DRAM: Cycles Active [%]	4.01
DRAM: Dram Sectors [%]	3.07
L2: D Sectors Fill Device [%]	2.46
L1: Tmain Requests [%]	0.96
L1: Texin Sm2tex Req Cycles Active [%]	0.89
L2: D Sectors Fill Sysmem [%]	0.00
L1: Data Pipe Tex Wavefronts [%]	0
L1: Tex Writeback Active [%]	0
L1: M Xbar2l1tex Read Sectors Mem Dshared [%]	0
GPU: Compute Memory Request Throughput Internal Activity [%]	0
L1: F Wavefronts [%]	0
L2: D Atomic Input Cycles Active [%]	0

Floating Point Operations Roofline

Floating Point Operations Roofline (Double Precision)

Floating Point Operations Roofline (Half Precision)



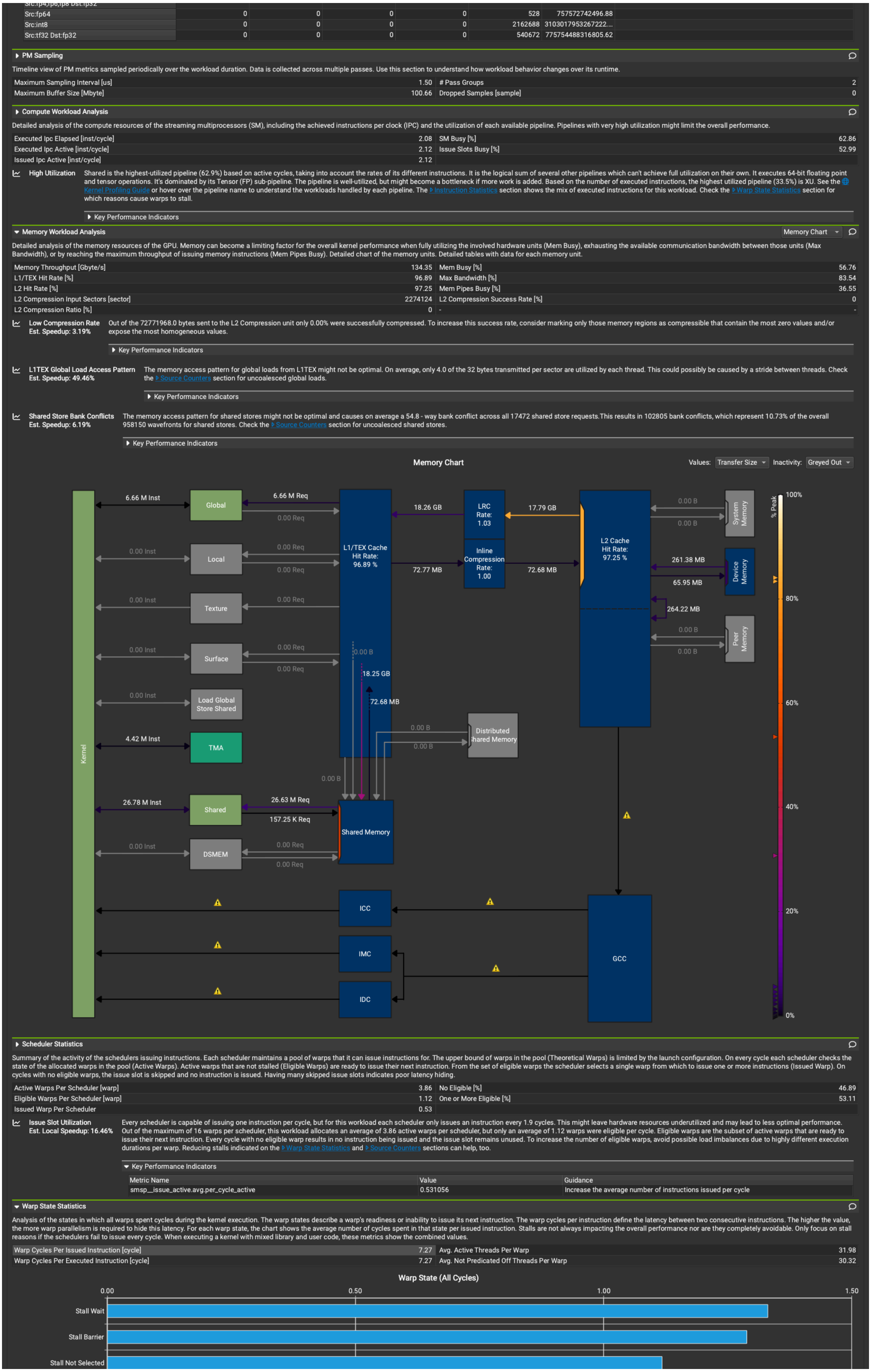
Floating Point Operations Roofline (Single Precision)

Tensor Core Operations Roofline

	# Operations	# Operations / Cycle	# Operations / s	Peak %	ʼeak Operations / Cycle	Peak Operations / s
Op:bgmma Src:int1	0	0	0	0	8650752	12412071813068890
Op:bmma Src:int1	0	0	0	0	5767168	8274714542045929
Op:hgmma Src:bf16 Dst:fp32 Sparsity:off	1163365711872	332806.28	477509404295002.31	61.55	540672	775754488316805.62
Op:hgmma Src:bf16 Dst:fp32 Sparsity:on	0	0	0	0	1081344	1551508976633611....
Op:hgmma Src:fp16 Sparsity:off	0	0	0	0	540672	775754488316805.62
Op:hgmma Src:fp16 Sparsity:on	0	0	0	0	1081344	1551508976633611....
Op:hgmma Src:tf32 Dst:fp32 Sparsity:off	0	0	0	0	270336	387877244158402.81
Op:hgmma Src:tf32 Dst:fp32 Sparsity:on	0	0	0	0	540672	775754488316805.62
Op:hmma Src:bf16 Dst:fp32 Sparsity:off	0	0	0	0	360448	517169658877870.56
Op:hmma Src:bf16 Dst:fp32 Sparsity:on	0	0	0	0	540672	775754488316805.62
Op:hmma Src:fp16 Dst:fp16 Sparsity:off	0	0	0	0	360448	517169658877870.56
Op:hmma Src:fp16 Dst:fp16 Sparsity:on	0	0	0	0	540672	775754488316805.62
Op:hmma Src:fp16 Dst:fp32 Sparsity:off	0	0	0	0	360448	517169658877870.56
Op:hmma Src:fp16 Dst:fp32 Sparsity:on	0	0	0	0	540672	775754488316805.62
Op:hmma Src:tf32 Dst:fp32 Sparsity:off	0	0	0	0	180224	258584829438935.28
Op:hmma Src:tf32 Dst:fp32 Sparsity:on	0	0	0	0	270336	387877244158402.81
Op:igmma Src:int8 Sparsity:off	0	0	0	0	1081344	1551508976633611....
Op:igmma Src:int8 Sparsity:on	0	0	0	0	2162688	3103017953267222....
Op:imma Src:int8 Sparsity:off	0	0	0	0	720896	1034339317755741....
Op:imma Src:int8 Sparsity:on	0	0	0	0	1081344	1551508976633611....
Src:bf16 Dst:fp32	1163365711872	332806.28	477509404295002.31	30.78	1081344	1551508976633611....
Src:fp16	0	0	0	0	1081344	1551508976633611....
Src:fp64	0	0	0	0	528	757572742496.88
Src:fp8 Sparsity:off	0	0	0	0	1081344	1551508976633611....
Src:fp8 Sparsity:on	0	0	0	0	2162688	3103017953267222....
Src:int1	0	0	0	0	8650752	12412071813068890
Src:int8	0	0	0	0	2162688	3103017953267222....
Src:tf32 Dst:fp32	0	0	0	0	540672	775754488316805.62

Tensor Core Operations Roofline

	# Operations	# Operations / Cycle	# Operations / s	Peak %	ʼeak Operations / Cycle	Peak Operations / s
Op:hmma Src:bf16 Dst:fp32 Sparsity:off	0	0	0	0	360448	517169658877870.56
Op:hmma Src:bf16 Dst:fp32 Sparsity:on	0	0	0	0	540672	775754488316805.62
Op:hmma Src:fp16 Dst:fp16 Sparsity:off	0	0	0	0	360448	517169658877870.56
Op:hmma Src:fp16 Dst:fp16 Sparsity:on	0	0	0	0	540672	775754488316805.62
Op:hmma Src:fp16 Dst:fp32 Sparsity:off	0	0	0	0	360448	517169658877870.56
Op:hmma Src:fp16 Dst:fp32 Sparsity:on	0	0	0	0	540672	775754488316805.62
Op:hmma Src:tf32 Dst:fp32 Sparsity:off	0	0	0	0	180224	258584829438935.28
Op:hmma Src:tf32 Dst:fp32 Sparsity:on	0	0	0	0	270336	387877244158402.81
Op:imma Src:int8 Sparsity:off	0	0	0	0	720896	1034339317755741....
Op:imma Src:int8 Sparsity:on	0	0	0	0	1081344	1551508976633611....
Op:utchmma Src:bf16 Dst:fp32 Sparsity:off						
Op:utchmma Src:bf16 Dst:fp32 Sparsity:on						
Op:utchmma Src:fp16 Dst:fp16 Sparsity:off						
Op:utchmma Src:fp16 Dst:fp16 Sparsity:on						
Op:utchmma Src:fp16 Dst:fp32 Sparsity:off						
Op:utchmma Src:fp16 Dst:fp32 Sparsity:on						
Op:utchmma Src:tf32 Dst:fp32 Sparsity:off						
Op:utchmma Src:tf32 Dst:fp32 Sparsity:on						
Op:utcimma Src:int8 Sparsity:off						
Op:utcimma Src:int8 Sparsity:on						
Op:utcomma Src:fp4 Dst:fp32 Sparsity:off						
Op:utcomma Src:fp4 Dst:fp32 Sparsity:on						
Op:utcqmma Src:fp4,fp6,fp8 Dst:fp16 Sparsity:off						
Op:utcqmma Src:fp4,fp6,fp8 Dst:fp16 Sparsity:on						
Op:utcqmma Src:fp4,fp6,fp8 Dst:fp32 Sparsity:off						
Op:utcqmma Src:fp4,fp6,fp8 Dst:fp32 Sparsity:on						
Src:bf16 Dst:fp32	1163365711872	332806.28	477509404295002.31	30.78	1081344	1551508976633611....
Src:fp16 Dst:fp16						
Src:fp16 Dst:fp32						
Src:fp4 Dst:fp32						
Src:fp4,fp6,fp8 Dst:fp16						



Src:fp4,fp6,fp8 Dst:fp32						
Src:fp64	0	0	0	0	528	757572742496.88
Src:int8	0	0	0	0	2162688	3103017953267222....
Src:tf32 Dst:fp32	0	0	0	0	540672	775754488316805.62

PM Sampling

Timeline view of PM metrics sampled periodically over the workload duration. Data is collected across multiple passes. Use this section to understand how workload behavior changes over its runtime.

Maximum Sampling Interval [us]	1.50	# Pass Groups	2
Maximum Buffer Size [Mbyte]	100.66	Dropped Samples [sample]	0

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	2.08	SM Busy [%]	62.86
Executed Ipc Active [inst/cycle]	2.12	Issue Slots Busy [%]	52.99
Issued Ipc Active [inst/cycle]	2.12		

High Utilization Shared is the highest-utilized pipeline (62.9%) based on active cycles, taking into account the rates of its different instructions. It is the logical sum of several other pipelines which can't achieve full utilization on their own. It executes 64-bit floating point and tensor operations. It's dominated by its Tensor (FP) sub-pipeline. The pipeline is well-utilized, but might become a bottleneck if more work is added. Based on the number of executed instructions, the highest utilized pipeline (33.5%) is XU. See the Kernel Profiling Guide or hover over the pipeline name to understand the workloads handled by each pipeline. The Instruction Statistics section shows the mix of executed instructions for this workload. Check the Warp State Statistics section for which reasons cause warps to stall.

Key Performance Indicators

Memory Workload Analysis

Memory Chart

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/s]	134.35	Mem Busy [%]	56.76
L1/TEX Hit Rate [%]	96.89	Max Bandwidth [%]	83.54
L2 Hit Rate [%]	97.25	Mem Pipes Busy [%]	36.55
L2 Compression Input Sectors [sector]	2274124	L2 Compression Success Rate [%]	0
L2 Compression Ratio [%]	0	-	-

Low Compression Rate
Est. Speedup: 3.19%
Out of the 72771968.0 bytes sent to the L2 Compression unit only 0.00% were successfully compressed. To increase this success rate, consider marking only those memory regions as compressible that contain the most zero values and/or expose the most homogeneous values.

Key Performance Indicators

L1TEX Global Load Access Pattern
Est. Speedup: 49.46%
The memory access pattern for global loads from L1TEX might not be optimal. On average, only 4.0 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the Source Counters section for uncoalesced global loads.

Key Performance Indicators

Shared Store Bank Conflicts
Est. Speedup: 6.19%
The memory access pattern for shared stores might not be optimal and causes on average a 54.8 - way bank conflict across all 17472 shared store requests.This results in 102805 bank conflicts, which represent 10.73% of the overall 958150 wavefronts for shared stores. Check the Source Counters section for uncoalesced shared stores.

Key Performance Indicators

Memory Chart

Values: Transfer Size Inactivity: Greyed Out

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	3.86	No Eligible [%]	46.89
Eligible Warps Per Scheduler [warp]	1.12	One or More Eligible [%]	53.11
Issued Warp Per Scheduler	0.53		

Issue Slot Utilization
Est. Local Speedup: 16.46%
Every scheduler is capable of issuing one instruction per cycle, but for this workload each scheduler only issues an instruction every 1.9 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this workload allocates an average of 3.86 active warps per scheduler, but only an average of 1.12 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the Warp State Statistics and Source Counters sections can help, too.

Key Performance Indicators

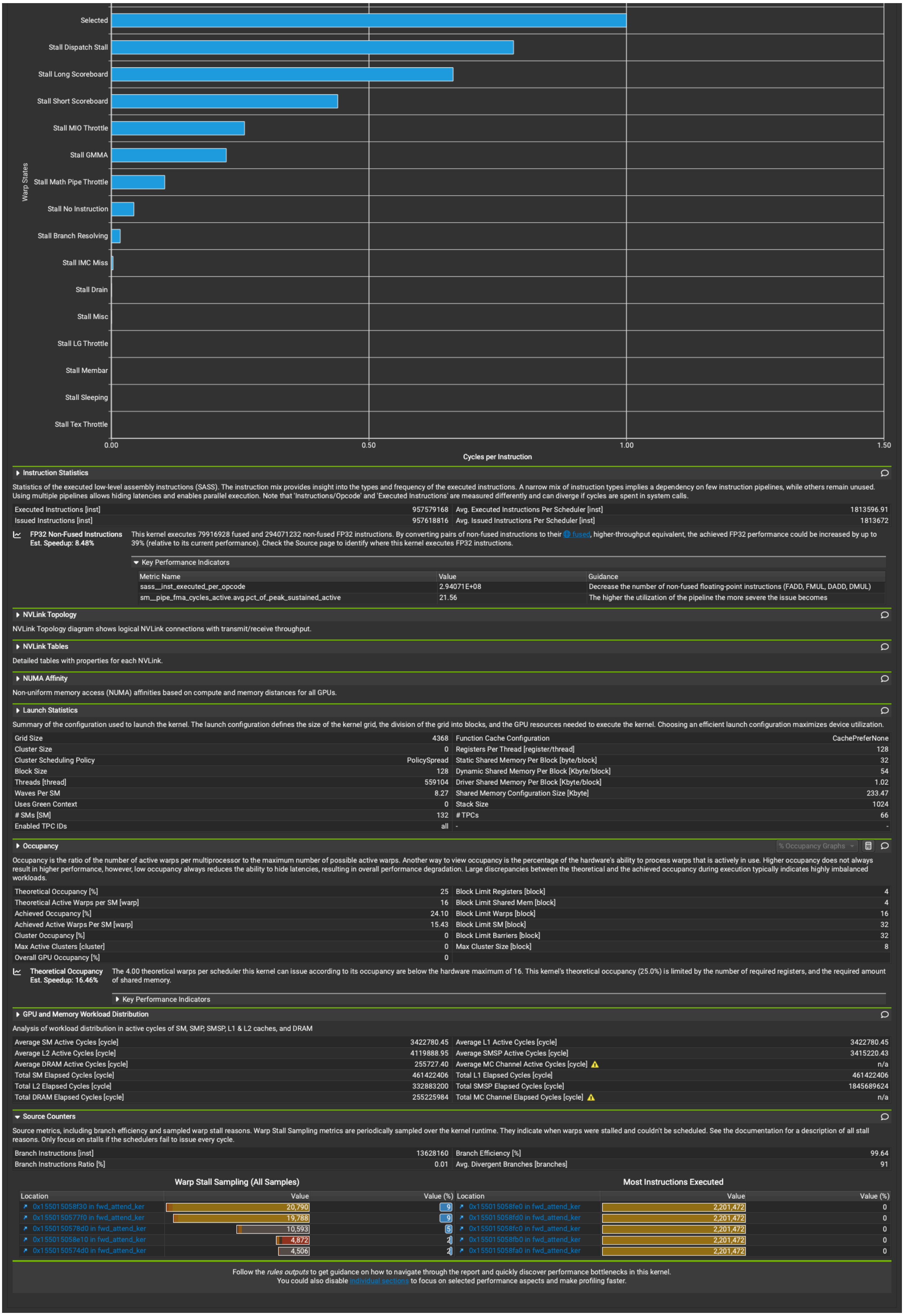
Metric Name	Value	Guidance
smsp__issue_active.avg.per_cycle_active	0.531056	Increase the average number of instructions issued per cycle

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	7.27	Avg. Active Threads Per Warp	31.98
Warp Cycles Per Executed Instruction [cycle]	7.27	Avg. Not Predicated Off Threads Per Warp	30.32

Warp State (All Cycles)



Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	957579168	Avg. Executed Instructions Per Scheduler [inst]	1813596.91
Issued Instructions [inst]	957618816	Avg. Issued Instructions Per Scheduler [inst]	1813672

FP32 Non-Fused Instructions
Est. Speedup: 8.48%

This kernel executes 79916928 fused and 294071232 non-fused FP32 instructions. By converting pairs of non-fused instructions to their fused, higher-throughput equivalent, the achieved FP32 performance could be increased by up to 39% (relative to its current performance). Check the Source page to identify where this kernel executes FP32 instructions.

Key Performance Indicators

Metric Name	Value	Guidance
sass__inst_executed_per_opcode	2.94071E+08	Decrease the number of non-fused floating-point instructions (FADD, FMUL, DADD, DMUL)
sm__pipe_fma_cycles_active.avg.pct_of_peak_sustained_active	21.56	The higher the utilization of the pipeline the more severe the issue becomes

NVLink Topology

NVLink Topology diagram shows logical NVLink connections with transmit/receive throughput.

NVLink Tables

Detailed tables with properties for each NVLink.

NUMA Affinity

Non-uniform memory access (NUMA) affinities based on compute and memory distances for all GPUs.

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	4368	Function Cache Configuration	CachePreferNone
Cluster Size	0	Registers Per Thread [register/thread]	128
Cluster Scheduling Policy	PolicySpread	Static Shared Memory Per Block [byte/block]	32
Block Size	128	Dynamic Shared Memory Per Block [Kbyte/block]	54
Threads [thread]	559104	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	8.27	Shared Memory Configuration Size [Kbyte]	233.47
Uses Green Context	0	Stack Size	1024
# SMs [SM]	132	# TPCs	66
Enabled TPC IDs	all	-	-

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	25	Block Limit Registers [block]	4
Theoretical Active Warps per SM [warp]	16	Block Limit Shared Mem [block]	4
Achieved Occupancy [%]	24.10	Block Limit Warps [block]	16
Achieved Active Warps Per SM [warp]	15.43	Block Limit SM [block]	32
Cluster Occupancy [%]	0	Block Limit Barriers [block]	32
Max Active Clusters [cluster]	0	Max Cluster Size [block]	8
Overall GPU Occupancy [%]	0		

Theoretical Occupancy
Est. Speedup: 16.46%

The 4.00 theoretical warps per scheduler this kernel can issue according to its occupancy are below the hardware maximum of 16. This kernel's theoretical occupancy (25.0%) is limited by the number of required registers, and the required amount of shared memory.

