

CUDA Kernel 面试背题笔记

本书整理自 LeetCUDA 项目，系统汇编了 CUDA 面试中最高频的 GPU Kernel 实现，涵盖从 Warp/Block 原语、Elementwise、Softmax、GEMM、FlashAttention 等面试核心知识体系。

每个 Kernel 配备详细的中文面试要点注释 (WHY + HOW)，并以递进式优化路线 (naive → tiling → vectorize → tensor core → warp specialized) 展示 GPU 性能调优思维。

30 个 Kernel · 9 个 Phase · 全中文注释

项目主页: <https://github.com/xlite-dev/LeetCUDA>

更新日期: 2026 年 7 月 24 日

```

1 #pragma once
2 // common.cuh: CUDA headers, macros, MMA/WGMMMA PTX, swizzle, TMA helpers
3
4 #include <cuda_runtime.h>
5 #include <float.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <math.h>
9 #include <cublas_v2.h>
10 #include <cuda.h>
11 #include <cuda/barrier>
12 #include <cuda/ptx>
13
14 #if defined(NOTES_V2_ENABLE_CUDNN)
15 #pragma nv_diag_suppress 128
16 #include < cudnn_frontend.h>
17 #pragma nv_diag_default 128
18 namespace fe = cudnn_frontend;
19 #endif
20
21 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS) && CUDART_VERSION < 13000
22 #error "NOTES_V2_ENABLE_TMA_MMA_WS requires CUDA Toolkit 13.0 or newer"
23 #endif
24
25 #define INT4(value) (reinterpret_cast<int4*>(&(value)))[0])
26 #define FLOAT4(value) (reinterpret_cast<float4*>(&(value)))[0])
27 #define HALF2(value) (reinterpret_cast<half2*>(&(value)))[0])
28
29 static constexpr int kWarpSize = 32;
30
31
32 // Phase 7b-1: MMA PTX 宏定义
33 // =====
34
35 // ---- gmem → smem: cp.async ----
36 // cp.async.commit_group / wait_group / wait_all 语义 (PTX ISA §9.7.9.25.3) :
37 // - commit_group: 将此前所有未提交的 cp.async 归入一个新的 async-group (per-thread) 。
38 // - wait_group N: 阻塞直到最多 N 个 async-group 尚未完成 (即 pending ≤ N) 。
39 //   N=0 → 等所有 group 完成。与 wgmma.wait_group 语义一致。
40 // - wait_all: 等价于 commit_group + wait_group 0。
41 #define CP_ASYNC_COMMIT_GROUP() asm volatile("cp.async.commit_group;\n" ::)
42 #define CP_ASYNC_WAIT_ALL() asm volatile("cp.async.wait_all;\n" ::)
43 #define CP_ASYNC_WAIT_GROUP(n) \
44     asm volatile("cp.async.wait_group %0;\n" :: "n"(n))
45
46 // 注意: cg 只支持 16 bytes, ca 支持 4/8/16 bytes
47 #define CP_ASYNC_CG(dst, src, bytes) \
48     asm volatile( \
49         "cp.async.cg.shared.global.L2::128B [%0], [%1], %2;\n" :: "r"(dst), \
50         "l"(src), "n"(bytes))
51
52 // ---- ldmatrix: smem → register (Tensor Core 专用) ----
53 // ldmatrix.sync.aligned.xN.m8n8.shared.b16
54 // 每次加载 8×8 的 half 矩阵片段到 1/2/4 条 32-bit 寄存器
55 // aligned: 要求 128-bit 对齐, trans: 转置加载
56 #define LDMATRIX_X4(R0, R1, R2, R3, addr) \
57     asm volatile( \
58         "ldmatrix.sync.aligned.x4.m8n8.shared.b16 {%0, %1, %2, %3}, [%4];\n" \
59         : "=r"(R0), "=r"(R1), "=r"(R2), "=r"(R3) \
60         : "r"(addr))
61
62 #define LDMATRIX_X2(R0, R1, addr) \

```

```

63     asm volatile("ldmatrix.sync.aligned.x2.m8n8.shared.b16 {%0, %1}, [%2];\n" \
64                 : "=r"(R0), "=r"(R1) \
65                 : "r"(addr)) \
66
67 // ldmatrix.x2.trans: 转置加载 (用于 NN 布局中需要 col-major B 矩阵的场景)
68 // FA 中 V[Bc,d] 为 row-major, 但 P@V 的 MMA 需要 col-major 的 B → 使用 trans
69 #define LDMATRIX_X2_T(R0, R1, addr) \
70     asm volatile( \
71         "ldmatrix.sync.aligned.x2.trans.m8n8.shared.b16 {%0, %1}, [%2];\n" \
72         : "=r"(R0), "=r"(R1) \
73         : "r"(addr)) \
74
75 // ---- mma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16 ----
76 // m16n8k16: M=16, N=8, K=16 (Ampere Tensor Core 的基本 tile)
77 // row.col: A 是 row-major, B 是 col-major
78 // f16.f16.f16.f16: A/B 是 f16, C/D 是 f16 (f32 累加版本用 f32.f16.f16.f32)
79 // 2 个输出寄存器 (RD0, RD1), 4 个 A 寄存器 + 2 个 B 寄存器
80 // C 矩阵大小 16x8=128 元素 = 128 个 half; 32 线程分担, 每线程 4 half = 2 个 uint32
81 #define HMMA16816(RD0, RD1, RA0, RA1, RA2, RA3, RB0, RB1, RC0, RC1) \
82     asm volatile( \
83         "mma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16 {%0, %1}, {%2, %3, " \
84         "%4, %5}, {%6, %7}, {%8, %9};\n" \
85         : "=r"(RD0), "=r"(RD1) \
86         : "r"(RA0), "r"(RA1), "r"(RA2), "r"(RA3), "r"(RB0), "r"(RB1), "r"(RC0), \
87         "r"(RC1)) \
88
89 // mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
90 // f32 accumulator variant: 4 output registers (RD0-RD3), A/B still f16.
91 // C matrix 16x8=128 elements, 32 threads × 4 f32 = 4 uint32 per thread.
92 #define HMMA16816F32(RD0, RD1, RD2, RD3, RA0, RA1, RA2, RA3, RB0, RB1, RC0, \
93                     RC1, RC2, RC3) \
94     asm volatile( \
95         "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 {%0, %1, %2, " \
96         "%3}, {%4, %5, %6, %7}, {%8, %9}, {%10, %11, %12, %13};\n" \
97         : "=r"(RD0), "=r"(RD1), "=r"(RD2), "=r"(RD3) \
98         : "r"(RA0), "r"(RA1), "r"(RA2), "r"(RA3), "r"(RB0), "r"(RB1), "r"(RC0), \
99         "r"(RC1), "r"(RC2), "r"(RC3)) \
100
101 // ---- MMA 辅助函数 ----
102 // div_ceil: 整数除法向上取整
103 #define HOST_DEVICE_INLINE __device__ __host__ inline
104 HOST_DEVICE_INLINE int div_ceil(int a, int b) {
105     return (a % b != 0) ? (a / b + 1) : (a / b);
106 }
107
108 // =====
109
110 // Phase 7b-3: HGEMM MMA Swizzle — m16n8k16 + multistage pipeline + TN 布局
111 // + XOR swizzle 消除 smem bank conflict (统一循环版)
112 // =====
113 // 面试要点 (Swizzle 原理 — smem Bank Conflict 消除):
114 // - 问题: ldmatrix 以 8x8 片段 (m8n8) 从 smem 加载 16x16 的矩阵。同一 warp 中
115 //   不同 lane 访问的地址在 bank 上产生冲突 → 串行化 (most common: 2/4-way)。
116 // - XOR swizzle 方案: 对列地址做 `((j>>3) ^ (i>>2)) % 2 << 3` 的 XOR 置换,
117 //   将连续 4 行的 bank 映射打散。每 4 行为一组的 pattern 交替:
118 //   rows 0~3: 列 0→地址偏移 0, 列 8→地址偏移 8
119 //   rows 4~7: 列 0→地址偏移 8, 列 8→地址偏移 0 (XOR 翻转)
120 //   rows 8~11: repeat rows 0~3 pattern
121 //   rows 12~15: repeat rows 4~7 pattern
122 // - 效果: 原来 n-way 的 bank conflict 降低到 1-way (fully conflict-free)。
123 // - 优势: 无需 smem PAD (不浪费空间), 原理上完全消除特定 pattern 的 bank conflict。
124 // - 局限性: 要求 kColStride ≤ 16 (即 BK ≤ 16), kStep ∈ {4,8}。
125 //   对 MMA m16n8k16 的 BK=16 而言刚好满足。

```

```

126 //
127 // 参考: LeetCUDA/kernels/hgemm/mma/swizzle/hgemm_mma_stage_tn_swizzle.cu
128 // 参考: https://zhuanlan.zhihu.com/p/4746910252
129
130 // ---- Swizzle 辅助函数 ----
131
132 // i: row index; j: col index.
133 // Returns the chunk-level (8 fp16 = 16 B per chunk) column swizzle, i.e.
134 // 0 / 8 / 16 / ... that the caller adds to the chunk-aligned base column.
135 // The within-chunk offset (`j & 7`) is preserved by the caller.
136 //
137 // kColStride matches the equivalent CuTe ``Swizzle<B, M, S>`` swizzle
138 // hardware mode used by TMA bulk-tensor descriptors:
139 // * kColStride = 16 fp16 = 32 B/row -> Swizzle<1, 4, 3> = SWIZZLE_32B
140 // * kColStride = 32 fp16 = 64 B/row -> Swizzle<2, 4, 3> = SWIZZLE_64B
141 // * kColStride = 64 fp16 = 128 B/row -> Swizzle<3, 4, 3> = SWIZZLE_128B
142 // The XOR mask is derived from the byte-address bit positions used by the
143 // underlying CuTe swizzle pattern: bits {4..(4+B-1)} XOR bits
144 // {7..(7+B-1)} of the absolute byte address. Translating to (i, j) with
145 // row stride ``kColStride * 2`` bytes:
146 // * 32B (B=1) : (j>>3) ^ (i>>2) [chunk in {0, 1}]
147 // * 64B (B=2) : chunk(2b) ^ {(i>>1)&1, (i>>2)&1} [{0..3}]
148 // * 128B (B=3) : chunk(3b) ^ {i&1, (i>>1)&1, (i>>2)&1} [{0..7}]
149 // source: LeetCUDA/ffpa-attn/cuffpa/swizzle.cuh
150 //
151 // v1/v2 实现 (由编译宏 NOTES_V2_ENABLE_SWIZZLE_V2 门控):
152 // swizzle_v1_impl : 原手写 XOR 分支 (permuted<kColStride,8>), 按列宽展开.
153 // swizzle_v2_impl : 用本地 SwizzleBMS<B,M,S> 镜像 cute::Swizzle<B,M,S>::apply
154 // 级公式统一表达 (16/32/64); kColStride=8 回退 v1.
155 // swizzle : 公开派发器, 宏门控, 接口与原 swizzle<kColStride> 一致.
156 // v2 不是新算法, 而是把 v1 的手写展开重新表达为 cute 的统一位置换公式,
157 // 便于脱离 cute 框架理解 swizzle 本质. v1/v2 bit-exact 等价.
158 template <const int kColStride = 16, const int kStep = 8>
159 static __host__ __device__ __forceinline__ int permuted(int i, int j) {
160     static_assert(kColStride == 16 || kColStride == 32 || kColStride == 64 ||
161         kColStride == 8,
162         "kColStride must be one of {8, 16, 32, 64} (matches "
163         "SWIZZLE_32B/64B/128B + the "
164         "kStep=4 narrow legacy case).");
165     static_assert(kStep == 4 || kStep == 8, "kStep must be 8 or 4.");
166     static_assert(kColStride % kStep == 0,
167         "kColStride must be multiple of kStep.");
168     if constexpr (kStep == 4) {
169         static_assert(kColStride <= 16, "kStep=4 only supports (kColStride <= 16).");
170         return (((j >> 2) ^ (i >> 2)) % (kColStride >> 2)) << 2;
171     } else if constexpr (kColStride == 16) {
172         return (((j >> 3) ^ (i >> 2)) & 1) << 3; // SWIZZLE_32B
173     } else if constexpr (kColStride == 32) {
174         const int chunk = (j >> 3) & 3;
175         const int xor_mask = ((i >> 1) & 1) | (((i >> 2) & 1) << 1);
176         return (chunk ^ xor_mask) << 3; // SWIZZLE_64B
177     } else { // kColStride == 64
178         const int chunk = (j >> 3) & 7;
179         const int xor_mask =
180             (i & 1) | (((i >> 1) & 1) << 1) | (((i >> 2) & 1) << 2);
181         return (chunk ^ xor_mask) << 3; // SWIZZLE_128B
182     }
183 }
184
185 // Manually SMEM swizzling for bank conflict free.
186 // -----
187 // [INFO] Assert smem store layout col_stride <= 16, prefer 16. |
188 // [INFO] For logical_col_stride > 16, we have to permute the |

```

```

189 // [INFO] smem store layout using col major ZigZag method: |
190 // [INFO] e.g, --> Q smem logical layout [Br][64]. |
191 // [INFO] --> col major ZigZag permuted --> |
192 // [INFO] --> Q smem store layout [4][Br][16]. |
193 // ----- |
194 // ----- |
195 // -----swizzle layout----- |
196 // -----logical col 0~64, step 8----- |
197 // -----smem col 0~16, step 8----- |
198 // ----- |
199 // |bank |b 0~3 |b 4~7 |b 0~3 |b 4~7 |b 0~3 |b 4~7 |b 0~3 |b 4~7 | |
200 // |row 0 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | |
201 // |bank |b 8~11|b12~15|b 8~11|b12~15|b 8~11|b12~15|b 8~11|b12~15| |
202 // |row 1 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | |
203 // |bank |b16~19|b20~23|b16~19|b20~23|b16~19|b20~23|b16~19|b20~23| |
204 // |row 2 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | |
205 // |bank |b24~27|b28~31|b24~27|b28~31|b24~27|b28~31|b24~27|b28~31| |
206 // |row 3 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | |
207 // ----- |
208 // |bank |b 0~3 |b 4~7 |b 0~3 |b 4~7 |b 0~3 |b 4~7 |b 0~3 |b 4~7 | |
209 // |row 4 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | |
210 // |bank |b 8~11|b12~15|b 8~11|b12~15|b 8~11|b12~15|b 8~11|b12~15| |
211 // |row 5 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | |
212 // |bank |b16~19|b20~23|b16~19|b20~23|b16~19|b20~23|b16~19|b20~23| |
213 // |row 6 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | |
214 // |bank |b24~27|b28~31|b24~27|b28~31|b24~27|b28~31|b24~27|b28~31| |
215 // |row 7 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | |
216 // ----- |
217 // |bank |b 0~3 |b 4~7 |b 0~3 |b 4~7 |b 0~3 |b 4~7 |b 0~3 |b 4~7 | |
218 // |row 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | |
219 // |bank |b 8~11|b12~15|b 8~11|b12~15|b 8~11|b12~15|b 8~11|b12~15| |
220 // |row 9 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | |
221 // |bank |b16~19|b20~23|b16~19|b20~23|b16~19|b20~23|b16~19|b20~23| |
222 // |row 10| 0 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | |
223 // |bank |b24~27|b28~31|b24~27|b28~31|b24~27|b28~31|b24~27|b28~31| |
224 // |row 11| 0 | 8 | 0 | 8 | 0 | 8 | 0 | 8 | |
225 // ----- |
226 // |bank |b 0~3 |b 4~7 |b 0~3 |b 4~7 |b 0~3 |b 4~7 |b 0~3 |b 4~7 | |
227 // |row 12| 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | |
228 // |bank |b 8~11|b12~15|b 8~11|b12~15|b 8~11|b12~15|b 8~11|b12~15| |
229 // |row 13| 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | |
230 // |bank |b16~19|b20~23|b16~19|b20~23|b16~19|b20~23|b16~19|b20~23| |
231 // |row 14| 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | |
232 // |bank |b24~27|b28~31|b24~27|b28~31|b24~27|b28~31|b24~27|b28~31| |
233 // |row 15| 8 | 0 | 8 | 0 | 8 | 0 | 8 | 0 | |
234 // ----- |
235 template <const int kColStride = 16>
236 static __host__ __device__ __forceinline__ int swizzle_v1_impl(int i, int j) {
237     return permuted<kColStride, 8>(i, j);
238 }
239
240 // Swizzle v2: cute::Swizzle<B,M,S> 位级镜像（脱离 cute 框架独立理解）
241 //
242 // v2 双重目的：
243 // (1) 探索优化 v1 的可能 -- v1 按 kColStride 手写 XOR 分支展开，已被 nvcc
244 // strength-reduce 到极简位运算；v2 用 cute 的统一位置换公式重新表达，
245 // 预期中性，但统一表达式可能更利于合并到地址算术（bonus）。
246 // (2) 把 cute Swizzle 的核心位逻辑从 cute 框架提取出来独立理解 -- 让 notes-v2
247 // 不依赖 cute 头即可展示 Swizzle<B,M,S> 的位级语义，便于读者脱离 cute
248 // 模板体系直接掌握 swizzle 原理。
249 //
250 // 注释叙述参考：上方 hgemm_mma_stages_tn_cute 处的 Swizzle<B,M,S> 原理解释
251 // （二维逻辑空间 + 列置换），此处适配 v2 的 (B,M,S)=(1/2/3, 4, 3) 语境。

```

```

252 //
253 // Swizzle<B,M,S> 的核心不是改变逻辑 Tensor 的 shape, 而是把一维 byte offset
254 // 重新解释为一个二维逻辑空间, 再把二维坐标映射回 bank-conflict-free 的物理
255 // offset:
256 // 1. 连续的 2^M 个元素组成二维空间中的一个基本元素 (元素宽度);
257 // 2. 连续的 2^S 个基本元素组成一行 (列数);
258 // 3. 二维空间包含 2^B 行 (行数, 这些行的 chunk 互相置换);
259 // 4. 对二维坐标做列置换: icol' = irow ^ icol;
260 // 5. 保留基本元素内部的低 M 位, 再将置换后的二维坐标编码回 offset.
261 //
262 // 位级实现 (与 cutlass/include/cute/swizzle.hpp 一致):
263 // bit 布局: 0bxxxxxxxxxxxxxxxxYYxxxxxxZZxxxx
264 //           ^--^ MBase: 保留不参与置换的低位数
265 //           ^-^ ^-^ BBits: XOR 掩码位数
266 //           ^-----^ SShift: YYY 相对 ZZZ 的位移
267 // apply(off) = off ^ shiftr(off & yyy_msk, S)
268 // yyy_msk    = ((1<<B)-1) << (M + max(0,S)) // YYY 掩码位置
269 // zzz_msk    = ((1<<B)-1) << (M - min(0,S)) // ZZZ 掩码位置 (文档用)
270 //
271 // 模板参数物理/语义含义:
272 // B (BBits) : 二维逻辑空间的行数位 = 2^B 行, 即 swizzle 周期覆盖 2^B 个 chunk
273 //             (这些 chunk 互相置换); 同时是 TMA 硬件 swizzle 模式
274 //             SWIZZLE_{32,64,128}B 的位数 (B=1/2/3).
275 // M (MBase) : 基本元素宽度位 = 2^M 个连续元素组成一个基本元素. 对 fp16 +
276 //             8-element chunk, 基本元素 = 16B = 2^4, 故 M=4; 地址 bit[0..M-1]
277 //             在 apply 中保持不变 (保留).
278 // S (SShift) : 二维空间每行列数位 = 2^S 个基本元素组成一行; 同时是 YYY 掩码
279 //             相对 ZZZ 掩码的位移. S>0 表示 YYY 在更高位, 向右移 S 位到 ZZZ
280 //             位置后 XOR. 本 notes-v2 三模式 S 恒为 3 (YYY 在 bit[7..7+B-1],
281 //             即行索引 i 的低 B 位; ZZZ 在 bit[4..4+B-1], 即列 chunk 索引).
282 //
283 // 完整 swizzle 周期覆盖 2^(M+S+B) 个元素 (fp16 下 2^(M+S+B+1) 字节):
284 // (B,M,S)=(1,4,3): 256 elem / 512B (SWIZZLE_32B)
285 // (B,M,S)=(2,4,3): 512 elem / 1024B (SWIZZLE_64B)
286 // (B,M,S)=(3,4,3): 1024 elem / 2048B (SWIZZLE_128B)
287 template <int B, int M, int S>
288 struct SwizzleBMS {
289     static_assert(M >= 0, "MBase must be non-negative.");
290     static_assert(B > 0, "BBits must be positive.");
291     static_assert((S > 0 ? S : -S) >= B,
292         "abs(SShift) must be >= BBits (cute::Swizzle constraint).");
293
294     static constexpr int bit_msk = (1 << B) - 1;
295     static constexpr int yyy_msk = bit_msk << (M + (S > 0 ? S : 0));
296     static constexpr int zzz_msk = bit_msk << (M - (S < 0 ? S : 0));
297
298     // apply: 对 byte offset 做 XOR 位置换, ZZZ ^= (YYY >> S).
299     static __host__ __device__ __forceinline__ int apply(int offset) {
300         if constexpr (S >= 0) {
301             return offset ^ ((offset & yyy_msk) >> S);
302         } else {
303             return offset ^ ((offset & yyy_msk) << (-S));
304         }
305     }
306 };
307
308 // swizzle_v2_impl: 用 SwizzleBMS<B,M,S>::apply 计算 chunk 级列 swizzle.
309 //
310 // 流程: (i,j) -> byte offset off=(i*kColStride+j)*sizeof(half) -> SwizzleBMS
311 //         位级置换 -> 取 [M, M+B) 位作为物理 chunk 索引 -> 乘 kStep 还原为列偏移.
312 //
313 // kColStride -> (B,M,S) 映射表 (对应 TMA 硬件 swizzle 模式):
314 // kColStride=16 fp16=32B/row -> (B,M,S)=(1,4,3) = SWIZZLE_32B

```

```

315 // kColStride=32 fp16=64B/row -> (B,M,S)=(2,4,3) = SWIZZLE_64B
316 // kColStride=64 fp16=128B/row -> (B,M,S)=(3,4,3) = SWIZZLE_128B
317 //
318 // kColStride=8 (kStep=4 legacy): 无对应 cute 标准 TMA swizzle 模式, 回退到
319 // swizzle_v1_impl. 该路径当前无任何 kernel 使用, 回退保证接口契约零变更.
320 //
321 // 等价性: 对 (16,32,64), v2 输出与 v1 bit-exact 相同 (见 test_swizzle_equiv).
322 // 证明: off 的 bit[4..4+B-1] 恰是 (j>>3) 低 B 位 (因 i*kColStride 在这些位
323 // 为 0); bit[7..7+B-1] 恰是 i 低 B 位 (因 j<kColStride 不贡献高位). apply
324 // 把 YYY(bit[7..]) XOR 到 ZZZ(bit[4..]), 取 [M,M+B)=bit[4..4+B-1] 即得
325 // (j>>3)^i 的低 B 位 = v1 的 chunk^xor_mask.
326 template <const int kColStride = 16>
327 static __host__ __device__ __forceinline__ int swizzle_v2_impl(int i, int j) {
328     if constexpr (kColStride == 8) {
329         return swizzle_v1_impl<kColStride>(i, j);
330     } else {
331         static_assert(kColStride == 16 || kColStride == 32 || kColStride == 64,
332             "v2 supports kColStride in {16, 32, 64} (cute TMA swizzle); "
333             "8 falls back to v1.");
334         constexpr int B = (kColStride == 16) ? 1 : (kColStride == 32) ? 2 : 3;
335         constexpr int M = 4;
336         constexpr int S = 3;
337         constexpr int kStep = 8;
338         const int off = (i * kColStride + j) * (int)sizeof(half);
339         const int sw = SwizzleBMS<B, M, S>::apply(off);
340         return ((sw >> M) & ((1 << B) - 1)) * kStep;
341     }
342 }
343
344 // swizzle: 公开派发器, 由编译宏 NOTES_V2_ENABLE_SWIZZLE_V2 门控 v1/v2.
345 // 未定义宏 (默认) -> swizzle_v1_impl (原手写 XOR 分支, 历史默认路径).
346 // 定义宏 -> swizzle_v2_impl (cute Swizzle<B,M,S> 位级镜像).
347 // 接口与原 swizzle<kColStride>(i,j) 完全一致, 所有 kernel 调用点无需改动.
348 // v1/v2 等价性由 host 端 test_swizzle_equiv (--swizzle-eq-check) 验证.
349
350 #if defined(NOTES_V2_ENABLE_WGMMA) || defined(NOTES_V2_ENABLE_TMA_MMA_WS)
351 // TMA/mbarrier helpers.
352 //
353 // Two implementations gated by NOTES_V2_FORCE_INLINE_ASYNC_PROXY:
354 // - Defined (default): raw `asm volatile` PTX. ptxas keeps setmaxnreg
355 //   because raw asm carries no source annotation that ptxas would treat as
356 //   an "extern call" boundary (warning C7506 otherwise drops the hint).
357 // - Undef'd: cuda::ptx:: / cuda::device:: C++ wrappers. Cleaner API but
358 //   ptxas drops setmaxnreg with C7506 even though the wrappers inline to
359 //   identical PTX; useful for comparing behavior / debugging.
360 //
361 // cuda::barrier (init/arrive/wait) is shared by both paths: it inlines to raw
362 // mbarrier PTX and does not trigger C7506, so it needs no gating.
363 //
364 // Mirrors the raw-PTX style of tmp/LeetGPU/CUDA/22_GEMM/sm90_wgmma_tma_ws_pingpong.cu.
365 #if defined(NOTES_V2_FORCE_INLINE_ASYNC_PROXY)
366
367 static __device__ __forceinline__ uint32_t
368 cast_smem_ptr_to_uint(void const *ptr) {
369     return static_cast<uint32_t>(__cvta_generic_to_shared(ptr));
370 }
371
372 static __device__ __forceinline__ void tma_fence_proxy_async_shared_cta() {
373     asm volatile("fence.proxy.async.shared::cta;\n" ::: "memory");
374 }
375
376 // cp.async.bulk.tensor.2d: 2D TMA copy from global to shared::cluster.
377 // [dst] = shared memory destination (smem addr, 32-bit)

```

```

378 // [desc, {c0, c1}] = CUDensorMap + (minor_coord, major_coord) coords
379 // [mbar]           = mbarrier in shared memory (smem addr, 32-bit)
380 // mbarrier::complete_tx::bytes: barrier flips phase when TMA bytes land.
381 // L2::cache_hint omitted (EVICT_NORMAL); matches the cuda::ptx default.
382 static __device__ __forceinline__ void tma_load_2d(
383     void *dst, const CUDensorMap *tensor_map, int minor_coord, int major_coord,
384     cuda::barrier<cuda::thread_scope_block> &barrier) {
385     uint64_t gmem_int_desc = reinterpret_cast<uint64_t>(tensor_map);
386     uint32_t smem_int_ptr = cast_smem_ptr_to_uint(dst);
387     uint32_t smem_int_mbar =
388         cast_smem_ptr_to_uint(reinterpret_cast<uint64_t *>(&barrier));
389     asm volatile(
390         "cp.async.bulk.tensor.2d.shared::cluster.global.mbarrier::complete_tx::bytes"
391         " [%0], [%1, {%3, %4}], [%2];"
392         :
393         : "r"(smem_int_ptr), "l"(gmem_int_desc), "r"(smem_int_mbar),
394           "r"(minor_coord), "r"(major_coord)
395         : "memory");
396 }
397
398 // mbarrier.arrive.expect_tx: register one arrival on the mbarrier and declare
399 // that `bytes` of async copy traffic is expected before the phase flips.
400 // Equivalent to cuda::ptx::mbarrier_arrive_expect_tx(release, cta, shared).
401 static __device__ __forceinline__ void tma_arrive_expect_tx(
402     cuda::barrier<cuda::thread_scope_block> &barrier, uint32_t bytes) {
403     uint32_t smem_int_mbar =
404         cast_smem_ptr_to_uint(reinterpret_cast<uint64_t *>(&barrier));
405     asm volatile(
406         "mbarrier.arrive.expect_tx.shared::cta.b64 _, [%0], %1;\n"
407         :
408         : "r"(smem_int_mbar), "r"(bytes)
409         : "memory");
410 }
411
412 #else // !NOTES_V2_FORCE_INLINE_ASYNC_PROXY: use cuda::ptx / cuda::device wrappers
413
414 static __device__ __forceinline__ void tma_fence_proxy_async_shared_cta() {
415     #if CUDART_VERSION >= 13020
416         cuda::ptx::fence_proxy_async(cuda::ptx::space_shared);
417     #else
418         cuda::device::experimental::fence_proxy_async_shared_cta();
419     #endif
420 }
421
422 static __device__ __forceinline__ void tma_load_2d(
423     void *dst, const CUDensorMap *tensor_map, int minor_coord, int major_coord,
424     cuda::barrier<cuda::thread_scope_block> &barrier) {
425     #if CUDART_VERSION >= 13020
426         const int32_t coords[] {minor_coord, major_coord};
427         auto *barrier_handle = cuda::device::barrier_native_handle(barrier);
428         cuda::ptx::cp_async_bulk_tensor(cuda::ptx::space_cluster,
429                                         cuda::ptx::space_global, dst, tensor_map,
430                                         coords, barrier_handle);
431     #else
432         cuda::device::experimental::cp_async_bulk_tensor_2d_global_to_shared(
433             dst, tensor_map, minor_coord, major_coord, barrier);
434     #endif
435 }
436
437 static __device__ __forceinline__ void tma_arrive_expect_tx(
438     cuda::barrier<cuda::thread_scope_block> &barrier, uint32_t bytes) {
439     #if CUDART_VERSION >= 13020
440         auto *barrier_handle = cuda::device::barrier_native_handle(barrier);

```

```

441 [[maybe_unused]] auto token = cuda::ptx::mbarrier_arrive_expect_tx(
442     cuda::ptx::sem_release, cuda::ptx::scope_cta, cuda::ptx::space_shared,
443     barrier_handle, bytes);
444 #else
445 [[maybe_unused]] auto token =
446     cuda::device::barrier_arrive_tx(barrier, 1, bytes);
447 #endif
448 }
449
450 #endif // NOTES_V2_FORCE_INLINE_ASYNC_PROXY
451
452 // Warpgroup-level register rebalancing for warp specialization.
453 // setmaxnreg.dec releases registers (producer TMA path needs few),
454 // setmaxnreg.inc requests registers (consumer MMA path needs many).
455 // Both require all warps in a warpgroup to execute the same instruction,
456 // and require __launch_bounds__(N, 1) so the compiler permits up to 256
457 // regs/warp (otherwise the hint may have no effect). Supported on sm_90a/sm_120a.
458 // __forceinline__ is mandatory: without it ptexas drops setmaxnreg with
459 // warning C7506 "ignored to maintain compatibility into 'extern' call",
460 // because a non-inlined call boundary forces a fixed register convention.
461 //
462 // On sm_120a (Blackwell, CUDA 13.2) ptexas drops setmaxnreg with C7506 even
463 // when the PTX is fully inlined (no call.uni), because ptexas treats
464 // cp.async.bulk.tensor (TMA) usage as an implicit extern-call boundary.
465 // sm_90a (Hopper) is unaffected. Gate the *call sites* with
466 // NOTES_V2_ENABLE_SETMAXNREGS so sm_120a builds stay warning-free and avoid
467 // the register-allocation side effects of __launch_bounds__(N,1) until ptexas
468 // is fixed. The function templates themselves are always defined.
469 #if defined(NOTES_V2_ENABLE_SETMAXNREGS)
470     #define NOTES_V2_REG_DEALLOC(N) warpgroup_reg_dealloc<N>()
471     #define NOTES_V2_REG_ALLOC(N) warpgroup_reg_alloc<N>()
472 #else
473     #define NOTES_V2_REG_DEALLOC(N) ((void)0)
474     #define NOTES_V2_REG_ALLOC(N) ((void)0)
475 #endif
476
477 template <uint32_t kNumRegs>
478 __device__ __forceinline__ void warpgroup_reg_dealloc()
479 {
480     asm volatile("setmaxnreg.dec.sync.aligned.u32 %0;\n" : : "n"(kNumRegs));
481 }
482
483 template <uint32_t kNumRegs>
484 __device__ __forceinline__ void warpgroup_reg_alloc()
485 {
486     asm volatile("setmaxnreg.inc.sync.aligned.u32 %0;\n" : : "n"(kNumRegs));
487 }
488 #endif
489
490 #if defined(NOTES_V2_ENABLE_WGMMA)
491 // =====
492 // Phase 7d: HGEMM WGMMA — m64n128k16 + TMA + Warp Specialization (Hopper)
493 // =====
494 // 面试要点 (WGMMA vs MMA 对比) :
495 //   - MMA: warp 级 (32 threads) , 同步执行
496 //   - WGMMA: warpgroup 级 (128 threads = 4 warps) , 异步执行 (fire-and-forget)
497 //   - m64n128k16: M=64, N=128, K=16 → 一次处理 64×128×16=131K 个乘加 (MMA m16n8k16的 4x16=64倍)
498 //   - TMA (Tensor Memory Accelerator): 硬件 DMA, 2D 寻址, 零寄存器开销
499 //   - Warp Specialization: Producer 做 TMA, Consumer 做 WGMMA, 通过 barrier 同步
500 //   - 128B swizzle: shared memory 的 128B swizzle 模式, 避免 bank conflict
501
502 // ---- WGMMA 辅助函数 ----
503 // wgmma.commit_group / wait_group 语义 (PTX ISA §9.7.15.7) :