Key Performance Indicators

GPU and Memory Workload Distribution

Analysis of workload distribution in active cycles of SM, SMP, SMSP, L1 & L2 caches, and DRAM

Average SM Active Cycles [cycle]	3422780.45	Average L1 Active Cycles [cycle]	3422780.45
Average L2 Active Cycles [cycle]	4119888.95	Average SMSP Active Cycles [cycle]	3415220.43
Average DRAM Active Cycles [cycle]	255727.40	Average MC Channel Active Cycles [cycle] ⚠	n/a
Total SM Elapsed Cycles [cycle]	461422406	Total L1 Elapsed Cycles [cycle]	461422406
Total L2 Elapsed Cycles [cycle]	332883200	Total SMSP Elapsed Cycles [cycle]	1845689624
Total DRAM Elapsed Cycles [cycle]	255225984	Total MC Channel Elapsed Cycles [cycle] ⚠	n/a

Source Counters

Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

Branch Instructions [inst]	13628160	Branch Efficiency [%]	99.64
Branch Instructions Ratio [%]	0.01	Avg. Divergent Branches [branches]	91

Warp Stall Sampling (All Samples)

Location	Value	Value (%)
0x155015058f30 in fwd_attend_ker	20,790	9
0x1550150577f0 in fwd_attend_ker	19,788	9
0x1550150578d0 in fwd_attend_ker	10,593	5
0x155015058e10 in fwd_attend_ker	4,872	2
0x1550150574d0 in fwd_attend_ker	4,506	2

Most Instructions Executed

Location	Value	Value (%)
0x155015058fe0 in fwd_attend_ker	2,201,472	0
0x155015058fd0 in fwd_attend_ker	2,201,472	0
0x155015058fc0 in fwd_attend_ker	2,201,472	0
0x155015058fb0 in fwd_attend_ker	2,201,472	0
0x155015058fa0 in fwd_attend_ker	2,201,472	0

Follow the *rules outputs* to get guidance on how to navigate through the report and quickly discover performance bottlenecks in this kernel.
You could also disable individual sections to focus on selected performance aspects and make profiling faster.

后续还有可能的优化:

- prefetch进一步优化?
- fusion

20260225

- profile 更长的seq_len下的表现，不同稀疏度?
- 加上kvcache的decode

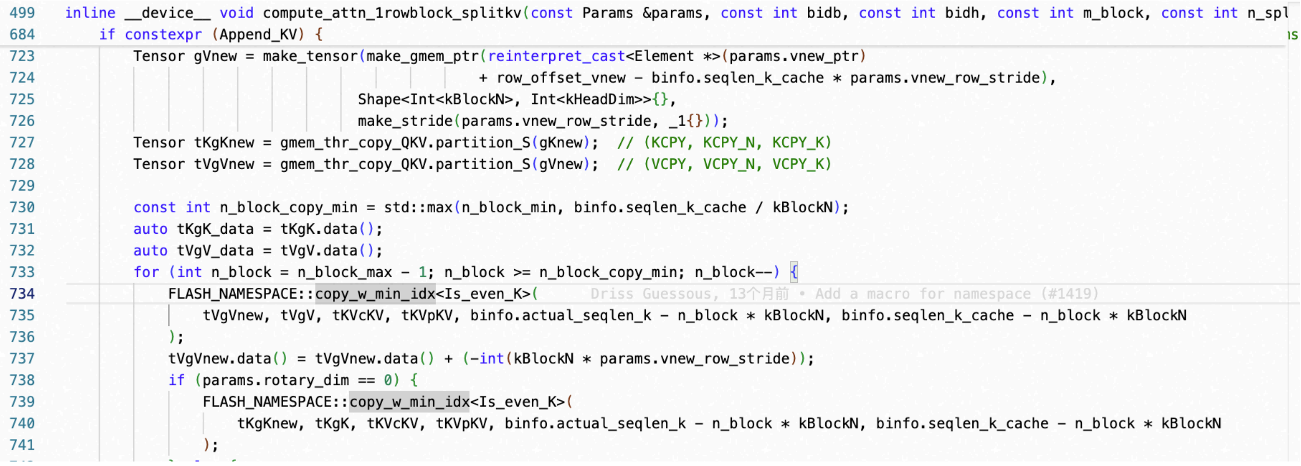
分析FA with KV

fa with kvcache的调用:

```
flash_attn.modules.mha 里调用 flash_attn_with_kvcache(...)
→ flash_attn/flash_attn_interface.py: flash_attn_with_kvcache
→ C++ 扩展 flash_attn_gpu.fwd_kvcache
→ csrc/flash_attn/flash_api.cpp: mha_fwd_kvcache
→ run_mha_fwd(params, stream, /*force_split_kernel=*/...)
→ run_mha_fwd_splitkv_dispatch<...>(...)
→ run_flash_splitkv_fwd<...>(...)
→ flash_fwd_splitkv_kernel<...><<<...>>>(params)
→ compute_attn_splitkv<...>(params)
→ compute_attn_1rowblock_splitkv<...>(...) (
```

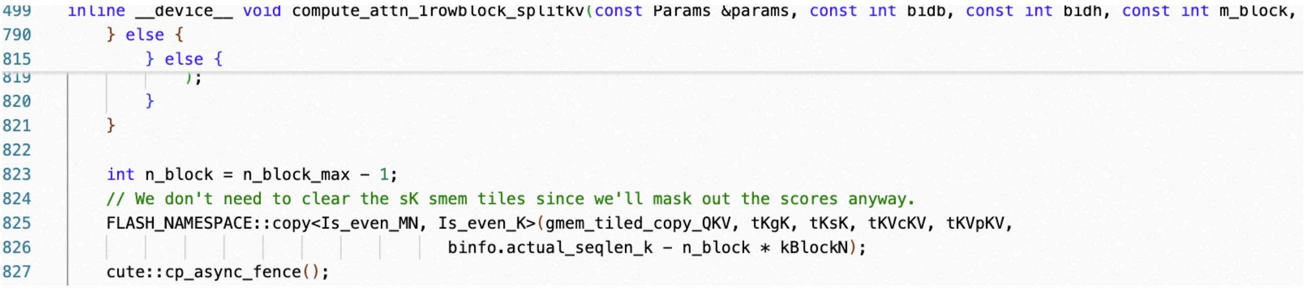
compute_attn_1rowblock_splitkv中：

- **写cache**: 拼接当前step的新的kv和旧的kv，传入hbm中历史kvcache的物理指针地址，与当前step计算的kv拼接，从而更新kvcache



```
inline __device__ void compute_attn_1rowblock_splitkv(const Params &params, const int bidb, const int bidh, const int m_block, const int n_spl
    if constexpr (Append_KV) {
        Tensor gVnew = make_tensor(make_gmem_ptr(reinterpret_cast<Element *>(params.vnew_ptr)
                                                + row_offset_vnew - binfo.seqlen_k_cache * params.vnew_row_stride),
                                  Shape<Int<kBlockN>, Int<kHeadDim>>{},
                                  make_stride(params.vnew_row_stride, _1{}));
        Tensor tKgKnew = gmem_thr_copy_QKV.partition_S(gKnew);  // (KCPY, KCPY_N, KCPY_K)
        Tensor tVgVnew = gmem_thr_copy_QKV.partition_S(gVnew);  // (VCPY, VCPY_N, VCPY_K)

        const int n_block_copy_min = std::max(n_block_min, binfo.seqlen_k_cache / kBlockN);
        auto tKgK_data = tKgK.data();
        auto tVgV_data = tVgV.data();
        for (int n_block = n_block_max - 1; n_block >= n_block_copy_min; n_block--) {
            FLASH_NAMESPACE::copy_w_min_idx<Is_even_K>(       Driss Guessous, 13个月前 • Add a macro for namespace (#1419)
                tVgVnew, tVgV, tKVcKV, tKVpKV, binfo.actual_seqlen_k - n_block * kBlockN, binfo.seqlen_k_cache - n_block * kBlockN
            );
            tVgVnew.data() = tVgVnew.data() + (-int(kBlockN * params.vnew_row_stride));
            if (params.rotary_dim == 0) {
                FLASH_NAMESPACE::copy_w_min_idx<Is_even_K>(
                    tKgKnew, tKgK, tKVcKV, tKVpKV, binfo.actual_seqlen_k - n_block * kBlockN, binfo.seqlen_k_cache - n_block * kBlockN
                );
```

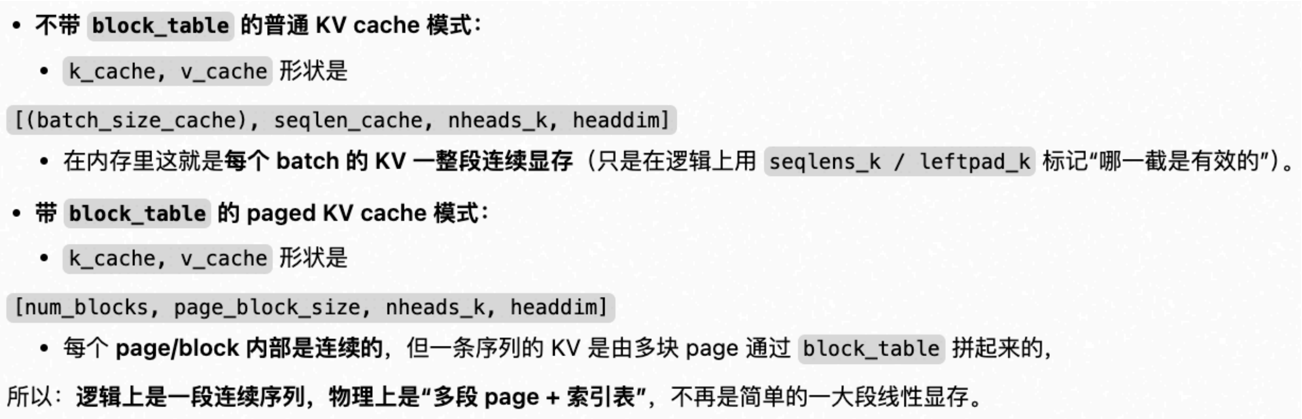
- **读cache**: 把kvcache load到shared memory



```
inline __device__ void compute_attn_1rowblock_splitkv(const Params &params, const int bidb, const int bidh, const int m_block,
    } else {
        } else {
            );
        }
    }

    int n_block = n_block_max - 1;
    // We don't need to clear the sK smem tiles since we'll mask out the scores anyway.
    FLASH_NAMESPACE::copy<Is_even_MN, Is_even_K>(gmem_tiled_copy_QKV, tKgK, tKsK, tKVcKV, tKVpKV,
                                       binfo.actual_seqlen_k - n_block * kBlockN);
    cute::cp_async_fence();
```

- 之后就进入flash attention的计算，分块加载kvcache并计算
  - Q：每个 CTA 负责一块，只load 一次，整个 K 序列复用
  - 当前 iteration 在用 “上一轮已经拷到 shared 的 K/V” 做 GEMM
  - 同时预取下一块 K/V 到 shared

其中的kvcache的显存：



- **不带 `block_table` 的普通 KV cache 模式：**
  - `k_cache, v_cache` 形状是

`[(batch_size_cache), seqlen_cache, nheads_k, headdim]`

  - 在内存里这就是**每个 batch 的 KV 一整段连续显存**（只是在逻辑上用 `seqlens_k / leftpad_k` 标记“哪一截是有效的”）。
- **带 `block_table` 的 paged KV cache 模式：**
  - `k_cache, v_cache` 形状是

`[num_blocks, page_block_size, nheads_k, headdim]`

  - 每个 **page/block 内部是连续的**，但一条序列的 KV 是由多块 page 通过 `block_table` 拼起来的，

所以：**逻辑上是一段连续序列，物理上是“多段 page + 索引表”**，不再是简单的一大段线性显存。

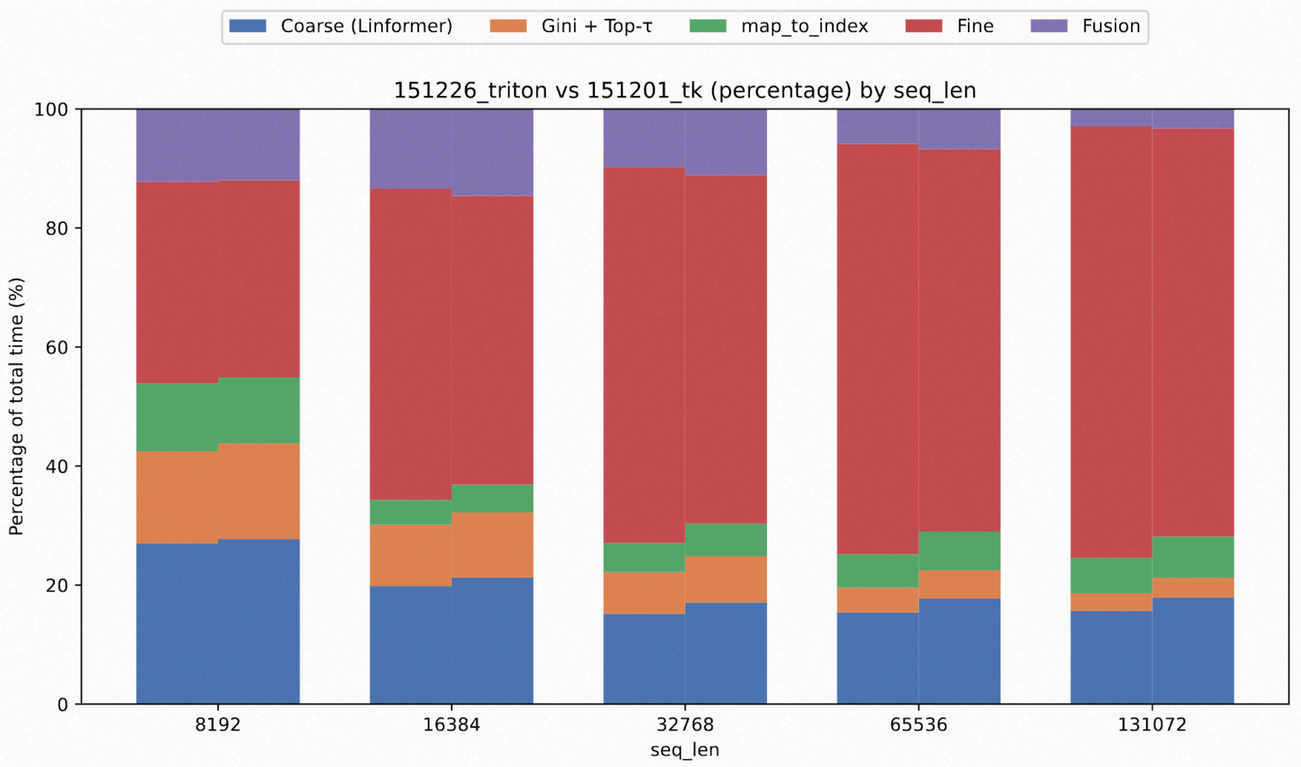
fa是直接计算出q, k, v，fa with kvcache相比fa会多一个过程：传入历史kvcache在gpu hbm中的物理指针地址，与当前step计算的kv拼接，从而更新kvcache。之后加载kvcache到gpu的shared memory的过程并分块计算和普通的fa是一样的。

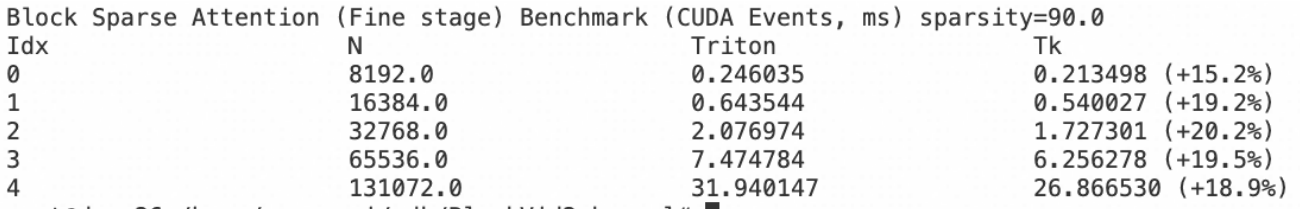
如果我们这个场景下考虑kvcache的话，当前block计算时需要读取之前所有block的kvcache，并把当前block的kv的值和历史kvcache拼接。因为fa中kvcache应该是按照连续或者block_table进行管理的，所以读取的时候是连续的一段。一个潜在的问题就是拼接的时候，当前block的kv值可能会有很多并且内存不连续token，拼接的时候或许会有访存的问题。不过拼接完之后的flash attn计算就是相同的方式了。

不同seq_len下的性能

先不考虑dyanmic block的选择，直接指定sparsity，定为90%

参考vsa中使用的seq len，这里用8192 16384 32768 65536 131072





```
Block Sparse Attention (Fine stage) Benchmark (CUDA Events, ms) sparsity=90.0
Idx                     N                       Triton                  Tk
0                       8192.0                  0.246035                0.213498 (+15.2%)
1                       16384.0                 0.643544                0.540027 (+19.2%)
2                       32768.0                 2.076974                1.727301 (+20.2%)
3                       65536.0                 7.474784                6.256278 (+19.5%)
4                       131072.0                31.940147               26.866530 (+18.9%)
```

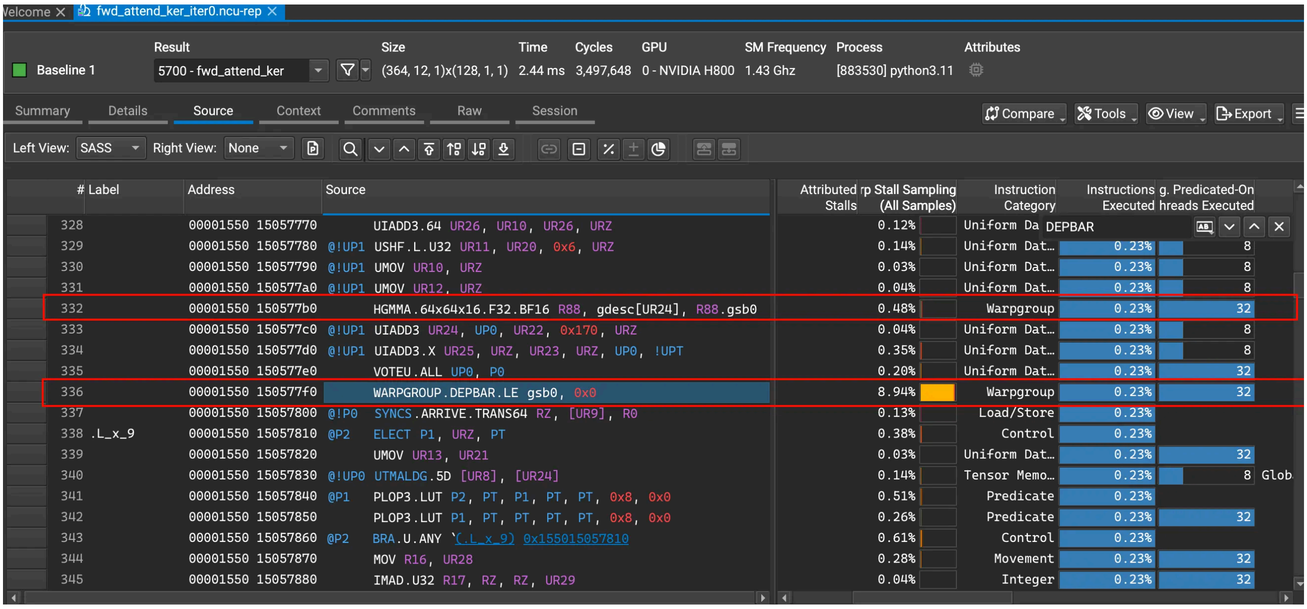
seq_len 越大，fine阶段的时间占比越大

~~with kvcache~~

~~想要确认一下在 attention 计算时，预取 kv block 的时间和 q 计算的具体的时间。~~

~~由于 flashattention 的原理，具体执行时 qkv 是分块做的预取和计算。原tk的代码利用了 hopper 架构的 `tma` 指令可以实现异步加载，所以可以 overlap **preload k/v block 和 compute q&k/v**的执行。~~

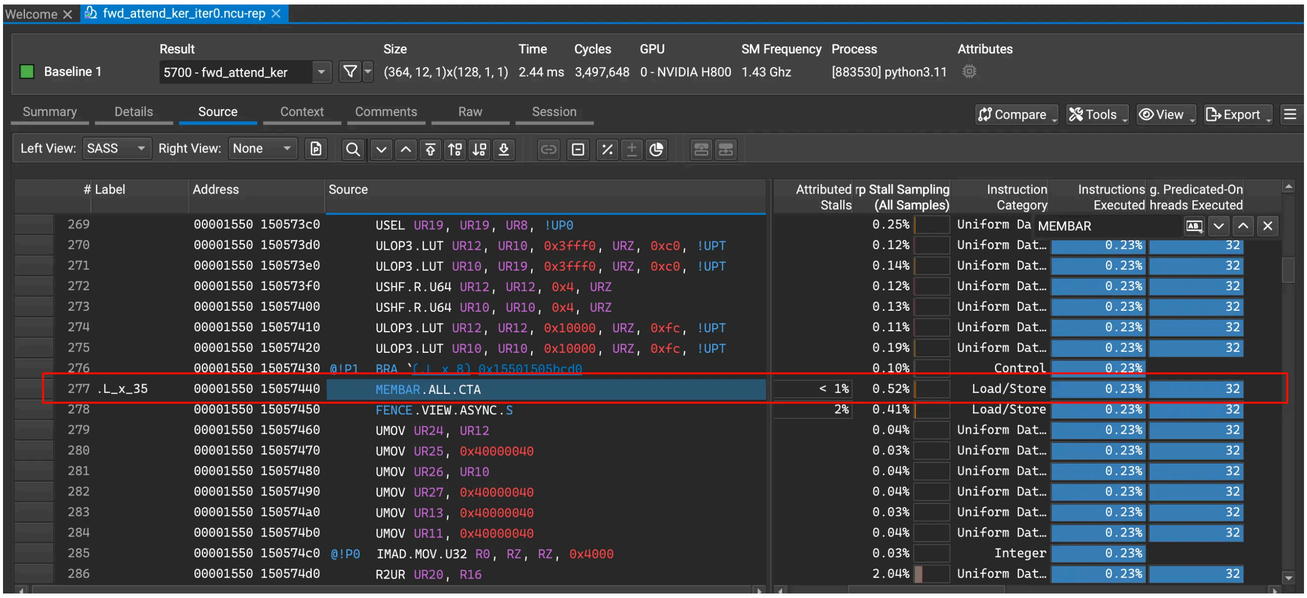
~~代码中具体是 `wait(k_smem_arrived, kv_idx % 2);` 的指令来等待 k block 完全加载完成，从而实现同步，**如果能知道这个 `wait` 的执行时间，就能知道 attention 中 k block 等待 load 时间**，判断和 compute 时间哪个更长。经了解可以通过SASS来看。~~



Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Baseline 1 — 5700 - fwd_attend_ker	(364, 12, 1)x(128, 1, 1)	2.44 ms	3,497,648	0 - NVIDIA H800	1.43 Ghz	[883530] python3.11	

Left View: SASS Right View: None

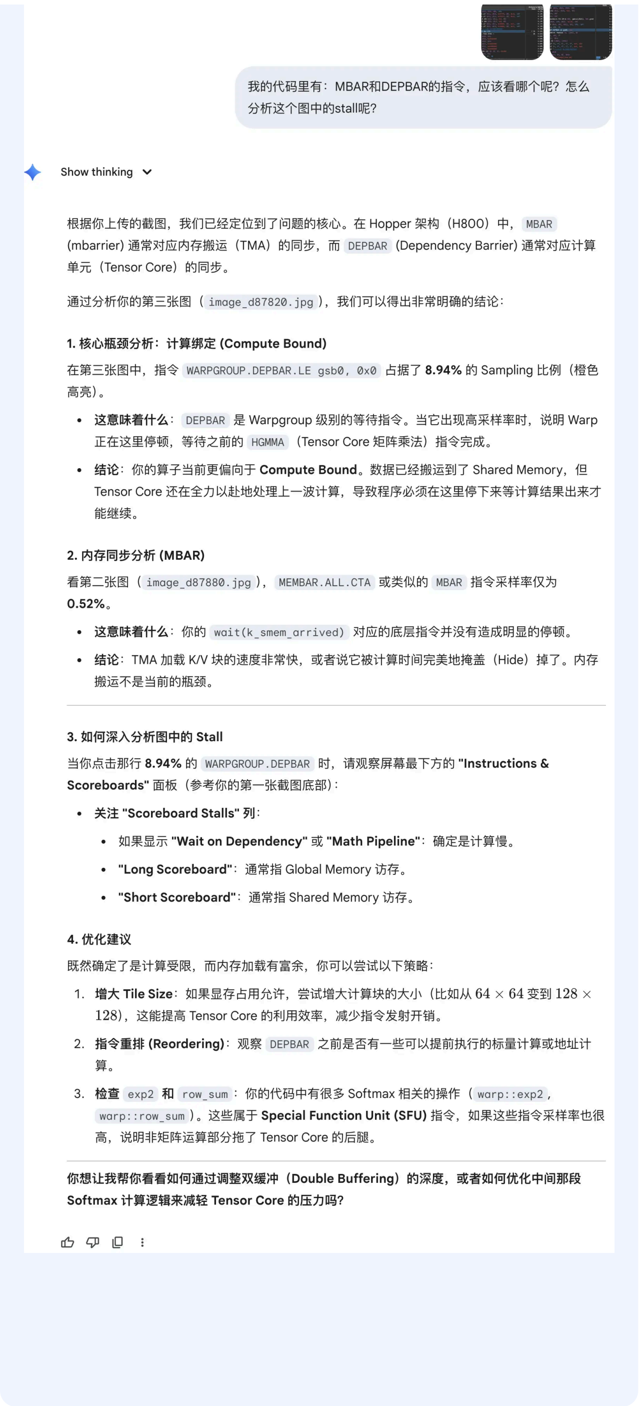
#	Label	Address	Source	Attributed Stalls	rp Stall Sampling (All Samples)	Instruction Category	Instructions Executed	g. Predicated-On hreads Executed	
328		00001550 15057770	`UIADD3.64 UR26, UR10, UR26, URZ`		0.12%	Uniform Da			
329		00001550 15057780	`@!UP1 USHF.L.U32 UR11, UR20, 0x6, URZ`		0.14%	Uniform Dat…	0.23%	8	
330		00001550 15057790	`@!UP1 UMOV UR10, URZ`		0.03%	Uniform Dat…	0.23%	8	
331		00001550 150577a0	`@!UP1 UMOV UR12, URZ`		0.04%	Uniform Dat…	0.23%	8	
332		00001550 150577b0	`HGMMA.64x64x16.F32.BF16 R88, gdesc[UR24], R88.gsb0`		0.48%	Warpgroup	0.23%	32	
333		00001550 150577c0	`@!UP1 UIADD3 UR24, UP0, UR22, 0x170, URZ`		0.04%	Uniform Dat…	0.23%	8	
334		00001550 150577d0	`@!UP1 UIADD3.X UR25, URZ, UR23, URZ, UP0, !UPT`		0.35%	Uniform Dat…	0.23%	8	
335		00001550 150577e0	`VOTEU.ALL UP0, P0`		0.20%	Uniform Dat…	0.23%	32	
336		00001550 150577f0	`WARPGROUP.DEPBAR.LE gsb0, 0x0`		8.94%	Warpgroup	0.23%	32	
337		00001550 15057800	`@!P0 SYNCS.ARRIVE.TRANS64 RZ, [UR9], R0`		0.13%	Load/Store	0.23%		
338	.L_x_9	00001550 15057810	`@P2 ELECT P1, URZ, PT`		0.38%	Control	0.23%		
339		00001550 15057820	`UMOV UR13, UR21`		0.03%	Uniform Dat…	0.23%	32	
340		00001550 15057830	`@!UP0 UTMALDG.5D [UR8], [UR24]`		0.14%	Tensor Memo…	0.23%	8	Glob
341		00001550 15057840	`@P1 PLOP3.LUT P2, PT, P1, PT, PT, 0x8, 0x0`		0.51%	Predicate	0.23%		
342		00001550 15057850	`PLOP3.LUT P1, PT, PT, PT, PT, 0x8, 0x0`		0.26%	Predicate	0.23%	32	
343		00001550 15057860	``@P2 BRA.U.ANY `(.L_x_9) 0x155015057810``		0.61%	Control	0.23%		
344		00001550 15057870	`MOV R16, UR28`		0.28%	Movement	0.23%	32	
345		00001550 15057880	`IMAD.U32 R17, RZ, RZ, UR29`		0.04%	Integer	0.23%	32	



Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Baseline 1 — 5700 - fwd_attend_ker	(364, 12, 1)x(128, 1, 1)	2.44 ms	3,497,648	0 - NVIDIA H800	1.43 Ghz	[883530] python3.11	

Left View: SASS Right View: None

#	Label	Address	Source	Attributed Stalls	rp Stall Sampling (All Samples)	Instruction Category	Instructions Executed	g. Predicated-On hreads Executed
269		00001550 150573c0	`USEL UR19, UR19, UR8, !UP0`		0.25%	Uniform Da		
270		00001550 150573d0	`ULOP3.LUT UR12, UR10, 0x3fff0, URZ, 0xc0, !UPT`		0.12%	Uniform Dat…	0.23%	32
271		00001550 150573e0	`ULOP3.LUT UR10, UR19, 0x3fff0, URZ, 0xc0, !UPT`		0.14%	Uniform Dat…	0.23%	32
272		00001550 150573f0	`USHF.R.U64 UR12, UR12, 0x4, URZ`		0.12%	Uniform Dat…	0.23%	32
273		00001550 15057400	`USHF.R.U64 UR10, UR10, 0x4, URZ`		0.13%	Uniform Dat…	0.23%	32
274		00001550 15057410	`ULOP3.LUT UR12, UR12, 0x10000, URZ, 0xfc, !UPT`		0.11%	Uniform Dat…	0.23%	32
275		00001550 15057420	`ULOP3.LUT UR10, UR10, 0x10000, URZ, 0xfc, !UPT`		0.19%	Uniform Dat…	0.23%	32
276		00001550 15057430	``@!P1 BRA `(.L_x_8) 0x15501505bcd0``		0.10%	Control	0.23%	
277	.L_x_35	00001550 15057440	`MEMBAR.ALL.CTA`	< 1%	0.52%	Load/Store	0.23%	32
278		00001550 15057450	`FENCE.VIEW.ASYNC.S`	2%	0.41%	Load/Store	0.23%	32
279		00001550 15057460	`UMOV UR24, UR12`		0.04%	Uniform Dat…	0.23%	32
280		00001550 15057470	`UMOV UR25, 0x40000040`		0.03%	Uniform Dat…	0.23%	32
281		00001550 15057480	`UMOV UR26, UR10`		0.04%	Uniform Dat…	0.23%	32
282		00001550 15057490	`UMOV UR27, 0x40000040`		0.04%	Uniform Dat…	0.23%	32
283		00001550 150574a0	`UMOV UR13, 0x40000040`		0.03%	Uniform Dat…	0.23%	32
284		00001550 150574b0	`UMOV UR11, 0x40000040`		0.04%	Uniform Dat…	0.23%	32
285		00001550 150574c0	`@!P0 IMAD.MOV.U32 R0, RZ, RZ, 0x4000`		0.03%	Integer	0.23%	
286		00001550 150574d0	`R2UR UR20, R16`		2.04%	Uniform Dat…	0.23%	32



我的代码里有：MBAR和DEPBAR的指令，应该看哪个呢？怎么分析这个图中的stall呢？

根据你上传的截图，我们已经定位到了问题的核心。在 Hopper 架构（H800）中，`MBAR` (mbarrier) 通常对应内存搬运（TMA）的同步，而 `DEPBAR` (Dependency Barrier) 通常对应计算单元（Tensor Core）的同步。

通过分析你的第三张图（`image_d87820.jpg`），我们可以得出非常明确的结论：

1. 核心瓶颈分析：计算绑定 (Compute Bound)

在第三张图中，指令 `WARPGROUP.DEPBAR.LE gsb0, 0x0` 占据了 **8.94%** 的 Sampling 比例（橙色高亮）。

- **这意味着什么**：`DEPBAR` 是 Warpgroup 级别的等待指令。当它出现高采样率时，说明 Warp 正在这里停顿，等待之前的 `HGMMA`（Tensor Core 矩阵乘法）指令完成。
- **结论**：你的算子当前更偏向于 **Compute Bound**。数据已经搬运到了 Shared Memory，但 Tensor Core 还在全力以赴地处理上一波计算，导致程序必须在这里停下来等计算结果出来才能继续。

2. 内存同步分析 (MBAR)

看第二张图（`image_d87880.jpg`），`MEMBAR.ALL.CTA` 或类似的 `MBAR` 指令采样率仅为 **0.52%**。

- **这意味着什么**：你的 `wait(k_smem_arrived)` 对应的底层指令并没有造成明显的停顿。
- **结论**：TMA 加载 K/V 块的速度非常快，或者说它被计算时间完美地掩盖（Hide）掉了。内存搬运不是当前的瓶颈。

---

3. 如何深入分析图中的 Stall

当你点击那行 **8.94%** 的 `WARPGROUP.DEPBAR` 时，请观察屏幕最下方的 **"Instructions & Scoreboards"** 面板（参考你的第一张截图底部）：

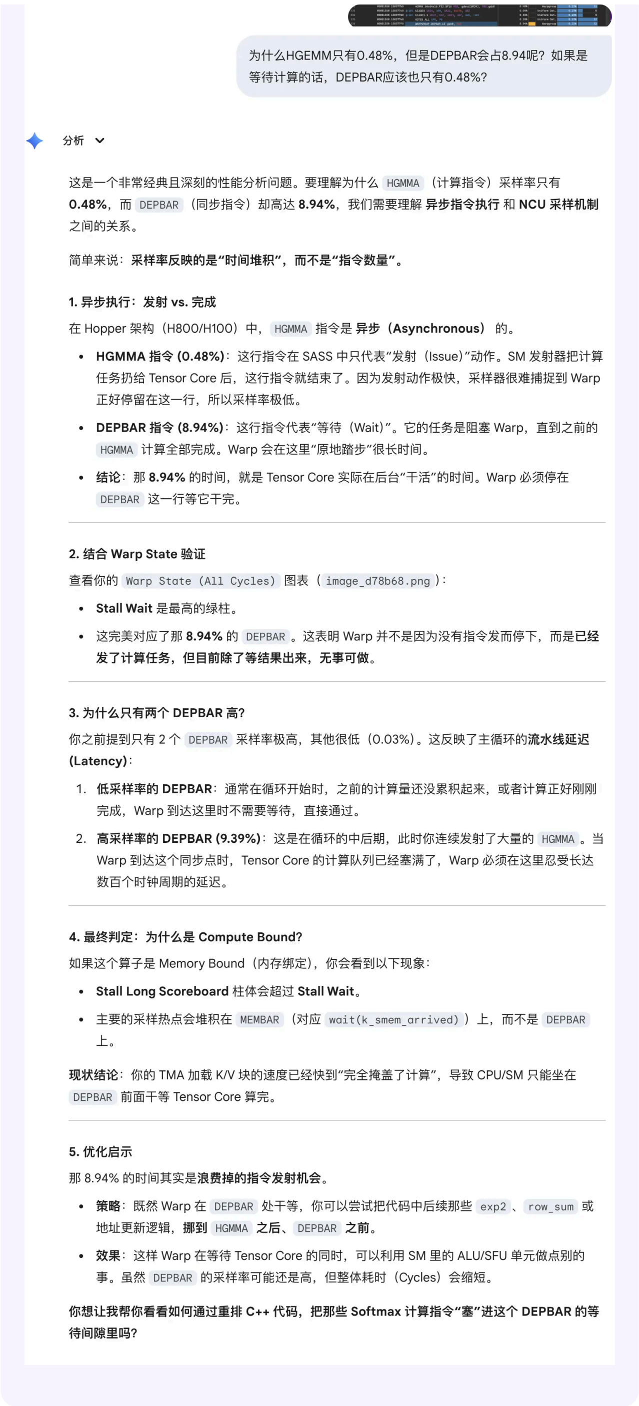
- **关注 "Scoreboard Stalls" 列**：
  - 如果显示 **"Wait on Dependency"** 或 **"Math Pipeline"**：确定是计算慢。
  - **"Long Scoreboard"**：通常指 Global Memory 访存。
  - **"Short Scoreboard"**：通常指 Shared Memory 访存。

4. 优化建议

既然确定了是计算受限，而内存加载有富余，你可以尝试以下策略：

1. **增大 Tile Size**：如果显存占用允许，尝试增大计算块的大小（比如从 64×64 变到 128×128），这能提高 Tensor Core 的利用效率，减少指令发射开销。
2. **指令重排 (Reordering)**：观察 `DEPBAR` 之前是否有一些可以提前执行的标量计算或地址计算。
3. **检查 `exp2` 和 `row_sum`**：你的代码中有很多 Softmax 相关的操作（`warp::exp2`，`warp::row_sum`）。这些属于 **Special Function Unit (SFU)** 指令，如果这些指令采样率也很高，说明非矩阵运算部分拖了 Tensor Core 的后腿。

你想让我帮你看看如何通过调整双缓冲（Double Buffering）的深度，或者如何优化中间那段 Softmax 计算逻辑来减轻 Tensor Core 的压力吗？



为什么HGEMM只有0.48%，但是DEPBAR会占8.94呢？如果是等待计算的话，DEPBAR应该也只有0.48%?

这是一个非常经典且深刻的性能分析问题。要理解为什么 `HGMMA`（计算指令）采样率只有 **0.48%**，而 `DEPBAR`（同步指令）却高达 **8.94%**，我们需要理解 **异步指令执行** 和 **NCU 采样机制** 之间的关系。

简单来说：**采样率反映的是“时间堆积”，而不是“指令数量”。**

1. 异步执行：发射 vs. 完成

在 Hopper 架构（H800/H100）中，`HGMMA` 指令是 **异步（Asynchronous）** 的。

- **HGMMA 指令 (0.48%)**：这行指令在 SASS 中只代表“发射（Issue）”动作。SM 发射器把计算任务扔给 Tensor Core 后，这行指令就结束了。因为发射动作极快，采样器很难捕捉到 Warp 正好停留在这一行，所以采样率极低。
- **DEPBAR 指令 (8.94%)**：这行指令代表“等待（Wait）”。它的任务是阻塞 Warp，直到之前的 `HGMMA` 计算全部完成。Warp 会在这里“原地踏步”很长时间。
- **结论**：那 **8.94%** 的时间，就是 Tensor Core 实际在后台“干活”的时间。Warp 必须停在 `DEPBAR` 这一行等它干完。

---

2. 结合 Warp State 验证

查看你的 `Warp State (All Cycles)` 图表（`image_d78b68.png`）：

- **Stall Wait** 是最高的绿柱。
- 这完美对应了那 **8.94%** 的 `DEPBAR`。这表明 Warp 并不是因为没有指令发而停下，而是**已经发了计算任务，但目前除了等结果出来，无事可做。**

---

3. 为什么只有两个 DEPBAR 高?

你之前提到只有 2 个 `DEPBAR` 采样率极高，其他很低（0.03%）。这反映了主循环的**流水线延迟 (Latency)**：

1. **低采样率的 DEPBAR**：通常在循环开始时，之前的计算量还没累积起来，或者计算正好刚刚完成，Warp 到达这里时不需要等待，直接通过。
2. **高采样率的 DEPBAR (9.39%)**：这是在循环的中后期，此时你连续发射了大量的 `HGMMA`。当 Warp 到达这个同步点时，Tensor Core 的计算队列已经塞满了，Warp 必须在这里忍受长达数百个时钟周期的延迟。

---

4. 最终判定：为什么是 Compute Bound?