```

```

504 // - commit_group: 将此前所有未提交的 wgmma.mma_async 归入一个新的 wgmma-group
505 //   (per-warpgroup, 故需 .sync.aligned 确保所有线程同步执行)。
506 // - wait_group N: 阻塞直到最多 N 个 wgmma-group 尚未完成 (pending ≤ N)。
507 //   N=0 → 等所有 group 完成。* 与 cp.async.wait_group 语义一致 (PTX ISA §9.7.9.25.3.3),
508 //   都是 "wait until only N or fewer groups are pending"。
509 #define WGMMA_FENCE() asm volatile("wgmma.fence.sync.aligned;\n" ::: "memory")
510 #define WGMMA_COMMIT_GROUP() \
511     asm volatile("wgmma.commit_group.sync.aligned;\n" ::: "memory")
512 #define WGMMA_WAIT_GROUP(n) \
513     asm volatile("wgmma.wait_group.sync.aligned %0;\n" :: "n"(n) : "memory")
514
515 // SMEM_DESC_ENCODE: 将 byte 偏移编码为 WGMMA descriptor 位域中的 14-bit 值。
516 // 编码规则 (PTX ISA §9.7.15.5.1.2.2) :
517 //   encoded = (x & 0x3FFFF) >> 4
518 // 其中 0x3FFFF = 2^18 - 1 = 262143, 只保留低 18 bit。
519 // >>4 是因为 shared memory 地址/偏移必须 16B 对齐 (2^4), 低 4 bit 恒为 0,
520 // 可省略以扩大可表示范围。
521 // 解码: decoded = encoded << 4 (回到原始 byte 值)。
522 #define SMEM_DESC_ENCODE(x) (((uint64_t)(x)) & 0x3FFFF) >> 0x4
523
524 // make_smem_desc: 构造 WGMMA (warpgroup MMA) 的 64-bit shared memory 矩阵描述符。
525 // 硬件 WGMMA (如 wgmma.mma_async.m64n128k16) 需要 descA/descB 两个 64-bit 描述符,
526 // 来定位 shared memory 中的 A/B tile 并告知 swizzle 模式。
527 //
528 // 64-bit descriptor 位域布局 (参见 PTX ISA §9.7.15.5.1.2.2 & CUTLASS GmmaDescriptor) :
529 // -----
530 // | 位域           | 范围       | 大小 | 含义
531 // |-----|-----|-----|-----|
532 // | start_address   | [0, 14)    | 14   | smem 基址编码
533 // | (unused)        | [14, 16)   | 2    | -
534 // | leading_byte_offset | [16, 30)   | 14   | 主要维度 (K) 的字节步长; swizzle 下硬件未使用 (assumed=1)
535 // | (unused)        | [30, 32)   | 2    | -
536 // | stride_byte_offset | [32, 46)   | 14   | 跨越维度 (M/N) 的字节步长: K-Major 下为 8-row stripe 间距
537 // | (unused)        | [46, 49)   | 3    | -
538 // | base_offset     | [49, 52)   | 3    | swizzle pattern 偏移
539 // | (unused)        | [52, 62)   | 10   | -
540 // | layout_type     | [62, 64)   | 2    | swizzle 模式
541 // -----
542 // layout_type: 0=None, 1=128B swizzle, 2=64B swizzle, 3=32B swizzle
543 //
544 // K-Major + 128B swizzle + half(16-bit) 的几何推导 (PTX ISA §9.7.15.5.1.2) :
545 // - 128B swizzle atom 在 128-bit 单位下为 8×8
546 // - 归一化到 half(2B) 后: atom 覆盖 8×(128/16)=64 个 K 方向元素 × 8 个 M 方向元素
547 // - 即每个 atom = 64(K) × 8(M) 个 half = 64×8×2 = 1024 bytes
548 // - 这是 stride_byte_offset=1024 的来源
549 //
550 // - leading_byte_offset 在 K-Major swizzle 下 unused (hardware assumes 1),
551 //   此处填 16 仅为占位, 编码后为 1。
552 //
553 // - base_offset=0 (bits [49,52) 未显式置位), 表示 smem ptr 已 1024B 对齐。
554 //   若未对齐需计算 (pattern_start_addr >> 7) & 0x7。
555 //
556 // 参考: CUTLASS GmmaDescriptor (cute/arch/mma_sm90_desc.hpp)
557 // 注: descA/descB 共用此函数; 对 B 而言物理存储为 [BN, BK], 详见 WgmmaSMem 注释。
558 __device__ inline uint64_t make_smem_desc(half *ptr) {
559     // __cvta_generic_to_shared: 将通用地址空间中的指针转换为 shared memory 地址
560     // (一个 32-bit 的 smem byte 偏移量, 相对于当前 CTA 的 shared memory 基址)。
561     uint32_t addr = static_cast<uint32_t>(__cvta_generic_to_shared(ptr));
562
563     // 从全零开始: 所有位域初始化为 0。
564     uint64_t desc = 0x0000000000000000;
565
566     // — bits [0, 14): start_address —

```

```

567 // SMEM_DESC_ENCODE(addr) 将 addr 右移 4 位 (丢弃低 4 个 0 bit) ,
568 // 只保留 14 位。硬件解码时左移 4 位恢复完整的 16B-aligned 地址。
569 // 可寻址范围:  $2^{(14+4)} = 2^{18} = 256K$  个 half = 512 KB 共享内存。
570 desc |= SMEM_DESC_ENCODE(addr);
571
572 // — bits [16, 30): leading_byte_offset —
573 // 原始值 16 (字节), 编码后为  $(16 \& 0x3FFFF) \gg 4 = 1$ 。
574 // << 16 将编码值放到 bits [16, 30)。
575 // K-Major + 128B swizzle 下该字段硬件未使用 (assumed to be 1, 参见 PTX ISA §9.7.15.5.1.2.1.1) ,
576 // 此处填 16 仅为占位, 使编码值为 1 满足硬件预期。
577 // 若为 MN-Major 或 INTERLEAVE 布局, LBO 有实际含义:
578 // 对 INTERLEAVE: 同一 8×2 brick 内第一列到第二列的字节偏移。
579 // 对 MN-Major swizzle: 偏移从首 (swizzle-byte-size/16) 行到下一组行。
580 desc |= SMEM_DESC_ENCODE((uint64_t)16) << 16;
581
582 // — bits [32, 46): stride_byte_offset —
583 // 原始值 1024 (字节), 编码后为  $(1024 \& 0x3FFFF) \gg 4 = 64$ 。
584 // << 32 将编码值放到 bits [32, 46)。
585 // K-Major 下定义为"从首 8 行到次 8 行的字节偏移" (PTX ISA §9.7.15.5.1.2.1.2) 。
586 // 1024 的来源 (K-Major + 128B swizzle + half) :
587 // 一个 128B swizzle atom 覆盖  $64(K) \times 8(M)$  个 half。
588 // 从 1 个 8-row stripe 到下一个 stripe 需要跨越  $8 \times 64 \times 2 = 1024$  bytes。
589 // 若为 MN-Major: SB0 是从首列到下一列的偏移 (8 列一组) 。
590 desc |= SMEM_DESC_ENCODE((uint64_t)1024) << 32;
591
592 // — bits [62, 64): layout_type (swizzle mode) —
593 //  $1llu \ll 62 =$  二进制 01 在 bits [62, 64) = layout_type = 1。
594 // 1 = 128B swizzle mode。
595 // 其他取值: 0=None, 2=64B swizzle, 3=32B swizzle。
596 // 128B swizzle 将 smem 中连续的行按 128B 粒度进行 XOR 重映射,
597 // 确保同一 warp 内各线程访问不同 bank 的同一偏移时不会产生 bank conflict。
598 desc |= 1llu << 62;
599
600 return desc;
601 }
602
603 // ---- WGMMMA PTX 指令宏 ----
604 // wgmma.mma_async.sync.aligned.m64n128k16.f16.f16.f16
605 // m64n128k16: M=64, N=128, K=16。一次 WGMMMA 计算  $D[64 \times 128] += A[64 \times 16] \times B[16 \times 128]$ 。
606 // f16.f16.f16: A=f16, B=f16, D=f16 (累加器也是 f16 精度)。
607 // 每线程 32 个 uint32 输出:  $D=64 \times 128=8192$  half, warpgroup 128 线程分担,
608 // 每线程 64 half = 32 uint32。
609 // descA/descB: shared memory 描述符 (由 make_smem_desc 生成, 64-bit 编码)。
610 // ScaleD: 0=清零累加器再写入, 1=与累加器中的旧值累加 (K 维迭代必须用 1)。
611 // ScaleA/ScaleB: 1=正常符号, -1=翻转符号 (此处始终用 1)。
612 // TransA/TransB: 0=K-major (K 维连续), 1=MN-major (M/N 维连续)。本实现始终用 0。
613 // 参考: PTX ISA §9.7.15.4 (wgmma.mma_async)
614 #define WGMMMA_M64N128K16_F16F16F16(d, sA, sB, ScaleD, ScaleA, ScaleB, TransA, \
615                                     TransB) \
616 { \
617     uint64_t desc_a = make_smem_desc(&(sA)[0]); \
618     uint64_t desc_b = make_smem_desc(&(sB)[0]); \
619     asm volatile( \
620         "{\n" \
621         "wgmma.mma_async.sync.aligned.m64n128k16.f16.f16.f16 " \
622         "{%0, %1, %2, %3, %4, %5, %6, %7, " \
623         "%8, %9, %10, %11, %12, %13, %14, %15, " \
624         "%16, %17, %18, %19, %20, %21, %22, %23, " \
625         "%24, %25, %26, %27, %28, %29, %30, %31}, " \
626         "%32, " \
627         "%33, " \
628         "%34, %35, %36, %37, %38;\n" \
629         "}\n"

```

```

630         : "+r"((d)[0][0]), "+r"((d)[0][1]), "+r"((d)[0][2]), "+r"((d)[0][3]), \
631         "+r"((d)[1][0]), "+r"((d)[1][1]), "+r"((d)[1][2]), "+r"((d)[1][3]), \
632         "+r"((d)[2][0]), "+r"((d)[2][1]), "+r"((d)[2][2]), "+r"((d)[2][3]), \
633         "+r"((d)[3][0]), "+r"((d)[3][1]), "+r"((d)[3][2]), "+r"((d)[3][3]), \
634         "+r"((d)[4][0]), "+r"((d)[4][1]), "+r"((d)[4][2]), "+r"((d)[4][3]), \
635         "+r"((d)[5][0]), "+r"((d)[5][1]), "+r"((d)[5][2]), "+r"((d)[5][3]), \
636         "+r"((d)[6][0]), "+r"((d)[6][1]), "+r"((d)[6][2]), "+r"((d)[6][3]), \
637         "+r"((d)[7][0]), "+r"((d)[7][1]), "+r"((d)[7][2]), "+r"((d)[7][3]) \
638     : "l"(desc_a), "l"(desc_b), "n"(int32_t(Scaled)), \
639     "n"(int32_t(ScaleA)), "n"(int32_t(ScaleB)), "n"(int32_t(TransA)), \
640     "n"(int32_t(TransB))); \
641 }
642
643 // ---- TMA Shared Memory Layout ----
644 // Multi-stage pipeline: kStages 个 stage, 每个 stage 存储 A[BM×BK] + B[BK×BN]
645 //
646 // 每个 stage 包含两块 smem:
647 //   A tile: [BM, BK] row-major → BK×BM 个 half, 地址连续
648 //   B tile: TMA 按 [BN, BK] 物理写入 (详见下方 ★), BN×BK 个 half
649 //
650 // ★ 关键区别 — WGMMMA 的 B 物理存储 vs 逻辑视图:
651 //
652 //   上层数据: 源矩阵 B^T [N, K] row-major (TN 布局, B 的列以行优先形式存储)。
653 //   物理存储: TMA 将 B^T 的 tile 写入 smem, 物理布局为 [BN, BK] row-major
654 //             (BN=128 行, BK=64 列, 每行 64 个连续的 half)。
655 //   TMA 的 smem_box_shape = (BK=64, BN=128) 定义了每个 tile 的 shape,
656 //   TMA 按行写入, 行内 BK 个 half 连续 → 物理上就是 [BN][BK]。
657 //   逻辑视图: WGMMMA 指令 (wgmma.mma_async.m64n128k16, PTX ISA §9.7.15.4)
658 //             通过 imm-trans-b=0 指定 B 为 K-major, 即逻辑上按 [BK, BN] 寻址。
659 //   实际 [BN, BK] → 逻辑 [BK, BN] 的"转置"是通过 descB 中的 128B
660 //   swizzle (layout_type=1) 在硬件地址重映射层完成的, 无需软件转置。
661 //
662 //   stride_byte_offset=1024 的来源 (K-major + 128B swizzle + f16, PTX ISA §9.7.15.5.1.2.1.2) :
663 //   128B swizzle atom = 8(N) × 8(K) × 128-bit = 8 rows × 64 个 half。
664 //   stride_byte_offset = 从当前 8-row stripe 到下一个 8-row stripe 的字节偏移
665 //                       = 8 rows × (BK=64) halves × 2 bytes = 1024。
666 //   这个值恰好等于物理 [BN, BK] 布局中连续 8 行的跨度 ← 进一步证实物理存储是 [BN, BK]。
667 //
668 //   与 MMA(TN) 的对比:
669 //   MMA (TN): smem 存 B^T [N, K] row-major, ldmatrix 按列解出 col-major B fragment。
670 //   WGMMMA:   smem 物理存 [BN, BK] (TMA 直写), WGMMMA 通过 swizzle 硬件以 K-major [BK, BN] 逻辑视图读取。
671 //   简言之: swizzle 桥接了 "物理 row-major [BN, BK]" 和 "逻辑 K-major [BK, BN]" 的差异。
672 template <int BM, int BN, int BK, int QSIZE> struct WgmmaSMem {
673     alignas(128) half A[BM * BK * QSIZE]; // A tile: row-major [BM, BK]
674     // B tile 标记:
675     //   - 物理存储: TMA 从 B^T[N,K] row-major 搬入, 物理上是 [BN, BK] row-major (BN 行, BK 列)
676     //   - 逻辑视图: WGMMMA 通过 descB 的 128B swizzle (PTX ISA §9.7.15.5.1.2) 将地址重映射,
677     //     以 K-major [BK, BN] 逻辑视图供 wgmma.mma_async (imm-trans-b=0, PTX ISA §9.7.15.4) 读取
678     //   - B[BN * BK * QSIZE] 与 B[BK * BN * QSIZE] 数值等价 (BN×BK = BK×BN),
679     //     但写 BN×BK 更能体现物理存储形态
680     alignas(128) half B[BN * BK * QSIZE];
681 };
682 #endif /* NOTES_V2_ENABLE_WGMMMA */
683
684 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
685 // SM120 不支持 WGMMMA, 但支持相同的 TMA 生产者协议和 warp 级 mma.sync。
686 // 保持 128B TMA swizzle 并显式声明物理布局供 ldmatrix 消费者使用。
687 template <int BM, int BN, int BK, int QSIZE> struct TmaMmaWSSMem {
688     static_assert(BK == 64, "The 128B swizzle helper below is specialized for BK=64");
689     half A[BM * BK * QSIZE];
690     half B[BN * BK * QSIZE];
691 };
692 #endif /* NOTES_V2_ENABLE_TMA_MMA_WS */

```

```

693
694 #if defined(NOTES_V2_ENABLE_WGMMMA) || defined(NOTES_V2_ENABLE_TMA_MMA_WS)
695 // ---- Host-side TMA Tensor Map helpers (WGMMMA test) ----
696 // 面试要点 (TMA descriptor 创建 — cuTensorMapEncodeTiled) :
697 //   - TMA descriptor 描述 global memory 中矩形的 shape/stride/dtype,
698 //     以及硬件搬运的 box (tile) 大小和 smem swizzle 模式
699 //   - 关键易错点: TMA shape 参数写的是 (W,H) 而不是 (H,W)!
700 //     对 row-major [M,K] 矩阵, TMA shape = (K,M), minor=K, major=M
701 //   - cuTensorMapEncodeTiled 是 CUDA Driver API, 需 #include <cuda.h> 并链接 -lcuda
702 //   - smem_box 维度顺序也是 (minor, major) = (BK, BM) 或 (BK, BN)
703 //   - Swizzle 模式通常选 CU_TENSOR_MAP_SWIZZLE_128B 配合 WGMMMA 的 128B swizzle
704 //
705 // * gmem_prob_stride + 1 的含义:
706 //   语法层面 — 数组名退化 + 指针算术:
707 //     gmem_prob_stride 类型为 uint64_t[5], 在表达式中退化为 uint64_t*
708 //     (指向首元素 gmem_prob_stride[0])。+ 1 偏移 1 个 uint64_t, 指向
709 //     gmem_prob_stride[1], 等价于 &gmem_prob_stride[1]。
710 //   语义层面 — 跳过隐式的最内维 stride:
711 //     cuTensorMapEncodeTiled 的 globalStrides 参数只接收 tensorRank-1 个
712 //     stride 值。最内维 (fastest-changing dimension) 的 stride 是隐式的,
713 //     固定等于 sizeof(element_type), 不需要显式传入。
714 //     对于 tensorRank=2 的 FP16 矩阵:
715 //       dim 0 (内维, K) : stride = sizeof(half)    ← 隐式, API 不需要
716 //       dim 1 (外维, M) : stride = sizeof(half)*K  ← 需要传给 API
717 //     gmem_prob_stride 数组的布局是内维在前:
718 //       [0] = sizeof(half)                        ← 内维, 不传
719 //       [1] = sizeof(half) * BlockMinorSize * blocks_width ← 外维, 传给 API
720 //     所以 + 1 的作用是跳过 [0], 让 API 从 [1] 开始读取, 正好得到外维 stride。
721 //
722 // * smem_box_stride 为什么不需要 +1?
723 //   与 globalStrides 不同, smemBoxStrides 参数接收完整的 tensorRank 个 stride,
724 //   最内维 stride 也必须显式传入 (不隐式)。
725 //   smem_box_stride[5] = {1, 1, 1, 1, 1} 表示 smem tile 中所有维度元素
726 //   都是连续存放的, 维度间没有 padding/stride。
727 //   对于 tensorRank=2: strides[0]=1 (内维 stride), strides[1]=1 (外维 stride),
728 //   即 smem 中第 (i,j) 元素的偏移 = i*1 + j*1 = i+j (元素连续排列)。
729 //   所以这里直接传 smem_box_stride (即 &smem_box_stride[0]), API 会读到全部
730 //   两个 stride 值, 不需要跳过任何一个。
731 //
732 // 参考: CUDA Programming Guide §TMA, PTX ISA §9.7.15.5, CUTLASS GmmaDescriptor
733
734 template <int BlockMajorSize, int BlockMinorSize>
735 __host__ static inline void create_tensor_map(CUtensorMap *tma_map,
736                                               half *gmem_ptr,
737                                               int blocks_height,
738                                               int blocks_width) {
739     void *gmem_address = (void *)gmem_ptr;
740     uint64_t gmem_prob_shape[5] = {(uint64_t)BlockMinorSize * blocks_width,
741                                     (uint64_t)BlockMajorSize * blocks_height,
742                                     1, 1, 1};
743     uint64_t gmem_prob_stride[5] = {
744         sizeof(half), sizeof(half) * BlockMinorSize * blocks_width, 0, 0, 0};
745     uint32_t smem_box_shape[5] = {uint32_t(BlockMinorSize),
746                                    uint32_t(BlockMajorSize), 1, 1, 1};
747     uint32_t smem_box_stride[5] = {1, 1, 1, 1, 1};
748     CUresult result = cuTensorMapEncodeTiled(
749         tma_map, CU_TENSOR_MAP_DATA_TYPE_FLOAT16, 2, gmem_address,
750         gmem_prob_shape, gmem_prob_stride + 1, smem_box_shape, smem_box_stride,
751         CU_TENSOR_MAP_INTERLEAVE_NONE, CU_TENSOR_MAP_SWIZZLE_128B,
752         CU_TENSOR_MAP_L2_PROMOTION_NONE, CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE);
753     if (result != CUDA_SUCCESS)
754         printf("cuTensorMapEncodeTiled failed: %d\n", (int)result);
755 }

```

```

756
757 // BlockMajorSize = major dim (outer), BlockMinorSize = minor dim (inner, contiguous).
758 // For row-major [Major, Minor] matrices.
759 // Default <128, 64> matches HGEMM A/B^T and FA Q; FA K/V use <64, 64>.
760 template <int BlockMajorSize = 128, int BlockMinorSize = 64>
761 __host__ static inline CUTensorMap *allocate_and_create_tensor_map(
762     half *src, int blocks_height, int blocks_width) {
763     CUTensorMap *tma_map_d;
764     cudaMalloc(&tma_map_d, sizeof(CUTensorMap));
765     CUTensorMap tma_map_host;
766     create_tensor_map<BlockMajorSize, BlockMinorSize>(&tma_map_host, src,
767                                                         blocks_height,
768                                                         blocks_width);
769     cudaMemcpy(tma_map_d, &tma_map_host, sizeof(CUTensorMap),
770               cudaMemcpyHostToDevice);
771     return tma_map_d;
772 }
773 #endif /* NOTES_V2_ENABLE_WGMMA || NOTES_V2_ENABLE_TMA_MMA_WS */

1 #pragma once
2 #include "common.cuh"
3 // base.cuh: Phase 0-5 (Warp/Block Reduce, Elementwise, Softmax/Norm, RoPE, Transpose)
4 // Phase 0: 面试框架速查 (纯注释, 面试开场必备的基础知识)
5 // =====
6
7 // ---- GPU 架构速查 ----
8 //
9 // SM (Streaming Multiprocessor) 内部结构:
10 // - Warp Scheduler x4: 每 SM 4 个 warp scheduler, 每个每周期可发射 1 条指令
11 // - Register File: 每 SM 65536 x 32-bit (4 bytes) = 256KB
12 // - Shared Memory / L1: 可配置, 最大 shared memory ~228KB (Hopper)
13 // - Tensor Cores: Hopper 每 SM 4 个; Blackwell 数量随型号/定义不同, 建议以官方 ISV guide 为准
14 // - Warp = 32 threads: 最小调度单元, SIMT 执行模型
15 //
16 // Memory Hierarchy 带宽数量级 (H100 参考):
17 // HBM3: ~3.35 TB/s (理论), 实际 ~2.5-3.0 TB/s
18 // L2 Cache: ~12 TB/s (50MB, 跨 SM 共享)
19 // L1/SMEM: ~19 TB/s (每 SM ~228KB)
20 // Register: ~0 延迟, ~100+ TB/s 等效带宽
21 //
22 // 关键瓶颈判断:
23 // Memory-bound: AI (Arithmetic Intensity) < 机器 FLOPS/带宽 比值
24 // Compute-bound: AI 足够大, 受限于计算吞吐
25 // Latency-bound: 线程不够多, 无法隐藏内存延迟
26 //
27 // Occupancy 公式:
28 // occupancy = active_warps / max_warps_per_SM
29 // 受三类资源分别取下限: 每线程寄存器数 → threads/SM; 每 block shared memory
30 // → blocks/SM; block 大小 → blocks/SM
31
32 // ---- 常见优化手段速查清单 ----
33 //
34 // 1. Coalesced Memory Access (合并访问)
35 // - 同一 warp 的线程访问连续的 128B 对齐地址 → 1 次内存事务
36 // - 否则产生多次事务 (最坏 32 次)
37 //
38 // 2. Tiling (分块)
39 // - 将数据从 HBM 分块加载到 shared memory 复用, 减少 HBM 访问
40 // - GEMM: Block Tile (BM×BN) + K Tile (BK)
41 //
42 // 3. Vectorized Memory Access (向量化)
43 // - 使用 float4/half2 等向量类型, 减少 load/store 指令数
44 // - float4 = 128-bit, 单条指令加载 16 bytes

```

```

45 //
46 // 4. Thread Tile (寄存器分块)
47 //   - 每个线程计算多个输出元素 (TM×TN)，提高计算密度
48 //   - 减少线程总数，降低同步开销
49 //
50 // 5. Bank Conflict Avoidance
51 //   - Shared memory 有 32 banks × 4 bytes
52 //   - 同 warp 多线程访问同一 bank 的不同地址 → bank conflict → 串行化
53 //   - 解决方案：PAD (在每行末尾加 1 个元素打破对齐)
54 //
55 // 6. Pipeline / Double Buffering (流水线)
56 //   - cp.async 异步拷贝下一批数据时同时做当前批的计算
57 //   - Stage 数 = 2/3/4, 权衡 shared memory 占用和延迟隐藏
58 //
59 // 7. Tensor Core (MMA / WGMMMA)
60 //   - MMA m16n8k16 (Ampere): warp 级指令, 单 warp 完成 16×8×16 的矩阵乘
61 //   - WGMMMA m64n128k16 (Hopper): warpgroup 级指令 (128 threads), 异步执行
62 //
63 // 8. Warp Specialization (Hopper+)
64 //   - Producer warpgroup 做 TMA 数据搬运, Consumer warpgroup 做计算
65 //   - 通过 cuda::barrier 同步, 完全解耦数据搬运和计算
66 //
67 // 9. TMA (Tensor Memory Accelerator, Hopper+)
68 //   - 硬件 DMA 引擎, 支持 1D~5D 寻址, 低寄存器开销
69 //   - 配合 cp.async.bulk 实现异步数据搬运
70
71 // ---- Roofline 分析公式 ----
72 //
73 // AI (Arithmetic Intensity) = FLOPs / Bytes_transferred
74 //
75 // GEMM (M=N=K=4096):
76 //   FLOPs = 2 × M × N × K = 2 × 40963 ≈ 137 GFLOPs
77 //   Bytes = (M×K + K×N + M×N) × sizeof(float) ≈ 200 MB
78 //   AI ≈ 137G / 200M ≈ 685 FLOPs/Byte → compute-bound (远超 H100 ridge point:
79 //   FP16 TC ≈ 295:1, FP32 ≈ 20:1)
80 //
81 // GEMV (M=4096, K=4096):
82 //   Bytes = (M×K + K + M) × sizeof(float) ≈ 67 MB
83 //   AI ≈ 33M / 67M ≈ 0.5 FLOPs/Byte → severely memory-bound
84 //
85 // Softmax (N=4096): AI ≈ (5×N) / (2×N×4) = 5/8 ≈ 0.625 FLOPs/Byte → memory-bound
86
87 // =====
88 // Phase 1: 头文件 + 宏定义 + 基础原语 (Warp Reduce / Block Reduce)
89 // =====
90 // 面试要点:
91 //   - warp_reduce: 用 __shfl_xor_sync 做蝶形归约, O(logN) 步, 无需 shared
92 //   memory
93 //   - block_reduce: 两级归约 (warp → shared memory → warp0 broadcast),
94 //     注意最后必须 broadcast 回所有线程 (__shfl_sync), 否则只有 warp0 知道结果
95 //   - 为什么不用 __shfl_down_sync? xor 模式所有线程做相同工作量, 更均衡
96
97 #include <algorithm>
98 #include <cstring>
99 #include <cuda_fp16.h>
100
101 // =====
102 // Phase 1a: Warp Reduce (warp 内归约, 纯寄存器操作, 无需 shared memory)
103 // =====
104
105 // Warp Reduce Sum — generic (used by both FP32 and FP16 contexts)
106 // 使用 __shfl_xor_sync 做蝶形归约 (butterfly reduction)
107 // 复杂度 O(logN), N=32 时仅需 5 步

```

```

108 // 模板参数: T=数据类型, kWarpWidth=segment width (默认 32)
109 // kWarpWidth 会作为 __shfl_xor_sync 的第 4 个参 width, 限制 shuffle 在同一 segment 内
110 // 当 kWarpWidth < 32 (如 FA 中 kWarpWidth=4) 时, 只有同 segment 的 lane 参与通信
111 // 蝶形归约示意 (以 warpSize=8 为例, 实际 warpSize=32 有 5 次迭代):
112 //
113 // 初始: 每个 lane 持有自己的值 v0..v7
114 // lane:  0    1    2    3    4    5    6    7
115 // val:  v0   v1   v2   v3   v4   v5   v6   v7
116 //
117 // mask=4 (第1次迭代, lane i 与 lane i^4 交换并累加):
118 //      ┌───────────┐
119 // 对:  (0,4) (1,5) (2,6) (3,7)
120 //
121 // lane:  0    1    2    3    4    5    6    7
122 // val: v0+v4 v1+v5 v2+v6 v3+v7 v4+v0 v5+v1 v6+v2 v7+v3
123 //
124 // mask=2 (第2次迭代, lane i 与 lane i^2 交换并累加):
125 //      ┌───┐ ┌───┐ ┌───┐ ┌───┐
126 // 对:  (0,2) (1,3) (4,6) (5,7)
127 //      └───┬───┬───┬───┘
128 //          前4个一组 后4个一组
129 // lane:  0    1    2    3    4    5    6    7 (每lane持有前一轮2个值的和再加本轮配对)
130 // val: Σ{0,2,4,6} Σ{1,3,5,7} Σ{0,2,4,6} Σ{1,3,5,7} Σ{0,2,4,6} Σ{1,3,5,7} Σ{0,2,4,6} Σ{1,3,5,7}
131 //      = v0+v2+v4+v6 ... (逐步归约)
132 //
133 // mask=1 (第3次迭代, lane i 与 lane i^1 交换并累加):
134 //      ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐
135 // 对:  (0,1) (2,3) (4,5) (6,7)
136 //
137 // lane:  0    1    2    3    4    5    6    7
138 // val: Σall Σall Σall Σall Σall Σall Σall Σall ← 所有 lane 拥有全归约结果!
139 //
140 // mask=0: 循环终止, 归约完成。
141 //
142 // 关键性质:
143 // - XOR 配对是对称的: lane i 的配对对象是 lane i^mask, 而 (i^mask)^mask = i
144 // - 每轮每个 lane 只和恰好 1 个其他 lane 通信 (一对一, 无冲突)
145 // - 每轮信息传递距离减半: 16→8→4→2→1 (距离减半, 信息翻倍)
146 // - O(log2 N) 步完成, 无需 shared memory, 纯寄存器操作
147 // - __shfl_xor_sync 第四个参数 kWarpWidth 限制 segment width:
148 //   当 kWarpWidth=4 时, 只有同 segment(4个一组)内的 lane 参与 shuffle
149 template <const int kWarpWidth = kWarpSize, typename T = float>
150 __device__ __forceinline__ T warp_reduce_sum(T val) {
151 #pragma unroll
152   for (int mask = kWarpWidth >> 1; mask >= 1; mask >>= 1) {
153     val += __shfl_xor_sync(0xffffffff, val, mask, kWarpWidth);
154   }
155   return val;
156 }
157
158 // Warp Reduce Max — generic
159 template <const int kWarpWidth = kWarpSize, typename T = float>
160 __device__ __forceinline__ T warp_reduce_max(T val) {
161 #pragma unroll
162   for (int mask = kWarpWidth >> 1; mask >= 1; mask >>= 1) {
163     val = max(val, __shfl_xor_sync(0xffffffff, val, mask, kWarpWidth));
164   }
165   return val;
166 }
167
168 // =====
169 // Phase 1b: Block Reduce (block 内归约, 两级: warp → shared memory → warp reduce)
170 // =====

```

```

171
172 // Block Reduce Sum — FP32 (增强版, 带 broadcast)
173 // 两级归约流程:
174 // 1. 每个 warp 内做 warp_reduce_sum → 得到每 warp 的一个值
175 // 2. warp leader (lane=0) 写入 shared memory
176 // 3. syncthreads 后, lane 0~kNumWarps-1 读取 shared memory
177 // 4. 所有 warp 内再做一次 warp_reduce<kNumWarps> → 得到最终结果
178 // 5. __shfl_sync broadcast 到所有线程 (关键! 否则每个warp只有lane<kNumWarps 知道结果)
179 template <const int kNumThreads = 256>
180 __device__ float block_reduce_sum(float val) {
181     constexpr int kNumWarps = (kNumThreads + kWarpSize - 1) / kWarpSize;
182     int warp = threadIdx.x / kWarpSize;
183     int lane = threadIdx.x % kWarpSize;
184     __shared__ float shared[kNumWarps];
185
186     float value = warp_reduce_sum<kWarpSize>(val);
187     if (lane == 0)
188         shared[warp] = value;
189     __syncthreads();
190     value = (lane < kNumWarps) ? shared[lane] : 0.0f;
191     value = warp_reduce_sum<kNumWarps>(value);
192     // 关键: broadcast 结果到所有线程, 后续用 result
193     // 做除法等操作时所有线程都能拿到
194     value = __shfl_sync(0xffffffff, value, 0, 32);
195     return value;
196 }
197
198 // Block Reduce Max — FP32 (增强版, 带 broadcast)
199 template <const int kNumThreads = 256>
200 __device__ float block_reduce_max(float val) {
201     constexpr int kNumWarps = (kNumThreads + kWarpSize - 1) / kWarpSize;
202     int warp = threadIdx.x / kWarpSize;
203     int lane = threadIdx.x % kWarpSize;
204     __shared__ float shared[kNumWarps];
205
206     float value = warp_reduce_max<kWarpSize>(val);
207     if (lane == 0)
208         shared[warp] = value;
209     __syncthreads();
210     value = (lane < kNumWarps) ? shared[lane] : -FLT_MAX;
211     value = warp_reduce_max<kNumWarps>(value);
212     value = __shfl_sync(0xffffffff, value, 0, 32);
213     return value;
214 }
215
216 // ---- Block Reduce Sum All: y = sum(a[0..N-1]) ----
217 // 多 block 各自做 warp→smem→warp0 reduce, 然后 atomicAdd 到全局 y
218 // 跨 block 求和的常见模式, 适合 N 较大时使用;
219 // Grid: ((N + 255) / 256, 1, 1)
220 // Block: (256, 1, 1)
221 // source: LeetCUDA/kernels/reduce/block_all_reduce.cu
222 template <const int kNumThreads = 256>
223 __global__ void block_reduce_all(float *a, float *y, int N) {
224     int tid = threadIdx.x;
225     int idx = blockIdx.x * kNumThreads + tid;
226     constexpr int kNumWarps = (kNumThreads + kWarpSize - 1) / kWarpSize;
227     __shared__ float shared[kNumWarps];
228
229     float val = (idx < N) ? a[idx] : 0.0f;
230     int warp = tid / kWarpSize;
231     int lane = tid % kWarpSize;
232
233     val = warp_reduce_sum<kWarpSize>(val);

```

```

234     if (lane == 0)
235         shared[warp] = val;
236     __syncthreads();
237
238     val = (lane < kNumWarps) ? shared[lane] : 0.0f;
239     if (warp == 0)
240         val = warp_reduce_sum<kNumWarps>(val);
241     if (tid == 0) // tid == 0, not lane 0. 只有 block 内的一个线程负责写回全局结果，避免重复累加
242         atomicAdd(y, val);
243 }
244
245 // ---- Dot Product: y = sum(a[i] * b[i]) ----
246 // 核心模式: elementwise 乘法 → block reduce → atomicAdd 全局累加
247 // Grid: ((N + 255) / 256, 1, 1)
248 // Block: (256, 1, 1)
249 // source: LeetCUDA/kernels/dot-product/dot_product.cu
250 template <const int kNumThreads = 256>
251 __global__ void dot(float *a, float *b, float *y, int N) {
252     int tid = threadIdx.x;
253     int idx = blockIdx.x * kNumThreads + tid;
254     constexpr int kNumWarps = (kNumThreads + kWarpSize - 1) / kWarpSize;
255     __shared__ float shared[kNumWarps];
256
257     float prod = (idx < N) ? a[idx] * b[idx] : 0.0f;
258     int warp = tid / kWarpSize;
259     int lane = tid % kWarpSize;
260
261     prod = warp_reduce_sum<kWarpSize>(prod);
262     if (lane == 0)
263         shared[warp] = prod;
264     __syncthreads();
265
266     prod = (lane < kNumWarps) ? shared[lane] : 0.0f;
267     if (warp == 0) // 只需要 warp 0 的线程继续 reduce 即可
268         prod = warp_reduce_sum<kNumWarps>(prod);
269     if (tid == 0) // tid == 0, not lane 0. 只有 block 内的一个线程负责写回全局结果，避免重复累加
270         atomicAdd(y, prod);
271 }
272
273 // Dot Product + float4
274 // Grid: ((N + 255) / 256, 1, 1), 每个block处理256元素，float4向量化后每个线程处理4元素
275 // Block: (64, 1, 1), 256/4=64
276 // 注意: 该版本默认输入地址满足 float4 对齐; 最适合 N 按 4 对齐的场景
277 // source: LeetCUDA/kernels/dot-product/dot_product.cu
278 template <const int kNumThreads = 256 / 4>
279 __global__ void dot_vec4(float *a, float *b, float *y, int N) {
280     int tid = threadIdx.x;
281     int idx = (blockIdx.x * kNumThreads + tid) * 4;
282     constexpr int kNumWarps = (kNumThreads + kWarpSize - 1) / kWarpSize;
283     __shared__ float shared[kNumWarps];
284
285     float4 reg_a = FLOAT4(a[idx]);
286     float4 reg_b = FLOAT4(b[idx]);
287     float prod = (idx < N) ? (reg_a.x * reg_b.x + reg_a.y * reg_b.y +
288                             reg_a.z * reg_b.z + reg_a.w * reg_b.w)
289                             : 0.0f;
290     int warp = tid / kWarpSize;
291     int lane = tid % kWarpSize;
292
293     prod = warp_reduce_sum<kWarpSize>(prod);
294     if (lane == 0)
295         shared[warp] = prod;
296     __syncthreads();