如果这个算子是 Memory Bound（内存绑定），你会看到以下现象：

- **Stall Long Scoreboard** 柱体会超过 **Stall Wait**。
- 主要的采样热点会堆积在 `MEMBAR`（对应 `wait(k_smem_arrived)`）上，而不是 `DEPBAR` 上。

现状结论：你的 TMA 加载 K/V 块的速度已经快到“完全掩盖了计算”，导致 CPU/SM 只能坐在 `DEPBAR` 前面干等 Tensor Core 算完。

---

5. 优化启示

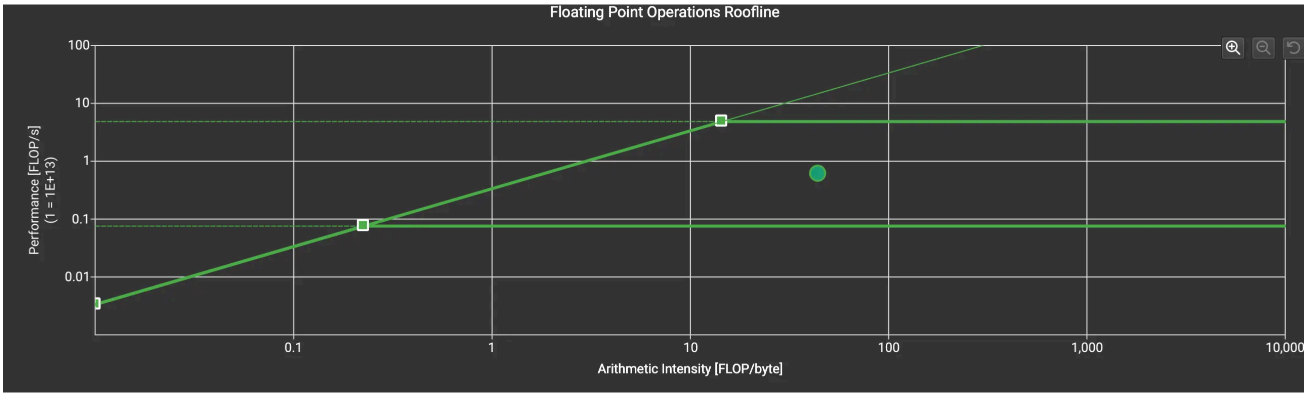
那 8.94% 的时间其实是**浪费掉的指令发射机会**。

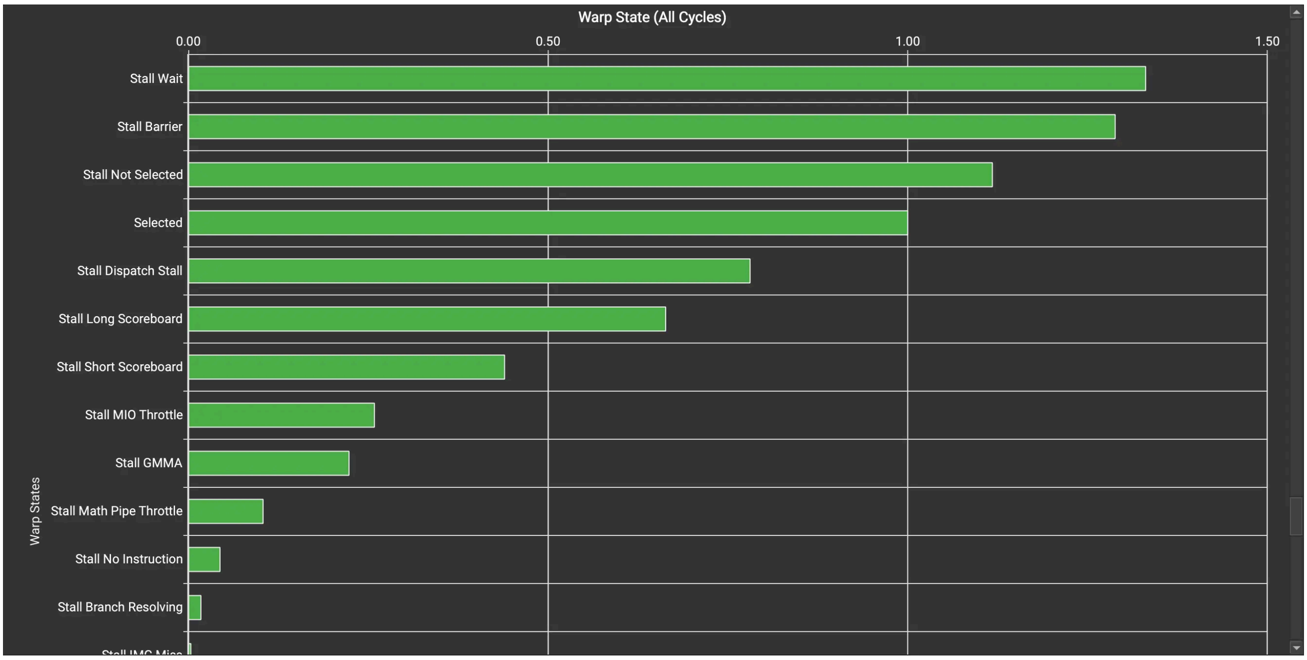
- **策略**：既然 Warp 在 `DEPBAR` 处干等，你可以尝试把代码中后续那些 `exp2`、`row_sum` 或地址更新逻辑，**挪到 `HGMMA` 之后、`DEPBAR` 之前。**
- **效果**：这样 Warp 在等待 Tensor Core 的同时，可以利用 SM 里的 ALU/SFU 单元做点别的事。虽然 `DEPBAR` 的采样率可能还是高，但整体耗时（Cycles）会缩短。

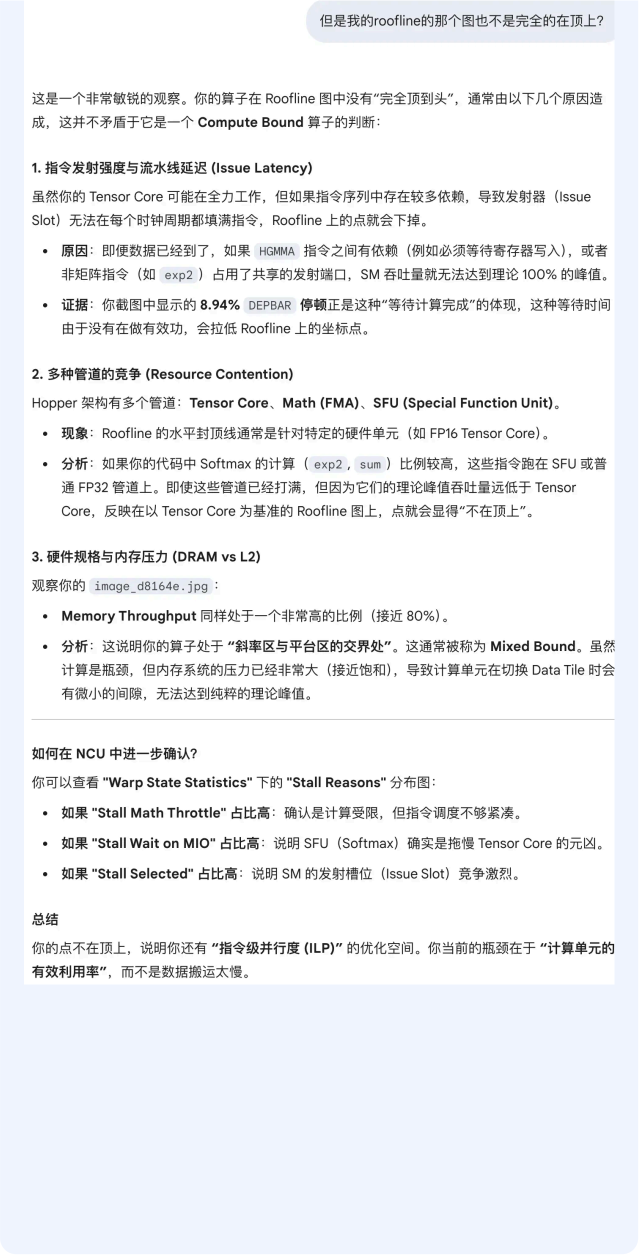
你想让我帮你看看如何通过重排 C++ 代码，把那些 Softmax 计算指令“塞”进这个 DEPBAR 的等待间隙里吗?

- ~~**sum:** `wait` **k block的指令其实已经执行完，此时访存kv block已经执行完成，在等待gemm的计算。**~~

~~但是为什么roofline的图点不在屋顶上，显示是compute bound呢？~~


Floating Point Operations Roofline
Performance [FLOP/s]
(1 = 1E+13)
100
10
1
0.1
0.01
0.1
1
10
100
1,000
10,000
Arithmetic Intensity [FLOP/byte]


Warp State (All Cycles)
Warp States
0.00
0.50
1.00
1.50
Stall Wait
Stall Barrier
Stall Not Selected
Selected
Stall Dispatch Stall
Stall Long Scoreboard
Stall Short Scoreboard
Stall MIO Throttle
Stall GMMA
Stall Math Pipe Throttle
Stall No Instruction
Stall Branch Resolving



但是我的roofline的那个图也不是完全的在顶上?

这是一个非常敏锐的观察。你的算子在 Roofline 图中没有“完全顶到头”，通常由以下几个原因造成，这并不矛盾于它是一个 **Compute Bound** 算子的判断：

1. 指令发射强度与流水线延迟 (Issue Latency)

虽然你的 Tensor Core 可能在全力工作，但如果指令序列中存在较多依赖，导致发射器（Issue Slot）无法在每个时钟周期都填满指令，Roofline 上的点就会下掉。

- **原因**：即便数据已经到了，如果 `HGMMA` 指令之间有依赖（例如必须等待寄存器写入），或者非矩阵指令（如 `exp2`）占用了共享的发射端口，SM 吞吐量就无法达到理论 100% 的峰值。
- **证据**：你截图中显示的 **8.94%** `DEPBAR` **停顿**正是这种“等待计算完成”的体现，这种等待时间由于没有在做有效功，会拉低 Roofline 上的坐标点。

2. 多种管道的竞争 (Resource Contention)

Hopper 架构有多个管道：**Tensor Core**、**Math (FMA)**、**SFU (Special Function Unit)**。

- **现象**：Roofline 的水平封顶线通常是针对特定的硬件单元（如 FP16 Tensor Core）。
- **分析**：如果你的代码中 Softmax 的计算（`exp2`，`sum`）比例较高，这些指令跑在 SFU 或普通 FP32 管道上。即使这些管道已经打满，但因为它们的理论峰值吞吐量远低于 Tensor Core，反映在以 Tensor Core 为基准的 Roofline 图上，点就会显得“不在顶上”。

3. 硬件规格与内存压力 (DRAM vs L2)

观察你的 `image_d8164e.jpg`：

- **Memory Throughput** 同样处于一个非常高的比例（接近 80%）。
- **分析**：这说明你的算子处于 **“斜率区与平台区的交界处”**。这通常被称为 **Mixed Bound**。虽然计算是瓶颈，但内存系统的压力已经非常大（接近饱和），导致计算单元在切换 Data Tile 时会有微小的间隙，无法达到纯粹的理论峰值。

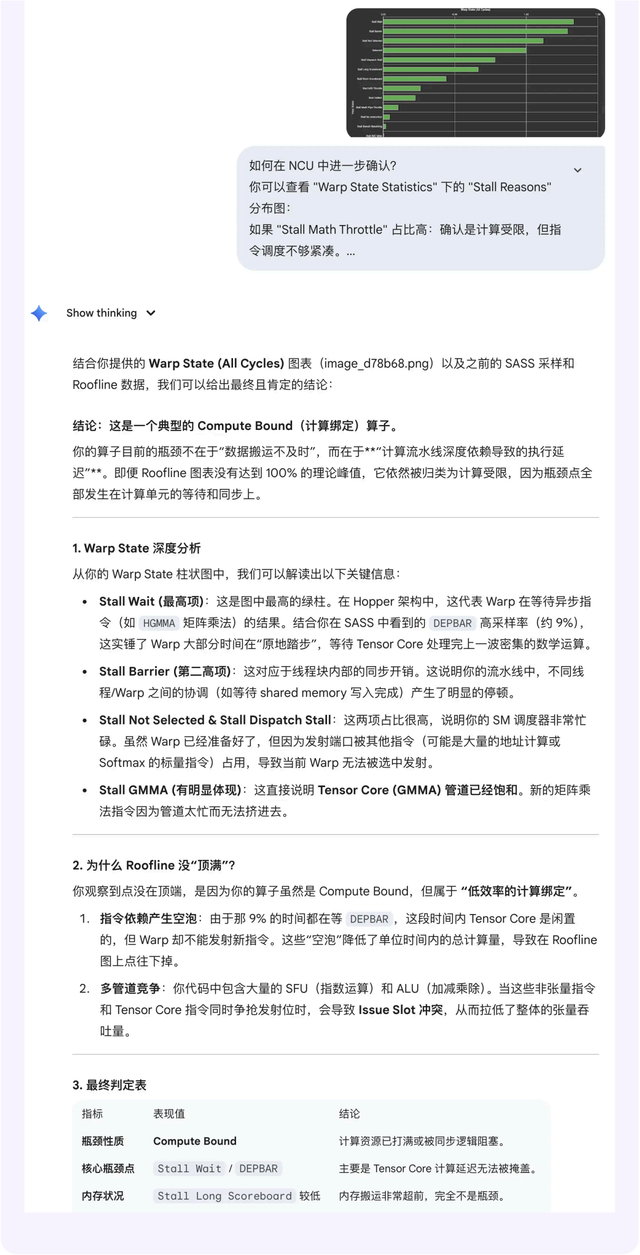
如何在 NCU 中进一步确认?

你可以查看 **"Warp State Statistics"** 下的 **"Stall Reasons"** 分布图：

- **如果 "Stall Math Throttle" 占比高**：确认是计算受限，但指令调度不够紧凑。
- **如果 "Stall Wait on MIO" 占比高**：说明 SFU（Softmax）确实是拖慢 Tensor Core 的元凶。
- **如果 "Stall Selected" 占比高**：说明 SM 的发射槽位（Issue Slot）竞争激烈。

总结

你的点不在顶上，说明你还有 **“指令级并行度 (ILP)”** 的优化空间。你当前的瓶颈在于 **“计算单元的有效利用率”**，而不是数据搬运太慢。



如何在 NCU 中进一步确认?
你可以查看 "Warp State Statistics" 下的 "Stall Reasons" 分布图：
如果 "Stall Math Throttle" 占比高：确认是计算受限，但指令调度不够紧凑。...

结合你提供的 **Warp State (All Cycles)** 图表（image_d78b68.png）以及之前的 SASS 采样和 Roofline 数据，我们可以给出最终且肯定的结论：

结论：这是一个典型的 Compute Bound（计算绑定）算子。

你的算子目前的瓶颈不在于“数据搬运不及时”，而在于**“计算流水线深度依赖导致的执行延迟”**。即便 Roofline 图表没有达到 100% 的理论峰值，它依然被归类为计算受限，因为瓶颈点全部发生在计算单元的等待和同步上。

---

1. Warp State 深度分析

从你的 Warp State 柱状图中，我们可以解读出以下关键信息：

- **Stall Wait (最高项)**：这是图中最高的绿柱。在 Hopper 架构中，这代表 Warp 在等待异步指令（如 `HGMMA` 矩阵乘法）的结果。结合你在 SASS 中看到的 `DEPBAR` 高采样率（约 9%），这实锤了 Warp 大部分时间在“原地踏步”，等待 Tensor Core 处理完上一波密集的数学运算。
- **Stall Barrier (第二高项)**：这对应于线程块内部的同步开销。这说明你的流水线中，不同线程/Warp 之间的协调（如等待 shared memory 写入完成）产生了明显的停顿。
- **Stall Not Selected & Stall Dispatch Stall**：这两项占比很高，说明你的 SM 调度器非常忙碌。虽然 Warp 已经准备好了，但因为发射端口被其他指令（可能是大量的地址计算或 Softmax 的标量指令）占用，导致当前 Warp 无法被选中发射。
- **Stall GMMA (有明显体现)**：这直接说明 **Tensor Core (GMMA) 管道已经饱和**。新的矩阵乘法指令因为管道太忙而无法挤进去。

---

2. 为什么 Roofline 没“顶满”?

你观察到点没在顶端，是因为你的算子虽然是 Compute Bound，但属于 **“低效率的计算绑定”**。

1. **指令依赖产生空泡**：由于那 9% 的时间都在等 `DEPBAR`，这段时间内 Tensor Core 是闲置的，但 Warp 却不能发射新指令。这些“空泡”降低了单位时间内的总计算量，导致在 Roofline 图上点往下掉。
2. **多管道竞争**：你代码中包含大量的 SFU（指数运算）和 ALU（加减乘除）。当这些非张量指令和 Tensor Core 指令同时争抢发射位时，会导致 **Issue Slot 冲突**，从而拉低了整体的张量吞吐量。

---

3. 最终判定表

指标	表现值	结论
瓶颈性质	**Compute Bound**	计算资源已打满或被同步逻辑阻塞。
核心瓶颈点	`Stall Wait` / `DEPBAR`	主要是 Tensor Core 计算延迟无法被掩盖。
内存状况	`Stall Long Scoreboard` 较低	内存搬运非常超前，完全不是瓶颈。

- **~~sum：roofline是由于其他计算和空泡导致打不到顶。~~**

- **~~sum: 所以通过上面两个分析，该算子在load稀疏的kv_block_cache的时候已经会被GEMM compute bound住，load的时间被计算overlap了，此时应该已经是一个compute bound的算子。~~**

> ~~如果我们这个场景下考虑kvcache的话，当前block计算时需要读取之前所有block的kvcache，并把当前block的kv的值和历史kvcache拼接。因为fa中kvcache应该是按照连续或者block_table进行管理的，所以读取的时候是连续的一段。一个潜在的问题就是拼接的时候，当前block的kv值可能会有很多并且内存不连续token，拼接的时候或许会有访存的问题。不过拼接完之后的flash attn计算就是相同的方式了。~~

~~**上面这里我上周分析到拼接会有稀疏访存的问题，但是我这里理解的不太对，**拼接的时候只会有一个block的new_K, new_V 通过store拼接到一块完整的old_KV cache的显存的后面，所以不是稀疏访存，只有连续的load old_KV和append store一块连续的new_KV。~~

~~下面是现在attention_kernel的input格式。~~

```
std::vector<torch::Tensor>
block_sparse_attention_forward(
    torch::Tensor q,
    torch::Tensor k,
    torch::Tensor v,
    torch::Tensor q2k_block_sparse_index, //每个q对应的稀疏的k,v block index
    torch::Tensor q2k_block_sparse_num,  // 每个q对应的稀疏的block数量，为了控制循环遍历稀疏块的次数
    torch::Tensor kv_block_size  // 每个kvblock内部的有效tokens数量
)
```

~~在传入attention kernel的时候需要是连续的一块的显存空间。如果考虑with kvcache，那么在该kernel之前就需要将old_kvcache和new_kvcache进行拼接，然后变成连续的一块显存，传入kernel。具体的完整执行也就是:~~

1. ~~新的一个block，输入是X，通过W_q, W_k, W_v得到new_Q, new_K, new_V，**然后和old_KV进行拼接形成full_K, full_V，然后变到一块连续的显存地址。**~~
2. ~~full_K,V通过pool计算会得到attn_score，从而可以进行dynamic sparse，此时可以得到q对应的q2k_block_sparse_index, q2k_block_sparse_num, kv_block_size等该kernel的其他输入参数，之后进行kernel的计算。~~

~~而在进入该kernel之前，old_kv已经是连续的一段显存，**load old_KV是连续的一段访存，而需要的就是把new_K, new_V拼接到该连续显存的后面，这也只是一个store的操作**~~

~~**而实际上在kernel内部，该已有的kernel已经解决了稀疏访存，其实就是通过tma异步的去load一个kv block块，和compute overlap起来，并且此时通过前面profile看稀疏访存已经被计算overlap了。**~~

~~**结论：如果kvcache的seq_len变得更长，**首先由于稀疏，每个q对应的q2k_block_sparse_num的数量也不会变得特别多，**即使一个q会对应很多的k,v block，或者是很少的k, v block**，在attention_kernel之前的操作是一样的，也就是拼接kvcache，而kernel内部的计算也是一样的，每一次都是取q,k,v的一个同样大小的tile在shared memory上进行计算，**所以即使再长，preload kv_block_cache的时间也同样会被计算overlap住。**~~

~~上面的profile结果是: Q的seq_len=23296，K,V的seq_len由于使用了linformer压缩，23296/64 = 364.~~

with kvcache

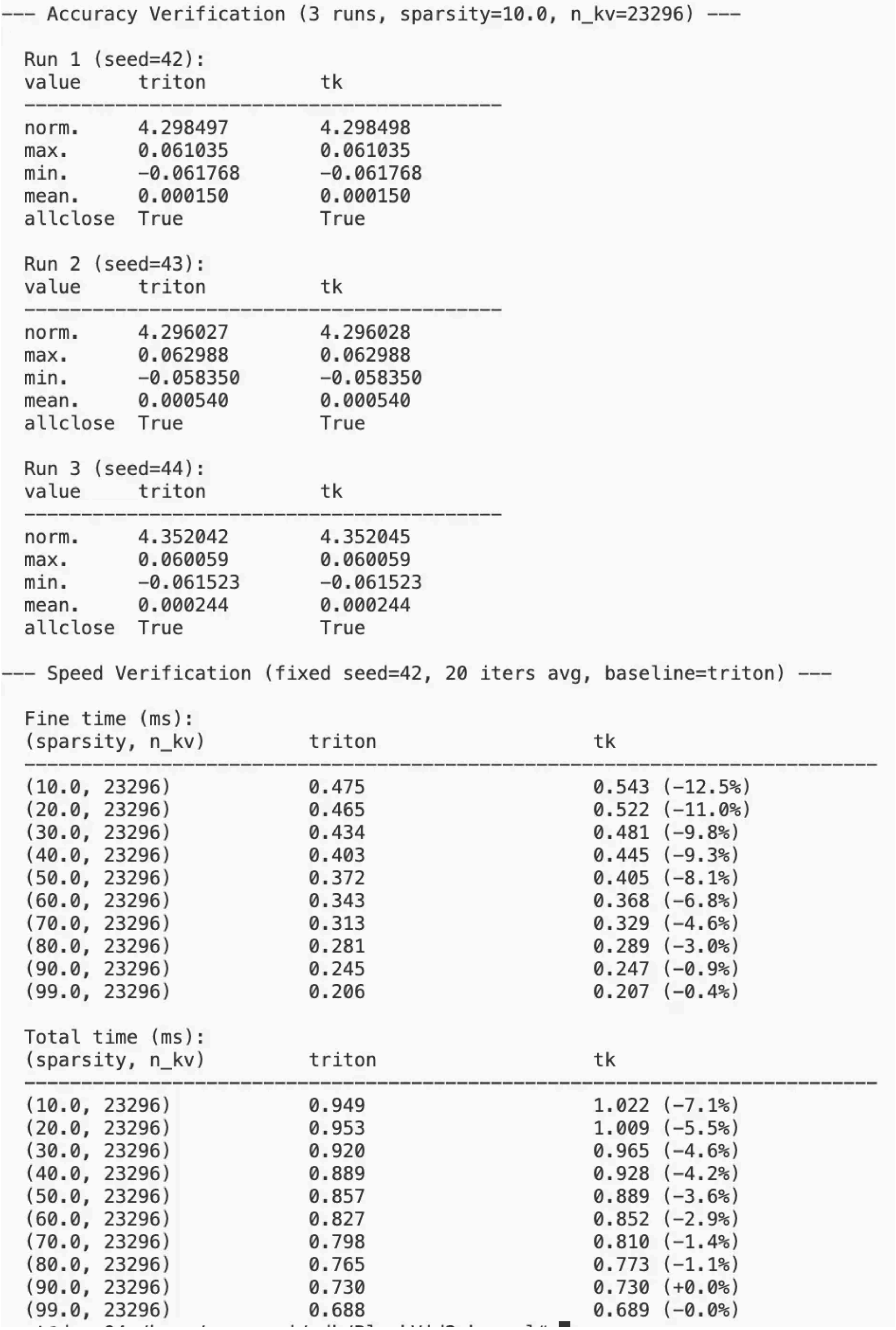
gemini误导了我，上面关于compute bound的事情是错的，实际上就是memory bound。以下分析没问题。