```

```

297 prod = (lane < kNumWarps) ? shared[lane] : 0.0f;
298 if (warp == 0)
299     prod = warp_reduce_sum<kNumWarps>(prod);
300 if (tid == 0)
301     atomicAdd(y, prod);
302 }
303
304
305
306 // =====
307 // Phase 2: Elementwise Ops (逐元素操作, 演示 coalesced access + vectorize)
308 // =====
309 // 面试要点:
310 //   - 逐元素操作是最简单的 kernel, 核心考点是 memory coalescing
311 //   - float4 向量化可将内存事务数减为 1/4, 大幅提升 bandwidth utilization
312 //   - grid/block 维度设计: grid(N/threads), block(threads), 一维即可
313
314 // ---- ReLU: y = max(0, x) ----
315 // Grid: ((N + 255) / 256, 1, 1)
316 // Block: (256, 1, 1)
317 // source: LeetCUDA/kernels/relu/relu.cu
318 __global__ void relu(float *x, float *y, int N) {
319     int idx = blockIdx.x * blockDim.x + threadIdx.x;
320     if (idx < N)
321         y[idx] = fmaxf(0.0f, x[idx]);
322 }
323
324 // ReLU + float4 向量化: 每个线程处理 4 个元素, 减少 75% 的 load/store 指令
325 // block(64)x4(float4)=256 元素/block, 与基础版吞吐相同
326 // Grid: ((N + 255) / 256, 1, 1)
327 // Block: (64, 1, 1)
328 // 注意: 该版本默认地址满足 float4 对齐; 最适合 N 按 4 对齐的场景
329 // source: LeetCUDA/kernels/relu/relu.cu
330 __global__ void relu_vec4(float *x, float *y, int N) {
331     int idx = (blockIdx.x * blockDim.x + threadIdx.x) * 4;
332     if (idx < N) {
333         float4 reg_x = FLOAT4(x[idx]); // 单条 128-bit load
334         float4 reg_y;
335         reg_y.x = fmaxf(0.0f, reg_x.x);
336         reg_y.y = fmaxf(0.0f, reg_x.y);
337         reg_y.z = fmaxf(0.0f, reg_x.z);
338         reg_y.w = fmaxf(0.0f, reg_x.w);
339         FLOAT4(y[idx]) = reg_y; // 单条 128-bit store
340     }
341 }
342
343 // ---- Elementwise Add: c = a + b ----
344 // Grid: ((N + 255) / 256, 1, 1)
345 // Block: (256, 1, 1)
346 // source: LeetCUDA/kernels/elementwise/elementwise.cu
347 __global__ void elementwise_add(float *a, float *b, float *c, int N) {
348     int idx = blockIdx.x * blockDim.x + threadIdx.x;
349     if (idx < N)
350         c[idx] = a[idx] + b[idx];
351 }
352
353 // Elementwise Add + float4 向量化
354 // Grid: ((N + 255) / 256, 1, 1)
355 // Block: (64, 1, 1), block(64)x4(float4)=256 元素/block
356 // 注意: 主路径要求 4 元素对齐; 尾部不足 4 个元素时回退到标量处理
357 // source: LeetCUDA/kernels/elementwise/elementwise.cu
358 __global__ void elementwise_add_vec4(float *a, float *b, float *c, int N) {
359     int idx = 4 * (blockIdx.x * blockDim.x + threadIdx.x);

```

```

360 if ((idx + 3) < N) {
361     float4 reg_a = FLOAT4(a[idx]);
362     float4 reg_b = FLOAT4(b[idx]);
363     float4 reg_c;
364     reg_c.x = reg_a.x + reg_b.x;
365     reg_c.y = reg_a.y + reg_b.y;
366     reg_c.z = reg_a.z + reg_b.z;
367     reg_c.w = reg_a.w + reg_b.w;
368     FLOAT4(c[idx]) = reg_c;
369 } else if (idx < N) {
370     for (int i = 0; (idx + i) < N; i++) {
371         c[idx + i] = a[idx + i] + b[idx + i];
372     }
373 }
374 }
375
376 // ---- Histogram: y[a[i]]++ ----
377 // 演示 atomicAdd 的用法：多个线程可能同时更新同一个 bin
378 // Grid: ((N + 255) / 256, 1, 1)
379 // Block: (256, 1, 1)
380 // source: LeetCUDA/kernels/histogram/histogram.cu
381 __global__ void histogram(int *a, int *y, int N) {
382     int idx = blockIdx.x * blockDim.x + threadIdx.x;
383     if (idx < N)
384         atomicAdd(&y[a[idx]], 1);
385 }
386
387 // ---- Merge Attn States: 合并 Split-KV Attention 的部分结果 ----
388 // 实现 FlashInfer 论文 (arXiv 2501.01005) Section 2.2 的 attention 合并逻辑。
389 //
390 // 面试要点：
391 // - 应用场景：Split-KV attention 将序列沿 K 维分段计算 attention，
392 //   每段产出部分结果 (O_i, LSE_i)，本 kernel 将两段按指数权重加权合并。
393 // - 数学公式 (5 步推导)：
394 //   Step 1 — max 归一化 (数值稳定，与 safe softmax 同理)：
395 //       L_max = max(LSE_1, LSE_2)
396 //   Step 2 — 还原指数权重 (un-normalized)：
397 //       w_1 = exp(LSE_1 - L_max), w_2 = exp(LSE_2 - L_max)
398 //   Step 3 — 归一化为混合比例：
399 //       alpha = w_1 / (w_1 + w_2), beta = w_2 / (w_1 + w_2)
400 //       满足 alpha + beta = 1
401 //   Step 4 — 逐元素加权合并：
402 //       O = alpha * O_1 + beta * O_2
403 // - LSE 布局 [num_heads, num_tokens] (与 FlashAttention-2 惯例一致，
404 //   lse[head_idx][token_idx])
405 // - Output 布局 [num_tokens, num_heads, head_size] (token 维在最外，
406 //   展平为 [T0H0, T0H1, ..., T1H0, ...])
407 // -  $-\infty$  LSE  $\rightarrow$   $-\infty$ ：空 attention 段 (causal mask 等导致全部 score
408 //   为  $-\infty$ ) 的 LSE 可能为  $+\infty$ ，替换后  $\exp(-\infty - L_{\max}) = 0$ ，
409 //   该段权重退化为 0
410 // - 128-bit 向量化：uint4 = 4 × float，每线程处理 4 个输出元素，
411 //   pack load  $\rightarrow$  FMA  $\rightarrow$  pack store 一条流水线
412 // -  $AI \approx 3 \text{ FLOP} / 20 \text{ bytes} \approx 0.15 \text{ FLOPS/Byte} \rightarrow$  severely memory-bound
413 //
414 // 线程到数据的 3D 映射 (以 num_tokens=2, num_heads=2, head_size=8 为例)：
415 //   kPackSize = 16 / sizeof(float) = 4 (每线程 4 个 float = 128-bit)
416 //   threads_per_head = head_size / 4 = 2
417 //   total_threads = num_tokens * num_heads * threads_per_head = 8
418 //
419 //   global_idx:      0      1      2      3      4      5      6      7
420 //   token_head_idx:  0      0      1      1      2      2      3      3
421 //   (第几个 (token, head) 对, 行优先)
422 //   pack_idx:        0      1      0      1      0      1      0      1

```

```

423 //      (该 (token,head) 对内第几个 pack)
424 // token_idx:      0      0      0      0      1      1      1      1
425 //      (token_head_idx / num_heads, token 变化慢)
426 // head_idx:      0      0      1      1      0      0      1      1
427 //      (token_head_idx % num_heads, head 变化快)
428 // pack_offset:    0      4      0      4      0      4      0      4
429 //      (pack_idx * 4, 该 pack 在 head_size 中的起始元素偏移)
430 // head_offset:    0      0      8      8      16     16     24     24
431 //      (token_idx * num_heads * head_size + head_idx * head_size,
432 //      该 (token,head) 对在 output 展平数组中的起始偏移)
433 //
434 // Grid: ((total_threads + 127) / 128, 1, 1)
435 // Block: (128, 1, 1)
436 // source: LeetCUDA/kernels/openai-triton/merge-attn-states/cuda_merge_attn_states.cu
437 __global__ void merge_attn_states(
438     float *output,           // [num_tokens, num_heads, head_size]
439     const float *prefix_output, // [num_tokens, num_heads, head_size]
440     const float *prefix_lse,   // [num_heads, num_tokens]
441     const float *suffix_output, // [num_tokens, num_heads, head_size]
442     const float *suffix_lse,   // [num_heads, num_tokens]
443     int num_tokens, int num_heads, int head_size) {
444
445     constexpr int kNumThreads = 128;
446     constexpr int kPackSize = 16 / sizeof(float); // 4 floats = 128-bit
447     using pack_t = uint4;
448
449     // 每个 (token, head) 对需要 threads_per_head 个线程覆盖 head_size 个元素
450     const int threads_per_head = head_size / kPackSize;
451     // 实际有效总线程数, 超过这个数的线程会被忽略, block是按照kNumThreads=128
452     // 来分配的, 那么最后一个block的线程数可能会超过实际需要的线程数
453     const int total_threads = num_tokens * num_heads * threads_per_head;
454
455     const int global_idx = blockIdx.x * kNumThreads + threadIdx.x;
456     if (global_idx >= total_threads)
457         return;
458
459     // global_idx → (token_head_idx, pack_idx) → (token_idx, head_idx, pack_offset)
460     // token_head_idx: 第几个 (token, head) 对, 行优先展平
461     const int token_head_idx = global_idx / threads_per_head;
462     // pack_idx: 该 (token, head) 对内第几个 pack, 0 ~ threads_per_head-1
463     const int pack_idx = global_idx % threads_per_head;
464
465     // token_head_idx 分解为 (token_idx, head_idx)
466     const int token_idx = token_head_idx / num_heads; // token 变化慢 (外维)
467     const int head_idx = token_head_idx % num_heads; // head 变化快 (内维)
468
469     // pack_offset: 该 pack 覆盖的元素在 head_size 中的起始偏移 (0, 4, 8, ...)
470     const int pack_offset = pack_idx * kPackSize;
471     // head_offset: 该 (token, head) 对在 output 展平一维数组中的起始偏移
472     const int head_offset = token_idx * num_heads * head_size + head_idx * head_size;
473
474     // 定位到当前 (token, head) 的输出段起始
475     const float *prefix_head = prefix_output + head_offset;
476     const float *suffix_head = suffix_output + head_offset;
477     float *output_head = output + head_offset;
478
479     // LSE 布局 [num_heads, num_tokens]: lse[head_idx][token_idx]
480     float p_lse = prefix_lse[head_idx * num_tokens + token_idx];
481     float s_lse = suffix_lse[head_idx * num_tokens + token_idx];
482
483     // inf → -inf: 空 attention 段 LSE 可能为 +inf, 替换后权重退化为 0
484     p_lse = isinf(p_lse) ? -INFINITY : p_lse;
485     s_lse = isinf(s_lse) ? -INFINITY : s_lse;

```

```

486 // Step 1: max 归一化 (与 safe softmax 同理, 防 exp 溢出)
487 const float max_lse = fmaxf(p_lse, s_lse);
488 p_lse -= max_lse;
489 s_lse -= max_lse;
490
491 // Step 2-3: 指数还原 → 混合比例 alpha, beta
492 const float p_se = expf(p_lse);
493 const float s_se = expf(s_lse);
494 const float out_se = p_se + s_se;
495 const float p_scale = p_se / out_se; // alpha = w_1 / (w_1 + w_2)
496 const float s_scale = s_se / out_se; // beta = w_2 / (w_1 + w_2)
497
498 // Step 4: 逐元素加权合并 0 = alpha * 0_1 + beta * 0_2
499 // 128-bit 向量化: uint4 load → per-element FMA → uint4 store
500 if (pack_offset < head_size) {
501     pack_t p_pack =
502         reinterpret_cast<const pack_t *>(prefix_head)[pack_idx];
503     pack_t s_pack =
504         reinterpret_cast<const pack_t *>(suffix_head)[pack_idx];
505     pack_t o_pack;
506
507 #pragma unroll
508     for (int i = 0; i < kPackSize; ++i) {
509         const float p_v = reinterpret_cast<const float *>(&p_pack)[i];
510         const float s_v = reinterpret_cast<const float *>(&s_pack)[i];
511         const float o_v = p_v * p_scale + (s_v * s_scale); // FMA
512         reinterpret_cast<float *>(&o_pack)[i] = o_v;
513     }
514
515     reinterpret_cast<pack_t *>(output_head)[pack_idx] = o_pack;
516 }
517 }
518
519 // =====
520 // Phase 3: Reduce 类 Ops — Softmax / RMS Norm / Layer Norm
521 // =====
522 // 面试要点:
523 //   - Softmax 三种实现递进: naive (溢出) → safe (2-pass, max 减法) →
524 //     online (1-pass, 增量更新)
525 //   - Online Softmax 是 FlashAttention-2 的数学基础
526 //   - RMS Norm vs Layer Norm: RMS 只需 1 次 reduce, Layer Norm 需要 2 次 (mean
527 //     + variance)
528 //   - Per-token 设计: 一个 block 处理一个 token, 无需跨 block 同步
529
530 // =====
531 // Phase 3a: Softmax — 三级递进 (面试核心考点)
532 // =====
533 // 面试常问: 「Softmax 有哪些实现方式? 各有什么优缺点?」
534 // 回答线索: naive(溢出) → safe → online
535
536 // ---- Level 1: 基础 Softmax (per-token, 无 max 减法, 数值不稳定) ----
537 // grid(S*h/h, h), block(h), 一个 block 处理一个 token
538 // 问题: x 值很大时 exp(x) 溢出为 inf
539 template <const int kNumThreads = 256>
540 // Grid: (S, 1, 1), S=batch*seq_len, DISPATCH_SOFTMAX_F32_PER_TOKEN_KERNEL
541 // Block: (H, 1, 1), 由外层 dispatch 选择 H=32/64/128/256/512/1024, 一个 block 处理一个 token
542 // source: LeetCode/kernels/softmax/softmax.cu
543 __global__ void softmax_per_token(float *x, float *y, int N) {
544     const int tid = threadIdx.x;
545     const int idx = blockIdx.x * blockDim.x + tid;
546
547     float exp_val = (idx < N) ? expf(x[idx]) : 0.0f;

```

```

549 float exp_sum = block_reduce_sum<kNumThreads>(exp_val);
550 if (idx < N)
551     y[idx] = exp_val / exp_sum;
552 }
553
554 // ---- Level 2: Safe Softmax (2-pass: 先 max 再 exp, 数值稳定) ----
555 // 面试重点: 为什么 Softmax 需要 Safe?
556 //   - expf 溢出阈值约 88.7: exp(88)≈1.65e38 (接近 float32 上限 3.4e38), exp(89)≈4.5e38 已溢出为 inf
557 //   - 减去 max 后: exp(x - max) ≤ exp(0) = 1.0, 永不超过 1
558 //   - 数学等价性: softmax(x) = softmax(x - c) 对任意常数 c 成立
559 //   - 代价: 2 次 block reduce (先 max, 再 sum), 但仍 O(N/B) 高效
560 // Grid: (S, 1, 1)
561 // Block: (H, 1, 1), 由外层 dispatch 选择 H=32/64/128/256/512/1024
562 // source: LeetCUDA/kernels/softmax/softmax.cu
563 template <const int kNumThreads = 256>
564 __global__ void safe_softmax_per_token(float *x, float *y, int N) {
565     const int tid = threadIdx.x;
566     const int idx = blockIdx.x * blockDim.x + tid;
567
568     // Pass 1: block reduce max — 找最大值
569     float val = (idx < N) ? x[idx] : -FLT_MAX;
570     float max_val = block_reduce_max<kNumThreads>(val);
571
572     // Pass 2: exp(x - max) → block reduce sum
573     float exp_val = (idx < N) ? expf(x[idx] - max_val) : 0.0f;
574     float exp_sum = block_reduce_sum<kNumThreads>(exp_val);
575
576     if (idx < N)
577         y[idx] = exp_val / exp_sum;
578 }
579
580 // ---- Level 3: Online Safe Softmax (FlashAttention-2的数学基础) ----
581 // 面试重点:
582 // 两种使用场景 (公式不同, 但结果等价):
583 //   1) 单元素增量更新 (online update, 处理新元素 x_i):
584 //       m_new = max(m_old, x_i)
585 //       d_new = d_old * exp(m_old - m_new) + exp(x_i - m_new)
586 //   2) 二元合并 (binary merge, 合并两个部分累加器, warp/block reduce 使用):
587 //       m = max(m1, m2) safe softmax用的max值
588 //       d = d1*exp(m1-m) + d2*exp(m2-m) softmax用的分母值
589 //       当 m1≥m2 时退化为 d1 + d2*exp(m2-m1) (d_bigger + d_smaller*exp(m_smaller - m_bigger))
590 // 算法来源: "Online normalizer calculation for softmax" (arXiv:1805.02867)
591 // Warp Reduce for Online Softmax — binary merge of two partial (m,d) accumulators.
592 //
593 // 不同于单元素增量更新公式 (m_new = max(m_old, x_i); d_new = d_old*exp(m_old-m_new) + exp(x_i-m_new)),
594 // warp reduce 中每次合并的是两个**已各自归一化到各自 max 的部分累加器**:
595 //   (m1,d1): max 和 Σexp(x_j - m1), 覆盖集合 S1
596 //   (m2,d2): max 和 Σexp(x_k - m2), 覆盖集合 S2
597 //
598 // 合并公式 (对称, 不依赖"新/旧"概念):
599 //   m = max(m1, m2)
600 //   d = d1*exp(m1-m) + d2*exp(m2-m) ← m1≥m2 时退化为 d1 + d2*exp(m2-m1)
601 //
602 // 该操作满足结合律 → XOR butterfly 任意归约顺序均保证全局正确结果。
603 struct __align__(8) MD {
604     float m; // running max
605     float d; // running denominator (sum of exp(x - max))
606 };
607
608 template <const int kWarpWidth = kWarpSize>
609 __device__ __forceinline__ MD warp_reduce_md(MD md1) {
610 #pragma unroll
611     for (int mask = kWarpWidth >> 1; mask >= 1; mask >>= 1) {

```

```

612 MD md2;
613 md2.m = __shfl_xor_sync(0xffffffff, md1.m, mask, kWarpWidth);
614 md2.d = __shfl_xor_sync(0xffffffff, md1.d, mask, kWarpWidth);
615
616 MD b_m = (md1.m > md2.m) ? md1 : md2; // max
617 MD s_m = (md1.m > md2.m) ? md2 : md1;
618
619 // b_m.d 无需 rescale (其基准 max 已是全局 max) ,
620 // s_m.d 需 rescale: exp(s_m.m - b_m.m)
621 md1.d = b_m.d + s_m.d * __expf(s_m.m - b_m.m);
622 md1.m = b_m.m;
623 }
624 return md1;
625 }
626
627 // 注意: 这里默认一个 block 处理一个 token; 边界线程的 d=0 只参与归约, 不会写回 y
628 // Grid: (S, 1, 1)
629 // Block: (H, 1, 1), 由外层 dispatch 选择 H=32/64/128/256/512/1024
630 // source: LeetCUDA/kernels/softmax/softmax.cu
631 template <const int kNumThreads = 256>
632 __global__ void online_safe_softmax_per_token(const float *x, float *y, int N) {
633     int tid = threadIdx.x;
634     int idx = blockIdx.x * kNumThreads + threadIdx.x;
635     const int kNumWarps = (kNumThreads + kWarpSize - 1) / kWarpSize;
636     __shared__ MD shared[kNumWarps];
637
638     float val = (idx < N) ? x[idx] : -FLT_MAX;
639     int warp = tid / kWarpSize;
640     int lane = tid % kWarpSize;
641
642     // 初始化: 每个线程持有一个 (max, denom) 对
643     MD md;
644     md.m = idx < N ? val : -FLT_MAX;
645     md.d = idx < N ? 1.0f : 0.0f;
646
647     // 第一级规约: warp_reduce_md 在归约中自动更新 m 和 d
648     md = warp_reduce_md<kWarpSize>(md);
649
650     if (lane == 0)
651         shared[warp] = md;
652     __syncthreads();
653
654     // 第二级归约: 每个 warp 结果再做一次 warp_reduce_md (复用 block_reduce 模式)
655     md = lane < kNumWarps ? shared[lane] : MD{-FLT_MAX, 0.0f};
656     md = warp_reduce_md<kWarpSize>(md); // 用kWarpSize确保每个lane都能拿到最终结果
657
658     // 用全局 max 和 denom 做最终 softmax
659     float d_inv = __fdivdef(1.0f, md.d);
660     // 边界线程即使看到 d=0 的填充值, 也不会走到写回路径
661     if (idx < N) {
662         y[idx] = __expf(val - md.m) * d_inv;
663     }
664 }
665
666 // =====
667 // Phase 3b: RMS Normalization (1-pass reduce)
668 // =====
669 // 面试要点:
670 // - RMS Norm:  $y = (x / \text{rms}(x)) * g$ ,  $1/\text{rms}(x) = \text{rsqrt}(\text{mean}(x^2))$ 
671 // - 只需 1 次 block reduce (sum of squares), 比 Layer Norm 少 1 次同步
672 // - Llama 系列使用 RMS Norm
673 // - grid(N, K/K), block(K): 一行一个 block
674 // Grid: (N, 1, 1), N=batch*seq_len, 每行一个 block

```

```

675 // Block: (128, 1, 1), kNumThreads=K=128 (K>128 时调整模板参数)
676 // source: LeetCUDA/kernels/rms-norm/rms_norm.cu
677 template <const int kNumThreads = 128>
678 __global__ void rms_norm(float *x, float *y, float g, int N, int K) {
679     int tid = threadIdx.x;
680     int idx = blockIdx.x * blockDim.x + threadIdx.x;
681     const float epsilon = 1e-5f;
682
683     __shared__ float s_variance;
684     float value = (idx < N * K) ? x[idx] : 0.0f;
685     float variance = value * value;
686     variance = block_reduce_sum<kNumThreads>(variance);
687     if (tid == 0)
688         s_variance = rsqrtf(variance / (float)K + epsilon); // 1/rms(x)
689     __syncthreads();
690     if (idx < N * K)
691         y[idx] = (value * s_variance) * g;
692 }
693
694 // RMS Norm + float4
695 // Grid: (N, 1, 1)
696 // Block: (32, 1, 1), 128/4=32; 对应一行 K 元素按 4 个一组交给 32 个线程处理
697 // 注意: 该版本默认 K 按 4 对齐, 且输入/输出地址满足 float4 对齐
698 // source: LeetCUDA/kernels/rms-norm/rms_norm.cu
699 template <const int kNumThreads = 128 / 4>
700 __global__ void rms_norm_vec4(float *x, float *y, float g, int N, int K) {
701     int tid = threadIdx.x;
702     int idx = (blockIdx.x * blockDim.x + threadIdx.x) * 4;
703     const float epsilon = 1e-5f;
704
705     __shared__ float s_variance;
706     float4 reg_x = FLOAT4(x[idx]);
707     float variance = (idx < N * K) ? (reg_x.x * reg_x.x + reg_x.y * reg_x.y +
708                                     reg_x.z * reg_x.z + reg_x.w * reg_x.w)
709                                     : 0.0f;
710     variance = block_reduce_sum<kNumThreads>(variance);
711     if (tid == 0)
712         s_variance = rsqrtf(variance / (float)K + epsilon);
713     __syncthreads();
714     float4 reg_y;
715     reg_y.x = reg_x.x * s_variance * g;
716     reg_y.y = reg_x.y * s_variance * g;
717     reg_y.z = reg_x.z * s_variance * g;
718     reg_y.w = reg_x.w * s_variance * g;
719     if (idx < N * K)
720         FLOAT4(y[idx]) = reg_y;
721 }
722
723 // =====
724 // Phase 3c: Layer Normalization (2-pass reduce)
725 // =====
726 // 面试要点:
727 //   - Layer Norm:  $y = ((x - \text{mean}) / \text{std}) * g + b$ ,  $\text{std} = \sqrt{\text{variance}}$ ,  $\text{variance} = \text{mean}((x - \text{mean})^2)$ 
728 //   - 需要 2 次 block reduce: 先 mean (sum/K), 再 variance (sum((x-mean)^2)/K)
729 //   - 两次 __syncthreads 必须到位, 否则 s_mean 未对所有线程可见就计算 variance
730 // Grid: (N, 1, 1), 一行一个 block
731 // Block: (128, 1, 1), kNumThreads=K=128
732 // source: LeetCUDA/kernels/layer-norm/layer_norm.cu
733 template <const int kNumThreads = 128>
734 __global__ void layer_norm(float *x, float *y, float g, float b, int N, int K) {
735     int tid = threadIdx.x;
736     int idx = blockIdx.x * blockDim.x + threadIdx.x;
737     const float epsilon = 1e-5f;

```

```

738
739 __shared__ float s_mean;
740 __shared__ float s_variance;
741 float value = (idx < N * K) ? x[idx] : 0.0f;
742
743 // Pass 1: compute mean
744 float sum = block_reduce_sum<kNumThreads>(value);
745 if (tid == 0)
746     s_mean = sum / (float)K;
747 __syncthreads(); // 必须等待 s_mean 对所有线程可见
748
749 // Pass 2: compute variance = (x - mean)2
750 float variance = (value - s_mean) * (value - s_mean);
751 variance = block_reduce_sum<kNumThreads>(variance);
752 if (tid == 0)
753     s_variance = rsqrtf(variance / (float)K + epsilon); // 1/std
754 __syncthreads(); // 必须等待 s_variance 对所有线程可见
755
756 if (idx < N * K)
757     y[idx] = ((value - s_mean) * s_variance) * g + b;
758 }
759
760 // Layer Norm + float4
761 // Grid: (N, 1, 1)
762 // Block: (32, 1, 1), 128/4=32; 对应一行 K 元素按 4 个一组交给 32 个线程处理
763 // 注意: 该版本默认 K 按 4 对齐, 且输入/输出地址满足 float4 对齐
764 // source: LeetCUDA/kernels/layer-norm/layer_norm.cu
765 template <const int kNumThreads = 128 / 4>
766 __global__ void layer_norm_vec4(float *x, float *y, float g, float b, int N, int K) {
767     int tid = threadIdx.x;
768     int idx = (blockIdx.x * blockDim.x + threadIdx.x) * 4;
769     const float epsilon = 1e-5f;
770
771     __shared__ float s_mean;
772     __shared__ float s_variance;
773     float4 reg_x = FLOAT4(x[idx]);
774     float value = (idx < N * K) ? (reg_x.x + reg_x.y + reg_x.z + reg_x.w) : 0.0f;
775
776     float sum = block_reduce_sum<kNumThreads>(value);
777     if (tid == 0)
778         s_mean = sum / (float)K;
779     __syncthreads();
780
781     float4 reg_x_hat;
782     reg_x_hat.x = reg_x.x - s_mean;
783     reg_x_hat.y = reg_x.y - s_mean;
784     reg_x_hat.z = reg_x.z - s_mean;
785     reg_x_hat.w = reg_x.w - s_mean;
786     float variance = reg_x_hat.x * reg_x_hat.x + reg_x_hat.y * reg_x_hat.y +
787                     reg_x_hat.z * reg_x_hat.z + reg_x_hat.w * reg_x_hat.w;
788     variance = block_reduce_sum<kNumThreads>(variance);
789     if (tid == 0)
790         s_variance = rsqrtf(variance / (float)K + epsilon);
791     __syncthreads();
792
793     float4 reg_y;
794     reg_y.x = reg_x_hat.x * s_variance * g + b;
795     reg_y.y = reg_x_hat.y * s_variance * g + b;
796     reg_y.z = reg_x_hat.z * s_variance * g + b;
797     reg_y.w = reg_x_hat.w * s_variance * g + b;
798     if (idx < N * K)
799         FLOAT4(y[idx]) = reg_y;
800 }

```

```

801
802 // =====
803 // Phase 4: RoPE — 旋转位置编码 (Rotary Position Embedding)
804 // =====
805 // 面试要点:
806 //   - RoPE 数学公式: 对每对相邻维度做 2D 旋转
807 //     [x1']   [cos(θ)  -sin(θ)] [x1]
808 //     [x2'] = [sin(θ)   cos(θ)] [x2]
809 //   -  $\theta_i = 1 / (\text{theta}^{(2i/d)})$ , theta=10000.0f (Llama 风格)
810 //   - token_pos = idx / N: token 在序列中的位置
811 //   - token_idx = idx % N: token 内的维度索引
812 //   - 输入 [seq_len, hidden_size], 输出同形状
813
814 // Grid: ((seq_len * N + 255) / 256, 1, 1)
815 // Block: (256, 1, 1)
816 // source: LeetCUDA/kernels/rope/rope.cu
817 __global__ void rope(float *x, float *out, int seq_len, int N) {
818     int idx = blockIdx.x * blockDim.x + threadIdx.x;
819     float x1 = x[idx * 2];
820     float x2 = x[idx * 2 + 1];
821     int token_pos = idx / N; // 序列位置
822     int token_idx = idx % N; // 维度索引
823
824     // 频率计算:  $\theta_i = 1 / (10000^{(2i/d)})$ 
825     float exp_v = 1.0f / powf(10000.0f, 2 * token_idx / (N * 2.0f));
826     float sin_v = sinf(token_pos * exp_v);
827     float cos_v = cosf(token_pos * exp_v);
828
829     // 2D 旋转
830     float out1 = x1 * cos_v - x2 * sin_v;
831     float out2 = x1 * sin_v + x2 * cos_v;
832     out[idx * 2] = out1;
833     out[idx * 2 + 1] = out2;
834 }
835
836 // =====
837 // Phase 5: Mat Transpose — 矩阵转置 (Bank Conflict 专题)
838 // =====
839 // 面试要点 (Bank Conflict 专题):
840 //   - Shared memory 有 32 个 bank, 每个 bank 4 bytes (32-bit)
841 //   - 同一 warp 的多个线程访问同一 bank 的不同地址 → bank conflict
842 //   - n-way bank conflict: n 个线程冲突 → 访问串行化为 n 次
843 //   - 解决方案: PAD (在每行末尾加 1 个元素, 打破地址对齐)
844 //
845 // 转置的四步演进:
846 //   naive: 非合并写入 (列优先写) → 每个 warp 产生 32 次内存事务
847 //   shared: 写入 smem (行优先) → 从 smem 读取 (列优先) → 合并写入 gmem
848 //   BCF: smem 布局 [kWarpSize_S*4][kWarpSize_S+PAD] = [64][17], PAD=1 加在第二维消除 bank conflict
849 //   merge_write: 进一步将 4 次 separate store 合并为 1 次 float4 store
850 // 注: 本文件仅实现 Level 1(naive) 与 Level 4(BCF+merge_write), Level 2/3 省略
851
852 // 每个线程处理 1 个元素, block(16,16)
853 // 读: x 按行优先访问, warp 的 32 线程访问 32 个连续地址 → 合并读取 ✓
854 // 写: y 按列优先写入, 32 线程跨 row 行分散 → 非合并写入 x (32 次内存事务)
855 // Grid: ((col + 15) / 16, (row + 15) / 16, 1), 每线程 1 元素
856 // Block: (16, 16, 1)
857 // source: LeetCUDA/kernels/mat-transpose/mat_transpose.cu
858 __global__ void mat_transpose(float *x, float *y, const int row, const int col) {
859     const int c = blockIdx.x * blockDim.x + threadIdx.x; // col in x
860     const int r = blockIdx.y * blockDim.y + threadIdx.y; // row in x
861     if (r < row && c < col) {
862         // x[r][c] → y[c][r]; x stride=col, y (transposed) stride=row
863         y[c * row + r] = x[r * col + c];

```

```

864 }
865 }
866
867 // Grid: ((col + 15) / 16, (row + 63) / 64, 1), 每线程 4 元素(float4)
868 // Block: (16, 16, 1)
869 // 注意: 该版本默认按 float4 打包写回; 最适合 row 能按 4 对齐的场景
870 // source: LeetCUDA/kernels/mat-transpose/mat_transpose.cu
871 __global__ void mat_transpose_padded(
872     float *x, float *y, const int row, const int col) {
873     const int tx = threadIdx.x;
874     const int ty = threadIdx.y;
875
876     constexpr int TILE = 16;
877     constexpr int PAD = 1;
878     // Bank conflict fix: 每行多 1 个元素, 打破 32-bank 对齐
879     __shared__ float tile[TILE * 4][TILE + PAD]; // 64x16
880
881     // x 空间坐标; 每线程覆盖 4 行 (actual row = x_r * 4)
882     const int x_c = blockIdx.x * TILE + tx;
883     const int x_r = blockIdx.y * TILE + ty;
884
885     if (x_r * 4 < row && x_c < col) {
886         // Step 1: 4 rows x 1 col → smem (row-major);
887 #pragma unroll
888         for (int i = 0; i < 4; ++i) {
889             tile[ty * 4 + i][tx] = x[(x_r * 4 + i) * col + x_c];
890         }
891         __syncthreads();
892
893         // Step 2: Transposed read from smem
894         float4 reg_trans;
895         reg_trans.x = tile[tx * 4 + 0][ty];
896         reg_trans.y = tile[tx * 4 + 1][ty];
897         reg_trans.z = tile[tx * 4 + 2][ty];
898         reg_trans.w = tile[tx * 4 + 3][ty];
899
900         // y 空间坐标: y_r = x 的列, y_c = x 的行
901         const int y_r = blockIdx.x * TILE + ty; // = x col
902         const int y_c = (blockIdx.y * TILE + tx) * 4; // = x row
903
904         // Coalesced write: y[y_r][y_c .. y_c+3]
905         FLOAT4(y[y_r * row + y_c]) = reg_trans;
906     }
907 }
908
909 // =====
1
2 #pragma once
3 #include "base.cuh"
4 // sgemv.cuh: Phase 6 GEMV
5 // Phase 6: GEMV — 矩阵向量乘 (M or N = 1, 纯 memory-bound 算子, warp-per-row 策略)
6 // =====
7 // 面试要点:
8 // - GEMV 是典型的 memory-bound 算子:  $AI \approx O(1)$ , 瓶颈在内存带宽
9 // - 核心策略: warp-per-row (每个 warp 负责矩阵的一行)
10 // - 不同 K 值对应不同分块策略:
11 //   K=32 倍数: 一个 warp 的 32 线程恰好覆盖 K 维
12 //   K=128 倍数: 每个线程用 float4 处理 4 元素, warp 覆盖 128
13 //   K=16 < 32: 一个 warp 用不满, 用 kRowPerWarp=2 让每个 warp 处理 2 行
14 //
15 // a: M×K, x: K×1, y: M×1, 计算:  $y = a * x$ ; N = 1
16 // ---- SGEMV K32: 基础 warp-per-row ----

```

```

17 // 设计: block(32, 4), blockDim.x=kWarpSize=32 (K 需为 32 倍数时一轮覆盖, 否则内层循环 kNumWarps 次)
18 // grid(M/4), 每个 warp 负责一行
19 // K 为 32 的倍数时, warp 的 32 个线程恰好覆盖 K 维
20 // Grid: ((M + 3) / 4, 1, 1), 每 block 处理 4 行
21 // Block: (32, 4, 1), 每行 1 warp
22 // 注意: 该版本最适合 K 按 32 对齐; 当 K 更小时通常切到 K16 这类专用分支
23 // source: LeetCUDA/kernels/sgemv/sgemv.cu
24 __global__ void sgemv_k32(float *a, float *x, float *y, int M, int K) {
25     int tx = threadIdx.x; // 0~31
26     int ty = threadIdx.y; // 0~3
27     int lane = tx % kWarpSize; // 0~31
28     int m = blockIdx.x * blockDim.y + ty; // 全局行号
29     if (m < M) {
30         float sum = 0.0f;
31         // 沿 K 维的迭代数 = ceil(K/32), 每个 warp 要累加完整的K, 那么
32         // 每个thread就要负责累加NUM_ITERS个元素, NUM_ITERS = ceil(K/32)
33         const int NUM_ITERS = (K + kWarpSize - 1) / kWarpSize;
34 #pragma unroll
35         for (int w = 0; w < NUM_ITERS; ++w) {
36             // 假设K是32的整倍数, m * K 本行的起始地址, x: Kx1
37             int k = w * kWarpSize + lane;
38             sum += a[m * K + k] * x[k];
39         }
40         sum = warp_reduce_sum<kWarpSize>(sum);
41         // 每个 warp 处理一行, lane 0 写回结果
42         if (lane == 0)
43             y[m] = sum;
44     }
45 }
46
47 // ---- SGEMV K128: float4 向量化 ----
48 // 每个线程处理 4 个元素(float4), 一个 warp 覆盖 128 个元素
49 // Grid: ((M + 3) / 4, 1, 1)
50 // Block: (32, 4, 1)
51 // 注意: 该版本最适合 K 按 128 对齐, 且 x/a 的地址满足 float4 对齐
52 // source: LeetCUDA/kernels/sgemv/sgemv.cu
53 __global__ void sgemv_k128(float *a, float *x, float *y, int M, int K) {
54     int tx = threadIdx.x; // 0~31
55     int ty = threadIdx.y; // 0~3
56     int lane = tx % kWarpSize; // 0~31
57     int m = blockDim.y * blockIdx.x + ty;
58
59     if (m < M) {
60         float sum = 0.0f;
61         // 沿 K 维的迭代数 = ceil(K/128), 每个 warp 每轮用 float4 覆盖 128 个 K 元素
62         const int NUM_ITERS = (((K + kWarpSize - 1) / kWarpSize) + 4 - 1) / 4;
63 #pragma unroll
64         for (int w = 0; w < NUM_ITERS; ++w) {
65             int k = (w * kWarpSize + lane) * 4;
66             float4 reg_x = FLOAT4(x[k]);
67             float4 reg_a = FLOAT4(a[m * K + k]);
68             sum += (reg_a.x * reg_x.x + reg_a.y * reg_x.y + reg_a.z * reg_x.z +
69                 reg_a.w * reg_x.w);
70         }
71         sum = warp_reduce_sum<kWarpSize>(sum);
72         if (lane == 0)
73             y[m] = sum;
74     }
75 }
76
77 // ---- SGEMV K16: K < WarpSize, kRowPerWarp=2 ----
78 // 面试亮点: K=16 < 32, 一个 warp 可以处理多行
79 // kRowPerWarp=2, kNumLanePerRow=16, 前 16 个 lane 处理 row0, 后 16 个 lane 处理 row1

```

```

80 // Grid: ((M + 7) / 8, 1, 1), NUM_ROWS=8
81 // Block: (32, 4, 1)
82 // 注意: 这一版是面向 K=16 的专用写法; kRowPerWarp=2 时一个 warp 同时处理 2 行
83 // source: LeetCUDA/kernels/sgemv/sgemv.cu
84 template <const int kRowPerWarp = 2>
85 __global__ void sgemv_k16(float *A, float *x, float *y, int M, int K) {
86     constexpr int kNumLanePerRow = (kWarpSize + kRowPerWarp - 1) / kRowPerWarp; // 16
87     int tx = threadIdx.x;
88     int ty = threadIdx.y;
89     int lane = tx % kWarpSize;
90     int k = lane % kNumLanePerRow; // row0: 0~15, t0~t15; row1: 0~15, t16~t31
91     int m = (blockDim.y * blockDim.x + ty) * kRowPerWarp + lane / kNumLanePerRow;
92     if (m < M) {
93         float sum = A[m * K + k] * x[k];
94         // 按照kNumLanePerRow=16, 分2组各自做 warp reduce sum, k==0的lane写回结果
95         sum = warp_reduce_sum<kNumLanePerRow>(sum);
96         // 注意: 判断条件是 k == 0, 不是 lane == 0!
97         if (k == 0)
98             y[m] = sum;
99     }
100 }
101
102 // =====
1
#pragma once
#include "base.cuh"
// sgemm.cuh: Phase 7a SGEMM
// Phase 7: GEMM — 矩阵矩阵乘 (GPU 最重要的算子, 面试核心考点)
// =====
// 面试要点 (GEMM 优化五层金字塔):
// Level 1 — Tiling (分块 + shared memory): 将数据从 HBM 搬到 SMEM 复用
// Level 2 — Thread Tile (寄存器分块): 每个线程计算 TM×TN
// 个元素, 提高计算密度 Level 3 — Vectorize (向量化访存): float4/half2, 减少
// load/store 指令数 Level 4 — Tensor Core (MMA
// m16n8k16): 硬件矩阵乘单元, warp 级指令 Level 5 — Warp Specialization +
// TMA (WGMMMA m64n128k16): Hopper 异步执行
//
// 计算密度递进:
// Level 1: AI ≈ B_K / (2×sizeof) ≈ 32/8 = 4 → 仍是 memory-bound
// Level 2: AI ≈ TM×TN×B_K / (2×sizeof) ≈ 8×8×8/8 = 64 → compute-bound
// Level 4: Tensor Core 提供硬件加速的 256 FMA/cycle/warp → 大幅提升吞吐
//
// =====
// Phase 7a: SGEMM (非 Tensor Core 路径)
// =====
// ---- Level 1: SGEMM — Block Tile 32×32 + K Tile 32 ----
// 最基础的 tiling 实现, 演示 shared memory 的核心用法
// C = A x B, C[M, N] = A[M, K] x B[K, N]
// BM=BN=32, BK=32, block(32, 32), 一个线程计算 c 的一个元素
// Grid: ((N + 31) / 32, (M + 31) / 32, 1)
// Block: (32, 32, 1), 1024 线程
// source: LeetCUDA/kernels/sgemm/sgemm.cu
__global__ void sgemm(float *a, float *b, float *c, int M, int N, int K) {
    constexpr int BM = 32; // vec 版: 32x4 = 128
    constexpr int BN = 32; // vec 版: 32x4 = 128
    constexpr int BK = 32;
    __shared__ float s_a[BM][BK], s_b[BK][BN]; // 32x32x4=4KB smem, float = 4 bytes

    int bx = blockDim.x;
    int by = blockDim.y;
    int tx = threadIdx.x;
    int tid = threadIdx.y * blockDim.x + tx;

```

```

40
41 // 线程到 smem 的映射: 32x32 线程, 每个线程加载 a 和 b 各 1 个元素
42 // 技巧: 一般来说 “/” 表示线程不是连续排布的, “%” 表示线程是连续排布的
43 // 因此, 在需要考虑连续访问的维度使用 “%”, 比如, 连续的线程访问列方向连续的元素
44 // A[M, K], M的stride=K, K的stride=1 → 线程连续访问 K 维度 → 用 %,
45 // 线程不连续访问 M 维度 → 用 /;
46 int load_smem_a_m = tid / 32; // row 0~31 由 32 线程加载;
47 int load_smem_a_k = tid % 32; // col 0~31 由 32 线程加载;
48 int load_smem_b_k = tid / 32; // row 0~31 由 32 线程加载;
49 int load_smem_b_n = tid % 32; // col 0~31 由 32 线程加载;
50 int load_gmem_a_m = by * BM + load_smem_a_m; // gmem row;
51 int load_gmem_b_n = bx * BN + load_smem_b_n; // gmem col;
52
53 float sum = 0.f; // 遍历完整的K, slice K;
54 // 这里不用pragma unroll, 因为K不是编译器常量, 编译器无法展开循环
55 for (int bk = 0; bk < (K + BK - 1) / BK; ++bk) {
56     int load_gmem_a_k = bk * BK + load_smem_a_k; // A [M, K]
57     int load_gmem_a_addr = load_gmem_a_m * K + load_gmem_a_k;
58     s_a[load_smem_a_m][load_smem_a_k] = a[load_gmem_a_addr];
59     int load_gmem_b_k = bk * BK + load_smem_b_k; // B [K, N]
60     int load_gmem_b_addr = load_gmem_b_k * N + load_gmem_b_n;
61     s_b[load_smem_b_k][load_smem_b_n] = b[load_gmem_b_addr];
62     __syncthreads(); // 确保整个 smem tile 加载完毕
63
64 #pragma unroll
65     for (int k = 0; k < BK; ++k) {
66         int comp_smem_a_m = load_smem_a_m; // vec 版: 0~127, 0, 1, 2, ... (连续)
67         int comp_smem_b_n = load_smem_b_n; // vec 版: 0~127, 0, 4, 8, ... (间隔)
68         sum += s_a[comp_smem_a_m][k] * s_b[k][comp_smem_b_n];
69     }
70     __syncthreads(); // 确保 smem 不会在下一轮加载时被覆盖
71 }
72 int store_gmem_c_m = load_gmem_a_m; // vec 版: 0~127, 0, 1, 2, ... (连续) [128x128]
73 int store_gmem_c_n = load_gmem_b_n; // vec 版: 0~127, 0, 4, 8, ... (间隔)
74 int store_gmem_c_addr = store_gmem_c_m * N + store_gmem_c_n;
75 c[store_gmem_c_addr] = sum; // C [M, N] = A[M, K] x B[K, N]
76 }
77
78 // ---- Level 1+: SGEMM Vec4 — Block Tile 128x128 + K Tile 32 + Thread Tile 4x4 ----
79 // 在 Level 1 基础上引入两层优化:
80 // 1) float4 向量化加载: A/B 各用 1 条 128-bit load 取代 4 条 32-bit load
81 // 2) Thread Tile 4x4: 每线程计算 16 个 C 元素, 提升计算/访存比 (AI 从
82 // BK/2≈16 提升到 TM*TN*BK/2≈256), 减少线程总数带来的同步开销
83 // C = A x B, C[M, N] = A[M, K] x B[K, N], A/B 均 row-major
84 // BM=BN=128, BK=32, block(32, 32)=1024 线程, 每线程负责 4x4=16 个 C 元素
85 // 1024 x 16 = 16384 = 128 x 128 ✓
86 //
87 // 线程到 4x4 tile 的映射 (与加载映射解耦, 独立计算更清晰):
88 // m_tile = tid / 32 (0~31), 每 tile 4 行 → 行 [m_tile*4, m_tile*4+3], 覆盖 0~127
89 // n_tile = tid % 32 (0~31), 每 tile 4 列 → 列 [n_tile*4, n_tile*4+3], 覆盖 0~127
90 //
91 // 加载映射 (每线程 4 个元素, float4):
92 // A[128][32]: a_m = tid/8 (8 线程/行), a_k = (tid%8)*4 (4 列/线程) → 8x4=32 列 ✓
93 // B[32][128]: b_k = tid/32 (32 线程/行), b_n = (tid%32)*4 (4 列/线程) → 32x4=128 列 ✓
94 // row-major 下 A[m][k..k+3] 与 B[k][n..n+3] 均连续 → float4 load 合法
95 //
96 // △ Bank Conflict 提示 (面试加分点):
97 // s_b[32][128] 上 warp 内 32 线程按 stride=4 访问 (tid%32 决定列 0,4,8,...,124)
98 // → 每 4 个线程落同一 bank 不同地址 → 4-way bank conflict。生产代码可用
99 // s_b[BK][BN+1] PAD 打散, 这里保持最简布局便于讲解。
100 //
101 // Grid: ((N + 127) / 128, (M + 127) / 128, 1)
102 // Block: (32, 32, 1), 1024 线程

```

```

103 // 假设: M/N 为 128 的倍数, K 为 32 的倍数 (与 Level 1 naive 版一致的边界约定)
104 // source: LeetCUDA/kernels/sgemm/sgemm.cu (vec4 variant)
105 __global__ void sgemm_vec4(float *a, float *b, float *c, int M, int N, int K) {
106     constexpr int BM = 128;
107     constexpr int BN = 128;
108     constexpr int BK = 32;
109     __shared__ float s_a[BM][BK]; // 128*32*4 = 16KB, float = 4 bytes
110     __shared__ float s_b[BK][BN]; // 32*128*4 = 16KB
111
112     int bx = blockIdx.x;
113     int by = blockIdx.y;
114     int tx = threadIdx.x;
115     int tid = threadIdx.y * blockDim.x + tx; // 0~1023
116
117     // 加载 A: 每线程加载 s_a[a_m][a_k..a_k+3] 共 4 个元素
118     int load_smem_a_m = tid / 8; // 0~127, 8 线程/行
119     int load_smem_a_k = (tid % 8) * 4; // 0,4,...,28
120     // 加载 B: 每线程加载 s_b[b_k][b_n..b_n+3] 共 4 个元素
121     int load_smem_b_k = tid / 32; // 0~31, 32 线程/行
122     int load_smem_b_n = (tid % 32) * 4; // 0,4,...,124
123
124     int load_gmem_a_m = by * BM + load_smem_a_m;
125     int load_gmem_b_n = bx * BN + load_smem_b_n;
126
127     // 4x4 Thread Tile 基址 (独立于加载映射), 这里compute索引的计算逻辑要和
128     // load索引的计算逻辑分开, load/compute是可以独立索引的, 理解这点很重要。
129     // 目标C Tile为[BM,BN]=[128x128], 有32x32线程, 则每个线程处理4x4 tile
130     // 那么, 就可以不重不漏地覆盖[32x4,32x4]=[128x128]的大小
131     int comp_smem_a_m_base = (tid / 32) * 4; // 0,4,8,...,124
132     int comp_smem_b_n_base = (tid % 32) * 4; // 0,4,8,...,124
133
134     float sum[4][4] = {0.f};
135     for (int bk = 0; bk < (K + BK - 1) / BK; ++bk) {
136         int load_gmem_a_k = bk * BK + load_smem_a_k; // A [M, K]
137         int load_gmem_a_addr = load_gmem_a_m * K + load_gmem_a_k;
138         FLOAT4(s_a[load_smem_a_m][load_smem_a_k]) = FLOAT4(a[load_gmem_a_addr]); // s_a [BM,BK]
139         int load_gmem_b_k = bk * BK + load_smem_b_k; // B [K, N]
140         int load_gmem_b_addr = load_gmem_b_k * N + load_gmem_b_n;
141         FLOAT4(s_b[load_smem_b_k][load_smem_b_n]) = FLOAT4(b[load_gmem_b_addr]); // s_b [BK,BN]
142         __syncthreads();
143
144     #pragma unroll
145         for (int k = 0; k < BK; ++k) {
146             // 每次迭代加载 4 个 A 元素 + 4 个 B 元素, 再做 4x4=16 次 FMA
147             float a_vals[4] = {s_a[comp_smem_a_m_base + 0][k],
148                               s_a[comp_smem_a_m_base + 1][k],
149                               s_a[comp_smem_a_m_base + 2][k],
150                               s_a[comp_smem_a_m_base + 3][k]};
151             float b_vals[4] = {s_b[k][comp_smem_b_n_base + 0],
152                               s_b[k][comp_smem_b_n_base + 1],
153                               s_b[k][comp_smem_b_n_base + 2],
154                               s_b[k][comp_smem_b_n_base + 3]};
155
156         #pragma unroll
157         for (int i = 0; i < 4; ++i) {
158             #pragma unroll
159             for (int j = 0; j < 4; ++j) {
160                 sum[i][j] += a_vals[i] * b_vals[j];
161             }
162         }
163         __syncthreads();
164     }
165 }

```

```

166 // 存储 4x4: 每行 4 个元素连续 → 可用 float4 store (要求 N 为 4 的倍数以保证对齐)
167 int store_gmem_c_m = by * BM + comp_smem_a_m_base;
168 int store_gmem_c_n = bx * BN + comp_smem_b_n_base;
169 #pragma unroll
170 for (int i = 0; i < 4; ++i) {
171     int store_gmem_c_addr = (store_gmem_c_m + i) * N + store_gmem_c_n;
172     float4 reg_c;
173     reg_c.x = sum[i][0];
174     reg_c.y = sum[i][1];
175     reg_c.z = sum[i][2];
176     reg_c.w = sum[i][3];
177     FLOAT4(c[store_gmem_c_addr]) = reg_c;
178 }
179 }
180
181 // =====
1
1 #pragma once
2 #include "base.cuh"
3 // hgemm.cuh: Phase 7b-d HGEMM (MMA/Swizzle/CuTe/WGMMMA/TMA_MMA_WS)
4
5 // Phase 7b: HGEMM — Tensor Core 路径 (MMA m16n8k16 + WGMMMA m64n128k16)
6 // =====
7 // 面试要点:
8 //   - MMA (Matrix Multiply-Accumulate): Ampere+ 的 Tensor Core 指令
9 //   - m16n8k16 含义: M=16, N=8, K=16 的矩阵乘, 结果 [16x8] = [16x16]x[16x8]
10 //   - ldmatrix: 从 shared memory 加载 16x16 的矩阵片段到寄存器 (4 条 32-bit
11 //     寄存器)
12 //   - Multistage Pipeline: s2/s3/s4 个 stage, 用 cp.async 异步加载下一批数据
13 //   - Block Swizzle: 在 grid 维度做 swizzle, 改善 L2 cache locality
14 //
15 // ★ TN 布局详解 (面试高频考点):
16 //   TN 命名约定: 来自 BLAS 的 op(A) × op(B) 语义
17 //   BLAS 源自 Fortran, 默认列优先 (column-major) 存储
18 //   N = Normal (列优先, BLAS 原生格式)
19 //   T = Transposed (行优先, 相对 BLAS 来说是"转置过的")
20 //   第一个字母 → A 的 op, 第二个字母 → B 的 op
21 //   所以 TN 表示: A 是行优先 (相对 BLAS=Transposed), B 是列优先 (相对
22 //   BLAS=Normal) 即: C = op(A) × op(B) = AT × B? 不对! 在 row-major
23 //   视角下: TN = A row-major [M×K], BT row-major [N×K] (等价于 B col-major [K×N]) 在 cuBLAS
24 //   调用中: cublasGemmEx(..., CUBLAS_OP_T, CUBLAS_OP_N, ...)
25 //   T on A: BLAS 把 row-major 的 A 视为 AT, 传 T 表示"转置回去"
26 //   N on B: B 已经是 BLAS 原生的 col-major, 无需转置
27 //
28 //   记忆口诀: TN = A行(T) B列(N), 第一字母 A 第二字母 B
29 //   T = row-major (行优先, 对 BLAS 来说是 transposed)
30 //   N = col-major (列优先, BLAS native = normal)
31 //
32 //   LeetCUDA _nn 布局对比 (A/B 均 row-major 自然存储): C[M×N] = A[M×K] × B[K×N]
33 //   - A: row-major [M, K], B: row-major [K, N]
34 //   - 按 N=col-major/T=row-major 约定, 二者 BLAS 视角均为 T → cuBLAS 等效 (T,T)
35 //   - 问题: ldmatrix 默认加载 col-major, B 是 row-major 需要 .trans
36 //
37 //   TN 布局: C[M×N] = A[M×K] × B[K×N], B 以 BT=[N×K] row-major 存储 (A 行优先, B 列优先)
38 //   - A [M×K]: row-major → 全局索引 A[m*K + k], smem s_a[BM][BK]
39 //   - BT [N×K]: row-major → 全局索引 B[n*K+k] 即访问原 B 元素 (k,n) (△ 内维连续的是 K)
40 //   - 优势: BT 已是 row-major, ldmatrix 无需 .trans, 天然匹配 MMA row.col
41 //   MMA 指令: mma.sync.aligned.m16n8k16.row.col
42 //   - row.col: A 输入 row-major, B 输入 col-major → 与 TN 布局天然匹配
43 //
44 //   WGMMMA (Warp Group MMA, Hopper+):
45 //   - warpgroup 级指令 (128 threads = 4 warps), 异步执行
46 //   - m64n128k16: 一次处理 64x128x16 的矩阵乘 (总 tile 量 131072, 是 MMA m16n8k16 的 64 倍)

```

```

47 // - Warp Specialization: Producer(128 threads) 做 TMA 搬运, Consumer(128
48 // threads) 做计算
49 // - TMA: 硬件 DMA, ~零寄存器开销, 支持 1D~5D 寻址
50
51 // =====
52 // Phase 7b-2: HGEMM MMA — m16n8k16 + multistage pipeline + TN 布局 (统一循环版)
53 // =====
54 // 面试重点 — Tile Hierarchy:
55 // MMA Atom: m16n8k16 (1 条 MMA 指令处理的最小 tile)
56 // MMA Tile (more warps): 2x4=8 个 MMA atom → [2x16, 8x4]=[32,32]
57 // VAL Tile (more values): 4x4=16 expand → [32x4, 32x4]=[128,128]
58 // 实际: kMmaTileM=2, kMmaTileN=4, kValTileM=4, kValTileN=4
59 // → BM=16x2x4=128, BN=8x4x4=128, Warps=2x4=8, Threads=8x32=256
60 //
61 // ★ TN 布局 (T=A 行优先, N=B 列优先):
62 // - A[M][K]: row-major → 全局索引 A[m*K + k], smem s_a[BM][BK]
63 // - B[K][N]: col-major (等价于 B^T[N][K] row-major) → 全局索引 B[n*K+k]
64 // - ldmatrix A: 用 x4 (非转置), A row-major 原生匹配
65 // - ldmatrix B: 用 x2 (非转置), B^T row-major 逐行加载 = B 的列, 天然匹配 MMA row.col
66 // - MMA 指令: mma.sync.aligned.m16n8k16.row.col → 天然匹配 TN 布局
67 //
68 // ★ 统一循环设计 (k 从 0 开始, 消除尾端重复代码):
69 // - k 从 0 开始: 每次迭代 k 直接对应 tile k, sel = k % kStages (零偏移)
70 // - 加载语义为“预取未来”: 迭代 k 加载 tile (k+kStages-1) 供后续使用
71 // - cp.async 条件化: 仅当 k+kStages-1 < NUM_K_TILES 时加载
72 // - WAIT_GROUP 自适应: 满载期用 kStages-2, 尾部排空用 0
73 //
74 // ★ Block Swizzle: 把 C 的逻辑 tile 布局改写为紧凑的二维发射窗口, 增加 A/B tile 的 L2 reuse 机会。
75 // C tile 坐标为 (by, bx), by 沿 M 方向, bx 沿 N 方向; 一个 bx 读取同一块 B^T[bx*BN:(bx+1)*BN, :]
76 // (TN 布局中 B^T[N][K] 为 row-major), 一个 by 读取同一块 A[by*BM:(by+1)*BM, :]。
77 // 下图只画 4 个逻辑上相邻的 CTA; SM0~SM3 仅表示可能的发射序列, 不是硬件 SM 绑定或调度保证。
78 //
79 // No Swizzle: grid = (tiles_n, tiles_m, 1), blockIdx.x = bx。
80 // C-Matrix (同一 by 行; CTA 先在很宽的 N 方向范围内展开):
81 //
82 // N / bx → 0 1 2 3 ...
83 // M ↓
84 // by = 0 | SM0 | SM1 | SM2 | SM3 | ...
85 // | A 0,B0 | A 0,B1 | A 0,B2 | A 0,B3 |
86 // |-----|
87 //
88 // 同一 by 的 CTA 复用 A[by], 但分别读取 B^T 0、B^T 1 等不同 B tile。只有 scheduler
89 // 推进到下一条 by 行、再次遇到相同 bx 时, 才有机会复用 B; 当 tiles_n 很大时, 这段时间内
90 // L2 更容易被其他 B tile 挤占。
91 //
92 // Thread Block Swizzle: 先把 N 切成 S 个连续窗口; 一个 z slice 只含 grid_x 个 bx。
93 // 下图用 grid_x=2 展示一个窄窗口, scheduler 更早进入下一条 by 行:
94 //
95 // N / local x → 0 1
96 // M ↓
97 // by = 0 | SM0 | SM1 | → A 0; B^T 0, B^T 1
98 // |-----|
99 // by = 1 | SM2 | SM3 | → A 1; B^T 0, B^T 1
100 // |-----|
101 //
102 // 对于这个 2x2 窗口: 同一行横向复用 A (SM0/SM1、SM2/SM3), 同一列纵向复用 B
103 // (SM0/SM2、SM1/SM3)。真实 grid_x 不必等于 2, 但缩小它会缩短再次访问相同 bx 的距离。
104 // 此优化只提高 scheduler 在相近时间执行这些 CTA、命中 L2 的机会, 不保证 CTA 的 z 顺序、
105 // 缓存驻留或 L2 hit。
106 //
107 // Launch (kBlockSwizzle=false, 当前默认路径):
108 // grid = (ceil(N / BN), ceil(M / BM), 1), blockIdx.x 直接就是 N-tile 编号 bx。
109 // Launch (kBlockSwizzle=true, 需由 launch 端显式构造 3D grid):

```

```

110 // tiles_n = ceil(N / BN), S = ceil(N / 2048)
111 // grid = (ceil(tiles_n / S), ceil(M / BM), S)
112 // bx = blockIdx.z * grid.x + blockIdx.x, col_start = bx * BN。
113 // 例如 N=8192、BN=128: tiles_n=64, S=4, grid.x=16; z=0/1/2/3 分别覆盖
114 // bx=0..15/16..31/32..47/48..63, 即 columns [0,2048)/[2048,4096)/[4096,6144)/[6144,8192)。
115 // grid.x*grid.z 可能产生 bx>=tiles_n 的冗余 CTA。当前 kernel 仍要求 M/N/K 分别按 BM/BN/BK
116 // 对齐; ceil 只保证逻辑 tile 编号不遗漏, 并不使部分尾 tile 变为安全。
117 // Block: (256, 1, 1), 8 warps
118 // source: LeetCUDA/kernels/hgemm/mma/basic/hgemm_mma_stage_tn.cu
119 // =====
120 template <const int kMmaM = 16,           // MMA atom M dim (m16n8k16)
121           const int kMmaN = 8,           // MMA atom N dim
122           const int kMmaK = 16,          // MMA atom K dim, also BK tile = kMmaK
123           const int kMmaTileM = 2,       // warps along M, 2 → warp tile M = 32
124           const int kMmaTileN = 4,       // warps along N, 4 → warp tile N = 32
125           const int kValTileM = 4,       // value-repeat along M, BM = 16*2*4 = 128
126           const int kValTileN = 4,       // value-repeat along N, BN = 8*4*4 = 128
127           const int kStages = 3,         // cp.async pipeline depth
128           const int kBlockSwizzle = 0>   // 1 enables 3D grid swizzle for L2 locality
129 __global__ void __launch_bounds__(256)
130 hgemm_mma_stages_tn(half *A, half *B, half *C, int M, int N, int K) {
131     static_assert(kBlockSwizzle == 0 || kBlockSwizzle == 1, "kBlockSwizzle must be 0 or 1");
132     static_assert(kStages >= 2, "kStages must be >= 2 for cp.async pipeline");
133     // Block Swizzle: 0 时 bx=blockIdx.x; 1 时将 (blockIdx.z, blockIdx.x)
134     // 线性化为原始 N-tile 编号 bx=z*gridDim.x+x。by 始终是 M-tile 编号。
135     const int bx = ((int)kBlockSwizzle) * blockIdx.z * gridDim.x + blockIdx.x;
136     const int by = blockIdx.y;
137     constexpr int BM = kMmaM * kMmaTileM * kValTileM; // 16*2*4=128
138     constexpr int BN = kMmaN * kMmaTileN * kValTileN; // 8*4*4=128
139     constexpr int BK = kMmaK; // 16
140
141     // Dynamic shared memory: kStages 个 stage 的 A 和 B
142     // TN 布局: s_a[BM][BK]=[128][16](A row-major), s_b[BN][BK]=[128][16](B^T
143     // row-major, 即 B col-major [K×N] 在 smem 中按 B^T[N×K] 存储)
144     // 原始实现会按配置决定是否给 A/B 的 K 维加 PAD; 尤其 B 在 TN 布局下常见会额外
145     // 加 B_PAD 来打散 bank 映射, 避免按列访问时出现明显 bank conflict。这里先保留最简 PAD=0 版本。
146     extern __shared__ half smem[];
147     half *s_a = smem;
148     half *s_b = smem + kStages * BM * BK; // A 和 B 连续存放
149     constexpr int s_a_stage_offset = BM * BK; // 128*16
150     constexpr int s_b_stage_offset = BN * BK; // 128*16 △ BN(128)×BK(16) = B^T row-major
151     const int tid = threadIdx.y * blockDim.x + threadIdx.x;
152     const int warp_id = tid / kWarpSize; // 0~7
153     const int lane_id = tid % kWarpSize; // 0~31
154     // warp_m变化快(0->0,1->1), warp_n变化慢([0,1]->0,[2,3]->1,...), 因此,
155     // 这种2x4的MMA(Warp) layout是按照col major的顺序来排列MMA0~MMA7的
156     //
157     //           N direction (warp_n)
158     //           0      1      2      3
159     // M direction (warp_m) 0  MMA0   MMA2   MMA4   MMA6
160     //                      1  MMA1   MMA3   MMA5   MMA7
161     // MMA的排布方式不是唯一的, 现在这样排是因为结合MMA/VAL Tile之后, 刚好能覆盖整个
162     // C Tile[128,128]。只要调整MMA/VAL Tile, 这里的排布方式也可以跟着调整。要注意的
163     // 是, MMA0-7逻辑上是可以认为是并行执行的, 各自的计算结果累计加到对应的C Tile位置上。
164     const int warp_m = warp_id % kMmaTileM; // 0,1 (M 方向 2 个 warp) kMmaTileM = 2
165     const int warp_n = warp_id / kMmaTileM; // 0,1,2,3 (N 方向 4 个 warp)
166
167     // 线程到 global memory 的映射 (用于加载 A 和 B) 共 256 个线程
168     // TN 布局关键: A[m*K+k] 是 row-major, B^T[n*K+k] 是 row-major (内维连续的是 K)
169     int load_smem_a_m = tid / 2; // 0~127
170     int load_smem_a_k = (tid % 2 == 0) ? 0 : 8; // 0, 8
171     int load_smem_b_n = tid / 2; // 0~127 → B^T 的 N 方向 (row-major 的行)
172     int load_smem_b_k = (tid % 2 == 0) ? 0 : 8; // 0, 8 → B^T 的 K 方向 (row-major 的列)
173     int load_gmem_a_m = by * BM + load_smem_a_m; // C/A 全局行号 = M 方向的 tile 起始 + 线程偏移

```

```

173 int load_gmem_b_n = bx * BN + load_smem_b_n; // C/B 全局列号 = N 方向的 tile 起始 + 线程偏移
174 if (load_gmem_a_m >= M || load_gmem_b_n >= N)
175     return;
176
177 // 8个Warps(MMAs)的排布是M方向2个, N方向4个, 则MMA Atom一次性能处理的tile大小是[2*16, 4*8]=[32, 32].
178 // CUDA中每个block能放的线程数是有上限的 (warp数量有限), 一般为4或者8个warps。为了能处理更大的C tile,
179 // 需要把MMA Atom的tile再M方向和N方向各自重复4次, 得到[128, 128]的C block tile, 这就是Value Tile的作用。
180 // MMA Tile对应到cutlass cute中的TiledMMA的概念, Value Tile对应到cutlass cute中的PermuteMNK的概念。
181 uint32_t RC[kValTileM][kValTileN][2] = {0}; // 初始化为 0
182
183 // CVTA: 一次转换 smem 基地址, 避免每次 cp.async 都做转换
184 uint32_t smem_a_base_ptr = __cvta_generic_to_shared(s_a);
185 uint32_t smem_b_base_ptr = __cvta_generic_to_shared(s_b);
186
187 // 预加载前 (kStages-1) 个 stage
188 // TN 布局: A 的 gmem 索引用 m*K+k(row-major), B^T 用 n*K+k(row-major, 即 B col-major [K×N])
189 #pragma unroll
190 for (int k = 0; k < (kStages - 1); ++k) {
191     int load_gmem_a_k = k * BK + load_smem_a_k;
192     int load_gmem_a_addr = load_gmem_a_m * K + load_gmem_a_k; // A: [m][k]
193     uint32_t load_smem_a_ptr = (smem_a_base_ptr +
194         (k * s_a_stage_offset + load_smem_a_m * BK + load_smem_a_k) * sizeof(half)
195     );
196     CP_ASYNC_CG(load_smem_a_ptr, &A[load_gmem_a_addr], 16);
197
198     int load_gmem_b_k = k * BK + load_smem_b_k;
199     // B^T: [n][k] row-major (即 B[k][n] col-major) Δ
200     int load_gmem_b_addr = load_gmem_b_n * K + load_gmem_b_k;
201     uint32_t load_smem_b_ptr = (smem_b_base_ptr +
202         (k * s_b_stage_offset + load_smem_b_n * BK + load_smem_b_k) * sizeof(half)
203     );
204     CP_ASYNC_CG(load_smem_b_ptr, &B[load_gmem_b_addr], 16);
205
206     CP_ASYNC_COMMIT_GROUP();
207 }
208
209 CP_ASYNC_WAIT_GROUP(kStages - 2); // 允许有 kStages-2 个group未完成
210 __syncthreads();
211
212 // 统一循环: k 从 0 开始, 每次迭代负责 tile k (加载 + 计算合并为单循环)
213 const int NUM_K_TILES = div_ceil(K, BK);
214 // 此处不用pragma unroll, 因为 K 不是编译期常量, 因此NUM_K_TILES 也不是编译期常量
215 for (int k = 0; k < NUM_K_TILES; ++k) {
216     int smem_sel = k % kStages; // 计算 tile k 的 stage
217     int smem_sel_next = (k + kStages - 1) % kStages; // 预加载目标 stage
218
219     // 条件加载: 预加载 tile (k+kStages-1) 供将来使用。因为在prefetch loop中已经
220     // load了(kStages-1) 个 stage, 因此这里是从 tile (k+kStages-1) 开始预加载
221     // TN 布局: A 的 gmem 地址用 m*K+k (row-major), B^T 的 gmem 地址用 n*K+k
222     // (row-major, 内维连续的是 K)
223     if (k + kStages - 1 < NUM_K_TILES) {
224         int load_gmem_a_k = (k + kStages - 1) * BK + load_smem_a_k;
225         int load_gmem_a_addr = load_gmem_a_m * K + load_gmem_a_k; // A: row-major [m][k]
226         int load_gmem_b_k = (k + kStages - 1) * BK + load_smem_b_k;
227         int load_gmem_b_addr = load_gmem_b_n * K + load_gmem_b_k; // B^T: row-major [n][k]
228
229         uint32_t load_smem_a_ptr =
230             (smem_a_base_ptr + (smem_sel_next * s_a_stage_offset +
231                 load_smem_a_m * BK + load_smem_a_k) *
232                 sizeof(half));
233         CP_ASYNC_CG(load_smem_a_ptr, &A[load_gmem_a_addr], 16);
234
235         uint32_t load_smem_b_ptr =