结论：如果kvcache的seq_len变得更长，首先由于稀疏，每个q对应的q2k_block_sparse_num的数量也不会变得特别多，**即使一个q会对应很多的k,v block，或者是很少的k, v block**，在attention_kernel之前的操作是一样的，也就是拼接kvcache，而kernel内部的计算也是一样的，每一次都是取q,k,v的一个同样大小的tile在shared memory上进行计算，

此时考虑直接看decode，考虑每次只有1个q_block，64tokens，k,v为23296 tokens。

由于不好知道实际的dynamic sparse ratio，这里我还是先用topk的方法去模拟sparsity ratio，这部分的开销不是主要部分。我这里就profile了sparsity从10到99的数值下的kernel的加速情况。

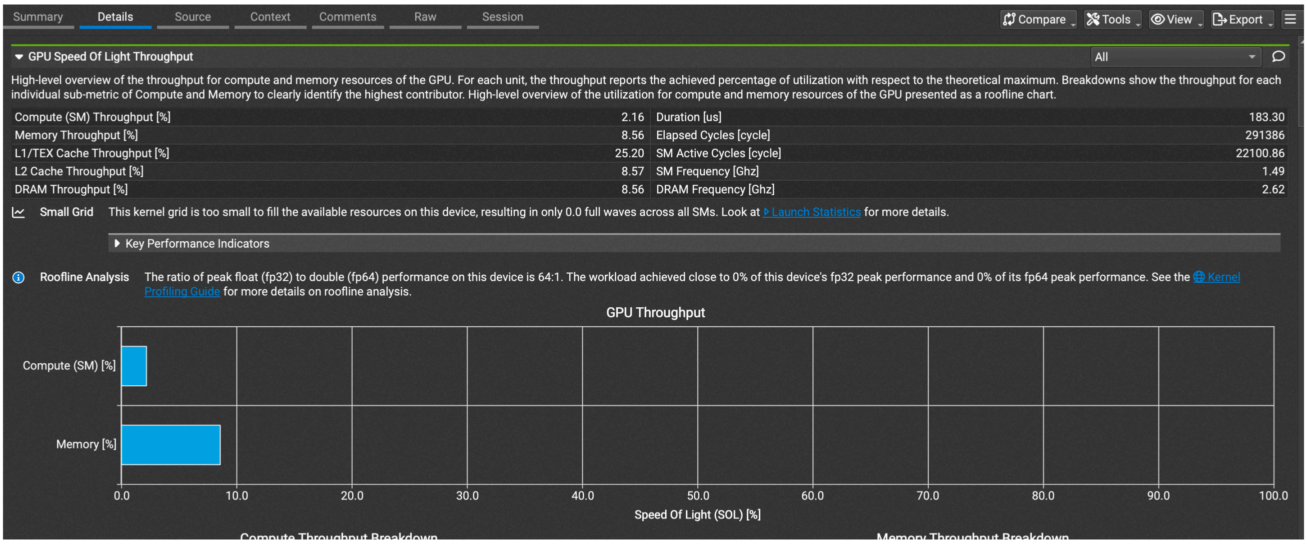
无优化版本



```
--- Accuracy Verification (3 runs, sparsity=10.0, n_kv=23296) ---

  Run 1 (seed=42):
  value     triton          tk
  ----------------------------------------
  norm.     4.298497        4.298498
  max.      0.061035        0.061035
  min.      -0.061768       -0.061768
  mean.     0.000150        0.000150
  allclose  True            True

  Run 2 (seed=43):
  value     triton          tk
  ----------------------------------------
  norm.     4.296027        4.296028
  max.      0.062988        0.062988
  min.      -0.058350       -0.058350
  mean.     0.000540        0.000540
  allclose  True            True

  Run 3 (seed=44):
  value     triton          tk
  ----------------------------------------
  norm.     4.352042        4.352045
  max.      0.060059        0.060059
  min.      -0.061523       -0.061523
  mean.     0.000244        0.000244
  allclose  True            True

--- Speed Verification (fixed seed=42, 20 iters avg, baseline=triton) ---

  Fine time (ms):
  (sparsity, n_kv)         triton                   tk
  ----------------------------------------------------------------------------
  (10.0, 23296)            0.475                    0.543 (-12.5%)
  (20.0, 23296)            0.465                    0.522 (-11.0%)
  (30.0, 23296)            0.434                    0.481 (-9.8%)
  (40.0, 23296)            0.403                    0.445 (-9.3%)
  (50.0, 23296)            0.372                    0.405 (-8.1%)
  (60.0, 23296)            0.343                    0.368 (-6.8%)
  (70.0, 23296)            0.313                    0.329 (-4.6%)
  (80.0, 23296)            0.281                    0.289 (-3.0%)
  (90.0, 23296)            0.245                    0.247 (-0.9%)
  (99.0, 23296)            0.206                    0.207 (-0.4%)

  Total time (ms):
  (sparsity, n_kv)         triton                   tk
  ----------------------------------------------------------------------------
  (10.0, 23296)            0.949                    1.022 (-7.1%)
  (20.0, 23296)            0.953                    1.009 (-5.5%)
  (30.0, 23296)            0.920                    0.965 (-4.6%)
  (40.0, 23296)            0.889                    0.928 (-4.2%)
  (50.0, 23296)            0.857                    0.889 (-3.6%)
  (60.0, 23296)            0.827                    0.852 (-2.9%)
  (70.0, 23296)            0.798                    0.810 (-1.4%)
  (80.0, 23296)            0.765                    0.773 (-1.1%)
  (90.0, 23296)            0.730                    0.730 (+0.0%)
  (99.0, 23296)            0.688                    0.689 (-0.0%)
```

split-k

此时的访存和计算utilization都非常低。因为实际在kernel中计算的q只有一个block，所以可以利用的sm就只有一个，可以使用split-k进行优化。



GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

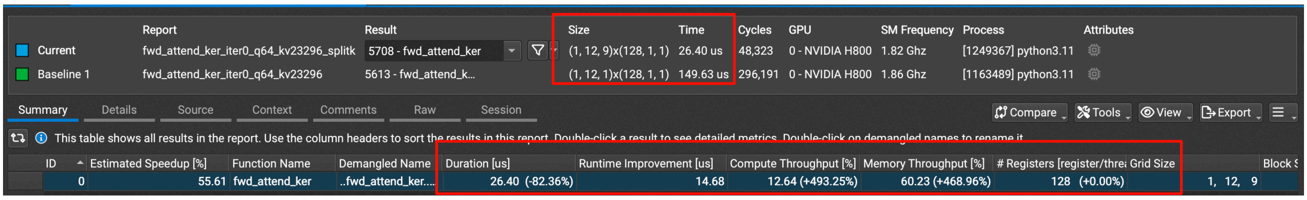
Compute (SM) Throughput [%]	2.16	Duration [us]	183.30
Memory Throughput [%]	8.56	Elapsed Cycles [cycle]	291386
L1/TEX Cache Throughput [%]	25.20	SM Active Cycles [cycle]	22100.86
L2 Cache Throughput [%]	8.57	SM Frequency [Ghz]	1.49
DRAM Throughput [%]	8.56	DRAM Frequency [Ghz]	2.62

Small Grid This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full waves across all SMs. Look at Launch Statistics for more details.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The workload achieved close to 0% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

下面是对比：

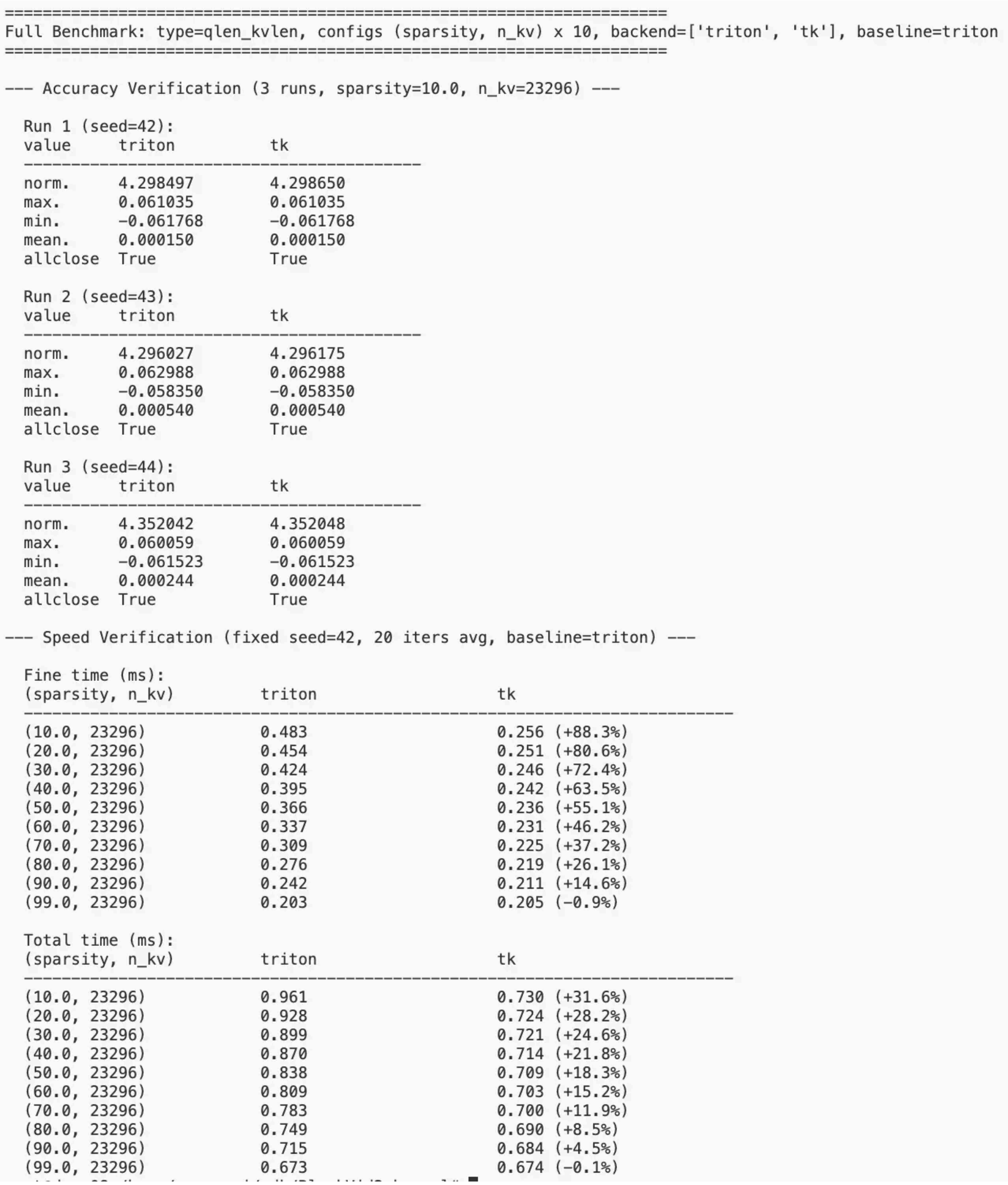
memory throughput大大增加，compute throughput也增加，最终时间从149.63 -> 26.4，提升5.67倍。



	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
🟦 Current	fwd_attend_ker_iter0_q64_kv23296_splitk	5708 - fwd_attend_ker	(1, 12, 9)x(128, 1, 1)	26.40 us	48,323	0 - NVIDIA H800	1.82 Ghz	[1249367] python3.11	
🟩 Baseline 1	fwd_attend_ker_iter0_q64_kv23296	5613 - fwd_attend_k...	(1, 12, 1)x(128, 1, 1)	149.63 us	296,191	0 - NVIDIA H800	1.86 Ghz	[1163489] python3.11	

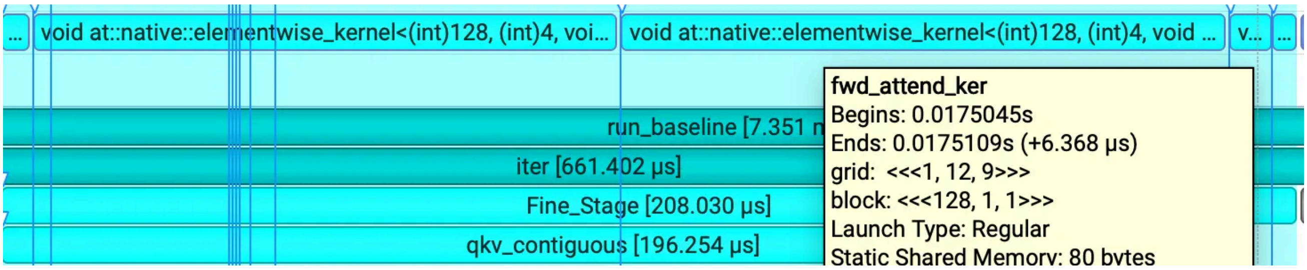
This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.

ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration [us]	Runtime Improvement [us]	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/threa	Grid Size	Block S
0	55.61	fwd_attend_ker	..fwd_attend_ker....	26.40 (-82.36%)	14.68	12.64 (+493.25%)	60.23 (+468.96%)	128 (+0.00%)	1, 12, 9	



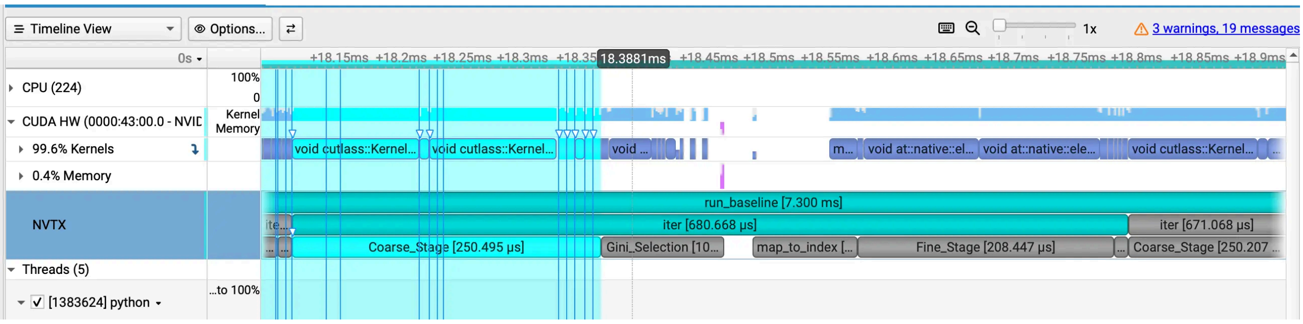
```
======================================================================
Full Benchmark: type=qlen_kvlen, configs (sparsity, n_kv) x 10, backend=['triton', 'tk'], baseline=triton
======================================================================

--- Accuracy Verification (3 runs, sparsity=10.0, n_kv=23296) ---

  Run 1 (seed=42):
  value     triton          tk
  ------------------------------------------
  norm.     4.298497        4.298650
  max.      0.061035        0.061035
  min.      -0.061768       -0.061768
  mean.     0.000150        0.000150
  allclose  True            True

  Run 2 (seed=43):
  value     triton          tk
  ------------------------------------------
  norm.     4.296027        4.296175
  max.      0.062988        0.062988
  min.      -0.058350       -0.058350
  mean.     0.000540        0.000540
  allclose  True            True

  Run 3 (seed=44):
  value     triton          tk
  ------------------------------------------
  norm.     4.352042        4.352048
  max.      0.060059        0.060059
  min.      -0.061523       -0.061523
  mean.     0.000244        0.000244
  allclose  True            True

--- Speed Verification (fixed seed=42, 20 iters avg, baseline=triton) ---

  Fine time (ms):
  (sparsity, n_kv)         triton                   tk
  --------------------------------------------------------------------------
  (10.0, 23296)            0.483                    0.256 (+88.3%)
  (20.0, 23296)            0.454                    0.251 (+80.6%)
  (30.0, 23296)            0.424                    0.246 (+72.4%)
  (40.0, 23296)            0.395                    0.242 (+63.5%)
  (50.0, 23296)            0.366                    0.236 (+55.1%)
  (60.0, 23296)            0.337                    0.231 (+46.2%)
  (70.0, 23296)            0.309                    0.225 (+37.2%)
  (80.0, 23296)            0.276                    0.219 (+26.1%)
  (90.0, 23296)            0.242                    0.211 (+14.6%)
  (99.0, 23296)            0.203                    0.205 (-0.9%)

  Total time (ms):
  (sparsity, n_kv)         triton                   tk
  --------------------------------------------------------------------------
  (10.0, 23296)            0.961                    0.730 (+31.6%)
  (20.0, 23296)            0.928                    0.724 (+28.2%)
  (30.0, 23296)            0.899                    0.721 (+24.6%)
  (40.0, 23296)            0.870                    0.714 (+21.8%)
  (50.0, 23296)            0.838                    0.709 (+18.3%)
  (60.0, 23296)            0.809                    0.703 (+15.2%)
  (70.0, 23296)            0.783                    0.700 (+11.9%)
  (80.0, 23296)            0.749                    0.690 (+8.5%)
  (90.0, 23296)            0.715                    0.684 (+4.5%)
  (99.0, 23296)            0.673                    0.674 (-0.1%)
```

因为 k, v copy 到连续访存的时间大概为0.2ms


...
void at::native::elementwise_kernel<(int)128, (int)4, voi...
void at::native::elementwise_kernel<(int)128, (int)4, void ...
v...
...
run_baseline [7.351 n
iter [661.402 μs]
Fine_Stage [208.030 μs]
qkv_contiguous [196.254 μs]

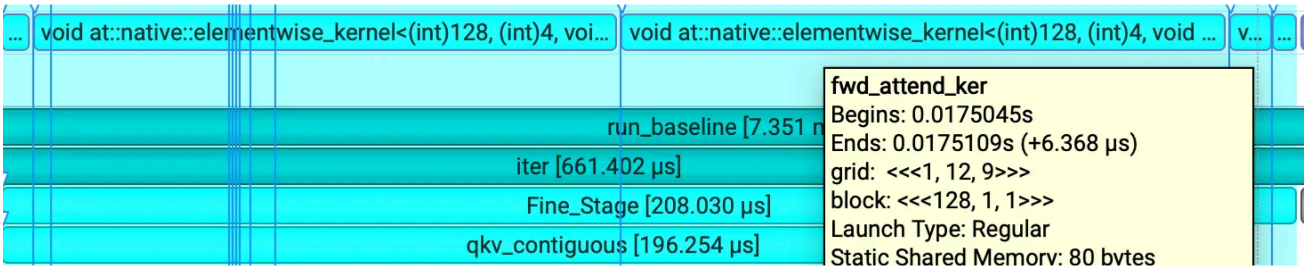
fwd_attend_ker
Begins: 0.0175045s
Ends: 0.0175109s (+6.368 μs)
grid: <<<1, 12, 9>>>
block: <<<128, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 80 bytes

此时 fine stage 的计算时间已经很短了



这里还有点发现，这里是sparse branch中的sparse_block_attention的模块，我这里因为是模拟的，所以kvcache不是连续的，在执行kernel前需要contiguous拷贝到一块连续的显存，这块占比的时间其实挺大。