```

```

236         (smem_b_base_ptr + (smem_sel_next * s_b_stage_offset +
237                             load_smem_b_n * BK + load_smem_b_k) *
238                             sizeof(half));
239     CP_ASYNC.CG(load_smem_b_ptr, &B[load_gmem_b_addr], 16);
240     CP_ASYNC.COMMIT_GROUP();
241 }
242
243 // ldmatrix: 从 smem_sel 加载 A 和 B 到寄存器
244 // TN 布局关键: A 用 x4 (非转置), 因为 A 是 row-major; B 用 x2 (非转置), smem 中
245 // B^T 为 row-major, 逐行加载即得 B 的列, 天然匹配 col-major B
246 uint32_t RA[kValTileM][4]; // [4][4], M方向4次repeat, A regs = 4 uint32_t per MMA
247 uint32_t RB[kValTileN][2]; // [4][2], N方向4次repeat, B regs = 2 uint32_t per MMA
248
249 // ldmatrix.x4: 加载 A 的 m16k16 片段 (row-major A, 非转置)
250 #pragma unroll
251 for (int i = 0; i < kValTileM; ++i) {
252     // {0,1} * (16 * 4) + i * 16 = {0,64} + {0,16,32,48} = {0,16,32,48,64,80,96,112}
253     // 其中 warp_m {0,1}; warp_m_0 offsets {0,16,32,48}; warp_m_1 offsets {64,80,96,112}
254     int warp_smem_a_m = warp_m * (kMmaM * kValTileM) + i * kMmaM;
255     // {0,16,32,...,112} + {0~15} = {0~127}, 按照col-major的顺序访问A的4个8x8 matrix (16x16)
256     // ldmatrix.{...}.x4.{...} 需要warp内32个线程都参与, 每个线程提供一个有效的, 不重叠的addr
257     int lane_smem_a_m = warp_smem_a_m + lane_id % 16; // t{0...15}=0~15, t{16...31}=0~15
258     int lane_smem_a_k = (lane_id / 16) * 8; // 0, 8, t{0...15}=0, t{16...31}=8
259     uint32_t lane_smem_a_ptr =
260         (smem_a_base_ptr +
261          (smem_sel * s_a_stage_offset + lane_smem_a_m * BK + lane_smem_a_k) *
262          sizeof(half));
263     LDMATRIX_X4(RA[i][0], RA[i][1], RA[i][2], RA[i][3], lane_smem_a_ptr);
264 }
265
266 // ldmatrix.x2: 加载 B 的 k16n8 片段 (非转置)
267 // 为什么不用 .trans? 因为 smem 中存的是 B^T row-major [N][K],
268 // ldmatrix 逐行加载 B^T 的行 = B 的列, 天然给出 col-major B fragment → 直接匹配 MMA row.col
269 #pragma unroll
270 for (int j = 0; j < kValTileN; ++j) {
271     // {0,...,3} * (8 * 4) + j * 8 = {0,32,64,96} + {0,8,16,24} = {0,8,...,120}
272     // warp_n_0 offsets {0,8,16,24}; warp_n_1 offsets {32,40,48,56}
273     // warp_n_2 offsets {64,72,80,88}; warp_n_3 offsets {96,104,112,120}
274     int warp_smem_b_n = warp_n * (kMmaN * kValTileN) + j * kMmaN;
275     // {0,8,...,120} + {0~7} = {0~127}, 按照row-major的顺序访问B^T的2个8x8 matrix (8x16)
276     // ldmatrix.{...}.x2.{...} 需要warp内前16个线程参与, 后16个线程传的addr会被忽略
277     int lane_smem_b_n = warp_smem_b_n + lane_id % 8; // t{0...7}=0~7, t{8...15}=0~7
278     int lane_smem_b_k = ((lane_id / 8) % 2) * 8; // t{0...7}=0, t{8...15}=8
279     uint32_t lane_smem_b_ptr =
280         (smem_b_base_ptr +
281          (smem_sel * s_b_stage_offset + lane_smem_b_n * BK + lane_smem_b_k) *
282          sizeof(half));
283     LDMATRIX_X2(RB[j][0], RB[j][1], lane_smem_b_ptr);
284 }
285
286 // MMA compute: 每个Warp(MMA) 在M方向重复kValTileM(4)次, 在N方向重复kValTileN(4)次
287 #pragma unroll
288 for (int i = 0; i < kValTileM; ++i) {
289     #pragma unroll
290     for (int j = 0; j < kValTileN; ++j) {
291         MMA16816(RC[i][j][0], RC[i][j][1], // C fragment
292                 RA[i][0], RA[i][1], RA[i][2], RA[i][3], // A fragment
293                 RB[j][0], RB[j][1], // B fragment
294                 RC[i][j][0], RC[i][j][1]);
295     }
296 }
297
298 // 自适应等待: 流水线满载期用 kStages-2, 尾部排空用 0

```

```

299 if (k + kStages - 1 < NUM_K_TILES) {
300     CP_ASYNC_WAIT_GROUP(kStages - 2);
301 } else { // 对于尾部的 k, 等待所有剩余的 cp.async 完成
302     CP_ASYNC_WAIT_GROUP(0);
303 }
304 __syncthreads();
305 }
306
307 // Epilogue: 寄存器 → global memory (通过 warp shuffle + 128-bit store)
308 {
309     for (int i = 0; i < kValTileM; ++i) {
310         // RC[kValTileM][kValTileN][2] 中 RC[...][...][2] 中保存了2个 uint32
311         // 寄存器, 每个uint32 寄存器表示两个临近的fp16值: {c0,c1}, 然后RC[...][...][0]
312         // 和RC[...][...][1]代表的是按照col-major排布的2个8x8子矩阵上 (不同物理行, 跨8x8)
313         // 同一个位置上的元素, 也就是, 实际代表了2个不同的行的元素, 因此要分开RC0, RC1;
314         // RC0表示第一行, 8个half, 可以用4个uint32寄存器来装; 同理, RC1表示第二行。
315         uint32_t RC0[kValTileN][4]; // 32 bits x 4 = 128 bits = 8 half
316         uint32_t RC1[kValTileN][4]; // 32 bits x 4 = 128 bits = 8 half
317         // =====
318         // MMA m16n8k16 C fragment — a single 16x8 matrix (registers per warp):
319         // Thread t holds 4 half values → RC[0] for rows 0-7, RC[1] for rows 8-15.
320         // Mapping: row = t/4, col-pair = t%4 (each uint32 = 2 half values).
321         //
322         //      cols: c0-1    c2-3    c4-5    c6-7
323         //      r0:  T0{c0,c1} T1{c2,c3} T2{c4,c5} T3{c6,c7}  }
324         //      r1:  T4{c0,c1} T5{c2,c3} T6{c4,c5} T7{c6,c7}  }
325         //      r2:  T8{c0,c1} T9{c2,c3} T10{c4,c5} T11{c6,c7} } RC[0]
326         //      ...
327         //      r7:  T28{c0,c1} T29{c2,c3} T30{c4,c5} T31{c6,c7} }
328         //      r8:  T0{c0,c1} T1{c2,c3} T2{c4,c5} T3{c6,c7}  }
329         //      r9:  T4{c0,c1} T5{c2,c3} T6{c4,c5} T7{c6,c7}  }
330         //      r10: T8{c0,c1} T9{c2,c3} T10{c4,c5} T11{c6,c7} } RC[1]
331         //      ...
332         //      r15: T28{c0,c1} T29{c2,c3} T30{c4,c5} T31{c6,c7} }
333         //
334         // Within a 4-lane group (e.g. T0-T3 for row 0):
335         // lane+0 holds {c0,c1}, lane+1 holds {c2,c3},
336         // lane+2 holds {c4,c5}, lane+3 holds {c6,c7}.
337         // shfl 从 lane+1/+2/+3 各收 2 个 half 到 lane+0, 4 次 shuffle 凑齐
338         // 一行 8 个 half = {c0,c1,c2,c3,c4,c5,c6,c7}, 再由 lane%4==0 128-bit store.
339         // =====
340 #pragma unroll
341         for (int j = 0; j < kValTileN; ++j) {
342             RC0[j][0] = RC[i][j][0];
343             RC1[j][0] = RC[i][j][1];
344             RC0[j][1] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 1);
345             RC0[j][2] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 2);
346             RC0[j][3] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 3);
347             RC1[j][1] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 1);
348             RC1[j][2] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 2);
349             RC1[j][3] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 3);
350         }
351         // 每 4 个 lane 中只有 lane 0 做 128-bit store
352         // lane_id / 4 → 行号映射 (每 4 个连续 lane 负责上8x8矩阵和下8x8矩阵各一行):
353         //
354         // | lane_id | lane_id/4 | RC[0] 的行 | RC[1] 的行 |
355         // |-----|-----|-----|-----|
356         // | 0~3   | 0        | row 0     | row 8     |
357         // | 4~7   | 1        | row 1     | row 9     |
358         // | 8~11  | 2        | row 2     | row 10    |
359         // | 12~15 | 3        | row 3     | row 11    |
360         // | 16~19 | 4        | row 4     | row 12    |
361         // | 20~23 | 5        | row 5     | row 13    |

```

```

362 // | 24~27 | 6 | row 6 | row 14 |
363 // | 28~31 | 7 | row 7 | row 15 |
364 // |-----|
365 if (lane_id % 4 == 0) {
366     // {0,1} * (16 * 4) + i * 16 = {0,64} + {0,16,32,48} = {0,16,32,48,64,80,96,112}
367     int store_warp_smem_c_m = warp_m * (kMmaM * kValTileM) + i * kMmaM; // smem row
368     // 这里用 lane_id / 4 → {0,1,2,3,4,5,6,7}, 对应 RC0/RC1 (+8) 的行号, 表示2个8x8矩阵的行号
369     int store_lane_gmem_c_m = by * BM + store_warp_smem_c_m + lane_id / 4; // gmem row
370 #pragma unroll
371     for (int j = 0; j < kValTileN; ++j) {
372         // {0,...,3} * (8 * 4) + j * 8 = {0,32,64,96} + {0,8,16,24} = {0,8,...,120}
373         int store_warp_smem_c_n = warp_n * (kMmaN * kValTileN) + j * kMmaN; // smem col
374         int store_lane_gmem_c_n = bx * BN + store_warp_smem_c_n; // gmem col
375         int store_gmem_c_addr_0 = store_lane_gmem_c_m * N + store_lane_gmem_c_n; // 1-th 8x8 matrix
376         int store_gmem_c_addr_1 = (store_lane_gmem_c_m + 8) * N + store_lane_gmem_c_n; // 2-th 8x8 matrix
377         // 128-bit store: 一次写入 8 个 half
378         *reinterpret_cast<float4*>(&C[store_gmem_c_addr_0]) =
379             *reinterpret_cast<float4*>(&RC0[j][0]);
380         *reinterpret_cast<float4*>(&C[store_gmem_c_addr_1]) =
381             *reinterpret_cast<float4*>(&RC1[j][0]);
382     }
383 }
384 }
385 }
386 }
387
388 // =====
389 template <const int kColStride = 16>
390 static __device__ __forceinline__ int swizzle(int i, int j) {
391     #if defined(NOTES_V2_ENABLE_SWIZZLE_V2)
392         return swizzle_v2_impl<kColStride>(i, j);
393     #else
394         return swizzle_v1_impl<kColStride>(i, j);
395     #endif
396 }
397
398 // =====
399 // Phase 7b-3: HGEMM MMA — m16n8k16 + multistage pipeline + TN 布局 + XOR swizzle
400 // + Register Double Buffering (generic kValTileK, default BK=64)
401 // =====
402 // 在 Phase 7b-2 的 smem XOR swizzle 基础上增加寄存器双缓冲:
403 // - kValTileK: 每个 BK tile 包含 kValTileK 个 kMmaK slice (BK = kMmaK * kValTileK)
404 // - 统一 BK tile: 所有 k_step slice 连续存放在同一个 BK 宽的 smem tile 中
405 // - RA[2][kValTileM][4] / RB[2][kValTileN][2]: 双份寄存器乒乓切换 (与 kValTileK 无关)
406 // - ldmatrix 与 MMA 计算重叠: 加载 k_step+1 的同时用另一组寄存器做 k_step 计算
407 // - 支持任意 kValTileK >= 2; 默认 kValTileK=4 (BK=64), kStages=2 推荐 64KB smem
408 //
409 // ★ smem 布局:
410 // s_a: [stage0][stage1]...[stage(kStages-1)], 每个 stage = BM × BK = 128 × BK
411 // s_b: 紧接 s_a 之后
412 // 每个 stage 内 k_step slice 列偏移 = k_step * kMmaK
413 //
414 // ★ 寄存器双缓冲时间线 (每 k 迭代内):
415 // Step 1: G→S cp.async 预取 stage(k+kStages-1) 全部 k_step (条件)
416 // Step 2: MMA k_step=0 (用 reg[load], 已在上轮 post-wait 中 ldmatrix)
417 // Step 3: for k_step=1..kValTileK-1: ldmatrix(列偏移 k_step*kMmaK)→reg[store], flip, MMA→reg[load]
418 // Step 4: adaptive wait + __syncthreads
419 // Step 5: S→R ldmatrix 预加载 stage(k+1) k_step=0 → reg[store] (条件)
420 //
421 // 参考: LeetCUDA/kernels/hgemm/mma/swizzle/hgemm_mma_stage_tn_swizzle_x2.cu
422 // Grid: ((N+127)/128/S, (M+127)/128, S), S=(N+2047)/2048, 3D block swizzle
423 // Block: (256, 1, 1), 8 warps
424 template <const int kMmaM = 16, // MMA atom M dim (m16n8k16)

```

```

425     const int kMmaN = 8,           // MMA atom N dim
426     const int kMmaK = 16,          // MMA atom K dim
427     const int kMmaTileM = 2,       // warps along M, 2 → warp tile M = 32
428     const int kMmaTileN = 4,       // warps along N, 4 → warp tile N = 32
429     const int kValTileM = 4,       // value-repeat along M, BM = 16*2*4 = 128
430     const int kValTileN = 4,       // value-repeat along N, BN = 8*4*4 = 128
431     const int kValTileK = 4,       // MMA_K slices per BK tile, BK = kMmaK * kValTileK
432     const int kStages = 2,         // cp.async pipeline depth (2 for kValTileK>=4 to fit smem)
433     const int kBlockSwizzle = 0>   // 1 enables 3D grid swizzle for L2 locality
434 __global__ void __launch_bounds__(256)
435     hgemm_mma_stages_tn_swizzle(half *A, half *B, half *C, int M, int N, int K) {
436     static_assert(kValTileK >= 2, "Register double buffering requires kValTileK >= 2");
437     static_assert(kBlockSwizzle == 0 || kBlockSwizzle == 1, "kBlockSwizzle must be 0 or 1");
438     static_assert(kStages >= 2, "kStages must be >= 2 for cp.async pipeline");
439     // Block Swizzle: 在 grid x 维度做 swizzle, 改善 L2 cache 局部性
440     const int bx = ((int)kBlockSwizzle) * blockIdx.z * gridDim.x + blockIdx.x;
441     const int by = blockIdx.y;
442     constexpr int BM = kMmaM * kMmaTileM * kValTileM; // 16*2*4=128
443     constexpr int BN = kMmaN * kMmaTileN * kValTileN; // 8*4*4=128
444     constexpr int BK = kMmaK * kValTileK;             // 16 * kValTileK
445
446     // kStages stages, each with BM×BK for A, BN×BK for B
447     // smem: kValTileK 个 kMmaK slice 连续存放在同一个 BK 宽 tile 中
448     extern __shared__ half smem[];
449     half *s_a = smem;
450     half *s_b = smem + kStages * BM * BK;
451     constexpr int s_a_stage_offset = BM * BK;
452     constexpr int s_b_stage_offset = BN * BK;
453
454     const int tid = threadIdx.y * blockDim.x + threadIdx.x;
455     const int warp_id = tid / kWarpSize;
456     const int lane_id = tid % kWarpSize;
457     const int warp_m = warp_id % kMmaTileM; // 0,1 (M 方向 2 个 warp) kMmaTileM = 2
458     const int warp_n = warp_id / kMmaTileM; // 0,1,2,3 (N 方向 4 个 warp)
459
460     // 线程到 global memory 的映射 (用于加载 A 和 B) 共 256 个线程
461     // TN 布局关键: A[m*K+k] 是 row-major, B^T[n*K+k] 是 row-major (内维连续的是 K)
462     // 注意: smem_a_k 和 smem_b_k 依然使用 0/8, 在后续的 kValTileK 循环中
463     // 会加上 k_step*kMmaK, 最终覆盖全部 BK 列
464     int load_smem_a_m = tid / 2;           // 0~127
465     int load_smem_a_k = (tid % 2 == 0) ? 0 : 8; // 0, 8
466     int load_smem_b_n = tid / 2;           // 0~127 → B^T 的 N 方向 (row-major 的行)
467     int load_smem_b_k = (tid % 2 == 0) ? 0 : 8; // 0, 8 → B^T 的 K 方向 (row-major 的列)
468     int load_gmem_a_m = by * BM + load_smem_a_m; // C/A 全局行号 = M 方向的 tile 起始 + 线程偏移
469     int load_gmem_b_n = bx * BN + load_smem_b_n; // C/B 全局列号 = N 方向的 tile 起始 + 线程偏移
470     if (load_gmem_a_m >= M || load_gmem_b_n >= N)
471         return;
472
473     // 8个Warps(MMAs)的排布是M方向2个, N方向4个, 则MMA Atom一次性能处理的tile大小是[2*16,4*8]=[32,32].
474     // CUDA中每个block能放的线程数是有上限的 (warp数量有限), 一般为4或者8个warps。为了能处理更大的C tile,
475     // 需要把MMA Atom的tile再M方向和N方向各自重复4次, 得到[128,128]的C block tile, 这就是Value Tile的作用。
476     // MMA Tile对应到cutlass cute中的TiledMMA的概念, Value Tile对应到cutlass cute中的PermuteMNK的概念。
477     uint32_t RC[kValTileM][kValTileN][2] = {0}; // 初始化为 0
478
479     uint32_t smem_a_base_ptr = __cvta_generic_to_shared(s_a);
480     uint32_t smem_b_base_ptr = __cvta_generic_to_shared(s_b);
481
482     // Prefetch (kStages-1) stages: load all k_step slices for each early stage
483     #pragma unroll
484     for (int k = 0; k < (kStages - 1); ++k) {
485     #pragma unroll
486         for (int k_step = 0; k_step < kValTileK; ++k_step) {
487             int load_gmem_a_k = k * BK + (k_step * kMmaK) + load_smem_a_k;

```

```

488     int load_gmem_a_addr = load_gmem_a_m * K + load_gmem_a_k;
489     int load_gmem_b_k = k * BK + (k_step * kMmaK) + load_smem_b_k;
490     int load_gmem_b_addr = load_gmem_b_n * K + load_gmem_b_k;
491
492     uint32_t load_smem_a_ptr =
493         (smem_a_base_ptr +
494          (k * s_a_stage_offset + load_smem_a_m * BK +
495           k_step * kMmaK +
496           swizzle<kMmaK>(load_smem_a_m, load_smem_a_k)) *
497           sizeof(half));
498     CP_ASYNC.CG(load_smem_a_ptr, &A[load_gmem_a_addr], 16);
499
500     uint32_t load_smem_b_ptr =
501         (smem_b_base_ptr +
502          (k * s_b_stage_offset + load_smem_b_n * BK +
503           k_step * kMmaK +
504           swizzle<kMmaK>(load_smem_b_n, load_smem_b_k)) *
505           sizeof(half));
506     CP_ASYNC.CG(load_smem_b_ptr, &B[load_gmem_b_addr], 16);
507 }
508 CP_ASYNC.COMMIT_GROUP();
509 }
510
511 CP_ASYNC.WAIT_GROUP(kStages - 2);
512 __syncthreads();
513
514 // Register double buffers: RA[0/1] for k_step=0/1 A data, RB[0/1] for B data
515 uint32_t RA[2][kValTileM][4];
516 uint32_t RB[2][kValTileN][2];
517 int reg_st_idx = 0; // write target for ldmatrix
518 int reg_ld_idx = 1; // read source for MMA
519
520 // Initial ldmatrix: load stage 0, S -> R, k_step=0 (0~15列) -> reg[0]
521 // 此时, reg_st_idx = 0, 对0位置的寄存器buffer做初始化。
522 #pragma unroll
523 for (int i = 0; i < kValTileM; ++i) {
524     int warp_smem_a_m = warp_m * (kMmaM * kValTileM) + i * kMmaM;
525     int lane_smem_a_m = warp_smem_a_m + lane_id % 16;
526     int lane_smem_a_k = (lane_id / 16) * 8;
527     uint32_t lane_smem_a_ptr =
528         (smem_a_base_ptr +
529          (0 * s_a_stage_offset + lane_smem_a_m * BK +
530           swizzle<kMmaK>(lane_smem_a_m, lane_smem_a_k)) *
531           sizeof(half));
532     LDMATRIX_X4(RA[reg_st_idx][i][0], RA[reg_st_idx][i][1],
533                RA[reg_st_idx][i][2], RA[reg_st_idx][i][3],
534                lane_smem_a_ptr);
535 }
536 #pragma unroll
537 for (int j = 0; j < kValTileN; ++j) {
538     int warp_smem_b_n = warp_n * (kMmaN * kValTileN) + j * kMmaN;
539     int lane_smem_b_n = warp_smem_b_n + lane_id % 8;
540     int lane_smem_b_k = ((lane_id / 8) % 2) * 8;
541     uint32_t lane_smem_b_ptr =
542         (smem_b_base_ptr +
543          (0 * s_b_stage_offset + lane_smem_b_n * BK +
544           swizzle<kMmaK>(lane_smem_b_n, lane_smem_b_k)) *
545           sizeof(half));
546     LDMATRIX_X2(RB[reg_st_idx][j][0], RB[reg_st_idx][j][1],
547                lane_smem_b_ptr);
548 }
549
550 // 统一循环: k 从 0 开始, 条件 G->S / S->R / wait 处理所有边界情况

```

```

551 // BK = kMmaK * kValTileK, 每个 k 迭代覆盖 BK 个 K 元素
552 const int NUM_K_TILES = div_ceil(K, BK);
553 for (int k = 0; k < NUM_K_TILES; ++k) {
554     int smem_sel = k % kStages;
555     int smem_sel_next = (k + kStages - 1) % kStages;
556
557     // G→S: 条件预取 stage(k+kStages-1) 全部 k_step slice
558     if (k + kStages - 1 < NUM_K_TILES) {
559 #pragma unroll
560         for (int k_step = 0; k_step < kValTileK; ++k_step) {
561             int load_gmem_a_k =
562                 (k + kStages - 1) * BK + (k_step * kMmaK) + load_smem_a_k;
563             int load_gmem_a_addr = load_gmem_a_m * K + load_gmem_a_k;
564             int load_gmem_b_k =
565                 (k + kStages - 1) * BK + (k_step * kMmaK) + load_smem_b_k;
566             int load_gmem_b_addr = load_gmem_b_n * K + load_gmem_b_k;
567
568             uint32_t load_smem_a_ptr =
569                 (smem_a_base_ptr +
570                  (smem_sel_next * s_a_stage_offset + load_smem_a_m * BK +
571                   k_step * kMmaK +
572                   swizzle<kMmaK>(load_smem_a_m, load_smem_a_k)) *
573                  sizeof(half));
574             CP_ASYNC.CG(load_smem_a_ptr, &A[load_gmem_a_addr], 16);
575
576             uint32_t load_smem_b_ptr =
577                 (smem_b_base_ptr +
578                  (smem_sel_next * s_b_stage_offset + load_smem_b_n * BK +
579                   k_step * kMmaK +
580                   swizzle<kMmaK>(load_smem_b_n, load_smem_b_k)) *
581                  sizeof(half));
582             CP_ASYNC.CG(load_smem_b_ptr, &B[load_gmem_b_addr], 16);
583         }
584         CP_ASYNC.COMMIT_GROUP();
585     }
586
587     // 内层 k_step 循环: flip → ldmatrix(k_step+1) → MMA, 乒乓交替
588     // 初始: st=0, ld=1, reg[0]=preload k_step=0; reg[1]=未初始化
589     // 每轮: flip→ldmatrix next k_step→reg[st], MMA reg[ld] (k_step+1 的 ldmatrix 条件化)
590
591 #pragma unroll
592     for (int k_step = 0; k_step < kValTileK; ++k_step) {
593         reg_st_idx ^= 1; // 0 → 1; 1 → 0
594         reg_ld_idx ^= 1; // 1 → 0; 0 → 1
595
596         if (k_step + 1 < kValTileK) {
597             int smem_k_offset = (k_step + 1) * kMmaK;
598 #pragma unroll
599             for (int i = 0; i < kValTileM; ++i) {
600                 int warp_smem_a_m = warp_m * (kMmaM * kValTileM) + i * kMmaM;
601                 int lane_smem_a_m = warp_smem_a_m + lane_id % 16;
602                 int lane_smem_a_k = (lane_id / 16) * 8;
603                 uint32_t lane_smem_a_ptr =
604                     (smem_a_base_ptr +
605                      (smem_sel * s_a_stage_offset + lane_smem_a_m * BK +
606                       smem_k_offset +
607                       swizzle<kMmaK>(lane_smem_a_m, lane_smem_a_k)) *
608                      sizeof(half));
609                 LDMATRIX_X4(RA[reg_st_idx][i][0], RA[reg_st_idx][i][1],
610                             RA[reg_st_idx][i][2], RA[reg_st_idx][i][3],
611                             lane_smem_a_ptr);
612             }
613 #pragma unroll

```

```

614     for (int j = 0; j < kValTileN; ++j) {
615         int warp_smem_b_n = warp_n * (kMmaN * kValTileN) + j * kMmaN;
616         int lane_smem_b_n = warp_smem_b_n + lane_id % 8;
617         int lane_smem_b_k = ((lane_id / 8) % 2) * 8;
618         uint32_t lane_smem_b_ptr =
619             (smem_b_base_ptr +
620              (smem_sel * s_b_stage_offset + lane_smem_b_n * BK +
621               smem_k_offset +
622               swizzle<kMmaK>(lane_smem_b_n, lane_smem_b_k)) *
623               sizeof(half));
624         LDMATRIX_X2(RB[reg_st_idx][j][0], RB[reg_st_idx][j][1],
625                     lane_smem_b_ptr);
626     }
627 }
628
629 #pragma unroll
630 for (int i = 0; i < kValTileM; ++i) {
631     #pragma unroll
632     for (int j = 0; j < kValTileN; ++j) {
633         HMMMA16816(RC[i][j][0], RC[i][j][1],
634                    RA[reg_ld_idx][i][0], RA[reg_ld_idx][i][1],
635                    RA[reg_ld_idx][i][2], RA[reg_ld_idx][i][3],
636                    RB[reg_ld_idx][j][0], RB[reg_ld_idx][j][1],
637                    RC[i][j][0], RC[i][j][1]);
638     }
639 }
640 }
641
642 // 自适应等待: 满载期用 kStages-2, 尾部排空用 0
643 if (k + kStages - 1 < NUM_K_TILES) {
644     CP_ASYNC_WAIT_GROUP(kStages - 2);
645 } else {
646     CP_ASYNC_WAIT_GROUP(0);
647 }
648 __syncthreads();
649
650 // 从下一阶段加载 k_step=0 数据, 供下一轮 k 迭代使用, 此时 reg_st_idx=0
651 if (k + 1 < NUM_K_TILES) {
652     smem_sel = (k + 1) % kStages; // old smem_sel + 1
653     #pragma unroll
654     for (int i = 0; i < kValTileM; ++i) {
655         int warp_smem_a_m = warp_m * (kMmaM * kValTileM) + i * kMmaM;
656         int lane_smem_a_m = warp_smem_a_m + lane_id % 16;
657         int lane_smem_a_k = (lane_id / 16) * 8;
658         uint32_t lane_smem_a_ptr =
659             (smem_a_base_ptr +
660              (smem_sel * s_a_stage_offset + lane_smem_a_m * BK +
661               swizzle<kMmaK>(lane_smem_a_m, lane_smem_a_k)) *
662               sizeof(half));
663         LDMATRIX_X4(RA[reg_st_idx][i][0], RA[reg_st_idx][i][1],
664                     RA[reg_st_idx][i][2], RA[reg_st_idx][i][3],
665                     lane_smem_a_ptr);
666     }
667     #pragma unroll
668     for (int j = 0; j < kValTileN; ++j) {
669         int warp_smem_b_n = warp_n * (kMmaN * kValTileN) + j * kMmaN;
670         int lane_smem_b_n = warp_smem_b_n + lane_id % 8;
671         int lane_smem_b_k = ((lane_id / 8) % 2) * 8;
672         uint32_t lane_smem_b_ptr =
673             (smem_b_base_ptr +
674              (smem_sel * s_b_stage_offset + lane_smem_b_n * BK +
675               swizzle<kMmaK>(lane_smem_b_n, lane_smem_b_k)) *
676               sizeof(half));

```

```

677     LDMATRIX_X2(RB[reg_st_idx][j][0], RB[reg_st_idx][j][1],
678                 lane_smem_b_ptr);
679 }
680 }
681 }
682
683 // Epilogue: 复用 RA[2][4][4] 寄存器做 warp shuffle → 128-bit collective store
684 // 做RC的warp shuffle正好需要[2][4][4]的寄存器空间, RA[2][4][4]正好可以复用
685 for (int i = 0; i < kValTileM; ++i) {
686 #pragma unroll
687     for (int j = 0; j < kValTileN; ++j) {
688         RA[0][j][0] = RC[i][j][0];
689         RA[1][j][0] = RC[i][j][1];
690         RA[0][j][1] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 1);
691         RA[0][j][2] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 2);
692         RA[0][j][3] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 3);
693         RA[1][j][1] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 1);
694         RA[1][j][2] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 2);
695         RA[1][j][3] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 3);
696     }
697     if (lane_id % 4 == 0) {
698         int store_warp_smem_c_m = warp_m * (kMmaM * kValTileM) + i * kMmaM;
699         int store_lane_gmem_c_m = by * BM + store_warp_smem_c_m + lane_id / 4;
700 #pragma unroll
701         for (int j = 0; j < kValTileN; ++j) {
702             int store_warp_smem_c_n = warp_n * (kMmaN * kValTileN) + j * kMmaN;
703             int store_lane_gmem_c_n = bx * BN + store_warp_smem_c_n;
704             int store_gmem_c_addr_0 =
705                 store_lane_gmem_c_m * N + store_lane_gmem_c_n;
706             int store_gmem_c_addr_1 =
707                 (store_lane_gmem_c_m + 8) * N + store_lane_gmem_c_n;
708             *reinterpret_cast<float4*>(&C[store_gmem_c_addr_0]) =
709                 *reinterpret_cast<float4*>(&RA[0][j][0]);
710             *reinterpret_cast<float4*>(&C[store_gmem_c_addr_1]) =
711                 *reinterpret_cast<float4*>(&RA[1][j][0]);
712         }
713     }
714 }
715 }
716
717 // =====
718 // Phase 7c: HGEMM CuTe — CUTLASS CuTe DSL 实现 (SM80+, Tensor Core 全自动调度)
719 // =====
720 // 面试要点 (CuTe vs 手写 MMA PTX) :
721 // - CuTe (CUTLASS Templates) 是 NVIDIA CUTLASS 3.x 的核心 DSL, 提供编译期
722 //   Tensor 抽象, 自动推导 MMA、Copy、Swizzle 的线程-数据映射
723 // - 与手写 MMA PTX (Phase 7b) 的对比:
724 //   - 手写 MMA: 手动计算 smem 地址、手动 ldmatrix/mma PTX、手动 swizzle、手动 epilogue shuffle
725 //   - CuTe: 声明 TiledMMA / TiledCopy / SmemLayout, 编译器自动推导线程-数据映射,
726 //     cute::copy / cute::gemm 自动展开为高效指令序列
727 // - 核心抽象 ("CuTe 五要素") :
728 //   1. Tensor: 全局/共享/寄存器数据的逻辑视图 = (ptr, shape, stride)
729 //   2. TiledMMA: 描述 MMA 指令 + warp/warpgroup 排布 + value tile 的 compile-time type
730 //   3. TiledCopy: 描述数据搬运 (G→S, S→R, R→S, S→G) 的线程-元素映射
731 //   4. Layout + Swizzle: 描述 smem 的数据排布和 bank conflict 消除
732 //   5. Pipeline: cp.async + multistage, 由 cute::copy 自动管理
733 //
734 // CuTe kernel 的参数推导链 (launch wrapper 负责实例化, kernel 只接收实例化后的类型) :
735 //   MMA Atom (SM80_16x8x16_F16F16F16F16_TN)
736 //   → TiledMMA (EURepeat=MxNxK, ValTile=MxNxK)
737 //   → TiledCopy (G2S/S2R/R2S/S2G, 每种有 ThrLayout + ValLayout)
738 //   → SmemLayout (Atom + Swizzle + tile_to_shape + Stage)
739 //

```

```

740 // 调参口诀（面试速记）：
741 // BM/BN/BK 越大 → smem 越大 → occupancy 越低，但 K 循环更少 → 指令开销更低
742 // Swizzle<B,M,S> 的核心不是改变逻辑 Tensor 的 shape，而是把一维 offset
743 // 重新解释为一个二维逻辑空间，再把二维坐标映射回 bank-conflict-free 的物理 offset：
744 // 1. 连续的  $2^M$  个元素组成二维空间中的一个基本元素；（元素宽度）
745 // 2. 连续的  $2^S$  个基本元素组成一行；（列数）
746 // 3. 二维空间包含  $2^B$  行；（行数）
747 // 4. 对二维坐标做列置换：icol' = irow ^ icol；
748 // 5. 保留基本元素内部的低 M 位，再将置换后的二维坐标编码回 offset。
749 // 因此 M 描述基本元素宽度，S 描述二维空间的列数，B 描述二维空间的行数。
750 // CuTe 的位级实现等价于：
751 // offset' = offset ^ ((offset & YYY) >> S),
752 // YYY = ((1 << B) - 1) << (M + S)。
753 // 对 Swizzle<3,3,3>：每个基本元素有  $2^3=8$  个值，每行有  $2^3=8$  个基本元素，
754 // 二维空间有  $2^3=8$  行；元素地址位 [6:8] XOR 到 [3:5]，低 3 位保持不变。
755 // 一个完整 swizzle 周期覆盖  $2^{(M+S+B)}=2^9=512$  个元素（FP16 下为 1024B）。
756 // kStage：2=最低延迟，3/4=更好隐藏延迟 → 权衡 smem 占用
757 // EURepeat：在 M/N/K 方向重复 MMA atom 的执行单元布局，决定 TiledMMA 的线程排布
758 // 与逻辑 tile 组织；MMA atom 越多，需要的线程数就越多
759 //
760 // ★ 与手写 MMA Swizzle (Phase 7b-3) 的本质区别：
761 // - 手写 swizzle：在 smem 地址计算时手动 XOR 列索引
762 // - CuTe Swizzle：SmemLayout 声明式指定地址位的 XOR pattern，cute::copy 自动应用
763 // - CuTe 优势：类型安全，编译器可在编译期推导映射；布局变更通常只需修改类型定义
764 //
765 // ★ 本实现的 Tile 配置：
766 // - gmem/smem tile：BM=128, BN=256, BK=32, kStage=2；这是 A/B tile 的目标 shape。
767 // - MMA atom：SM80_16x8x16_F16F16F16F16_TN；EURepeat=2x2x1, MMA_P_T=32x32x16。
768 // - TiledMMA 的逻辑 MMA tile 是 32x32x16, G2S/S2R copy 再把它映射到更大的 BMxBN tile；
769 // BN=256 不是由“8x2x16”直接推导出的 MMA tile，而是本 wrapper 选择的 B tile 尺寸。
770 // - Smem Swizzle<3,3,3>：512-element address period 的 XOR 重排，针对 ldmatrix
771 // 的访问模式降低 bank conflict；512 是 swizzle 周期，不等于 A atom 的逻辑元素数。
772 // - Threads：size(MMA{}) = 128 (4 warps × 2x2 EU repeat)。
773 // - Dynamic smem：Stage=2 时 A[128x32] + B[256x32] 共 49,152 bytes (约 48 KiB)。
774 //
775 // 参考：LeetCUDA/kernels/hgemm/cutlass/hgemm_mma_stage_tn_cute.cu
776 // 参考：https://zhuanlan.zhihu.com/p/671419093 (CuTe Swizzle 详解)
777 // =====
778
779 #if defined(NOTES_V2_ENABLE_CUTE)
780
781 #include <cute/tensor.hpp>
782 #include <cutlass/arch/barrier.h>
783 #include <cutlass/device_kernel.h>
784 #include <cutlass/numeric_conversion.h>
785
786 // =====
787 // Phase 7c-1: CuTe HGEMM Kernel — TN 布局 + Smem Swizzle + Multistage Pipeline
788 // =====
789 // TN 布局：C[MxN] = A[MxK] × B^T[NxK]
790 // - A[M][K] row-major, B^T[N][K] row-major (等价 B col-major [KxN])
791 // - CuTe 中通过 make_tensor(make_gmem_ptr(ptr), shape, stride) 描述
792 //
793 // Pipeline 流程 (stage 是 K tile 的循环缓冲槽，不是一条完整计算链)：
794 // 1. PREFETCH：预加载 kStage-1 个 K tile (G→S via cp.async)，填满 stage。
795 // 2. 主循环 over K tiles：
796 // a. 从当前 stage 做 S→R: cute::copy(...) → ldmatrix；
797 // b. 对当前 K slice 做 MMA: cute::gemm(...) → mma.sync；
798 // c. 在处理当前 tile 的首个 MMA slice 时，异步提交下一个 tile 的 G→S；
799 // d. 当前 tile 的最后一个 K slice 完成后等待并切换 smem stage。
800 // 3. Epilogue：R→S (UniversalCopy 写入 C scratchpad) → S→G (128-bit store)。
801 //
802 // 面试对比 — 手写 MMA vs CuTe 代码量：

```

```

803 // 手写: ldmatrix PTX 地址计算 + swizzle XOR + mma PTX + epilogue shuffle (~200行)
804 // CuTe: partition + copy + gemm (~80行), 其余由 launch wrapper 的类型推导完成
805 template <typename T,                                // element type (half)
806         int BM, int BN, int BK,                      // block tile: BM=128, BN=256, BK=32
807         int kStage,                                  // cp.async pipeline depth (2)
808         typename TiledMMA,                          // MMA: SM80_16x8x16_F16F16F16_TN, EURepeat=2x2x1
809         typename G2SCopyA,                          // G→S A: SM80_CP_ASYNC_CACHEGLOBAL<uint128_t>
810         typename G2SCopyB,                          // G→S B: same as G2SCopyA
811         typename SmemLayoutA,                      // smem A: Swizzle<3,3,3> + 8×BK atom → (BM,BK,kStage)
812         typename SmemLayoutB,                      // smem B: same atom → (BN,BK,kStage)
813         typename SmemLayoutC,                      // C scratchpad: Swizzle<3,3,3> + 32×32 atom, pipe=4
814         typename S2RCopyAtomA,                     // S→R A: SM75_U32x4_LDSM_N (ldmatrix.x4)
815         typename S2RCopyAtomB,                     // S→R B: same as S2RCopyAtomA
816         typename R2SCopyAtomC,                     // R→S C: UniversalCopy<int> (32-bit store)
817         typename S2GCopyAtomC,                     // S→G C: UniversalCopy<uint128_t> (128-bit wide store)
818         typename S2GCopyC,                         // S→G TiledCopy: ThrLayout{32,4} × ValLayout{1,8}
819         const int BlockSwizzle = 0>                // 1 enables 3D grid swizzle for L2 locality
820 __global__ void hgemm_mma_stages_tn_cute(T *Aptr, T *Bptr, T *Dptr, int m,
821                                         int n, int k) {
822     using namespace cute;
823     static_assert(BlockSwizzle == 0 || BlockSwizzle == 1, "BlockSwizzle must be 0 or 1");
824     static_assert(kStage >= 2, "kStage must be >= 2 for cp.async pipeline");
825
826     // 动态 shared memory: KStage 个 A/B tile; C epilogue 复用当前 A stage 的空间。
827     extern __shared__ T shm_data[];
828     T *Ashm = shm_data;
829     T *Bshm = shm_data + cute::cosize(SmemLayoutA{});
830
831     int idx = threadIdx.x;
832     // BlockSwizzle: 0/1 控制是否启用 thread block swizzle, 改善 L2 cache 局部性
833     int ix = ((int)BlockSwizzle) * blockIdx.z * gridDim.x + blockIdx.x;
834     int iy = blockIdx.y; // M 方向 block 索引
835     // 当前 kernel 只有 CTA 级边界判断, 没有 tile 内的逐元素 predication; 调用方需保证
836     // M % BM == 0、N % BN == 0、K % BK == 0, 才能避免边界 tile 的越界访问或未写回。
837     if (iy * BM >= m || ix * BN >= n) return;
838
839     // CuTe Tensor 抽象: (ptr, shape, stride) 三元组描述数据布局
840     // make_stride(leading_dim, Int<1>{}) → row-major: 同行相邻元素步长=1, 跨行步长=leading_dim
841     Tensor A = make_tensor(make_gmem_ptr(Aptr), make_shape(m, k),
842                             make_stride(k, Int<1>{}));
843     Tensor B = make_tensor(make_gmem_ptr(Bptr), make_shape(n, k),
844                             make_stride(k, Int<1>{}));
845     Tensor D = make_tensor(make_gmem_ptr(Dptr), make_shape(m, n),
846                             make_stride(n, Int<1>{}));
847
848     // local_tile: 按 tiler 切出当前 CTA 的逻辑 tile; 返回 Tensor 仍保留未切出的 remainder mode。
849     // make_coord(iy, _): 固定 M/N 方向的 CTA 坐标, _ 保留 K 方向的 tile remainder:
850     // gA: (BM, BK, num_tile_k), gB: (BN, BK, num_tile_k), gD: (BM, BN)。
851     Tensor gA = local_tile(A, make_tile(Int<BM>{}, Int<BK>{}), make_coord(iy, _));
852     Tensor gB = local_tile(B, make_tile(Int<BN>{}, Int<BK>{}), make_coord(ix, _));
853     Tensor gD = local_tile(D, make_tile(Int<BM>{}, Int<BN>{}), make_coord(iy, ix));
854
855     // Shared memory Tensor: 按 SmemLayout 解读 smem 区域
856     auto sA = make_tensor(make_smem_ptr(Ashm), SmemLayoutA{}); // (BM, BK, kStage)
857     auto sB = make_tensor(make_smem_ptr(Bshm), SmemLayoutB{}); // (BN, BK, kStage)
858
859     // TiledMMA partition: 将 MMA 的 A/B/C tile 映射到各线程的寄存器 fragment
860     TiledMMA tiled_mma;
861     auto thr_mma = tiled_mma.get_slice(idx);
862     auto tCrA = thr_mma.partition_fragment_A(gA(_, _, 0)); // (MMA, MMA_M, MMA_K)
863     auto tCrB = thr_mma.partition_fragment_B(gB(_, _, 0)); // (MMA, MMA_N, MMA_K)
864     auto tCrD = thr_mma.partition_fragment_C(gD);           // (MMA, MMA_M, MMA_N)
865     clear(tCrD); // 累加器清零

```

```

866 // G2S TiledCopy: 描述 global → shared memory 的数据搬运 (128-bit cp.async)。
867 G2SCopyA g2s_tiled_copy_a;
868 auto g2s_thr_copy_a = g2s_tiled_copy_a.get_slice(idx);
869 auto tAgA_copy = g2s_thr_copy_a.partition_S(gA); // (CPY, CPY_M, CPY_K, num_tile_k)
870 auto tAsA_copy = g2s_thr_copy_a.partition_D(sA); // (CPY, CPY_M, CPY_K, kStage)
871
872 G2SCopyB g2s_tiled_copy_b;
873 auto g2s_thr_copy_b = g2s_tiled_copy_b.get_slice(idx);
874 auto tBgB_copy = g2s_thr_copy_b.partition_S(gB); // (CPY, CPY_N, CPY_K, num_tile_k)
875 auto tBsB_copy = g2s_thr_copy_b.partition_D(sB);
876
877 // S2R TiledCopy: 描述 shared → register 的数据搬运 (使用 ldmatrix)
878 // make_tiled_copy_A/B: 根据 TiledMMA 自动推导 S2R copy 的线程-数据映射
879 auto s2r_tiled_copy_a = make_tiled_copy_A(S2RCopyAtomA{}, tiled_mma);
880 auto s2r_thr_copy_a = s2r_tiled_copy_a.get_slice(idx);
881 auto tAsA = s2r_thr_copy_a.partition_S(sA); // (CPY, CPY_M, CPY_K, kStage)
882 auto tCrA_view = s2r_thr_copy_a.retile_D(tCrA); // (CPY, CPY_M, CPY_K) — 与 rA 寄存器布局对齐
883
884 auto s2r_tiled_copy_b = make_tiled_copy_B(S2RCopyAtomB{}, tiled_mma);
885 auto s2r_thr_copy_b = s2r_tiled_copy_b.get_slice(idx);
886 auto tBsB = s2r_thr_copy_b.partition_S(sB);
887 auto tCrB_view = s2r_thr_copy_b.retile_D(tCrB);
888
889 // PREFETCH: 预加载前 (kStage - 1) 个 K tile, 填满循环缓冲; 每次 copy 提交一个异步 G→S 操作。
890 int itile_to_read = 0, ismem_read = 0, ismem_write = 0;
891 #pragma unroll
892 for (int istage = 0; istage < kStage - 1; ++istage) {
893     cute::copy(g2s_tiled_copy_a, tAgA_copy(_, _, _, istage),
894               tAsA_copy(_, _, _, istage));
895     cute::copy(g2s_tiled_copy_b, tBgB_copy(_, _, _, istage),
896               tBsB_copy(_, _, _, istage));
897     cp_async_fence();
898     ++itile_to_read;
899     ++ismem_write;
900 }
901 cp_async_wait<kStage - 2>();
902 __syncthreads();
903
904 // K 维第一层循环 — 按 BK 大小将 K 切分为 num_k_tiles 个 tile。
905 // 外层 k_tile 遍历这些 BK-tile, 内层 k_step 遍历 tile 内的 MMA_K slice。
906 // 当前实现使用整除, 要求 K % BK == 0; 没有为尾部 K tile 做 predication/padding。
907 int num_k_tiles = k / BK;
908 // 循环前预加载首轮数据: 将第一个 K tile 的第 0 个 K slice 从 smem 加载到寄存器。
909 cute::copy(s2r_tiled_copy_a, tAsA(_, _, 0, ismem_read), tCrA_view(_, _, 0));
910 cute::copy(s2r_tiled_copy_b, tBsB(_, _, 0, ismem_read), tCrB_view(_, _, 0));
911
912 #pragma unroll 1
913 for (int k_tile = 0; k_tile < num_k_tiles; ++k_tile) {
914     // num_k_steps: BK tile 内 K 方向的 MMA 迭代次数, 取自 fragment 的第三 mode (K-mode)。
915     //
916     // 为什么只显式展开 K? MMA_M 和 MMA_N 去哪了?
917     // tCrA=(MMA, MMA_M, MMA_K), tCrB=(MMA, MMA_N, MMA_K), tCrD=(MMA, MMA_M, MMA_N)
918     // - K 方向: 每个 k_step 需要从 smem 加载**不同的** A/B 数据 (ldmatrix),
919     // 所以 K 循环必须显式写, 每次迭代做 S→R copy + cute::gemm。
920     // - M/N 方向: 同一个 k_step 内, 所有 M、N 位置的 MMA atom 共享
921     // 同一批寄存器数据。cute::gemm 根据 tiled_mma 的类型信息 (EURepeat=2×2)
922     // 在编译期自动展开为覆盖全部 (MMA_M, MMA_N) 对的 mma.sync 指令序列,
923     // 无需手动写 M/N 循环。——这等价于手写 MMA 中的:
924     // for (i=0; i<kValTileM; ++i)
925     //     for (j=0; j<kValTileN; ++j)
926     //         HMMA16816(RC[i][j][0], RC[i][j][1], ...);
927     //
928

```

```

929 // CUTLASS 推导链 (mma_atom.hpp) :
930 // make_tiled_mma 将 MMA_P_T::<2> (即 kMmaPK) 存入 PermutationMnK
931 // → TiledMMA::permutation_mnk<2>() 返回 kMmaPK
932 // → thrfrg_A 对 A(M,K) 按 (permutation_mnk<0>(), permutation_mnk<2>())
933 // 做 logical_divide, 即按 (kMmaPM, kMmaPK) tile 化 K 维:
934 // K 被分为 BK/kMmaPK 个 perm-K slice
935 // 随后按 AtomShape_MNK<2> (MMA_K_atom) 做 zipped_divide
936 // 每个 perm-K 含 kMmaEURepeatK 个 atom-K slice
937 // → partition_A 选当前线程后, K-mode size = BK / kMmaPK = 32 / 16 = 2
938 // 本配置: kMmaPK=16 (MMA_K_atom=16 × kMmaEURepeatK=1), BK=32 → num_k_steps=2
939 int num_k_steps = size<2>(tCrA);
940
941 #pragma unroll
942 for (int k_step = 0; k_step < num_k_steps; ++k_step) {
943     int k_step_next = (k_step + 1) % num_k_steps;
944
945     if (k_step == num_k_steps - 1) {
946         // 等待下一个 K tile 的 smem 数据就绪, 然后切换到下一个 stage。
947         cp_async_wait<kStage - 2>();
948         __syncthreads();
949         ismem_read = (ismem_read + 1) % kStage;
950     }
951
952     // S→R: 加载下一组 A/B fragment 到寄存器。
953     //
954     // cute::copy 原生支持多 mode——理论上可以一次加载全部 K slice:
955     // cute::copy(s2r_tiled_copy_a, tAsA(_, _, _, ismem_read), tCrA_view);
956     // 但这里刻意逐 k_step_next 加载, 目的是做**寄存器双缓冲**：
957     // 当前 k_step 做 MMA 计算时, ldmatrix 预加载 k_step_next 的数据,
958     // 让 S→R 延迟与 MMA 计算重叠。
959     // 如果一次性加载全部 K slice, S→R 和 MMA 就会彻底串行。
960     cute::copy(s2r_tiled_copy_a, tAsA(_, _, k_step_next, ismem_read),
961               tCrA_view(_, _, k_step_next));
962     cute::copy(s2r_tiled_copy_b, tBsB(_, _, k_step_next, ismem_read),
963               tCrB_view(_, _, k_step_next));
964
965     if (k_step == 0) {
966         // G→S: 提交下一个 K tile (整个 BK) 的异步拷贝。
967         if (itile_to_read < num_k_tiles) {
968             cute::copy(g2s_tiled_copy_a, tAgA_copy(_, _, _, itile_to_read),
969                       tAsA_copy(_, _, _, ismem_write));
970             cute::copy(g2s_tiled_copy_b, tBgB_copy(_, _, _, itile_to_read),
971                       tBsB_copy(_, _, _, ismem_write));
972             ++itile_to_read;
973             ismem_write = (ismem_write + 1) % kStage;
974         }
975         cp_async_fence();
976     }
977
978     // MMA: cute::gemm 内部根据 tiled_mma 的类型信息, 自动展开为覆盖
979     // 全部 (MMA_M, MMA_N) 对的 mma.sync 指令序列, 累加到 tCrD。
980     cute::gemm(tiled_mma, tCrD, tCrA(_, _, k_step), tCrB(_, _, k_step), tCrD);
981 }
982 }
983
984 // Epilogue: D 寄存器 → shared memory → global memory。
985 // 使用当前 A stage 作为 C scratchpad, 避免为 epilogue 单独分配完整的 C shared memory。
986 // 这里是“复用 storage”, 不是把 A Tensor 转换成 C Tensor:
987 // sA 的逻辑 shape 是 (BM, BK, kStage)=(128,32,2), 而 sA(_,_,ismem_read)
988 // 只选出其中一个 A-stage 的 storage 起点; sC 随后以完全独立的 SmemLayoutC
989 // 解释这同一段地址。当前 wrapper 的固定配置下:
990 // 一个 A stage: 128 × 32 = 4096 个 half;
991 // C scratchpad: 32 × 32 × 4 = 4096 个 half。

```

```

992 // 两者容量刚好相等, 但 logical shape 和 layout 不是同一个: A 的 layout atom
993 // 是 8x32, C 的 layout atom 是 32x32, 二者都含 Swizzle<3,3,3>, 却不能把
994 // C 数据再按 SmemLayoutA 读取。launch wrapper 的 static_assert 用当前配置
995 // 检查 C scratchpad 能放进一个 A pipe; 此处只别名该单个 stage, 不是整块双缓冲 sA。
996 //
997 // mainloop 已结束, 不会再通过 sA 的 A/B load view 消费这个 pipe 的旧 A 数据。
998 // 从这一行开始, 对这块地址的所有访问都通过 SmemLayoutC 派生的 sC 完成: R2S
999 // 用它写 C, S2G 用它读 C。因此 C 的 swizzle/address mapping 在写和读两侧一致。
1000 auto sC = make_tensor(sA(_, _, ismem_read).data(), SmemLayoutC{});
1001
1002 // R2S TiledCopy: 寄存器 → shared memory (使用 UniversalCopy 做类型转换)
1003 // R2SCopyAtomC = Copy_Atom<UniversalCopy<int>, T>; 以 32-bit 粒度搬运 T 数据
1004 auto r2s_tiled_copy_c = make_tiled_copy_C(R2SCopyAtomC{}, tiled_mma);
1005 auto r2s_thr_copy_c = r2s_tiled_copy_c.get_slice(idx);
1006 // tCrD 是 TiledMMA 产生的 accumulator fragment, 逻辑 shape 为
1007 // (MMA, MMA_M, MMA_N)。它的寄存器元素已经就位, 但该 layout 是为 MMA
1008 // accumulator 服务的, 并不直接按 R2S copy atom 的 source-value 顺序表达。
1009 //
1010 // retile_S 中的 S 指“这个 TiledCopy 的 source side”, 不是 shared memory。
1011 // 它基于 r2s_tiled_copy_c 的 reference layout 创建一个共享 tCrD.data() 的
1012 // zero-copy layout view, 将逻辑坐标表示为 (CPY, CPY_M, CPY_N)。这一步不读取
1013 // 或写入寄存器/共享内存, 不进行 dtype conversion, 也不交换 lane 间数据; 真正的
1014 // R2S 数据搬运发生在下面的 cute::copy(r2s_tiled_copy_c, ..., tCsC_r2s(...))。
1015 // 可将它与主循环的 s2r_thr_copy_a.retire_D(tCrA) 对照: 二者都是为了让已有
1016 // fragment 的 per-thread value ordering 匹配某个 TiledCopy 的 source/destination
1017 // 角色, 而不是另一次 memory transfer。
1018 // tCrD: (MMA, MMA_M, MMA_N) -> retire -> tCrC_r2s: (CPY, CPY_M, CPY_N)
1019 auto tCrC_r2s = r2s_thr_copy_c.retire_S(tCrD); // (CPY, CPY_M, CPY_N)
1020 // get_slice(idx) 已固定 CTA 内当前 logical copy thread; partition_D 再按
1021 // R2S TiledCopy 的 destination mapping 切分 sC。推导链:
1022 // MMA_P_T = Tile<kMmaPM, kMmaPN, kMmaPK> = Tile<32, 32, 16>
1023 // → TiledMMA::tile_size<0>(mma) = 32, tile_size<1>(mma) = 32
1024 // → make_tiled_copy_C 的 tiler = (32, 32)
1025 // → SmemLayoutC 的逻辑 M×N = (kMmaPM, kMmaPN) = (32, 32) 恰好等于该 tiler
1026 // → partition_D: zipped_divide(sC, tiler=(32,32)) 后 M/N 维无 remainder
1027 // → 结果 shape 的 M/N remainder = _1, _1
1028 // 最后一维 pipe=4 来自 SmemLayoutC 的第三维 kSmemLayoutCBatch, 而不是
1029 // “thread-level tensor 自动只剩 CPY”这一条规则。_1 表示该逻辑 mode 的 extent
1030 // 是 1, 并非该 mode 被删除或 C tile 没有 M/N 覆盖。
1031 auto tCsC_r2s = r2s_thr_copy_c.partition_D(sC); // (CPY, _1, _1, pipe)
1032
1033 // S2G TiledCopy: shared memory → global memory (128-bit store)
1034 // S2GCopyAtomC(Copy_Atom<UniversalCopy<cute::uint128_t>, T>) -> S2GCopyC
1035 S2GCopyC s2g_tiled_copy_c;
1036 auto s2g_thr_copy_c = s2g_tiled_copy_c.get_slice(idx);
1037 // tCsC_s2g 与 tCsC_r2s 都从同一个 sC 产生, 因而保留相同的 pipe=4; 前者是
1038 // S2G source mapping, 后者是 R2S destination mapping。tCgC_s2g 则面向完整
1039 // gD=(BM,BN)=(128,256), 相对于 S2G copy tiler 仍有非平凡的 M/N repetition,
1040 // 所以它保留 CPY_M、CPY_N。这里的 CPY/CPY_M/CPY_N 都是编译期 copy-partition
1041 // 的逻辑 mode, 不应直接理解为固定的 lane 数、warp 数或物理连续维度。
1042 auto tCsC_s2g = s2g_thr_copy_c.partition_S(sC); // (CPY, _1, _1, pipe)
1043 auto tCgC_s2g = s2g_thr_copy_c.partition_D(gD); // (CPY, CPY_M, CPY_N)
1044
1045 // group_modes<B,E> 将 layout 的半开区间 [B,E) 组合成一个嵌套 mode:
1046 // group_modes<1,3>(Tensor(CPY, CPY_M, CPY_N))
1047 // → Tensor(CPY, (CPY_M, CPY_N))
1048 // 它只重组 Tensor 的 layout, 不分配内存、不移动 data(), 也不改变元素访问的物理地址。
1049 // 第二个 mode 仍然保存原来 CPY_M/CPY_N 的层次关系, 只是现在可以用一个坐标
1050 // i+j 访问这个组合 mode; 因此这里的 group_modes 更准确地说“合并 mode”,
1051 // 而不是把底层数据拷贝或无条件 flatten 成普通一维数组。
1052 // 从 type/shape 结构看, 结果确实是 Tensor(CPY, (CPY_M, CPY_N)); 但 grouped
1053 // mode 的 cardinality 是两个子 mode 的乘积, 因此 size<1>(…) 可作为一个标量
1054 // 范围遍历其全部 logical coordinate。于是 (_, i+j) 中的 i+j 是该 nested mode

```

```

1055 // 的线性 logical coordinate, 不是在 C++ 中对二维 tuple 做加法, 更不是 raw
1056 // pointer offset; 最终物理地址仍由各自 Tensor 的 grouped layout 计算。
1057 //
1058 // 这里为什么要对 tCrC_r2s 和 tCgC_s2g 同时 group?
1059 //   - tCrC_r2s: 每个线程持有的寄存器 C fragment, 原始形状为 (CPY, CPY_M, CPY_N);
1060 //   - tCgC_s2g: 该线程对应的 global-memory C 输出位置, 形状与源 Tensor 对齐;
1061 //   - 两者使用相同的 [1,3] 合并后, 源/目的 Tensor 仍保持相同的坐标空间,
1062 //     cute::copy 可以按相同的 (CPY, i+j) 坐标完成寄存器到 global 的对应搬运。
1063 //
1064 // tCsC_r2s/tCsC_s2g 没有 group 第 3 个 pipe mode, 因为它们还需要显式保留
1065 // shared-memory C scratchpad 的 pipeline 维度; step=size<3>(tCsC_r2s) 表示
1066 // 每一轮可以复用的 C shared-memory pipeline 深度。外层 i 在合并后的 C 元素空间中
1067 // 按 step 前进, 内层 j 选择当前 pipeline slot: 寄存器先写入 sC(_,0,0,j),
1068 // 再从同一个 slot 写回 global memory。
1069 // 当前 kSmemLayoutCBatch=4, 因此 step=4。代码未对 i+j 做尾部 predication,
1070 // 这个循环依赖当前静态 layout 中 grouped extent 能被 pipe depth 整除; 不应将
1071 // “每轮正好处理 4 个 fragment” 误认为任意 tile/copy 配置都自动成立的通用规则。
1072 auto tCgC_s2gx = group_modes<1, 3>(tCgC_s2g);
1073 auto tCrC_r2sx = group_modes<1, 3>(tCrC_r2s);
1074
1075 int step = size<3>(tCsC_r2s); // C scratchpad 的 pipe 数 (由 kSmemLayoutCBatch=4 指定)
1076 // 双层循环的语义 (注意 step 与 CPY_N 无关):
1077 //   group_modes<1,3> 之后, tCrC_r2sx 的 shape 为 (CPY, (CPY_M, CPY_N))。
1078 //   size<1>(tCrC_r2sx) = CPY_M * CPY_N, 即该线程需处理的 C fragment 总数。
1079 //   外循环以 step=4 为步长, 内循环 j=0..3 每次处理 1 个 fragment, 将其写入
1080 //   第 j 号 C scratchpad pipe slot, 再读出写回 global。
1081 //
1082 //   这里 step=4 是 scratchpad pipeline 深度, 不是 CPY_N。CPY_N 的值取决于
1083 //   TiledMMA 的 C-layout 如何将 (128,256) 的 gD tile 分配到 128 个线程上,
1084 //   通常 ≠ 4。循环之所以成立, 是因为 grouped mode 用线性坐标 i+j 遍历整个
1085 //   CPY_M * CPY_N 的扁平空间——它不再区分 M 和 N 方向, 也不要求 CPY_N == step。
1086 //   唯一前提是 CPY_M * CPY_N 能被 step 整除 (当前静态 layout 编译期保证)。
1087 //
1088 //   第 j 号 pipe slot 在 sC 上的物理地址由 R2S partition_D / S2G partition_S
1089 //   各自推导; 因为二者都从同一个 sC (SmemLayoutC) 出发, pipe mode 保持一致,
1090 //   写入和读取命中同一段 shared memory。
1091 #pragma unroll
1092 for (int i = 0; i < size<1>(tCrC_r2sx); i += step) {
1093     // 每轮处理 step 个 fragment, 内层 j 选 pipe slot。
1094 #pragma unroll
1095     for (int j = 0; j < step; ++j) {
1096         // 这两行是 R2R staging: make_tensor_like<T> 分配当前线程私有、拥有 storage
1097         // 的 register Tensor; 普通 cute::copy 将 accumulator fragment materialize
1098         // 成输出元素类型 T 的临时 payload。
1099         //
1100         // cute::copy (无 copy-atom 的普通版) 内部按元素做
1101         //   dst(i) = static_cast<T>(static_cast<SrcType>(src(i)))
1102         // 因此当 accumulator 与 T 类型不同时, 它自动完成 dtype 转换; 当二者相同
1103         // (如本 kernel 的 f16 accumulator + T=half), cast 退化为 no-op。
1104         //
1105         // 即使类型一致, t 还有一个不可省略的作用: layout 归一化。
1106         // tCrC_r2sx(_, i+j) 的 layout 是 retile_S → group_modes → slice 层层
1107         // 叠加的 composed layout, 而 make_tensor_like<T> + cute::copy 把它
1108         // 物化到一个拥有简单 strides 的 owning register Tensor 中。后续
1109         // r2s_tiled_copy_c 的 copy_unpack 需要按 copy-atom 的 val-layout
1110         // 拆分 source 元素, 简单 owning layout 比复杂的 composed layout 更容易
1111         // 被编译器推导内联。上游同源实现将其概括为"cope with accumulator and
1112         // output data type difference", 实际承担了类型适配和 layout 归一化两个职责。
1113         //
1114         // 这不是 retile_S 的一部分, 也不是 warp shuffle: 源/目的都在当前线程的
1115         // register 中, 代码没有显式 __shfl_sync。不能仅根据这段 C++ 断言最终 SASS
1116         // 绝不会含 shuffle; 但语义上的跨线程 regrouping 不由此 R2R copy 承担, 而是
1117         // 由随后的 R2S -> shared memory -> S2G 路径完成。若特定 accumulator/output

```

```

1118 // type 与 R2S copy-atom contract 已直接兼容, 可以设计直接 R2S 的变体; 本例
1119 // 保持同源实现的通用 staging 写法。
1120 auto t = make_tensor_like<T>(tCrC_r2sx(_, i + j));
1121 cute::copy(tCrC_r2sx(_, i + j), t);
1122 // R2S 的 copy atom 才在此处将 thread-local payload 写入 j 号 C scratchpad
1123 // slot; SmemLayoutC 定义写入后的跨线程地址重组。
1124 //
1125 // cute::copy(r2s_tiled_copy_c, src, dst) 的调用链路:
1126 //   r2s_tiled_copy_c 是 TiledCopy, 但继承自 Copy_Atom<UniversalCopy<int>, T>
1127 //   → 匹配 copy(Copy_Atom<...> const&, src, dst) 重载 [copy.hpp:~L190]
1128 //   → src(t) 与 dst(tCsC_r2s(_, 0, 0, j)) 都是 rank-1 → 直接 copy_atom.call
1129 //   → copy_atom.call 内部: 若 size(src)==NumValSrc (atom 编译期 val-count),
1130 //     走 copy_unpack; 否则递归剥 mode, 最终都落到 UniversalCopy<int>::copy
1131 //     → 硬件层面就是逐 uint32_t 的 register-to-smem store [arch/copy.hpp:~L46]
1132 //
1133 // 这里没有任何运行时"查找表": t 的第 i 个元素之所以对应 dst 的第 i 个
1134 // shared memory offset, 是因为 pre-partition 阶段已经把映射烧进 layout 了:
1135 //   - t 的 layout 来自 retile_S → group_modes → slice → make_tensor_like
1136 //     → 元素顺序已按 R2S atom 的 source-side value ordering 排好
1137 //   - tCsC_r2s 的 layout 来自 r2s_thr_copy_c.partition_D(sC)
1138 //     → TiledCopy::tidfrg_D 用 LayoutCopy_TV + Tiler_MN + ValLayoutDst
1139 //     计算了当前线程每个 val 在 smem 中的物理 offset
1140 //   两者共享同一个 copy atom 的 ValLayoutSrc/ValLayoutDst 定义,
1141 //   因此逐元素 copy 天然把正确的数据写到了正确的地址。
1142 cute::copy(r2s_tiled_copy_c, t, tCsC_r2s(_, 0, 0, j));
1143 }
1144 // 这是 CTA 范围的 rendezvous: 所有线程完成当前批 R2S 写入后, S2G 才能从
1145 // sC 的重组布局读取完整数据。它承担了手写版中显式跨 lane 拼接所需的协作边界。
1146 __syncthreads();
1147
1148 // S→G: 从同一个 pipe slot 读回 global memory, 保持源/目的坐标一一对应。
1149 #pragma unroll
1150 for (int j = 0; j < step; ++j) {
1151   // S2GCopyC 的 atom 为 UniversalCopy<uint128_t>: 与 R2S 的 UniversalCopy<int>
1152   // (32-bit) 不同, 它一次搬运 128-bit (8 个 half), 映射为一条 wide store
1153   // (如 st.global.v4.f32 或 st.global.b128)。
1154   //
1155   // cute::copy(s2g_tiled_copy_c, src, dst) 的调用链路:
1156   //   s2g_tiled_copy_c 是 TiledCopy, 继承自 Copy_Atom<UniversalCopy<uint128_t>, T>
1157   //   → 匹配 copy(Copy_Atom<...> const&, src, dst) 重载 [copy.hpp:~L190]
1158   //   → src=tCsC_s2g(_, 0, 0, j) 与 dst=tCgC_s2gx(_, i+j) 都是 rank-1
1159   //   → 直接 copy_atom.call(src, dst)
1160   //   → copy_atom.call 内部: size(src)==NumValSrc → copy_unpack
1161   //     → 逐 128-bit chunk 调用 UniversalCopy<uint128_t>::copy
1162   //     → 硬件层面就是 shared-memory-to-global wide store [arch/copy.hpp:~L46]
1163   //
1164   // S2G 的 pre-partition 与 R2S 对称, 但 source/destination 角色互换:
1165   //   - tCsC_s2g 来自 s2g_thr_copy_c.partition_S(sC): TiledCopy::tidfrg_S
1166   //     用 LayoutCopy_TV + Tiler_MN + ValLayoutSrc 计算了当前线程每个 val
1167   //     在 sC (shared memory) 中的物理 offset。
1168   //   S2GCopyC 的 tiler = product(ThrLayout{32,4}, ValLayout{1,8})
1169   //     = (32×1, 4×8) = (32, 32) — 恰好与 R2S tiler 相同,
1170   //   因此 tCsC_s2g 也是 (CPY, _1, _1, pipe), _1,_1 来自 sC=(32,32,4) 无
1171   //   M/N remainder。
1172   //   - tCgC_s2gx(_, i+j) 来自 s2g_thr_copy_c.partition_D(gD) → group_modes
1173   //     → slice。gD=(128,256) 相对于 tiler (32,32) 有 4×8 个 tile repetition,
1174   //     因此 tCgC_s2g 为 (CPY, CPY_M, CPY_N), group_modes<1,3> 合并 M/N
1175   //     repetition 后得到 (CPY, (CPY_M,CPY_N)), 再用线性坐标 i+j 切片。
1176   //   两者共享同一个 copy atom 的 ValLayoutSrc/ValLayoutDst 定义,
1177   //   因此逐 chunk copy 天然从正确的 smem 地址读到正确的 gmem 地址。
1178   cute::copy(s2g_tiled_copy_c, tCsC_s2g(_, 0, 0, j), tCgC_s2gx(_, i + j));
1179 }
1180 // 所有线程都完成当前 pipe slots 的 S2G 读取后, 下一轮 R2S 才能覆盖这 4 个

```

```

1181 // scratchpad slots, 避免一部分线程仍在读取旧数据时被其他线程提前重写。
1182 __syncthreads();
1183 }
1184
1185 // 与 hgemm_mma_stages_tn 的手写 epilogue 对照: 手写版必须显式处理 MMA fragment
1186 // 的物理分布, 使用 RC0/RC1 暂存, 再用 __shfl_sync 从同一 warp 的相邻 lane 收齐
1187 // 8 个 half, 并由 lane_id % 4 == 0 的线程做 float4 (128-bit) global store。
1188 // CuTe 版没有把这些 lane 映射写死在 kernel 源码中: retile_S 表达 accumulator
1189 // fragment 与 R2S copy 的对应, R2S/sC/S2G 通过 shared-memory scratchpad 完成
1190 // CTA 范围的数据重组, S2GCopyC 以 uint128_t 表达 wide store。两者的共同目标都是
1191 // 将分散在计算线程寄存器中的 C fragment 变为适合连续 global-memory 写回的布局;
1192 // 区别是手写版直接编排 shuffle/store, CuTe 版将映射交给 TiledMMA/TiledCopy 类型推导。
1193 }
1194
1195 // =====
1196 // Phase 7c-2: CuTe HGEMM Launch Wrapper — 类型实例化 + Grid/Block 配置
1197 // =====
1198 // CuTe 的核心设计理念: "类型即配置" (Type-level configuration)
1199 // 所有的 MMA atom、TiledCopy、SmemLayout、Swizzle 都是 **编译期类型**,
1200 // kernel 接收这些类型的实例 (通常为 struct), 在编译期完成全部映射推导。
1201 //
1202 // 以下每个类型定义后面都标注了它在 kernel body 中的对应变量/用法, 形成 1:1 对照。
1203 //
1204 // 面试常问: 「CuTe 为什么比手写 PTX 简洁? 」
1205 // 答: 手写需要:
1206 //   1) 手动计算每线程的 smem 地址偏移
1207 //   2) 手动计算 ldmatrix 的 lane 映射
1208 //   3) 手动插入 swizzle XOR 到每个地址计算点
1209 //   4) 手动做 epilogue 的 warp shuffle 编排
1210 // CuTe 的 partition + copy + gemm 通过 TiledCopy/TiledMMA 的类型信息
1211 // 在编译期自动完成上述全部推导, 生成与手写等价的 PTX 指令序列。
1212 // =====
1213 template <typename T, const int Stages = 2, const int BlockSwizzle = 0>
1214 void launch_hgemm_mma_stages_tn_cute(T *a, T *b, T *c, int M, int N, int K) {
1215     using namespace cute;
1216
1217     // — Tile 尺寸 (kernel 模板参数 BM/BN/BK/kStage 的值) —
1218     //
1219     // kernel 用法对照:
1220     //   BM=128:   local_tile(A, make_tile(Int<BM>{}, Int<BK>{}), ...) → gA=(BM,BK,num_k_tiles)
1221     //             local_tile(D, make_tile(Int<BM>{}, Int<BN>{}), ...) → gD=(BM,BN)
1222     //   BN=256:   local_tile(B, make_tile(Int<BN>{}, Int<BK>{}), ...) → gB=(BN,BK,num_k_tiles)
1223     //             local_tile(D, ...) → gD 的列方向
1224     //   BK=32:    num_k_tiles = k / BK; 每个 BK tile 内 num_k_steps = BK/kMmaPK = 2 个 MMA_K slice
1225     //   kStage=2: sA/sB 的第三维 = (BM,BK,kStage); cp_async_wait<kStage-2>() 流水线同步
1226     //   kSmemLayoutCBatch=4: step = size<3>(tCsC_r2s) → C scratchpad pipe 深度 = 4
1227     auto BM = Int<128>{};
1228     auto BN = Int<256>{};
1229     auto BK = Int<32>{};
1230     auto KStage = Int<Stages>{};
1231     auto kSmemLayoutCBatch = Int<4>{};
1232
1233     // — SmemLayoutA / SmemLayoutB —
1234     // kernel 用法:
1235     // auto sA = make_tensor(make_smem_ptr(Ashm), SmemLayoutA{}); // (BM, BK, kStage)
1236     // auto sB = make_tensor(make_smem_ptr(Bshm), SmemLayoutB{}); // (BN, BK, kStage)
1237     // SmemLayout 由三部分组成: Swizzle<3,3,3> + atom layout(8×BK) + tile_to_shape 扩展到完整 tile+stage。
1238     //
1239     // 对当前 base layout shape=(8,32), stride=(32,1), T=half 的例子:
1240     //   - M=3: 连续 2^3=8 个 half 组成一个基本元素, 即 16 bytes;
1241     //   - S=3: 连续 2^3=8 个基本元素组成一行, 即 128 bytes;
1242     //   - B=3: 共有 2^3=8 行。
1243     // 于是一个 8×32 的逻辑 tile 正好对应 8 行 × 128B, 覆盖全部 32 个 shared-memory bank。