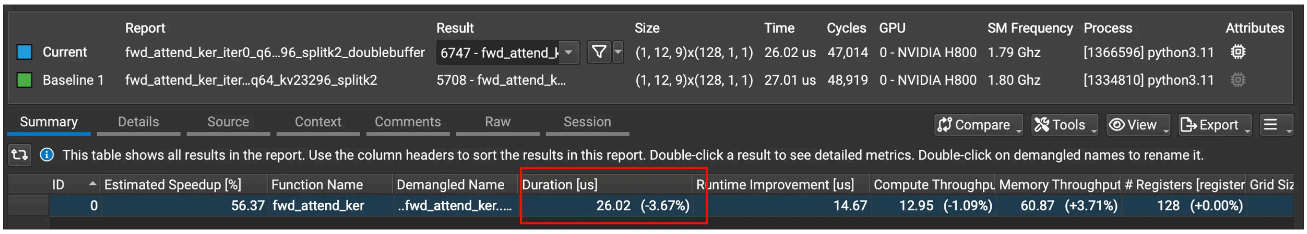
如果一开始就用连续的一段kvcache，这里的contiguous的开销占比其实也挺大的，不过这是在kernel之前的处理，和kernel的实现没有关系，需要在框架层再做一下，我粗略看了zeyu现在的代码的kvcache是layer粒度的，所以可能会存在kvcache碎片的情况。


void at::native::elementwise_kernel<(int)128, (int)4, voi...
void at::native::elementwise_kernel<(int)128, (int)4, void ...
v...
run_baseline [7.351 n
iter [661.402 μs]
Fine_Stage [208.030 μs]
qkv_contiguous [196.254 μs]

fwd_attend_ker
Begins: 0.0175045s
Ends: 0.0175109s (+6.368 μs)
grid: <<<1, 12, 9>>>
block: <<<128, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 80 bytes

double buffering

提升不大了。



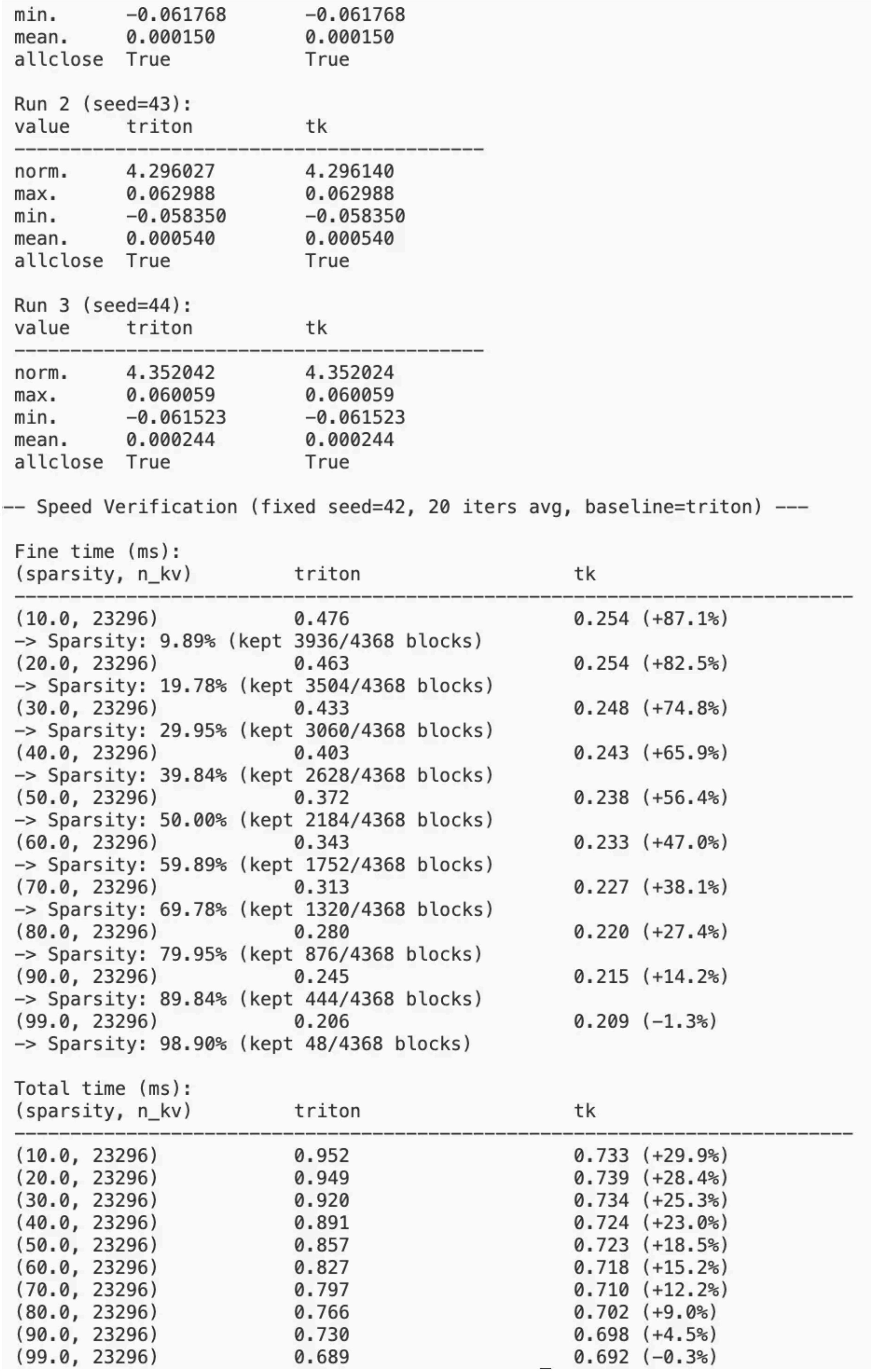
	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	fwd_attend_ker_iter0_q6...96_splitk2_doublebuffer	6747 - fwd_attend_k	(1, 12, 9)x(128, 1, 1)	26.02 us	47,014	0 - NVIDIA H800	1.79 Ghz	[1366596] python3.11	
Baseline 1	fwd_attend_ker_iter...q64_kv23296_splitk2	5708 - fwd_attend_k...	(1, 12, 9)x(128, 1, 1)	27.01 us	48,919	0 - NVIDIA H800	1.80 Ghz	[1334810] python3.11	

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.

ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration [us]	Runtime Improvement [us]	Compute Throughpu	Memory Throughput	# Registers [register	Grid Siz
0	56.37	fwd_attend_ker	..fwd_attend_ker....	26.02 (-3.67%)	14.67	12.95 (-1.09%)	60.87 (+3.71%)	128 (+0.00%)	

增大tma的load size (64->128)

用kitten感觉实现有点问题，尝试了下也没效果

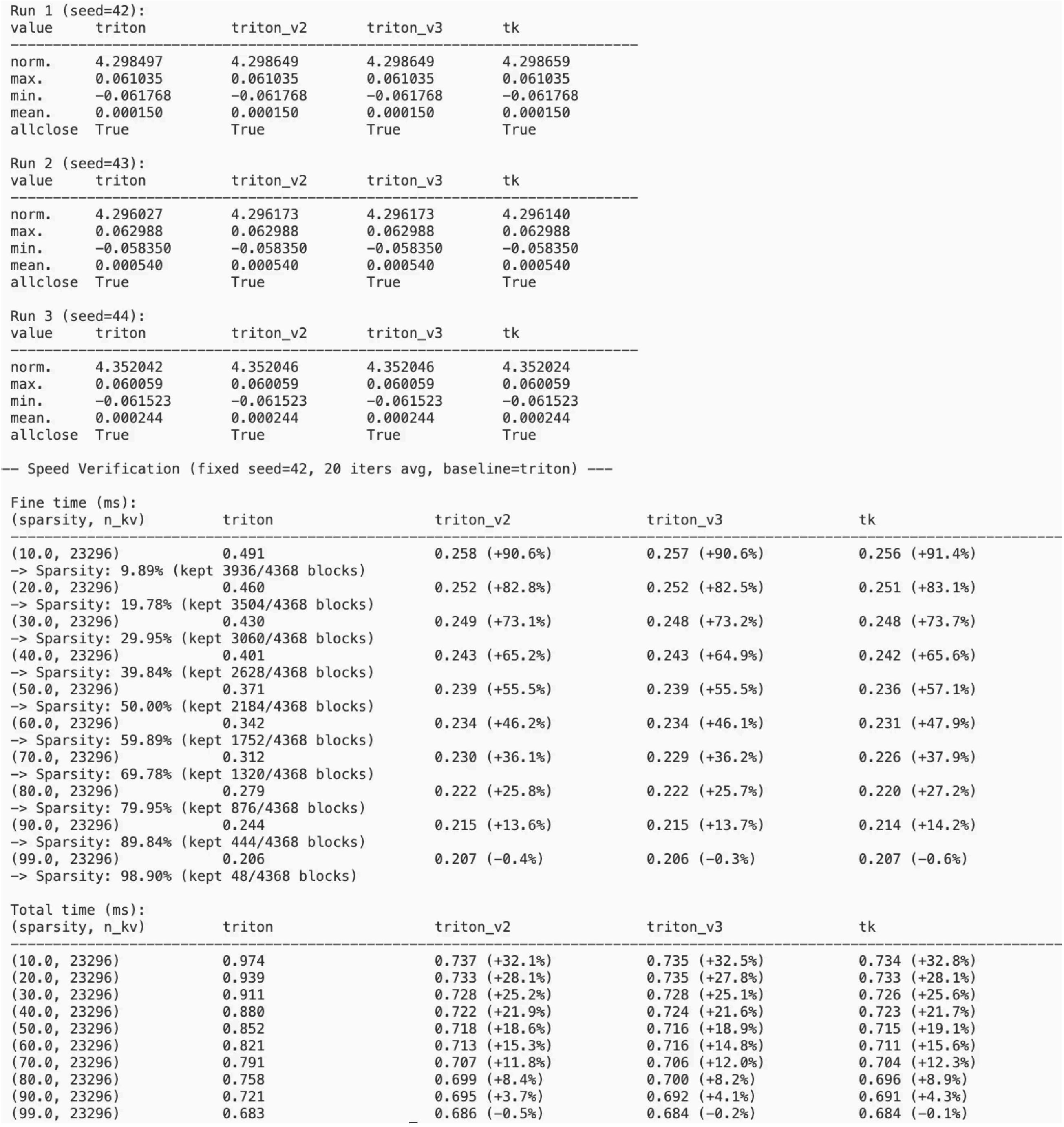


```
min.      -0.061768       -0.061768
mean.     0.000150        0.000150
allclose  True            True

Run 2 (seed=43):
value     triton          tk
------------------------------------------
norm.     4.296027        4.296140
max.      0.062988        0.062988
min.      -0.058350       -0.058350
mean.     0.000540        0.000540
allclose  True            True

Run 3 (seed=44):
value     triton          tk
------------------------------------------
norm.     4.352042        4.352024
max.      0.060059        0.060059
min.      -0.061523       -0.061523
mean.     0.000244        0.000244
allclose  True            True

-- Speed Verification (fixed seed=42, 20 iters avg, baseline=triton) ---

Fine time (ms):
(sparsity, n_kv)         triton                   tk
---------------------------------------------------------------------------
(10.0, 23296)            0.476                    0.254 (+87.1%)
-> Sparsity: 9.89% (kept 3936/4368 blocks)
(20.0, 23296)            0.463                    0.254 (+82.5%)
-> Sparsity: 19.78% (kept 3504/4368 blocks)
(30.0, 23296)            0.433                    0.248 (+74.8%)
-> Sparsity: 29.95% (kept 3060/4368 blocks)
(40.0, 23296)            0.403                    0.243 (+65.9%)
-> Sparsity: 39.84% (kept 2628/4368 blocks)
(50.0, 23296)            0.372                    0.238 (+56.4%)
-> Sparsity: 50.00% (kept 2184/4368 blocks)
(60.0, 23296)            0.343                    0.233 (+47.0%)
-> Sparsity: 59.89% (kept 1752/4368 blocks)
(70.0, 23296)            0.313                    0.227 (+38.1%)
-> Sparsity: 69.78% (kept 1320/4368 blocks)
(80.0, 23296)            0.280                    0.220 (+27.4%)
-> Sparsity: 79.95% (kept 876/4368 blocks)
(90.0, 23296)            0.245                    0.215 (+14.2%)
-> Sparsity: 89.84% (kept 444/4368 blocks)
(99.0, 23296)            0.206                    0.209 (-1.3%)
-> Sparsity: 98.90% (kept 48/4368 blocks)

Total time (ms):
(sparsity, n_kv)         triton                   tk
---------------------------------------------------------------------------
(10.0, 23296)            0.952                    0.733 (+29.9%)
(20.0, 23296)            0.949                    0.739 (+28.4%)
(30.0, 23296)            0.920                    0.734 (+25.3%)
(40.0, 23296)            0.891                    0.724 (+23.0%)
(50.0, 23296)            0.857                    0.723 (+18.5%)
(60.0, 23296)            0.827                    0.718 (+15.2%)
(70.0, 23296)            0.797                    0.710 (+12.2%)
(80.0, 23296)            0.766                    0.702 (+9.0%)
(90.0, 23296)            0.730                    0.698 (+4.5%)
(99.0, 23296)            0.689                    0.692 (-0.3%)
```

想到triton上也可以实现splitk

首先给triton的kernel支持了q_seqlen和kv_seqlen不等的情况

然后在triton上实现split-k的优化，效果也很好：



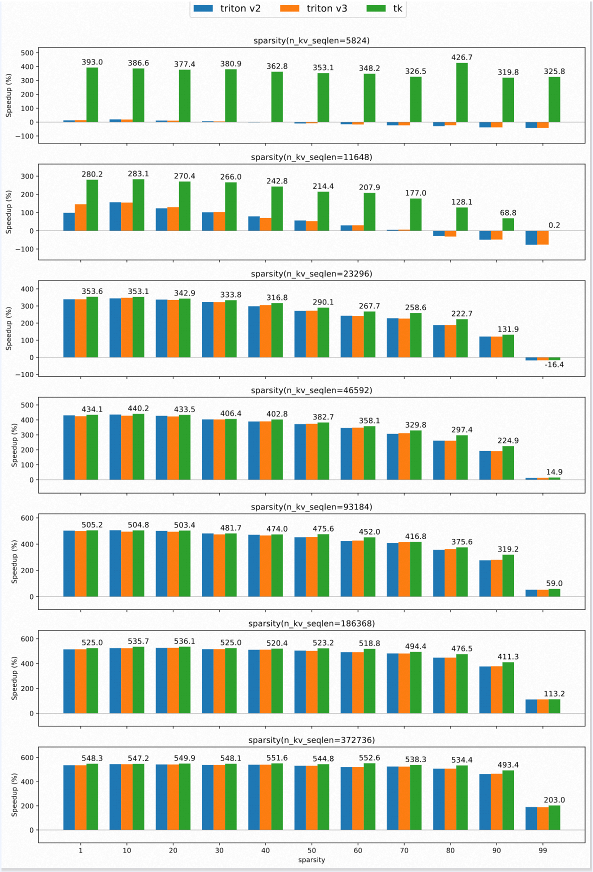
```
Run 1 (seed=42):
value     triton          triton_v2       triton_v3       tk
--------------------------------------------------------------------------
norm.     4.298497        4.298649        4.298649        4.298659
max.      0.061035        0.061035        0.061035        0.061035
min.      -0.061768       -0.061768       -0.061768       -0.061768
mean.     0.000150        0.000150        0.000150        0.000150
allclose  True            True            True            True

Run 2 (seed=43):
value     triton          triton_v2       triton_v3       tk
--------------------------------------------------------------------------
norm.     4.296027        4.296173        4.296173        4.296140
max.      0.062988        0.062988        0.062988        0.062988
min.      -0.058350       -0.058350       -0.058350       -0.058350
mean.     0.000540        0.000540        0.000540        0.000540
allclose  True            True            True            True

Run 3 (seed=44):
value     triton          triton_v2       triton_v3       tk
--------------------------------------------------------------------------
norm.     4.352042        4.352046        4.352046        4.352024
max.      0.060059        0.060059        0.060059        0.060059
min.      -0.061523       -0.061523       -0.061523       -0.061523
mean.     0.000244        0.000244        0.000244        0.000244
allclose  True            True            True            True

-- Speed Verification (fixed seed=42, 20 iters avg, baseline=triton) ---

Fine time (ms):
(sparsity, n_kv)         triton                   triton_v2                triton_v3                tk
----------------------------------------------------------------------------------------------------------------------------
(10.0, 23296)            0.491                    0.258 (+90.6%)           0.257 (+90.6%)           0.256 (+91.4%)
-> Sparsity: 9.89% (kept 3936/4368 blocks)
(20.0, 23296)            0.460                    0.252 (+82.8%)           0.252 (+82.5%)           0.251 (+83.1%)
-> Sparsity: 19.78% (kept 3504/4368 blocks)
(30.0, 23296)            0.430                    0.249 (+73.1%)           0.248 (+73.2%)           0.248 (+73.7%)
-> Sparsity: 29.95% (kept 3060/4368 blocks)
(40.0, 23296)            0.401                    0.243 (+65.2%)           0.243 (+64.9%)           0.242 (+65.6%)
-> Sparsity: 39.84% (kept 2628/4368 blocks)
(50.0, 23296)            0.371                    0.239 (+55.5%)           0.239 (+55.5%)           0.236 (+57.1%)
-> Sparsity: 50.00% (kept 2184/4368 blocks)
(60.0, 23296)            0.342                    0.234 (+46.2%)           0.234 (+46.1%)           0.231 (+47.9%)
-> Sparsity: 59.89% (kept 1752/4368 blocks)
(70.0, 23296)            0.312                    0.230 (+36.1%)           0.229 (+36.2%)           0.226 (+37.9%)
-> Sparsity: 69.78% (kept 1320/4368 blocks)
(80.0, 23296)            0.279                    0.222 (+25.8%)           0.222 (+25.7%)           0.220 (+27.2%)
-> Sparsity: 79.95% (kept 876/4368 blocks)
(90.0, 23296)            0.244                    0.215 (+13.6%)           0.215 (+13.7%)           0.214 (+14.2%)
-> Sparsity: 89.84% (kept 444/4368 blocks)
(99.0, 23296)            0.206                    0.207 (-0.4%)            0.206 (-0.3%)            0.207 (-0.6%)
-> Sparsity: 98.90% (kept 48/4368 blocks)

Total time (ms):
(sparsity, n_kv)         triton                   triton_v2                triton_v3                tk
----------------------------------------------------------------------------------------------------------------------------
(10.0, 23296)            0.974                    0.737 (+32.1%)           0.735 (+32.5%)           0.734 (+32.8%)
(20.0, 23296)            0.939                    0.733 (+28.1%)           0.735 (+27.8%)           0.733 (+28.1%)
(30.0, 23296)            0.911                    0.728 (+25.2%)           0.728 (+25.1%)           0.726 (+25.6%)
(40.0, 23296)            0.880                    0.722 (+21.9%)           0.724 (+21.6%)           0.723 (+21.7%)
(50.0, 23296)            0.852                    0.718 (+18.6%)           0.716 (+18.9%)           0.715 (+19.1%)
(60.0, 23296)            0.821                    0.713 (+15.3%)           0.716 (+14.8%)           0.711 (+15.6%)
(70.0, 23296)            0.791                    0.707 (+11.8%)           0.706 (+12.0%)           0.704 (+12.3%)
(80.0, 23296)            0.758                    0.699 (+8.4%)            0.700 (+8.2%)            0.696 (+8.9%)
(90.0, 23296)            0.721                    0.695 (+3.7%)            0.692 (+4.1%)            0.691 (+4.3%)
(99.0, 23296)            0.683                    0.686 (-0.5%)            0.684 (-0.2%)            0.684 (-0.1%)
```

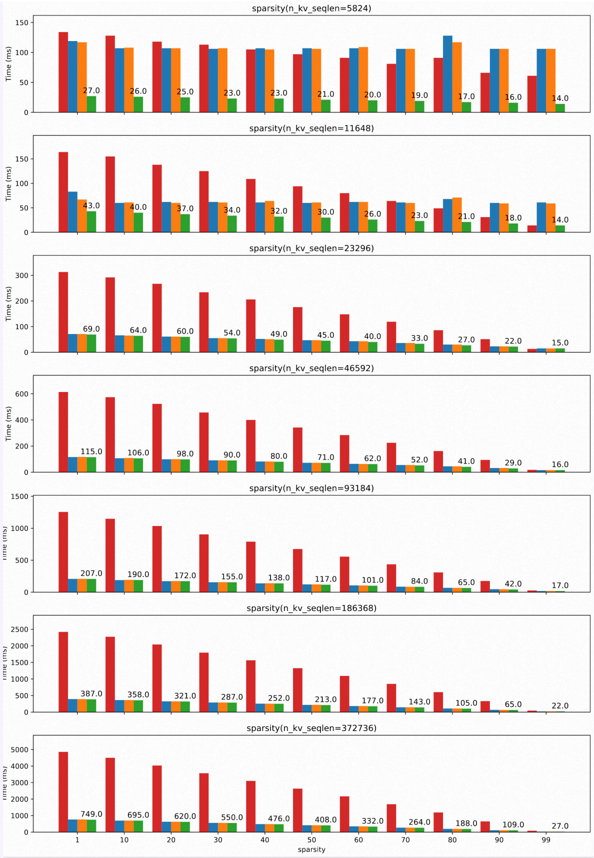
e2e test

decode的情况下（q_tokens=64, kv_tokens变化，下图每一行表示一个kv_tokens_num）sparse_block_attention的kernel执行时间

speedup:

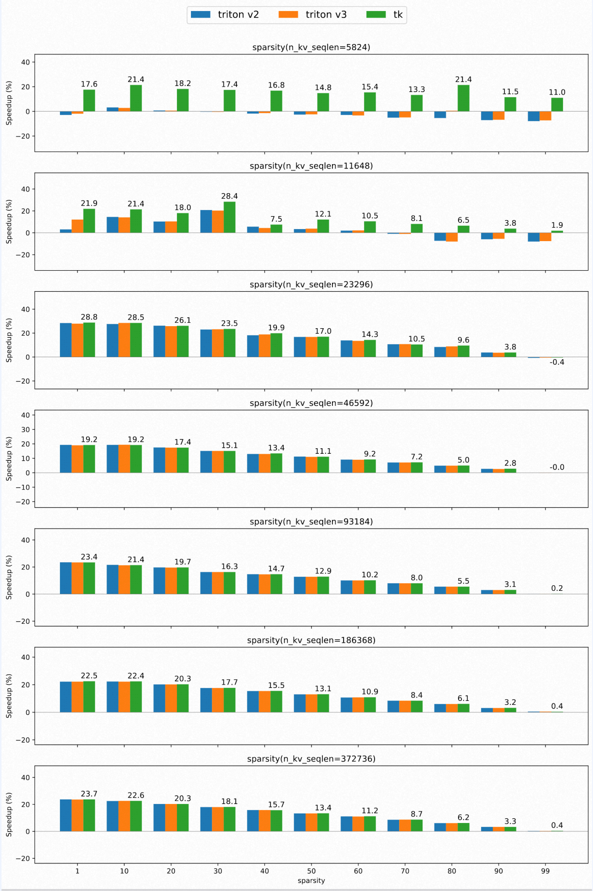
absolute time:



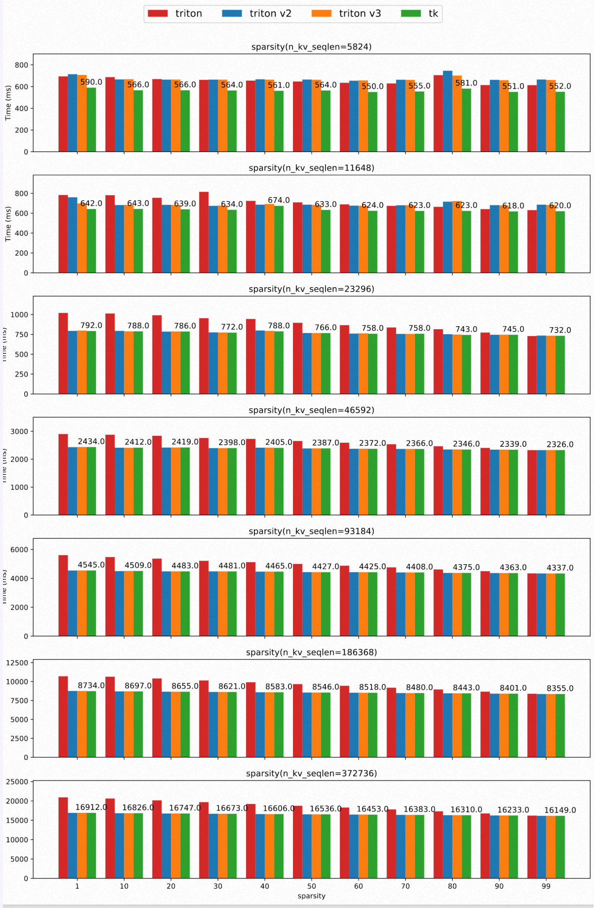


total的执行时间

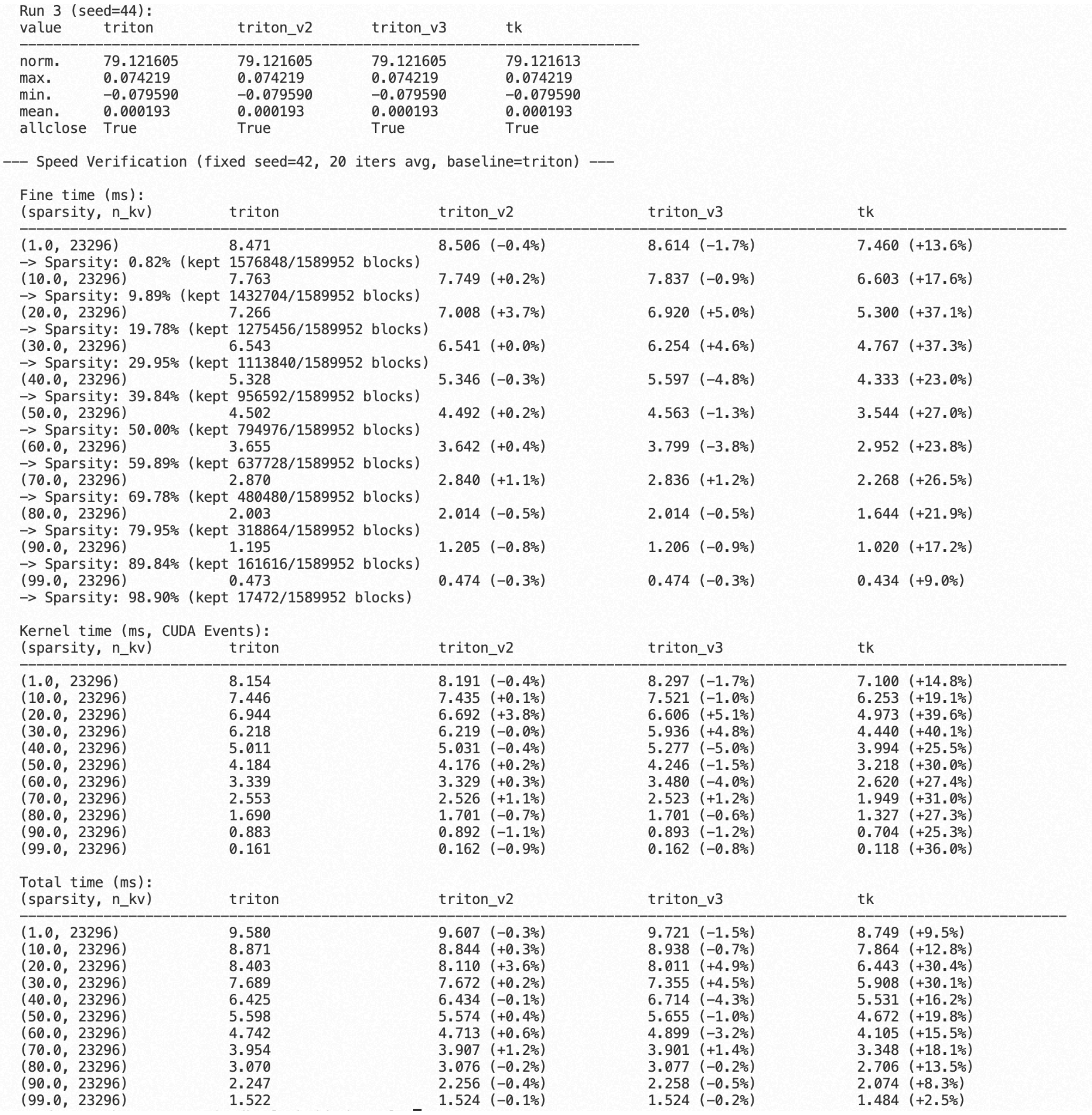
speedup



absolute time



如果不考虑kvcache，也就是prefill的情况下，q_seqlen和kv_seqlen相等，这里都取23296，使用vsa的原始thunderkitten的kernel。如果使用了splitk, double buffer反而有负优化。



```
  Run 3 (seed=44):
  value     triton          triton_v2       triton_v3       tk
  --------------------------------------------------------------------------
  norm.     79.121605       79.121605       79.121605       79.121613
  max.      0.074219        0.074219        0.074219        0.074219
  min.      -0.079590       -0.079590       -0.079590       -0.079590
  mean.     0.000193        0.000193        0.000193        0.000193
  allclose  True            True            True            True

--- Speed Verification (fixed seed=42, 20 iters avg, baseline=triton) ---

  Fine time (ms):
  (sparsity, n_kv)         triton                   triton_v2                triton_v3                tk
  -----------------------------------------------------------------------------------------------------------------------------
  (1.0, 23296)             8.471                    8.506 (-0.4%)            8.614 (-1.7%)            7.460 (+13.6%)
  -> Sparsity: 0.82% (kept 1576848/1589952 blocks)
  (10.0, 23296)            7.763                    7.749 (+0.2%)            7.837 (-0.9%)            6.603 (+17.6%)
  -> Sparsity: 9.89% (kept 1432704/1589952 blocks)
  (20.0, 23296)            7.266                    7.008 (+3.7%)            6.920 (+5.0%)            5.300 (+37.1%)
  -> Sparsity: 19.78% (kept 1275456/1589952 blocks)
  (30.0, 23296)            6.543                    6.541 (+0.0%)            6.254 (+4.6%)            4.767 (+37.3%)
  -> Sparsity: 29.95% (kept 1113840/1589952 blocks)
  (40.0, 23296)            5.328                    5.346 (-0.3%)            5.597 (-4.8%)            4.333 (+23.0%)
  -> Sparsity: 39.84% (kept 956592/1589952 blocks)
  (50.0, 23296)            4.502                    4.492 (+0.2%)            4.563 (-1.3%)            3.544 (+27.0%)
  -> Sparsity: 50.00% (kept 794976/1589952 blocks)
  (60.0, 23296)            3.655                    3.642 (+0.4%)            3.799 (-3.8%)            2.952 (+23.8%)
  -> Sparsity: 59.89% (kept 637728/1589952 blocks)
  (70.0, 23296)            2.870                    2.840 (+1.1%)            2.836 (+1.2%)            2.268 (+26.5%)
  -> Sparsity: 69.78% (kept 480480/1589952 blocks)
  (80.0, 23296)            2.003                    2.014 (-0.5%)            2.014 (-0.5%)            1.644 (+21.9%)
  -> Sparsity: 79.95% (kept 318864/1589952 blocks)
  (90.0, 23296)            1.195                    1.205 (-0.8%)            1.206 (-0.9%)            1.020 (+17.2%)
  -> Sparsity: 89.84% (kept 161616/1589952 blocks)
  (99.0, 23296)            0.473                    0.474 (-0.3%)            0.474 (-0.3%)            0.434 (+9.0%)
  -> Sparsity: 98.90% (kept 17472/1589952 blocks)

  Kernel time (ms, CUDA Events):
  (sparsity, n_kv)         triton                   triton_v2                triton_v3                tk
  -----------------------------------------------------------------------------------------------------------------------------
  (1.0, 23296)             8.154                    8.191 (-0.4%)            8.297 (-1.7%)            7.100 (+14.8%)
  (10.0, 23296)            7.446                    7.435 (+0.1%)            7.521 (-1.0%)            6.253 (+19.1%)
  (20.0, 23296)            6.944                    6.692 (+3.8%)            6.606 (+5.1%)            4.973 (+39.6%)
  (30.0, 23296)            6.218                    6.219 (-0.0%)            5.936 (+4.8%)            4.440 (+40.1%)
  (40.0, 23296)            5.011                    5.031 (-0.4%)            5.277 (-5.0%)            3.994 (+25.5%)
  (50.0, 23296)            4.184                    4.176 (+0.2%)            4.246 (-1.5%)            3.218 (+30.0%)
  (60.0, 23296)            3.339                    3.329 (+0.3%)            3.480 (-4.0%)            2.620 (+27.4%)
  (70.0, 23296)            2.553                    2.526 (+1.1%)            2.523 (+1.2%)            1.949 (+31.0%)
  (80.0, 23296)            1.690                    1.701 (-0.7%)            1.701 (-0.6%)            1.327 (+27.3%)
  (90.0, 23296)            0.883                    0.892 (-1.1%)            0.893 (-1.2%)            0.704 (+25.3%)
  (99.0, 23296)            0.161                    0.162 (-0.9%)            0.162 (-0.8%)            0.118 (+36.0%)

  Total time (ms):
  (sparsity, n_kv)         triton                   triton_v2                triton_v3                tk
  -----------------------------------------------------------------------------------------------------------------------------
  (1.0, 23296)             9.580                    9.607 (-0.3%)            9.721 (-1.5%)            8.749 (+9.5%)
  (10.0, 23296)            8.871                    8.844 (+0.3%)            8.938 (-0.7%)            7.864 (+12.8%)
  (20.0, 23296)            8.403                    8.110 (+3.6%)            8.011 (+4.9%)            6.443 (+30.4%)
  (30.0, 23296)            7.689                    7.672 (+0.2%)            7.355 (+4.5%)            5.908 (+30.1%)
  (40.0, 23296)            6.425                    6.434 (-0.1%)            6.714 (-4.3%)            5.531 (+16.2%)
  (50.0, 23296)            5.598                    5.574 (+0.4%)            5.655 (-1.0%)            4.672 (+19.8%)
  (60.0, 23296)            4.742                    4.713 (+0.6%)            4.899 (-3.2%)            4.105 (+15.5%)
  (70.0, 23296)            3.954                    3.907 (+1.2%)            3.901 (+1.4%)            3.348 (+18.1%)
  (80.0, 23296)            3.070                    3.076 (-0.2%)            3.077 (-0.2%)            2.706 (+13.5%)
  (90.0, 23296)            2.247                    2.256 (-0.4%)            2.258 (-0.5%)            2.074 (+8.3%)
  (99.0, 23296)            1.522                    1.524 (-0.1%)            1.524 (-0.2%)            1.484 (+2.5%)
```

此时with kvcache时虽然kernel的性能提升了4-5倍，但是尴尬的是，sparse branch中sparse block attention的时间占比不大，但是对于整体的时间优化其实不大。

n_q=64, n_kv=23296, sparsity=99%的情况下：

stage	time(us)	
linear branch	251.934	39.78%

dynamic selection	83.359	13.16%
kernel前处理map to index	79.135	12.50%
kernel前处理q,k,v contiguous	196.254	31.00%
sparse_block_attention kernel	**11.776**	**1.86%**
fusion	10.784	1.70%
	633.242	

另外尝试了一下cuda stream，把linear branch和sparse branch overlap起来，不过两个stream互相之间的kernel会发生竞争资源，影响大，total time反而变长了。