```

```

1244 // Swizzle 对每个基本元素的列坐标执行  $icol' = irow \wedge icol$ ,
1245 // 再将 (irow, icol') 和基本元素内部的 3 个低位编码回物理 offset。
1246 // 其位级形式为  $offset' = offset \wedge ((offset \& (0b111 \ll 6)) \gg 3)$ :
1247 // 地址位 [6:8] XOR 到 [3:5], 地址位 [0:2] 不参与置换。
1248 // composition(Swizzle, base_layout) 的语义是  $R(c) = Swizzle(base\_layout(c))$ :
1249 // 逻辑坐标仍由 base_layout 给出, 只有最终的 shared-memory offset 被重排。
1250 // tile_to_shape: 通过 blocked product 重复这个带 swizzle 的 block layout,
1251 // 使结果 shape 匹配  $BM \times BK \times kStage$ ; 默认按目标 shape 的 mode order 重复各维。
1252 using SmemLayoutAtom = decltype(composition(
1253     Swizzle<3, 3, 3>{},
1254     make_layout(make_shape(Int<8>{}, Int<BK>{}),
1255         make_stride(Int<BK>{}, Int<1>{}))));
1256 using SmemLayoutA = decltype(tile_to_shape(
1257     SmemLayoutAtom{}, make_shape(Int<BM>{}, Int<BK>{}, Int<kStage>{})));
1258 using SmemLayoutB = decltype(tile_to_shape(
1259     SmemLayoutAtom{}, make_shape(Int<BN>{}, Int<BK>{}, Int<kStage>{})));
1260
1261 // — MMA (TiledMMA) —
1262 // kernel 用法:
1263 // TiledMMA tiled_mma;
1264 // auto thr_mma = tiled_mma.get_slice(threadIdx.x); // 每线程的 MMA slice
1265 // auto tCrA = thr_mma.partition_fragment_A(gA(_,_,0)); // (MMA, MMA_M, MMA_K)
1266 // auto tCrB = thr_mma.partition_fragment_B(gB(_,_,0)); // (MMA, MMA_N, MMA_K)
1267 // auto tCrD = thr_mma.partition_fragment_C(gD); // (MMA, MMA_M, MMA_N)
1268 // cute::gemm(tiled_mma, tCrD, tCrA(_,_,k_step), tCrB(_,_,k_step), tCrD);
1269 // MMA 还决定 S2R/R2S copy 的 tiler: make_tiled_copy_A/B/C 都依赖 tiled_mma 的
1270 // tile_size<0/1>(mma) = (32,32) 来推导线程→数据映射。
1271 //
1272 // 推导链: SM80_16x8x16_F16F16F16F16_TN → MMA_Atom
1273 // → make_tiled_mma(atom, EURepeat{2,2,1}, ValTile{32,32,16})
1274 // → TiledMMA: 128 threads = 4 warps × (2×2 EU slices), 逻辑 MMA tile = 32×32×16
1275 // TN = A row-major, B col-major; 因此传给本 kernel 的 B 指针实际指向
1276 //  $B^T[N,K]$  的 row-major 存储 (等价于 GEMM 语义中的  $B[K,N]$  col-major)。
1277 using mma_op = SM80_16x8x16_F16F16F16F16_TN;
1278 using mma_traits = MMA_Traits<mma_op>;
1279 using mma_atom = MMA_Atom<mma_traits>;
1280 using mma_atom_shape = mma_traits::Shape_MNK; // (Int<16>, Int<8>, Int<16>)
1281
1282 static constexpr int kMmaEURepeatM = 2;
1283 static constexpr int kMmaEURepeatN = 2;
1284 static constexpr int kMmaEURepeatK = 1;
1285 static constexpr int kMmaPM = 1 * kMmaEURepeatM * get<0>(mma_atom_shape{}); // 32
1286 static constexpr int kMmaPN = 2 * kMmaEURepeatN * get<1>(mma_atom_shape{}); // 32
1287 static constexpr int kMmaPK = 1 * kMmaEURepeatK * get<2>(mma_atom_shape{}); // 16
1288 // kMmaPK=16 → kernel 中 num_k_steps = size<2>(tCrA) =  $BK/kMmaPK = 32/16 = 2$ 
1289
1290 using MMA_EU_RepeatT = decltype(make_layout(make_shape(
1291     Int<kMmaEURepeatM>{}, Int<kMmaEURepeatN>{}, Int<kMmaEURepeatK>{})));
1292 using MMA_P_T = Tile<Int<kMmaPM>, Int<kMmaPN>, Int<kMmaPK>>;
1293 using MMA = decltype(make_tiled_mma(mma_atom{}, MMA_EU_RepeatT{}, MMA_P_T{}));
1294
1295 // — G2SCopyA / G2SCopyB —
1296 // kernel 用法:
1297 // G2SCopyA g2s_tiled_copy_a; G2SCopyB g2s_tiled_copy_b;
1298 // cute::copy(g2s_tiled_copy_a, tAgA_copy(_,_,_,istage), tAsA_copy(_,_,_,istage));
1299 // cute::copy(g2s_tiled_copy_b, tBgB_copy(_,_,_,istage), tBsB_copy(_,_,_,istage));
1300 // G→S: 128-bit cp.async. ThrLayout{32,4} × ValLayout{1,8}:
1301 // 128 个线程, 每线程搬运  $1 \times 8 = 8$  个 half = 128 bits。
1302 using g2s_copy_op = SM80_CP_ASYNC_CACHEGLOBAL<cute::uint128_t>;
1303 using g2s_copy_traits = Copy_Traits<g2s_copy_op>;
1304 using g2s_copy_atom = Copy_Atom<g2s_copy_traits, T>;
1305 using G2SCopyA = decltype(make_tiled_copy(
1306     g2s_copy_atom{},

```

```

1307     make_layout(make_shape(Int<32>{}, Int<4>{}),
1308                 make_stride(Int<4>{}, Int<1>{})),
1309     make_layout(make_shape(Int<1>{}, Int<8>{}))));
1310 using G2SCopyB = G2SCopyA;
1311
1312 // — S2RCopyAtomA / S2RCopyAtomB —
1313 // kernel 用法:
1314 // auto s2r_tiled_copy_a = make_tiled_copy_A(S2RCopyAtomA{}, tiled_mma);
1315 // auto s2r_tiled_copy_b = make_tiled_copy_B(S2RCopyAtomB{}, tiled_mma);
1316 // cute::copy(s2r_tiled_copy_a, tAsA(_,_,k_step_next,ismem_read), tCrA_view(_,_,k_step_next));
1317 // cute::copy(s2r_tiled_copy_b, tBsB(_,_,k_step_next,ismem_read), tCrB_view(_,_,k_step_next));
1318 // S→R: ldmatrix (SM75_U32x4_LDSM_N)。make_tiled_copy_A/B 根据 TiledMMA 自动推导
1319 // 与 MMA fragment 对齐的 shared→register 映射, 无需手动指定 ThrLayout/ValLayout。
1320 using s2r_copy_op = SM75_U32x4_LDSM_N;
1321 using s2r_copy_traits = Copy_Traits<s2r_copy_op>;
1322 using s2r_copy_atom = Copy_Atom<s2r_copy_traits, T>;
1323 using S2RCopyAtomA = s2r_copy_atom;
1324 using S2RCopyAtomB = s2r_copy_atom;
1325
1326 // — SmemLayoutC —
1327 // kernel 用法:
1328 // auto sC = make_tensor(sA(_,_,ismem_read).data(), SmemLayoutC{});
1329 // 复用当前 A stage 的空间作为 C scratchpad
1330 // sC shape = (kMmaPM, kMmaPN, kSmemLayoutCBatch) = (32, 32, 4)
1331 // pipe 数 4 由 kSmemLayoutCBatch 指定 → step = size<3>(tCsC_r2s) = 4
1332 // 与 A/B 相同的 Swizzle<3,3,3> 规则, 但 base layout 是 32×32 (MMA tile 大小),
1333 // 再扩展为 4 个 epilogue scratchpad pipe slots。这 4 个 slots 不等同于主循环
1334 // 的 kStage K-tile pipeline。
1335 using SmemLayoutAtomC = decltype(composition(
1336     Swizzle<3, 3, 3>{},
1337     make_layout(make_shape(Int<kMmaPM>{}, Int<kMmaPN>{}),
1338                 make_stride(Int<kMmaPN>{}, Int<1>{}))));
1339 using SmemLayoutC = decltype(tile_to_shape(
1340     SmemLayoutAtomC{},
1341     make_shape(Int<kMmaPM>{}, Int<kMmaPN>{}, Int<kSmemLayoutCBatch>{})));
1342
1343 // — R2SCopyAtomC —
1344 // kernel 用法:
1345 // auto r2s_tiled_copy_c = make_tiled_copy_C(R2SCopyAtomC{}, tiled_mma);
1346 // cute::copy(r2s_tiled_copy_c, t, tCsC_r2s(_, 0, 0, j));
1347 // R→S: UniversalCopy<int> 以 32-bit 粒度将寄存器 payload 写入 C scratchpad。
1348 // make_tiled_copy_C 从 TiledMMA 的 tile_size<0/1>(mma)=(32,32) 推导 tiler,
1349 // 因此 R2S copy 的 tiler = (32,32), 与 SmemLayoutC 的 (32,32) 恰好匹配。
1350 using R2SCopyAtomC = Copy_Atom<UniversalCopy<int>, T>;
1351
1352 // — S2GCopyC —
1353 // kernel 用法:
1354 // S2GCopyC s2g_tiled_copy_c;
1355 // cute::copy(s2g_tiled_copy_c, tCsC_s2g(_, 0, 0, j), tCgC_s2gx(_, i+j));
1356 // S→G: UniversalCopy<uint128_t> 以 128-bit 粒度做 shared→global wide store。
1357 // ThrLayout{32,4} × ValLayout{1,8} → tiler = product((32,4),(1,8)) = (32,32),
1358 // 与 R2S tiler 相同, 因此 partition_S(sC) 得到的 pipe 维度一致。
1359 using S2GCopyAtomC = Copy_Atom<UniversalCopy<uint128_t>, T>;
1360 using S2GCopyC = decltype(make_tiled_copy(
1361     S2GCopyAtomC{},
1362     make_layout(make_shape(Int<32>{}, Int<4>{}),
1363                 make_stride(Int<4>{}, Int<1>{})),
1364     make_layout(make_shape(Int<1>{}, Int<8>{}))));
1365
1366 // — Grid/Block 配置 —
1367 // kernel 用法:
1368 // int idx = threadIdx.x;          // 0..127
1369 // int ix = ((int)BlockSwizzle) * blockIdx.z * gridDim.x + blockIdx.x;

```

```

1370 // int iy = blockIdx.y;           // M 方向 CTA 坐标
1371 // if (iy * BM >= m || ix * BN >= n) return; // CTA 级边界保护
1372 // 由于 kernel 没有 tile 内的逐元素 predication, 调用方需保证
1373 // M % BM == 0, N % BN == 0, K % BK == 0。
1374 // 当不启用 BlockSwizzle 时, BZ=1 退化为纯 2D grid, 行为不变。
1375 constexpr int kSwizzleStride = 2048;
1376 int BX = (N + BN - 1) / BN;
1377 int BY = (M + BM - 1) / BM;
1378 int BZ = BlockSwizzle ? (N + kSwizzleStride - 1) / kSwizzleStride : 1;
1379 BX = BlockSwizzle ? (BX + BZ - 1) / BZ : BX;
1380 dim3 block(size(MMA{})); // 128 threads (= size(TiledMMA))
1381 dim3 grid(BX, BY, BZ);
1382
1383 // — Dynamic Shared Memory 大小 —
1384 // kernel 用法:
1385 // extern __shared__ T shm_data[];
1386 // T *Ashm = shm_data;
1387 // T *Bshm = shm_data + cuted::cosize(SmemLayoutA{});
1388 // cosize(SmemLayoutA) = BM*BK*kStage = 128*32*2 = 8192 个 T
1389 // cosize(SmemLayoutB) = BN*BK*kStage = 256*32*2 = 16384 个 T
1390 // 合计 A+B = 24576 个 T; C epilogue 复用 A stage (128*32 = 4096 个 T)。
1391 // static_assert 保证 C scratchpad (kMmaPM*kMmaPN*kSmemLayoutCBatch = 32*32*4 = 4096)
1392 // 能放进一个 A stage (128*32 = 4096)。
1393 static constexpr int shm_size_AB =
1394     cuted::cosize(SmemLayoutA{}) + cuted::cosize(SmemLayoutB{});
1395 static constexpr int shm_size_C = cuted::cosize(SmemLayoutC{});
1396 static_assert(size<0>(SmemLayoutA{}) * size<1>(SmemLayoutA{}) >= size(SmemLayoutC{}),
1397     "C shared memory must fit within one A pipe");
1398 static constexpr int kShmSize =
1399     cuted::max(shm_size_AB, shm_size_C) * sizeof(T);
1400
1401 cudaFuncSetAttribute(
1402     hgemm_mma_stages_tn_cute<T, BM, BN, BK, KStage, MMA, G2SCopyA, G2SCopyB,
1403         SmemLayoutA, SmemLayoutB, SmemLayoutC,
1404         S2RCopyAtomA, S2RCopyAtomB, R2SCopyAtomC,
1405         S2GCopyAtomC, S2GCopyC, BlockSwizzle>,
1406     cudaFuncAttributeMaxDynamicSharedMemorySize, kShmSize);
1407
1408 hgemm_mma_stages_tn_cute<T, BM, BN, BK, KStage, MMA, G2SCopyA, G2SCopyB,
1409     SmemLayoutA, SmemLayoutB, SmemLayoutC, S2RCopyAtomA,
1410     S2RCopyAtomB, R2SCopyAtomC, S2GCopyAtomC, S2GCopyC,
1411     BlockSwizzle>
1412     <<<grid, block, kShmSize>>>(a, b, c, M, N, K);
1413 }
1414
1415 #endif /* NOTES_V2_ENABLE_CUTE */
1416 #if defined(NOTES_V2_ENABLE_WGMMA)
1417 // ---- WGMMA Kernel: Warp Specialization + TMA ----
1418 // 面试重点 — Warp Specialization (Hopper 最核心的编程模型变化):
1419 //
1420 // 背景: 传统 GEMM kernel 中, 所有线程同步地"加载→计算→存回",
1421 // 数据搬运和计算无法重叠。Hopper 的 TMA + WGMMA 支持将工作拆分为两个
1422 // warpgroup, 让数据搬运和矩阵乘完全异步执行:
1423 //
1424 // WG0 (128 threads, Producer): 仅 thread 0 提交 TMA 2D 拷贝指令,
1425 // 将 A/B tile 从 HBM 搬到 shared memory。
1426 // WG1 (128 threads, Consumer): 所有 128 个线程参与 WGMMA 矩阵乘。
1427 //
1428 // Producer 和 Consumer 通过 cuda::barrier (CTA 级别) 同步:
1429 // - full[stage]: Producer 发信号表示 stage stage 的数据已就绪, 可被使用
1430 // - empty[stage]: Consumer 发信号表示 stage stage 的使用完毕, 可以被覆盖
1431 //
1432 // 本节重点理解:

```

```

1433 // 1) 为什么 Producer 只需要 thread 0? TMA 是硬件 DMA 指令, 一次提交
1434 // 即可搬运整个 2D tile, 无需所有线程参与。
1435 // 2) barrier 的 arrive count = 129: 128 个 Consumer 线程 + 1 个 Producer 提交线程。
1436 // 3) 多 stage pipeline 使 Consumer 计算 stage stage 的同时,
1437 // Producer 可以搬运 stage (stage+1), 隐藏 HBM-SMEM 延迟。
1438 //
1439 // Tile Hierarchy (与 MMA m16n8k16 kernel 对比):
1440 //   WGMMMA Atom:      m64n128k16 (一次处理 64×128×16, 是 MMA 的 64 倍)
1441 //   K Tile:           BK=64, 每个 K tile 包含 BK/kWmmaK=4 个 WGMMMA atom
1442 //   M Tile:           BM=128, 每个 M tile 包含 BM/kWmmaM=2 个 WGMMMA atom
1443 //   N Tile:           BN=128, 单个 kWmmaN=128 即可覆盖, 无需在 N 方向分块
1444 //   Block Tile:       C[128,128] = A[128,64] × B[64,128]
1445 //   Threads:          256 = 2 warpgroups × 128 threads/warpgroup
1446 //   Producer(WG0):    128 threads (仅 thread 0 做 TMA 提交)
1447 //   Consumer(WG1):    128 threads = 4 warps (全部参与 WGMMMA 和写回)
1448 //
1449 // kStages=3: 3 级流水线。Consumer 滞后 Producer 最多 2 步, 确保 HBM-SMEM
1450 // 的延迟被计算完全掩盖。
1451 //
1452 // Grid: ((N+127)/128/S, (M+127)/128, S), S=(N+2047)/2048, 3D block swizzle
1453 // - grid.z = S 个 swizzle 分区, 将连续 block 打散到不同 N 区域改善 L2 命中
1454 // Block: (256, 1, 1), 2 warpgroups
1455 // source: LeetCUDA/kernels/hgemm/wgmma/hgemm_wgmma_fp16acc_stages_tn.cu
1456 template <const int kWmmaM = 64,           // WGMMMA atom M dim (m64n128k16)
1457           const int kWmmaN = 128,          // WGMMMA atom N dim
1458           const int kWmmaK = 16,           // WGMMMA atom K dim
1459           const int BM = 128,              // block tile M, 2 × kWmmaM
1460           const int BN = 128,              // block tile N, 1 × kWmmaN
1461           const int BK = 64,               // block tile K, 4 × kWmmaK
1462           const int kNumThreads = 256,     // 2 warpgroups × 128 threads
1463           const int kStages = 3,           // TMA pipeline depth (full/empty barriers)
1464           const int kBlockSwizzle = 0>     // 1 enables 3D grid swizzle for L2 locality
1465 __global__ void __launch_bounds__(kNumThreads, 1)
1466   hgemm_wgmma_stages_tn(
1467     int M, int N, int K, half *C,
1468     const CUtensorMap *__restrict__ tensorMapA,
1469     const CUtensorMap *__restrict__ tensorMapB) {
1470   static_assert(kBlockSwizzle == 0 || kBlockSwizzle == 1, "kBlockSwizzle must be 0 or 1");
1471   // 注意: tensorMapA/tensorMapB 需要由 host 侧按当前 tile 布局预先创建;
1472   // 对 row-major [M, K] 矩阵, TMA shape 参数写的是 (K, M) 而不是 (M, K),
1473   // 也就是TMA descriptor中把连续的维度写在最内层, 非连续的维度写在最外层。
1474   // 这是 TMA descriptor 最容易背错的地方之一。notes 这里只保留 kernel 主体,
1475   // 不展开宿主侧 create_tensor_map 细节。
1476
1477   // Block Swizzle: 在 grid x 维度做 swizzle, 改善 L2 cache 局部性
1478   // bx = blockIdx.z * gridDim.x + blockIdx.x, 将相邻 block 打散到不同 N 区域
1479   const int bx = ((int)kBlockSwizzle) * blockIdx.z * gridDim.x + blockIdx.x;
1480   const int by = blockIdx.y;
1481   constexpr int kConsumerThreads = kNumThreads / 2; // 128 threads = 1 warpgroup
1482   // kNumConsumers = (kNumThreads / 128) - 1 = 2 - 1 = 1 (1 个 consumer WG)
1483   // 这里的 -1 是因为 2 个 warpgroup 中 1 个是 producer, 剩余都是 consumer
1484   constexpr int kNumConsumers = (kNumThreads / kConsumerThreads) - 1; // 1 consumer WG
1485   // kWarpgroupM = BM / kNumConsumers: 每个 consumer warpgroup 负责的 M 行数
1486   // 当只有一个 consumer 时, 它负责全部 BM=128 行
1487   constexpr int kWarpgroupM = BM / kNumConsumers; // 128
1488
1489   // 边界检查: 确保当前 block 不超出 M/N 范围
1490   if (bx >= div_ceil(N, BN) || by >= div_ceil(M, BM))
1491     return;
1492
1493   // ---- Shared Memory 分配 ----
1494   // 动态 shared memory (由 host 侧通过 kernel launch 的 smem 参数指定大小)
1495   // TMA + WGMMMA 仅需 __align__(128), 而非 __align__(1024)。原因:

```

```

1496 // make_smem_desc() 构造 WGMMMA descriptor 时, start_address 字段
1497 // 完整编码了 __cvta_generic_to_shared 的绝对 32-bit 偏移量,
1498 // 硬件根据该绝对地址自动推导并补偿 swizzle phase 偏移 (等价于
1499 // descriptor 内置的 base_offset 机制)。因此即使 smem 基址未落到
1500 // 1024B 边界, WGMMMA 硬件也能正确读取 TMA 写入的数据。
1501 // 对比 hgemmm_tma_mma_ws_tn: 消费者使用 ldmatrix + 手写 swizzle<64>()
1502 // , 该函数无法感知 smem 绝对地址, 硬编码了 phase=0 的假设, 因此必须
1503 // __align__(1024) 来保证 phase 确实为 0。从健壮性角度看本 kernel 也
1504 // 应该用 __align__(1024); 这里使用 128 也可以正常运行。
1505 extern __shared__ __align__(1024) uint8_t smem_tma_wgmma_ws[];
1506 WgmmaSMem<BM, BN, BK, kStages> &s =
1507     *reinterpret_cast<WgmmaSMem<BM, BN, BK, kStages>*>(smem_tma_wgmma_ws);
1508 half *s_a = s.A;
1509 half *s_b = s.B;
1510
1511 // ---- cuda::barrier 初始化 ----
1512 //
1513 // ★ Barrier 机制速查 (arrive / wait 核心语义):
1514 //
1515 // cuda::barrier 是一个 CTA 级同步原语, 基于 **phase (奇偶交替)** 工作:
1516 //
1517 // init(bar, N): 设置 arrive_count = N。
1518 // 含义: 每 phase 需要 N 个线程各调用一次 arrive(), barrier 才翻转 phase。
1519 //
1520 // arrive(): 线程声明"我已到达此 barrier 点"。
1521 // - 非阻塞, 立即返回一个 token (包含当前 phase 值)。
1522 // - 不关心"谁"到达, 只关心"到达次数"是否达到 arrive_count。
1523 //
1524 // wait(token): 线程阻塞, 直到 barrier 的当前 phase ≠ token 中的 phase。
1525 // - 即: 等待 phase 翻转。
1526 // - Phase 翻转条件: (1) arrive 次数达到 arrive_count
1527 // (2) 若使用了 barrier_arrive_tx, 还需 TMA 字节全部写完
1528 //
1529 // 典型用法: b.wait(b.arrive())
1530 // - arrive() 注册自己到达, 拿到 token (当前 phase = P)
1531 // - wait(token) 阻塞直到 phase 翻转 (P → P+1)
1532 // - 如果自己是第 N 个到达者 → phase 立即翻转 → wait 立即返回 (不阻塞)
1533 // - 如果自己不是最后一个 → wait 阻塞, 直到第 N 个到达者触发翻转
1534 //
1535 // ★ 本 kernel 的 Pipeline 同步协议 (kStages=3, arrive_count=129):
1536 //
1537 // Producer (1 thread)          Consumer (128 threads)
1538 // -----
1539 //                                C0: arrive(empty[*]) ×128 [init: 标记所有 stage 为空]
1540 // ┌ for each k_tile: ─┐          ┌ for each k_tile: ─┐
1541 // | P1: arrive+wait(empty[q]) |   | C1: arrive+wait(full[q]) |
1542 // |   ↓ 等 stage q 被消费完   |   |   ↓ 等 TMA 数据就绪   |
1543 // | P2: TMA(A[q]) + TMA(B[q]) |   | C2: WGMMMA 计算       |
1544 // |   ↓ 异步拷贝             |   | C3: wait WGMMMA 完成   |
1545 // | P3: arrive_tx(full[q])    |   | C4: arrive(empty[q])   |
1546 // |   ↑ 通知: stage q 数据就绪 |   |   ↑ 通知: stage q 已消费完 |
1547 // └──────────────────┘          └──────────────────┘
1548 //
1549 // 关键不变式 (invariant):
1550 // - Producer 不会覆盖 Consumer 正在读的 stage
1551 // - Consumer 不会读 Producer 还没写完的 stage
1552 // - 通过 full/empty 两个 barrier 的 phase 交替来保证
1553 //
1554 // 每个 stage 有两个 barrier:
1555 // full[stage]: TMA 数据就绪信号。Producer 发 (arrive_tx), Consumer 等 (wait)。
1556 // empty[stage]: Stage 空闲信号。Consumer 发 (arrive), Producer 等 (wait)。
1557 //
1558 // arrive_count = 128 (consumer) + 1 (producer) = 129:

```

```

1559 // 每 phase, 128 个 consumer 线程 + 1 个 producer 线程都要 arrive 一次。
1560 #pragma nv_diag_suppress static_var_with_dynamic_init
1561 __shared__ cuda::barrier<cuda::thread_scope_block> full[kStages];
1562 __shared__ cuda::barrier<cuda::thread_scope_block> empty[kStages];
1563 #pragma nv_diag_default static_var_with_dynamic_init
1564
1565 // K 方向总 tile 数。要求 K 能被 BK 整除, 否则尾 tile 被丢弃。
1566 const int NUM_K_TILES = div_ceil(K, BK);
1567 const int wg_idx = threadIdx.x / kConsumerThreads; // 0=Producer, 1=Consumer
1568 const int wg_tid = threadIdx.x % kConsumerThreads; // 0~127 within warpgroup
1569
1570 // 初始化 barriers (仅 thread 0 执行)
1571 if (threadIdx.x == 0) {
1572     for (int i = 0; i < kStages; ++i) {
1573         // arrive_count = 129: 128 consumer + 1 producer
1574         // init 是 CUDA barrier 的 hidden friend 函数 (定义在 cuda::barrier 类体内),
1575         // 只能通过 ADL (Argument-Dependent Lookup) 调用。因为第一个参数 &full[i] 的类型是
1576         // cuda::barrier<cuda::thread_scope_block>*, 编译器通过 ADL 在 cuda 命名空间中自动找到它。
1577         init(&full[i], kConsumerThreads + 1); // 128 consumer + 1 producer
1578         init(&empty[i], kConsumerThreads + 1); // same
1579     }
1580     // fence_proxy_async_shared_cta: 确保 barrier 在 smem 中的初始化
1581     // 对 async proxy (TMA/WGMMMA) 可见。
1582     tma_fence_proxy_async_shared_cta();
1583 }
1584 __syncthreads();
1585
1586 // =====
1587 // ★ Warp Specialization 并发模型 — 为什么只有 if/else 没有 while?
1588 // =====
1589 //
1590 // 256 个线程在同一个 SM 上**并发执行**。if/else 只是分工, 不是先后顺序:
1591 // - WG0 (threadIdx.x 0~127): 全部走 if 分支, 做 Producer。
1592 // - WG1 (threadIdx.x 128~255): 全部走 else 分支, 做 Consumer。
1593 //
1594 // SM 的 warp scheduler 会交替调度来自 WG0 和 WG1 的 warp, 两者**同时**推进:
1595 // Producer: for (k_iter = 0 .. NUM_K_TILES) { P1→P2→P3 } ← 自己的循环
1596 // Consumer: for (k_iter = 0 .. NUM_K_TILES) { C1→C2→C3→C4 } ← 自己的循环
1597 //
1598 // 两个 warpgroups 各自独立迭代 K-tiles, 通过 full[] / empty[] barrier 同步:
1599 // - Producer 领先 Consumer 太多 → wait(empty[q]) 阻塞, 等 Consumer 消费完
1600 // - Consumer 领先 Producer 太多 → wait(full[q]) 阻塞, 等 TMA 搬运完
1601 //
1602 // 这是生产者-消费者模型的 GPU 实现: 不需要共享循环变量, barrier 的 phase
1603 // 交替天然保证了流水线串行化 (at most kStages-1 steps ahead)。
1604 // =====
1605 // Producer Warpgroup (WG0, threadIdx.x 0~127)
1606 // 职责: 提交 TMA 2D 拷贝, 将 A/B tile 从 HBM 异步搬运到 SMEM。
1607 // 只有 wg_tid==0 执行实际拷贝提交, 其余 127 个线程空闲。
1608 // =====
1609 if (wg_idx == 0) {
1610     NOTES_V2_REG_DEALLOC(40);
1611     if (wg_tid == 0) {
1612         // stage: 当前操作的 stage 索引 (round-robin 0 -> 1 -> 2 -> 0 -> ...)
1613         int stage = 0;
1614         for (int k = 0; k < NUM_K_TILES; ++k, ++stage) {
1615             if (stage == kStages)
1616                 stage = 0;
1617
1618             // Step P1: 等待 stage stage 变为"空" (可被覆盖写入)
1619             //
1620             // 模式 empty[stage].wait(empty[stage].arrive()):
1621             // - arrive(): Producer (1 个线程) 在 empty barrier 上注册到达,

```

```

1622 // 获得当前 phase 的 token。如果 Consumer 已经在 C4 步骤积累了
1623 // 128 次 arrive (来自上一轮迭代或 C0 初始化)，则加上这次共 129 次
1624 // → phase 立即翻转。
1625 // - wait(token): 阻塞直到 barrier phase ≠ token 中的 phase。
1626 // 由于 arrive() 是第 129 次到达，phase 在 arrive 时已翻转，
1627 // wait 看到 phase 已变 → 立即返回 (首轮不阻塞)。
1628 //
1629 // 语义: Producer 说"我准备好写 stage stage 了，Consumer 用完没有?"
1630 // 如果 Consumer 还没用完 → phase 未翻转 → wait 阻塞等待。
1631 // 如果 Consumer 已用完 → phase 已翻转 → wait 立即返回。
1632 empty[stage].wait(empty[stage].arrive());
1633
1634 // Step P2: 提交 TMA 2D 拷贝指令
1635 // cp_async_bulk_tensor_2d_global_to_shared:
1636 // 参数1(dst): smem 目标地址 (当前 stage 的 A/B tile 起始位置)
1637 // 参数2(tensorMap): Host 预创建的 TMA descriptor (描述源矩阵的 shape/stride/dtype)
1638 // 参数3/4(coords): (k_offset, m_offset) 或 (k_offset, n_offset) 的全局坐标
1639 // 参数5(barrier): 拷贝完成后自动 arrive 到此 barrier
1640 //
1641 // TMA 是硬件 DMA 引擎: 一次指令提交即可搬运整个 2D tile,
1642 // 无需线程逐元素搬运, 零寄存器开销。
1643 // TMA 2D 加载 A tile: coords = (k_offset, m_offset)
1644 tma_load_2d(&s_a[stage * BK * BM], tensorMapA, k * BK, by * BM,
1645             full[stage]);
1646
1647 // TMA 2D 加载 B tile: coords = (k_offset, n_offset)
1648 tma_load_2d(&s_b[stage * BK * BN], tensorMapB, k * BK, bx * BN,
1649             full[stage]);
1650
1651 // Step P3: 通知 Consumer: stage stage 的 TMA 数据已就绪
1652 //
1653 // barrier_arrive_tx(bar, arrive_count_update, byte_count):
1654 // - 在 bar 上注册 1 次到达 (arrive_count_update=1)
1655 // - 同时声明预期有 byte_count 字节将通过 async copy (TMA) 写入 smem
1656 // - Phase 翻转条件:
1657 //   (a) 总 arrive 次数达到 129 (128 consumer + 1 producer)
1658 //   (b) 所有声明的 async 字节已写入 smem
1659 // 两个条件都满足后 phase 才翻转, Consumer 的 wait() 才返回。
1660 // - 即: Consumer 不会在 TMA 数据完整到达前就开始读 smem。
1661 tma_arrive_expect_tx(full[stage], (BK * BN + BK * BM) * sizeof(half));
1662 }
1663 }
1664 }
1665 // =====
1666 // Consumer Warpgroup (WG1, threadIdx.x 128~255)
1667 // 职责: 等待 TMA 数据就绪 -> 发射 WGMMMA 做矩阵乘 -> 积累结果 -> 写回 C
1668 // 所有 128 个线程 (4 warps) 全部参与。
1669 // =====
1670 else {
1671 // Step C0: Consumer 初始化 — 标记所有 stage 为"空" (可被 Producer 写入)
1672 //
1673 // 所有 128 个 Consumer 线程对每个 stage 的 empty barrier 调用 arrive()。
1674 // 这是 Pipeline 的"预热"步骤——没有它, Producer 的 empty[stage].wait()
1675 // 在第一轮会永远阻塞 (因为 Producer 的 1 次 arrive 不足以凑够 129)。
1676 //
1677 // 注意: 此时每个 empty[i] 只有 128 次 arrive, 未达 129, phase 不翻转。
1678 // Producer 后续的 empty[stage].arrive() 作为第 129 次, 触发 phase 翻转。
1679 NOTES_V2_REG_ALLOC(232);
1680
1681 for (int i = 0; i < kStages; ++i) {
1682     [[maybe_unused]] auto token = empty[i].arrive();
1683 }
1684