triton的实现，装起来会简单一些：

复制代码

```
"""
Fused Attention
===============

This is a Triton implementation of the Flash Attention v2 algorithm from Tri Dao
(https://tridao.me/publications/flash2/flash2.pdf)

Credits: OpenAI kernel team
"""

import pytest
import torch
import triton
import triton.language as tl

# ──────────────────────────── SPARSE ADDITION BEGIN ───────────────────────────
import math  # small utility needed by the sparse wrapper
# ──────────────────────────── SPARSE ADDITION END ─────────────────────────────


# We don't run auto-tuning every time to keep the tutorial fast. Keeping
# the code below and commenting out the equivalent parameters is convenient for
# re-tuning.
configs = [
    triton.Config({'BLOCK_M': BM, 'BLOCK_N': BN}, num_stages=s, num_warps=w) \
    for BM in [64]\
    for BN in [64]\
    for s in [3, 4, 5, 6, 7, 8]\
    for w in [4, 8]\
]

# ──────────────────────────── SPARSE ADDITION BEGIN ───────────────────────────
# Split-K only (no double-buffer)
@triton.autotune(configs, key=["Q_CTX", "KV_CTX", "HEAD_DIM"])
@triton.jit
def _attn_fwd_sparse(Q, K, V, sm_scale,                         #
                     q2k_index, q2k_num, max_kv_blks,           #
                     variable_block_sizes,
                     M, Out,                                    #
                     stride_qz, stride_qh, stride_qm, stride_qk,
                     stride_kz, stride_kh, stride_kn, stride_kk,
                     stride_vz, stride_vh, stride_vk, stride_vn,
                     stride_oz, stride_oh, stride_om, stride_on,
                     Z, H, Q_CTX, KV_CTX,                       #
                     HEAD_DIM: tl.constexpr,                    #
                     BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr,
                     NUM_SPLITS: tl.constexpr):
    """
    64×64 **block-sparse** forward kernel with Split-K.
    When NUM_SPLITS > 1, each instance processes a chunk of KV blocks; partial results are
reduced later.
    """

    # ----- program-id mapping -----
    q_blk   = tl.program_id(0)          # Q-tile index
    off_hz  = tl.program_id(1)          # fused (batch, head)
    split_idx = tl.program_id(2)        # split index (0 when NUM_SPLITS=1)
    b       = off_hz // H
    h       = off_hz %  H
    q_tiles = Q_CTX // BLOCK_M
    meta_base = ((b * H + h) * q_tiles + q_blk)

    kv_blocks = tl.load(q2k_num  + meta_base)                # int32
    kv_ptr    = q2k_index + meta_base * max_kv_blks          # ptr to list

    # ----- Split-K: compute kv_start, kv_end for this split -----
    chunk_size = (kv_blocks + NUM_SPLITS - 1) // NUM_SPLITS
    kv_start = split_idx * chunk_size
    kv_end = kv_start + chunk_size
    kv_end = tl.minimum(kv_end, kv_blocks)
    num_kv_this_split = kv_end - kv_start

    # ----- output batch index: when split, write to (b * NUM_SPLITS + split_idx) -----
    out_b = b * NUM_SPLITS + split_idx
    q_off = (b.to(tl.int64) * stride_qz + h.to(tl.int64) * stride_qh)
    k_off = (b.to(tl.int64) * stride_kz + h.to(tl.int64) * stride_kh)
    v_off = (b.to(tl.int64) * stride_vz + h.to(tl.int64) * stride_vh)
    o_off = (out_b.to(tl.int64) * stride_oz + h.to(tl.int64) * stride_oh)

    Q_ptr = tl.make_block_ptr(
        base=Q + q_off, shape=(Q_CTX, HEAD_DIM),
        strides=(stride_qm, stride_qk),
        offsets=(q_blk * BLOCK_M, 0),
        block_shape=(BLOCK_M, HEAD_DIM), order=(1, 0))

    K_base = tl.make_block_ptr(
        base=K + k_off, shape=(HEAD_DIM, KV_CTX),
        strides=(stride_kk, stride_kn),
        offsets=(0, 0),
        block_shape=(HEAD_DIM, BLOCK_N), order=(0, 1))
    v_order: tl.constexpr = (0, 1) if V.dtype.element_ty == tl.float8e5 else (1, 0)
    V_base = tl.make_block_ptr(
        base=V + v_off, shape=(KV_CTX, HEAD_DIM),
        strides=(stride_vk, stride_vn),
        offsets=(0, 0),
        block_shape=(BLOCK_N, HEAD_DIM), order=v_order)

    O_ptr = tl.make_block_ptr(
        base=Out + o_off, shape=(Q_CTX, HEAD_DIM),
        strides=(stride_om, stride_on),
        offsets=(q_blk * BLOCK_M, 0),
        block_shape=(BLOCK_M, HEAD_DIM), order=(1, 0))

    # ----- accumulators -----
    offs_m = (out_b * H + h) * Q_CTX + q_blk * BLOCK_M + tl.arange(0, BLOCK_M)
    m_i = tl.full([BLOCK_M], -float("inf"), tl.float32)
    l_i = tl.zeros([BLOCK_M], dtype=tl.float32) + 1.0
    acc = tl.zeros([BLOCK_M, HEAD_DIM], dtype=tl.float32)
    qk_scale = sm_scale * 1.44269504  # 1/ln2
    q = tl.load(Q_ptr)

    # ----- sparse loop over K/V tiles in this split -----
    for i in tl.range(0, num_kv_this_split):
        idx_in_list = kv_start + i
        kv_idx = tl.load(kv_ptr + idx_in_list).to(tl.int32)
        block_size = tl.load(variable_block_sizes + kv_idx)
        K_ptr = tl.advance(K_base, (0, kv_idx * BLOCK_N))
        V_ptr = tl.advance(V_base, (kv_idx * BLOCK_N, 0))

        k = tl.load(K_ptr)
        qk = tl.dot(q, k)
        # mask out invalid columns
        mask = tl.arange(0, BLOCK_N) < block_size
        qk = tl.where(mask[None, :], qk, -float("inf"))

        m_ij = tl.maximum(m_i, tl.max(qk, 1) * qk_scale)
        p = tl.math.exp2(qk * qk_scale - m_ij[:, None])
        l_ij = tl.sum(p, 1)

        alpha = tl.math.exp2(m_i - m_ij)
        l_i = l_i * alpha + l_ij
        acc = acc * alpha[:, None]

        v = tl.load(V_ptr)
        acc = tl.dot(p.to(tl.bfloat16), v, acc)
        m_i = m_ij

    # ----- epilogue -----
    m_i += tl.math.log2(l_i)
    acc = acc / l_i[:, None]
    tl.store(M + offs_m, m_i)
    tl.store(O_ptr, acc.to(Out.type.element_ty))


@triton.jit
def _attn_fwd_reduce(O_partial, M_partial, O_final, M_final,
                     batch, num_splits, qo_heads, q_seq_len,
                     HEAD_DIM: tl.constexpr,
                     stride_o_partial_b, stride_o_partial_h, stride_o_partial_m,
stride_o_partial_n,
                    stride_o_final_b, stride_o_final_h, stride_o_final_m, stride_o_final_n,
                    stride_m_partial_b, stride_m_partial_h, stride_m_partial_m,
                    stride_m_final_b, stride_m_final_h, stride_m_final_m):
    """
    Reduce partial Split-K results: softmax-weighted combination of o_partial across splits.
    O_partial: [batch*num_splits, heads, seq, head_dim]
    M_partial: [batch*num_splits, heads, seq]  (lse in log2 space)
    """
    seq_idx = tl.program_id(0)
    head_idx = tl.program_id(1)
    b_idx = tl.program_id(2)
    d_idx = tl.arange(0, HEAD_DIM)

    if seq_idx >= q_seq_len:
        return

    # Find max_lse across splits
    max_lse = -1e20
    for s in range(num_splits):
        m_idx = (b_idx * num_splits + s) * stride_m_partial_b + head_idx * stride_m_partial_h +
seq_idx * stride_m_partial_m
        l = tl.load(M_partial + m_idx)
        max_lse = tl.maximum(max_lse, l)

    # Weighted sum: out = sum(exp2(lse_i - max_lse) * o_i) / sum(exp2(lse_i - max_lse))
    sum_exp = 0.0
    out_acc = tl.zeros([HEAD_DIM], dtype=tl.float32)
    for s in range(num_splits):
        m_idx = (b_idx * num_splits + s) * stride_m_partial_b + head_idx * stride_m_partial_h +
seq_idx * stride_m_partial_m
        o_idx = (b_idx * num_splits + s) * stride_o_partial_b + head_idx * stride_o_partial_h +
seq_idx * stride_o_partial_m
        lse = tl.load(M_partial + m_idx)
        exp_val = tl.math.exp2(lse - max_lse)
        sum_exp += exp_val
        o_val = tl.load(O_partial + o_idx + d_idx * stride_o_partial_n).to(tl.float32)
        out_acc += exp_val * o_val

    # Avoid div by zero when all splits empty
    sum_exp = tl.maximum(sum_exp, 1e-20)
    out_acc = out_acc / sum_exp
    lse_final = max_lse + tl.math.log2(sum_exp)

    # Store
    out_ptr = b_idx * stride_o_final_b + head_idx * stride_o_final_h + seq_idx * stride_o_final_m
    tl.store(O_final + out_ptr + d_idx * stride_o_final_n, out_acc.to(O_final.dtype.element_ty))
    m_ptr = b_idx * stride_m_final_b + head_idx * stride_m_final_h + seq_idx * stride_m_final_m
    tl.store(M_final + m_ptr, lse_final)
# ──────────────────────────── SPARSE ADDITION END ─────────────────────────────




@triton.jit
def _attn_bwd_preprocess(O, DO,  #
                         Delta,  #
                         Z, H, Q_CTX,  #
                         BLOCK_M: tl.constexpr, HEAD_DIM: tl.constexpr  #
                         ):
    off_m = tl.program_id(0) * BLOCK_M + tl.arange(0, BLOCK_M)
    off_hz = tl.program_id(1)
    off_n = tl.arange(0, HEAD_DIM)
    # load
    o = tl.load(O + off_hz * HEAD_DIM * Q_CTX + off_m[:, None] * HEAD_DIM + off_n[None, :])
    do = tl.load(DO + off_hz * HEAD_DIM * Q_CTX + off_m[:, None] * HEAD_DIM + off_n[None,
:]).to(tl.float32)
    delta = tl.sum(o * do, axis=1)
    # write-back
    tl.store(Delta + off_hz * Q_CTX + off_m, delta)


# The main inner-loop logic for computing dK and dV.
@triton.jit
def _attn_bwd_dkdv(dk, dv,  #
                   Q, k, v, sm_scale,  #
                   DO,  #
                   M, D,  #
                   k2q_index, k2q_num, max_q_blks,
                   variable_block_sizes,
                   stride_q_tok, stride_d,  #
                   H, KV_CTX, BLOCK_M1: tl.constexpr,  #
                   BLOCK_N1: tl.constexpr,  #
                   HEAD_DIM: tl.constexpr,  #
                   start_n, start_m, num_steps):
    offs_m = start_m + tl.arange(0, BLOCK_M1)
    offs_n = start_n + tl.arange(0, BLOCK_N1)
    offs_k = tl.arange(0, HEAD_DIM)
    qT_ptrs = Q + offs_m[None, :] * stride_q_tok + offs_k[:, None] * stride_d
    do_ptrs = DO + offs_m[:, None] * stride_q_tok + offs_k[None, :] * stride_d
    tl.static_assert(BLOCK_N1 % BLOCK_M1 == 0)
    step_m = BLOCK_M1
    kv_blk   = tl.program_id(0)
    off_hz  = tl.program_id(2)
    b       = off_hz // H
    h       = off_hz %  H
    kv_tiles = KV_CTX // BLOCK_N1
    meta_base = ((b * H + h) * kv_tiles + kv_blk)

    q_blocks = tl.load(k2q_num  + meta_base)                 # int32
    q_ptr    = k2q_index + meta_base * max_q_blks           # ptr to list
    block_size = tl.load(variable_block_sizes + kv_blk)



    for blk_idx in range(q_blocks*2):
        block_sparse_offset = (tl.load(q_ptr + blk_idx//2).to(tl.int32)*2 + blk_idx%2) *step_m
        qT = tl.load(qT_ptrs + block_sparse_offset * stride_q_tok)
        # Load m before computing qk to reduce pipeline stall.
        offs_m = start_m + block_sparse_offset + tl.arange(0, BLOCK_M1)
        m = tl.load(M + offs_m)
        qkT = tl.dot(k, qT)
        pT = tl.math.exp2(qkT - m[None, :])
        mask = tl.arange(0, BLOCK_N1) < block_size
        pT = tl.where(mask[:, None], pT, 0.0)

        do = tl.load(do_ptrs + block_sparse_offset * stride_q_tok)
        # Compute dV.
        ppT = pT
        ppT = ppT.to(tl.bfloat16)
        dv += tl.dot(ppT, do)
        # D (= delta) is pre-divided by ds_scale.
        Di = tl.load(D + offs_m)
        # Compute dP and dS.
        dpT = tl.dot(v, tl.trans(do)).to(tl.float32)
        dsT = pT * (dpT - Di[None, :])
        dsT = dsT.to(tl.bfloat16)
        dk += tl.dot(dsT, tl.trans(qT))
        # Increment pointers.
    return dk, dv



# the main inner-loop logic for computing dQ
@triton.jit
def _attn_bwd_dq(dq, q, K, V,  #
                 do, m, D,
                 q2k_index, q2k_num, max_kv_blks,
                 variable_block_sizes,
                 stride_kv_tok, stride_d,  #
                 H, Q_CTX,  #
                 BLOCK_M2: tl.constexpr,  #
                 BLOCK_N2: tl.constexpr,  #
                 HEAD_DIM: tl.constexpr,
                 start_m, start_n, num_steps):
    offs_m = start_m + tl.arange(0, BLOCK_M2)
    offs_n = start_n + tl.arange(0, BLOCK_N2)
    offs_k = tl.arange(0, HEAD_DIM)
    kT_ptrs = K + offs_n[None, :] * stride_kv_tok + offs_k[:, None] * stride_d
    vT_ptrs = V + offs_n[None, :] * stride_kv_tok + offs_k[:, None] * stride_d
    Di = tl.load(D + offs_m)
    tl.static_assert(BLOCK_M2 % BLOCK_N2 == 0)
    step_n = BLOCK_N2

    q_blk   = tl.program_id(0)
    off_hz  = tl.program_id(2)
    b       = off_hz // H
    h       = off_hz %  H
    q_tiles = Q_CTX // BLOCK_M2
    meta_base = ((b * H + h) * q_tiles + q_blk)

    kv_blocks = tl.load(q2k_num  + meta_base)                 # int32
    kv_ptr    = q2k_index + meta_base * max_kv_blks           # ptr to list


    for blk_idx in range(kv_blocks*2):
        block_sparse_offset = (tl.load(kv_ptr + blk_idx//2).to(tl.int32)*2 + blk_idx%2) *step_n
* stride_kv_tok
        block_size = tl.load(variable_block_sizes + blk_idx//2) - (blk_idx%2) * step_n
        kT = tl.load(kT_ptrs + block_sparse_offset)
        vT = tl.load(vT_ptrs + block_sparse_offset)
        qk = tl.dot(q, kT)
        p = tl.math.exp2(qk - m)
        mask = tl.arange(0, BLOCK_N2) < block_size.to(tl.int32)
        p = tl.where(mask[None, :], p , 0.0)
        # Compute dP and dS.
        dp = tl.dot(do, vT).to(tl.float32)
        ds = p * (dp - Di[:, None])
        ds = ds.to(tl.bfloat16)
        # Compute dQ.
        # NOTE: We need to de-scale dq in the end, because kT was pre-scaled.
        dq += tl.dot(ds, tl.trans(kT))
        # Increment pointers.
    return dq
```

```
@triton.jit
def _attn_bwd(Q, K, V, sm_scale,  #
              DO,  #
              DQ, DK, DV,  #
              M, D,
              q2k_index, q2k_num, max_kv_blks,
              k2q_index, k2q_num, max_q_blks,
              variable_block_sizes,
              stride_qz, stride_qh, stride_qm, stride_qk,
              stride_kz, stride_kh, stride_kn, stride_kk,
              stride_vz, stride_vh, stride_vk, stride_vn,
              H, Q_CTX, KV_CTX,  #
              BLOCK_M1: tl.constexpr,  #
              BLOCK_N1: tl.constexpr,  #
              BLOCK_M2: tl.constexpr,  #
              BLOCK_N2: tl.constexpr,  #
              HEAD_DIM: tl.constexpr):
    LN2 = 0.6931471824645996  # = ln(2)

    bhid = tl.program_id(2)
    off_chz = (bhid * Q_CTX).to(tl.int64)
    adj_q = (stride_qh * (bhid % H) + stride_qz * (bhid // H)).to(tl.int64)
    adj_kv = (stride_kh * (bhid % H) + stride_kz * (bhid // H)).to(tl.int64)
    pid = tl.program_id(0)

    # offset pointers: Q/DO/DQ use adj_q, K/V/DK/DV use adj_kv (different seq lengths)
    Q += adj_q
    K += adj_kv
    V += adj_kv
    DO += adj_q
    DQ += adj_q
    DK += adj_kv
    DV += adj_kv
    M += off_chz
    D += off_chz

    offs_k = tl.arange(0, HEAD_DIM)

    start_n = pid * BLOCK_N1
    start_m = 0

    offs_n = start_n + tl.arange(0, BLOCK_N1)

    dv = tl.zeros([BLOCK_N1, HEAD_DIM], dtype=tl.float32)
    dk = tl.zeros([BLOCK_N1, HEAD_DIM], dtype=tl.float32)

    # load K and V: use stride_kn, stride_kk for K; stride_vk, stride_vn for V
    k = tl.load(K + offs_n[:, None] * stride_kn + offs_k[None, :] * stride_kk)
    v = tl.load(V + offs_n[:, None] * stride_vk + offs_k[None, :] * stride_vn)

    num_steps = Q_CTX // BLOCK_M1

    dk, dv = _attn_bwd_dkdv(  #
        dk, dv,  #
        Q, k, v, sm_scale,  #
        DO,  #
        M, D,  #
        k2q_index, k2q_num, max_q_blks,
        variable_block_sizes,
        stride_qm, stride_qk,  #
        H, KV_CTX,  #
        BLOCK_M1, BLOCK_N1, HEAD_DIM,  #
        start_n, start_m, num_steps  #
    )

    dv_ptrs = DV + offs_n[:, None] * stride_vk + offs_k[None, :] * stride_vn
    tl.store(dv_ptrs, dv)

    dk *= sm_scale
    dk_ptrs = DK + offs_n[:, None] * stride_kn + offs_k[None, :] * stride_kk
    tl.store(dk_ptrs, dk)

    # DQ: only when pid < Q_CTX // BLOCK_M2 (cross-attention: Q blocks may be fewer than KV blocks)
    if pid < (Q_CTX // BLOCK_M2):
        start_m = pid * BLOCK_M2
        end_n = 0

        offs_m = start_m + tl.arange(0, BLOCK_M2)

        q = tl.load(Q + offs_m[:, None] * stride_qm + offs_k[None, :] * stride_qk)
        dq = tl.zeros([BLOCK_M2, HEAD_DIM], dtype=tl.float32)
        do = tl.load(DO + offs_m[:, None] * stride_qm + offs_k[None, :] * stride_qk)

        m = tl.load(M + offs_m)
        m = m[:, None]

        num_steps = KV_CTX // BLOCK_N2
        dq = _attn_bwd_dq(dq, q, K, V,  #
                          do, m, D,  #
                          q2k_index, q2k_num, max_kv_blks,
                          variable_block_sizes,
                          stride_kn, stride_kk,  # stride_kv_tok, stride_d for K/V
                          H, Q_CTX,  #
                          BLOCK_M2, BLOCK_N2, HEAD_DIM,  #
                          start_m, end_n, num_steps  #
                          )
        dq_ptrs = DQ + offs_m[:, None] * stride_qm + offs_k[None, :] * stride_qk
        dq *= LN2
        tl.store(dq_ptrs, dq)




# ──────────────────────────── SPARSE ADDITION BEGIN ───────────────────────────
def triton_block_sparse_attn_forward(q, k, v, q2k_index, q2k_num, variable_block_sizes):
    """
    Block-sparse attention forward with Split-K.
    Split-K: when base_blocks < 108, split KV blocks across multiple programs for better parallelism.
    """
    B, H, Q_CTX, D = q.shape
    _, _, KV_CTX, _ = k.shape
    sm_scale = 1.0 / math.sqrt(D)
    max_kv_blks = q2k_index.shape[-1]
    assert Q_CTX % 64 == 0, f"Q_CTX must be a multiple of 64, but got {Q_CTX}"
    assert Q_CTX // 64 == q2k_num.shape[-1], f"shape mismatch, Q_CTX // 64 = {Q_CTX // 64}, q2k_num.shape[-1] = {q2k_num.shape[-1]}"
    assert KV_CTX % 64 == 0, f"KV_CTX must be a multiple of 64, but got {KV_CTX}"

    BLOCK_M = 64
    base_blocks = (Q_CTX // BLOCK_M) * H * B
    num_splits = 1
    if base_blocks < 108:
        num_splits = (108 + base_blocks - 1) // base_blocks
        if num_splits > 64:
            num_splits = 64

    if num_splits == 1:
        o = torch.empty_like(q)
        M = torch.empty((B, H, Q_CTX), dtype=torch.float32, device=q.device)
        grid = (triton.cdiv(Q_CTX, BLOCK_M), B * H, 1)
        _attn_fwd_sparse[grid](
            q, k, v, sm_scale,
            q2k_index, q2k_num, max_kv_blks,
            variable_block_sizes,
            M, o,
            q.stride(0), q.stride(1), q.stride(2), q.stride(3),
            k.stride(0), k.stride(1), k.stride(2), k.stride(3),
            v.stride(0), v.stride(1), v.stride(2), v.stride(3),
            o.stride(0), o.stride(1), o.stride(2), o.stride(3),
            B, H, Q_CTX, KV_CTX,
            HEAD_DIM=D, NUM_SPLITS=1
        )
        return o, M

    # Split-K path: partial results then reduce
    o_partial = torch.empty((B * num_splits, H, Q_CTX, D), dtype=q.dtype, device=q.device)
    M_partial = torch.empty((B * num_splits, H, Q_CTX), dtype=torch.float32, device=q.device)
    o = torch.empty_like(q)
    M = torch.empty((B, H, Q_CTX), dtype=torch.float32, device=q.device)

    grid = (triton.cdiv(Q_CTX, BLOCK_M), B * H, num_splits)
    _attn_fwd_sparse[grid](
        q, k, v, sm_scale,
        q2k_index, q2k_num, max_kv_blks,
        variable_block_sizes,
        M_partial, o_partial,
        q.stride(0), q.stride(1), q.stride(2), q.stride(3),
        k.stride(0), k.stride(1), k.stride(2), k.stride(3),
        v.stride(0), v.stride(1), v.stride(2), v.stride(3),
        o_partial.stride(0), o_partial.stride(1), o_partial.stride(2), o_partial.stride(3),
        B, H, Q_CTX, KV_CTX,
        HEAD_DIM=D, NUM_SPLITS=num_splits
    )

    # Reduce partial results
    reduce_grid = (Q_CTX, H, B)
    _attn_fwd_reduce[reduce_grid](
        o_partial, M_partial, o, M,
        batch=B, num_splits=num_splits, qo_heads=H, q_seq_len=Q_CTX,
        HEAD_DIM=D,
        stride_o_partial_b=o_partial.stride(0), stride_o_partial_h=o_partial.stride(1),
        stride_o_partial_m=o_partial.stride(2), stride_o_partial_n=o_partial.stride(3),
        stride_o_final_b=o.stride(0), stride_o_final_h=o.stride(1),
        stride_o_final_m=o.stride(2), stride_o_final_n=o.stride(3),
        stride_m_partial_b=M_partial.stride(0), stride_m_partial_h=M_partial.stride(1),
        stride_m_partial_m=M_partial.stride(2),
        stride_m_final_b=M.stride(0), stride_m_final_h=M.stride(1), stride_m_final_m=M.stride(2),
    )

    return o, M

def triton_block_sparse_attn_backward(do, q, k, v, o, M, q2k_index, q2k_num, k2q_index, k2q_num,
variable_block_sizes):
    assert do.is_contiguous()
    B, H, Q_CTX, D = q.shape
    _, _, KV_CTX, _ = k.shape
    sm_scale = 1.0 / math.sqrt(D)
    dq = torch.empty_like(q)
    dk = torch.empty_like(k)
    dv = torch.empty_like(v)
    BLOCK_M1, BLOCK_N1, BLOCK_M2, BLOCK_N2 = 32, 64, 64, 32
    RCP_LN2 = 1.4426950408889634  # = 1.0 / ln(2)
    arg_k = k * (sm_scale * RCP_LN2)
    PRE_BLOCK = 64
    assert Q_CTX % PRE_BLOCK == 0
    assert KV_CTX % BLOCK_N1 == 0
    pre_grid = (Q_CTX // PRE_BLOCK, B * H)
    delta = torch.empty_like(M)
    _attn_bwd_preprocess[pre_grid](
        o, do,  #
        delta,  #
        B, H, Q_CTX,  #
        BLOCK_M=PRE_BLOCK, HEAD_DIM=D  #
    )

    max_q_blks = k2q_index.shape[-1]
    max_kv_blks = q2k_index.shape[-1]

    grid = (KV_CTX // BLOCK_N1, 1, B * H)
    _attn_bwd[grid](
        q, arg_k, v, sm_scale, do, dq, dk, dv,  #
        M, delta,  #
        q2k_index, q2k_num, max_kv_blks,
        k2q_index, k2q_num, max_q_blks,
        variable_block_sizes,
        q.stride(0), q.stride(1), q.stride(2), q.stride(3),
        k.stride(0), k.stride(1), k.stride(2), k.stride(3),
        v.stride(0), v.stride(1), v.stride(2), v.stride(3),
        H, Q_CTX, KV_CTX,  #
        BLOCK_M1=BLOCK_M1, BLOCK_N1=BLOCK_N1,  #
        BLOCK_M2=BLOCK_M2, BLOCK_N2=BLOCK_N2,  #
        HEAD_DIM=D  #
    )
```

tk的实现：

```
1. 安装
bash build.sh
2. benchmark
kernels/run.sh
```

要接入框架的话可以让gpt写，只要build.sh安装后就可以通过python的方式import。

如果需要tk我可以接进去，不过triton应该更好接