```

```

1685 // 累加器寄存器声明
1686 // d[kWarpgroupM / kWmmaM][kWmmaN / 16][4]:
1687 // - d[0][*][*]: M 方向第 1 个 WGMMMA atom (rows 0~63)
1688 // - d[1][*][*]: M 方向第 2 个 WGMMMA atom (rows 64~127)
1689 // - d[*][g][*]: N 方向第 g 组 16 列
1690 // - d[*][*][0..3]: 4 条 uint32 寄存器, 共 8 个 half (覆盖 16x16 子块)
1691 // 每个线程总共 2 * 8 * 4 = 2 * 32 = 64 uint32 = 128 half (每次WGMMMA Atom 32 uint32 输出)
1692 // 128 线程 * 128 half = 16384 half = 128 * 128 = BM*BN (刚好覆盖整个 C tile)
1693 uint32_t d[kWarpgroupM / kWmmaM][kWmmaN / 16][4] = {};
1694
1695 int stage = 0;
1696 // K 维外循环: 沿 K tile 迭代 (BK=64, 每个 K tile 做 4 次 WGMMMA 累加)
1697 for (int k = 0; k < NUM_K_TILES; ++k, ++stage) {
1698     if (stage == kStages)
1699         stage = 0;
1700
1701     // Step C1: 等待 TMA 数据就绪 (full 信号)
1702     //
1703     // 模式 full[stage].wait(full[stage].arrive()):
1704     // - 128 个 Consumer 线程各调用 arrive(), 共 128 次到达。
1705     // - 当前 phase 累积: 128/129, phase 尚未翻转。
1706     // - wait(token) 阻塞, 直到 Producer 的 barrier_arrive_tx (Step P3)
1707     //   贡献第 129 次到达 + TMA 字节全部写完 → phase 翻转 → wait 返回。
1708     //
1709     // 语义: Consumer 说"我准备好读 stage stage 了, 数据到了没有?"
1710     full[stage].wait(full[stage].arrive());
1711
1712     // Step C2: 发射 WGMMMA 指令序列
1713     //
1714     // WGMMMA 是异步指令 (fire-and-forget), 发射后立即返回, 不等待计算完成。
1715     // 标准流程: FENCE → 发射 WGMMMA → COMMIT → WAIT。
1716     //
1717     // WGMMMA_FENCE (wgmma.fence.sync.aligned) :
1718     // - 确保 TMA 写入 smem 的数据对 async proxy (WGMMMA) 可见
1719     // - 确保累加器寄存器 d[] 已准备好接收 WGMMMA 输出
1720     // - 本质是 proxy 间的内存序 (memory ordering), 不是传统 __syncthreads
1721     //
1722     // 每个 WGMMMA_M64N128K16_F16F16F16 指令:
1723     // D[64x128] += A[64x16] × B[16x128]
1724     // A/B 均为 K-major (imm-trans=0), 由 descA/descB 描述 smem 中的布局。
1725     // ScaleD=1: 累加。K 维有 BK/kWmmaK=4 次迭代, 每次都 ScaleD=1。
1726     WGMMMA_FENCE();
1727
1728     // M 维迭代: BM/kWmmaM = 128/64 = 2 个 WGMMMA atom
1729     // 每个 atom 处理 64 行 M 方向数据。
1730 #pragma unroll
1731     for (int m = 0; m < kWarpgroupM / kWmmaM; ++m) {
1732         // wgmma_sA 指向当前 stage 中 A tile 的第 m 个 64*BK 子块
1733         // s_a 布局: [kStages][BM][BK] -> stage * BK*BM + BK * (m*64)
1734         half *wgmma_sA = s_a + stage * BK * BM + BK * m * kWmmaM;
1735
1736         // K 维迭代: BK/kWmmaK = 64/16 = 4 次 WGMMMA (累加)
1737         // 每次处理 K=16 维的矩阵乘, 4 次累加后覆盖完整的 BK=64。
1738 #pragma unroll
1739         for (int k_step = 0; k_step < BK / kWmmaK; ++k_step) {
1740             // 第 k_step 次 WGMMMA:
1741             // A: wgmma_sA + k_step * kWmmaK (A 的 K 维起始位置)
1742             // B: s_b + stage * BK * BN + k_step * kWmmaK (B 的 K 维起始位置)
1743             // 注意 B 的 smem 布局是 [BK, BN] row-major, K-major 读取时从
1744             // 第 k_it*16 行开始取 16 行, 每行 BN=128 列。
1745             // ScaleD=1: 累加 (K 维迭代需要累积结果)
1746             WGMMMA_M64N128K16_F16F16F16(d[m], wgmma_sA + k_step * kWmmaK,
1747                 s_b + stage * BK * BN + k_step * kWmmaK,

```

```

1748         1, // Scaled=1: accumulate (不清零)
1749         1, 1, 0, 0);
1750     }
1751 }
1752
1753 // Step C3: 提交并等待 WGMMMA 完成
1754 //
1755 // WGMMMA_COMMIT_GROUP (wgmma.commit_group.sync.aligned):
1756 // 将自上一次 COMMIT 以来发射的所有 WGMMMA 归为一组, 提交到 async proxy 执行。
1757 // 类比 cp.async.commit_group, 但 scope 是 warpgroup 而非 per-thread。
1758 //
1759 // WGMMMA_WAIT_GROUP(0) (wgmma.wait_group.sync.aligned 0):
1760 // 阻塞直到最多 N 个 group 尚未完成 (pending ≤ N)。N=0 即等待所有 group。
1761 // * 语义与 cp.async.wait_group 完全一致 (PTX ISA §9.7.15.7.3 vs §9.7.9.25.3.3) :
1762 // 两者都是 "wait until only N or fewer of the most recent groups are pending"。
1763 // 此处用 0 是因为 Consumer 在本次迭代中只 commit 了 1 个 group,
1764 // 必须等它全部完成才能进入下一步 (读下一 stage 的数据)。
1765 WGMMMA_COMMIT_GROUP();
1766 WGMMMA_WAIT_GROUP(0);
1767
1768 // Step C4: 释放 stage stage — 通知 Producer 可以覆盖写入
1769 //
1770 // 128 个 Consumer 线程在 empty[stage] 上 arrive(), 为 **下一 phase** 积累到达次数。
1771 // 这 128 次 arrive 不会立即翻转 phase (还需 Producer 在下一轮的 1 次 arrive)。
1772 //
1773 // 语义: Consumer 说"stage stage 我已经读完了, 你可以放心覆盖了。"
1774 [[maybe_unused]] auto token = empty[stage].arrive();
1775 }
1776
1777 // =====
1778 // Epilogue: 将寄存器中的累加结果写回 global memory C
1779 // =====
1780 //
1781 // WGMMMA m64n128k16.f16.f16.f16 的输出寄存器映射:
1782 //
1783 // 对 warpgroup 内每个线程, 输出 64 个 half = 32 个 uint32。
1784 // 这些 half 分布在 d[2][8][4] 数组中:
1785 // d[m_it][g][0]: (row, col) 和 (row, col+2) 位置的 2 个 half
1786 // d[m_it][g][1]: (row+8, col) 和 (row+8, col+2) 位置的 2 个 half
1787 // d[m_it][g][2]: (row, col+8) 和 (row, col+10) 位置的 2 个 half
1788 // d[m_it][g][3]: (row+8, col+8) 和 (row+8, col+10) 位置的 2 个 half
1789 //
1790 // 其中:
1791 // row = warp * 16 + lane / 4 (0~63, 4 warps * 16 rows)
1792 // col = g * 16 + 2 * (lane % 4) (0~126, step 2, 8 g's * 16 cols)
1793 //
1794 // 每个线程覆盖一个 16x16 子块中的 16 个 half:
1795 // +-----+-----+
1796 // | reg[0] | reg[2] | <- rows [row, row+8)
1797 // |(col,+2)|(col+8)|
1798 // +-----+-----+
1799 // | reg[1] | reg[3] | <- rows [row+8, row+16)
1800 // |(col,+2)|(col+8)|
1801 // +-----+-----+
1802 // 每个 reg 包含 2 个连续 half (col 和 col+2), 用 uint32 存储。
1803 //
1804 // 三维遍历:
1805 // m_it=0: rows 0~63, m_it=1: rows 64~127
1806 // g=0..7: 每个覆盖 16 列, 8*16=128 列 ok
1807 // 每个线程写 4 uint32 * 2 half/uint32 * 8 g * 2 m_it = 128 half。
1808 // 128 线程 * 128 half = 16384 half = 128 * 128 ok
1809
1810 const int lane = wg_tid % 32;

```

```

1811     const int warp = wg_tid / 32;
1812     // row: 当前线程在当前 WGMMMA atom 中负责的起始行。
1813     // warp 0~3 各负责 16 行: rows [warp*16, warp*16+15]。
1814     // lane/4: 每 4 个连续 lane 负责同一行 (因为 WGMMMA fragment 中每行有 4 个 uint32,
1815     //         分别覆盖列对 {c0,c1}、{c2,c3}、{c8,c9}、{c10,c11}, 由 lane%4 区分)。
1816     //         因此 lane/4 给出该行在 16-row block 内的行号 (0~7 对应 rows 0~7,
1817     //         同一 warp 中 lane/4==0 的有 lane 0,1,2,3, 都负责 row 0)。
1818     const int row = warp * 16 + lane / 4;
1819     // block_C: 当前 C tile 在 global memory 中的起始地址
1820     // by*BM 是 M 方向偏移, bx*BN 是 N 方向偏移
1821     half *block_C = C + by * BM * N + bx * BN;
1822
1823     // 2 * (8 * 4) = 2 * 32 = 64 uint32 = 128 half
1824     // NOTE: 这里可以考虑通过 R->S, S->G 的方式做一次 128 bits写回。
1825 #pragma unroll
1826     for (int m = 0; m < kWarpgroupM / kWgmmaM; ++m) {
1827         int yo = m * kWgmmaM; // M 方向行偏移 (0 或 64)
1828 #pragma unroll
1829         for (int g = 0; g < kWgmmaN / 16; ++g) {
1830             int col = g * 16 + 2 * (lane % 4); // N 方向列偏移 (0,2,4,...,14 then 16,18,...)
1831             // 一次 uint32 store 写入 2 个 half (连续列 col 和 col+2)
1832             // 左上象限: (row, col)
1833             *reinterpret_cast<uint32_t*>(&block_C[(row + yo) * N + col]) = d[m][g][0];
1834             // 左下象限: (row+8, col)
1835             *reinterpret_cast<uint32_t*>(&block_C[(row + yo + 8) * N + col]) = d[m][g][1];
1836             // 右上象限: (row, col+8)
1837             *reinterpret_cast<uint32_t*>(&block_C[(row + yo) * N + col + 8]) = d[m][g][2];
1838             // 右下象限: (row+8, col+8)
1839             *reinterpret_cast<uint32_t*>(&block_C[(row + yo + 8) * N + col + 8]) = d[m][g][3];
1840         }
1841     }
1842 }
1843 }
1844
1845 #endif /* NOTES_V2_ENABLE_WGMMMA */
1846
1847 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
1848
1849 // 消费者 MMA 层级映射参考 hgemmm_mma_stages_tn_swizzle。与那个
1850 // 八 warp kernel 不同, 本 warp-specialized 消费者仅拥有四个 warp。
1851 template <const int kMmaM = 16,                // MMA atom M dim (m16n8k16)
1852           const int kMmaN = 8,                // MMA atom N dim
1853           const int kMmaK = 16,               // MMA atom K dim
1854           const int kMmaTileM = 2,            // consumer warps along M, warp tile M = 32
1855           const int kMmaTileN = 2,            // consumer warps along N, warp tile N = 16
1856           const int kValTileM = 4,            // value-repeat along M, BM = 16*2*4 = 128
1857           const int kValTileN = 8,            // value-repeat along N, BN = 8*2*8 = 128
1858           const int kValTileK = 4,            // MMA_K slices per BK tile, BK = 16*4 = 64
1859           const int kStages = 2,              // TMA full/empty pipeline depth
1860           const int kNumThreads = 256,        // 128 producer + 128 consumer threads
1861           const int kBlockSwizzle = 0>        // 1 enables 3D grid swizzle for L2 locality
1862 __global__ void __launch_bounds__(kNumThreads)
1863     hgemmm_tma_mma_ws_tn(
1864         int M, int N, int K, half *C,
1865         const CUtensorMap *__restrict__ tensorMapA,
1866         const CUtensorMap *__restrict__ tensorMapB) {
1867
1868     constexpr int BM = kMmaM * kMmaTileM * kValTileM;
1869     constexpr int BN = kMmaN * kMmaTileN * kValTileN;
1870     constexpr int BK = kMmaK * kValTileK;
1871     static_assert(kMmaM == 16 && kMmaN == 8 && kMmaK == 16, "This kernel uses mma.sync.m16n8k16");
1872     static_assert(kMmaTileM * kMmaTileN == 4, "The consumer warpgroup has exactly four warps");
1873     static_assert(BM == 128 && BN == 128 && BK == 64, "TMA desc and 128B swizzle require 128x128x64");

```

```

1874 static_assert(kNumThreads == 256, "Use one producer and one consumer warpgroup");
1875 static_assert(kBlockSwizzle == 0 || kBlockSwizzle == 1, "kBlockSwizzle must be 0 or 1");
1876
1877 const int bx = kBlockSwizzle * blockIdx.z * gridDim.x + blockIdx.x;
1878 const int by = blockIdx.y;
1879 constexpr int kConsumerThreads = kNumThreads / 2;
1880 static_assert(kConsumerThreads == kMmaTileM * kMmaTileN * kWarpSize,
1881             "Consumer threads must cover the MMA warp grid");
1882
1883 if (bx >= div_ceil(N, BN) || by >= div_ceil(M, BM))
1884     return;
1885
1886 // TMA CU_TENSOR_MAP_SWIZZLE_128B 要求 smem 基地址 1024 字节对齐,
1887 // 以确保硬件 swizzle phase 从零开始。消费者 swizzle<64>()
1888 // 假设零 phase; 其他对齐 (如 128) 会导致 phase 偏移, 产生不匹配的
1889 // 物理地址, 破坏整个输出。
1890 //
1891 // 为什么 hgemm_wgmma_stages_tn 只需要 __align__(128), 而本 kernel
1892 // 必须 __align__(1024)? 核心区别在于消费者如何感知 swizzle phase:
1893 //
1894 // WGMMMA 消费者: 通过 make_smem_desc() 构造 64-bit WGMMMA descriptor,
1895 // 其中 bits [49,52) 的 base_offset 字段编码了 smem 基址相对于
1896 // 1024B 边界的偏移量 ((addr >> 7) & 0x7)。硬件在读取 smem 时会
1897 // 用这个 base_offset 自动补偿 swizzle phase, 因此 WGMMMA 路径不
1898 // 需要 1024 字节对齐也能得到正确数据。
1899 //
1900 // 本 kernel 消费者: 使用 ldmatrix + 手写 swizzle<64>() 来计算
1901 // smem 物理地址。这个函数是纯软件公式, 没有任何 base_offset
1902 // 补偿机制。它假设 smem 基址恰好落在 1024B 边界上 (phase=0)。
1903 // 如果基址只满足 128B 对齐, TMA 硬件会以非零 phase 写入数据,
1904 // 而 ldmatrix 以零 phase 读取 → 物理地址彻底错位 → 全输出错误。
1905 //
1906 // 简言之: WGMMMA 用硬件 descriptor base_offset 自动解决 phase 偏移;
1907 // ldmatrix 路径没有等价机制, 必须靠 1024B 对齐来保证 phase=0。
1908 extern __shared__ __align__(1024) uint8_t smem_tma_mma_ws[];
1909 TmaMmaWSSMem<BM, BN, BK, kStages> &s =
1910     *reinterpret_cast<TmaMmaWSSMem<BM, BN, BK, kStages>*>(smem_tma_mma_ws);
1911 half *s_a = s.A;
1912 half *s_b = s.B;
1913
1914 #pragma nv_diag_suppress static_var_with_dynamic_init
1915 __shared__ cuda::barrier<cuda::thread_scope_block> full[kStages];
1916 __shared__ cuda::barrier<cuda::thread_scope_block> empty[kStages];
1917 #pragma nv_diag_default static_var_with_dynamic_init
1918
1919 const int NUM_K_TILES = div_ceil(K, BK);
1920 const int wg_idx = threadIdx.x / kConsumerThreads;
1921 const int wg_tid = threadIdx.x % kConsumerThreads;
1922
1923 if (threadIdx.x == 0) {
1924     for (int stage = 0; stage < kStages; ++stage) {
1925         init(&full[stage], kConsumerThreads + 1);
1926         init(&empty[stage], kConsumerThreads + 1);
1927     }
1928     tma_fence_proxy_async_shared_cta();
1929 }
1930 __syncthreads();
1931
1932 if (wg_idx == 0) {
1933     // 生产者 Warpgroup (wg_idx==0)
1934     NOTES_V2_REG_DEALLOC(40);
1935     if (wg_tid == 0) {
1936         int stage = 0;

```

```

1937     for (int k = 0; k < NUM_K_TILES; ++k, ++stage) {
1938         if (stage == kStages)
1939             stage = 0;
1940
1941         empty[stage].wait(empty[stage].arrive());
1942
1943         tma_load_2d(&s_a[stage * BM * BK], tensorMapA, k * BK, by * BM,
1944             full[stage]);
1945         tma_load_2d(&s_b[stage * BN * BK], tensorMapB, k * BK, bx * BN,
1946             full[stage]);
1947         tma_arrive_expect_tx(full[stage], (BM * BK + BN * BK) * sizeof(half));
1948     }
1949 }
1950 } else {
1951     // 消费者 Warpgroup (wg_idx==1)
1952     // 初始化: 标记所有 stage 为"空" (可被 Producer 写入)
1953     NOTES_V2_REG_ALLOC(232);
1954
1955     for (int stage = 0; stage < kStages; ++stage) {
1956         [[maybe_unused]] auto token = empty[stage].arrive();
1957     }
1958
1959     const int warp_id = wg_tid / kWarpSize;
1960     const int lane_id = wg_tid % kWarpSize;
1961     // WG1 内形成 2x2 warp grid, warp_m / warp_n 分别表示 warp 在 M/N 方向的索引。
1962     const int warp_m = warp_id % kMmaTileM; // kMmaTileM = 2, {0,1}
1963     const int warp_n = warp_id / kMmaTileM; // kMmaTileN = 2, {0,1}
1964     uint32_t RA[kValTileM][4];
1965     uint32_t RB[kValTileN][2];
1966     uint32_t RC[kValTileM][kValTileN][2] = {0};
1967
1968     const uint32_t smem_a_base_ptr = __cvta_generic_to_shared(s_a);
1969     const uint32_t smem_b_base_ptr = __cvta_generic_to_shared(s_b);
1970     int stage = 0;
1971     for (int k = 0; k < NUM_K_TILES; ++k, ++stage) {
1972         if (stage == kStages)
1973             stage = 0;
1974
1975         full[stage].wait(full[stage].arrive());
1976         tma_fence_proxy_async_shared_cta();
1977
1978         // 与 cp.async pipeline (hgemm_mma_stages_tn 等) 不同, 这里 ldmatrix +
1979         // mma.sync 之前/后 不需要 __syncthreads, 原因在于 TMA + async proxy 的
1980         // 同步机制与 warp specialization 的线程分工:
1981         //
1982         // 1. 生产者-消费者解耦: TMA 拷贝由 producer (wg_idx==0 的单个线程)
1983         //    通过 tma_load_2d + tma_arrive_expect_tx 提交, 而非所有 256 个线程
1984         //    各自做 cp.async。cp.async 需要 __syncthreads 确保所有线程的异步
1985         //    拷贝都完成后进入计算; 而 TMA 路径用 cuda::barrier 的 full/empty
1986         //    协议完成 CTA 级同步——full[stage].wait() 返回时, producer 已通过
1987         //    expect_tx 声明了完整的传输字节数, 且硬件保证这些字节已全部写入 smem。
1988         //
1989         // 2. async proxy fence 替代 __syncthreads: fence_proxy_async(space_shared)
1990         //    在 async proxy (TMA 写入通道) 和后续 smem load (ldmatrix 读取通道)
1991         //    之间建立内存序。它确保 fence 之前的所有 async proxy 写操作 (TMA)
1992         //    对 fence 之后的所有 smem 读操作 (ldmatrix) 可见。这是一个轻量级
1993         //    proxy 间 ordering, 不需要阻塞线程, 也不会让 warp 等待其他 warp。
1994         //
1995         // 3. 消费者内部 warp 间无依赖: WG1 内的 4 个 warp (2x2 grid) 各自独立
1996         //    读取 smem 中互不重叠的 A/B 区域 (warp_m 和 warp_n 分别索引不同的
1997         //    行区间)。它们之间不需要读写共享数据, 因此不需要 __syncthreads 来
1998         //    对齐 warp 间的执行进度。mma.sync 本身是 warp 级同步指令, 保证同一
1999         //    warp 内 32 个线程的 ldmatrix 和 mma 操作正确协作。

```

```

2000
2001 // 注意: 这里已经没有NUM_K_TILES循环了, 因为K loop load已经被WG0生产者处理了
2002 #pragma unroll
2003 for (int k_step = 0; k_step < kValTileK; ++k_step) {
2004
2005 #pragma unroll
2006 for (int i = 0; i < kValTileM; ++i) { // kValTileM = 4
2007 // kMmaM * kValTileM = 16*4 = 64, 每个消费者 warp 在 M 方向覆盖 64 行。
2008 const int warp_smem_a_m = warp_m * (kMmaM * kValTileM) + i * kMmaM;
2009 const int lane_smem_a_m = warp_smem_a_m + lane_id % 16;
2010 // 与 hgemm_mma_stages_tn_swizzle 不同: k_step * kMmaK 必须放在
2011 // lane_smem_a_k 中传给 swizzle<64>(), 而非像 swizzle kernel
2012 // 那样作为 smem_k_offset 加在 swizzle 外部。原因:
2013 // swizzle<64> 作用于完整 BK=64, swizzle 周期覆盖全部 64 列,
2014 // k_step=0 和 k_step=1 的 chunk 会被 XOR 交叉混合 → k_step
2015 // 偏移必须在 swizzle 内部参与 chunk 计算。
2016 // swizzle<kMmaK> 作用于 kMmaK=16, 每个 kMmaK slice 独立
2017 // swizzle, slice 之间互不跨越 → k_step * kMmaK 可以加在外部。
2018 const int lane_smem_a_k = (k_step * kMmaK) + (lane_id / 16) * 8;
2019 const uint32_t lane_smem_a_ptr = smem_a_base_ptr +
2020 (stage * BM * BK + lane_smem_a_m * BK +
2021 swizzle<64>(lane_smem_a_m, lane_smem_a_k)) *
2022 sizeof(half);
2023 LDMATRIX_X4(RA[i][0], RA[i][1], RA[i][2], RA[i][3], lane_smem_a_ptr);
2024 }
2025
2026 #pragma unroll
2027 for (int j = 0; j < kValTileN; ++j) { // kValTileN = 8
2028 // kMmaN * kValTileN = 8*8 = 64, 每个消费者 warp 在 B^T 的 N 方向覆盖 64 行。
2029 const int warp_smem_b_n = warp_n * (kMmaN * kValTileN) + j * kMmaN;
2030 const int lane_smem_b_n = warp_smem_b_n + lane_id % 8;
2031 const int lane_smem_b_k = (k_step * kMmaK) + ((lane_id / 8) % 2) * 8;
2032 const uint32_t lane_smem_b_ptr = smem_b_base_ptr +
2033 (stage * BN * BK + lane_smem_b_n * BK +
2034 swizzle<64>(lane_smem_b_n, lane_smem_b_k)) *
2035 sizeof(half);
2036 LDMATRIX_X2(RB[j][0], RB[j][1], lane_smem_b_ptr);
2037 }
2038
2039 #pragma unroll
2040 for (int i = 0; i < kValTileM; ++i) {
2041 #pragma unroll
2042 for (int j = 0; j < kValTileN; ++j) {
2043 HMMMA16816(RC[i][j][0], RC[i][j][1],
2044 RA[i][0], RA[i][1], RA[i][2], RA[i][3],
2045 RB[j][0], RB[j][1],
2046 RC[i][j][0], RC[i][j][1]);
2047 }
2048 }
2049 }
2050 [[maybe_unused]] auto token = empty[stage].arrive();
2051 }
2052
2053 // Epilogue: 复用 RA[0] 和 RA[1] (已在寄存器中) 作为 shuffle 缓冲,
2054 // 与 hgemm_mma_stages_tn_swizzle 的 epilogue 模式一致。四个相邻
2055 // lane 各持有两个 uint32 fragment; shuffle 汇聚为每行一个 float4,
2056 // 由 lane 0 发出对齐的 128-bit store。
2057 #pragma unroll
2058 for (int i = 0; i < kValTileM; ++i) {
2059 const int store_warp_smem_c_m = warp_m * (kMmaM * kValTileM) + i * kMmaM;
2060 const int store_lane_gmem_c_m = by * BM + store_warp_smem_c_m + lane_id / 4;
2061 #pragma unroll
2062 for (int j = 0; j < kValTileN; ++j) {

```

```

2063     const int store_warp_smem_c_n = warp_n * (kMmaN * kValTileN) + j * kMmaN;
2064     RA[0][0] = RC[i][j][0];
2065     RA[1][0] = RC[i][j][1];
2066     RA[0][1] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 1);
2067     RA[0][2] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 2);
2068     RA[0][3] = __shfl_sync(0xffffffff, RC[i][j][0], lane_id + 3);
2069     RA[1][1] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 1);
2070     RA[1][2] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 2);
2071     RA[1][3] = __shfl_sync(0xffffffff, RC[i][j][1], lane_id + 3);
2072     if (lane_id % 4 == 0) {
2073         const int store_lane_gmem_c_n = bx * BN + store_warp_smem_c_n;
2074         const int store_gmem_c_addr_0 =
2075             store_lane_gmem_c_m * N + store_lane_gmem_c_n;
2076         const int store_gmem_c_addr_1 =
2077             (store_lane_gmem_c_m + 8) * N + store_lane_gmem_c_n;
2078         *reinterpret_cast<float4*>(&C[store_gmem_c_addr_0]) =
2079             *reinterpret_cast<float4*>(&RA[0][0]);
2080         *reinterpret_cast<float4*>(&C[store_gmem_c_addr_1]) =
2081             *reinterpret_cast<float4*>(&RA[1][0]);
2082     }
2083 }
2084 }
2085 }
2086 }
2087
2088 #endif /* NOTES_V2_ENABLE_TMA_MMA_WS */

```

```

1  #pragma once
2  #include "base.cuh"
3  // flash_attn.cuh: Phase 8 FlashAttention 2/3 (MMA/TMA_WS/FA3/CuTe)
4
5  // Phase 8: FlashAttention-2 + MMA m16n8k16)
6  // =====
7  // 面试题点 (FlashAttention-2算法) :
8  // 1. 核心问题: 标准 Attention 的  $O(N^2)$  中间矩阵 ( $S=QK^T$ ) 必须写入 HBM,
9  //    但 HBM 带宽是瓶颈 → FlashAttention-2用 tiling + online softmax 避免写回
10 // 2. FA 三板斧:
11 //    a) Tiling: Q 分块 [Br,d], K/V 沿 seqLen 分块 [Bc,d]
12 //    b) Online Softmax: 迭代更新 m(行max) 和 l(行sum), 无需全局同步
13 //    c) Recomputation(反向): 反向传播时重新计算 S/P, 而非存储中间矩阵
14 // 3. Split-Q 设计: 所有 warp 共享同一块 K, 各 warp 处理 Q 的不同行片段
15 //    - warp_KV=0 (所有 warp 共享 K), warp_QP=warp_id (各 warp 不同 Q 行)
16 //    - 优点: 减少 warp 间通信和 shuffle
17 // 4. Online rescaling 公式 (FA 核心, arXiv:2307.08691) :
18 //    for each K,V tile:
19 //        S_cur = Q @ K^T // 未缩放, 存入 R_S
20 //        m_new = max(m_old, row_max(S_cur * scale))
21 //        P_cur = exp(S_cur * scale - m_new) // ← 写回 R_S 寄存器!
22 //        l_new = exp(m_old - m_new) * l_old + row_sum(P_cur)
23 //        O_new = diag(exp(m_old - m_new)) * O_old + P_cur @ V
24 //    O_final = O_new / l_final
25 // 5. 为什么 R_S 可以直接用作 P@V 的 A 矩阵?
26 //    - R_S 经过 softmax 后, 存储的是  $P = \exp(S - m)$ , 数据仍然是 half 精度
27 //    - 当前实现依赖 m16n8k16 这一路径下约定好的 fragment 布局, 使 softmax
28 //      后的 P 可以继续留在 R_S 中供后面的 P@V 直接消费
29 //    - 这是此实现的寄存器布局复用技巧, 不要背成“所有 MMA A/C fragment
30 //      都天然同构”的通用结论
31 //
32 // 本实现参考: FlashAttention-2 (Dao et al., arXiv:2307.08691)
33 // 从 LeetCUDA flash-attn/mma/basic/flash_attn_mma_split_q.cu 提取
34 // Grid: ((N + 63) / 64, B * H, 1), Br=64
35 // Block: (128, 1, 1), kNumThreads=kWarpSize×kMmaTileSeqLenQ×kMmaTileSeqLenK=128
36 // source: LeetCUDA/kernels/flash-attn/mma/basic/flash_attn_mma_split_q.cu

```

```

37
38 // =====
39 // FlashAttention-2 Split-Q Kernel (完整实现)
40 // =====
41 // Q,K,V,0: [batch_size, num_heads, seq_len, head_dim], [B,H,N,d]
42 //
43 // Tile 设计 (以 kHeadDim=64 为例) :
44 //   Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ = 16*{4,8}*1 = {64,128}
45 //   Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK = 8*1*8 = 64
46 //   Warp 布局: 4 warps, warp_QP 0~3 各处理 16 行, warp_KV=0 共享 K
47 //
48 // 执行流程:
49 //   1) 预加载 Q[Br,d] 到 smem (只加载一次, split-Q 的核心优势)
50 //   2) 外循环: 沿 K seqlen 分块迭代 (Tc = seqlen/Bc)
51 //   3)   3a: cp.async 加载当前 V + 预加载下一个 K (多 stage pipeline)
52 //   4)   3b: Q@K^T — 沿 head dim 内循环 → ldmatrix Q/K → HMMA16816
53 //   5)   3c: Online Safe Softmax — warp reduce max/row → exp(S*scale - max)
54 //       → 关键: 将 P 写回 R_S 寄存器 (替换 S), R_S 现在存储 P matrix
55 //   6)   3d: P@V — 沿 V_Bc 内循环 → ldmatrix V (transposed) → 直接用 R_S 做 A
56 //       → 当前实现依赖同一组寄存器布局约定, 避免为 P@V 额外重组 P fragment
57 //   7)   3e: Online rescaling — O_new = exp(m_old-m_new)*O_old + P@V
58 //   8) 最终 rescale: O_final = (1/l_final) * O_final
59 //   9) Epilogue: warp shuffle + 128-bit collective store
60 //
61 // ★ Pad 与手动 XOR swizzle 的 profile 结论 (SM120, B=1,H=32,N=4096,D=64) :
62 //   - XOR 能消除目标 ldmatrix lane pattern 的 bank conflict, 但收益是局部的;
63 //     XOR 本身及 `^` 的计算不是主要瓶颈, nvcc 已将该式化简为 bit operations。
64 //   - compact Q/K/V XOR 改变了 cp.async 的 shared destination pattern: LDGSTS
65 //     wavefronts 从 pad 的 30.15M 增至 68.16M (2.26x), long-scoreboard、
66 //     LG-throttle、MIO-throttle 分别约为 pad 的 3.25x、2.74x、1.60x。
67 //   - compact XOR 虽节省约 5 KiB smem, 但没有提高此 kernel 的 occupancy; 最终
68 //     约 120.0 TFLOPS, 显著低于 Q/K/V kPad=8 的 166.6 TFLOPS。
69 //   - 所以当前 kernel 默认对 Q/K/V 都用 kPad=8。保留 swizzle 路径是为了学习和消融: 评价
70 //     shared layout 必须同时观察 ldmatrix reads 与 cp.async/LDGSTS writes, 不能只看
71 //     bank-conflict counter, 也不能只优化 XOR 地址算术。
72 // ---- 寄存器填充辅助函数 (FlashAttention 用, 前移以供 TMA_MMA_WS kernel 使用) ----
73 template <typename T, int M, const int N, const int K = 2>
74 __device__ inline void fill_3D_regs(T (&R)[M][N][K], T val) {
75     #pragma unroll
76     for (int i = 0; i < M; ++i)
77         #pragma unroll
78         for (int j = 0; j < N; ++j)
79             #pragma unroll
80             for (int k = 0; k < K; ++k)
81                 R[i][j][k] = val;
82 }
83
84 template <typename T, int M, const int N = 2>
85 __device__ inline void fill_2D_regs(T (&R)[M][N], T val) {
86     #pragma unroll
87     for (int i = 0; i < M; ++i)
88         #pragma unroll
89         for (int j = 0; j < N; ++j)
90             R[i][j] = val;
91 }
92
93 template <
94     const int kHeadDim,          // head dim: 32, 64, 128
95     const int kMmaAtomM,         // 16 (MMA instruction M dimension)
96     const int kMmaAtomN,         // 8 (MMA instruction N dimension)
97     const int kMmaAtomK,         // 16 (MMA instruction K dimension)
98     const int kMmaAccF32,        // 0=f16 acc, 1=f32 acc (mma.sync.f32.f16.f16.f32)
99     const int kMmaTileSeqLenQ,   // MMA tiles along Q's M dim, 4 → Br=16*4=64

```

```

100     const int kMmaTileSeqLenK,    // MMA tiles along K's N dim, 1 → Bc basis=8
101     const int kMmaTileSeqLenP,    // MMA tiles for P@V M dim, must equal kMmaTileSeqLenQ
102     const int kMmaTileHeadDimV,    // MMA tiles for P@V N dim (head dim direction)
103     const int kValTileSeqLenQ,    // value tiles along Q's M, 1 → Br per warp=16
104     const int kValTileSeqLenK,    // value tiles along K's N, 8 → Bc_warp=8*8=64
105     const int kValTileSeqLenP,    // value tiles for P@V M dim, 1
106     const int kValTileHeadDimV,    // value tiles for P@V N dim, kHeadDim/(8*kMmaTileHeadDimV)
107     const int kStagesK,           // pipeline stages for K: >= 1; NO stages required for Q/V
108     const int kPadQ,              // Q row padding; 0 selects compact XOR swizzle
109     const int kPadK,              // K row padding; 0 selects compact XOR swizzle
110     const int kPadV>             // V row padding; 0 selects compact XOR swizzle
111 __global__ void __launch_bounds__(kWarpSize * kMmaTileSeqLenQ * kMmaTileSeqLenK)
112 flash_attn_mma_stages_split_q(
113     half *Q, half *K, half *V, half *O, int N, int H) {
114     static_assert(kStagesK >= 1, "kStagesK must be >= 1");
115     static_assert(kMmaAccF32 == 0 || kMmaAccF32 == 1, "kMmaAccF32 must be 0 or 1");
116     static_assert(kPadQ >= 0 && kPadK >= 0 && kPadV >= 0,
117         "Q/K/V padding must be non-negative");
118     constexpr int kNRegs = kMmaAccF32 ? 4 : 2;
119     constexpr bool kSwizzleQ = kPadQ == 0;
120     constexpr bool kSwizzleK = kPadK == 0;
121     constexpr bool kSwizzleV = kPadV == 0;
122     constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ; // 16*8*1=128
123     constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK; // 8*1*8=64
124     constexpr int kNumThreads = kWarpSize * kMmaTileSeqLenQ * kMmaTileSeqLenK; // 32*8*1=256
125     const int Tc = (N + Bc - 1) / Bc;
126     // 原始实现默认 seqLen 与 Bc 对齐; 最后一个不完整 tile 需要额外 pad/边界处理。
127     // 这里保留 ceil 写法是为了说明 tile 划分方式, 不等于当前实现已经完整处理了尾 tile。
128     const float scale = 1.0f / sqrtf((float)kHeadDim);
129
130     // Block indexing
131     const int Nb_id = blockIdx.y / H; // batch id
132     const int Nh_id = blockIdx.y % H; // head id
133     const int Q_tile_id = blockIdx.x; // tile id along Q's M dim (Br)
134     const int tid = threadIdx.x;
135     const int warp_id = tid / kWarpSize;
136     const int lane_id = tid % kWarpSize;
137     const int warp_QP = warp_id; // Split-Q: 每个 warp 处理不同的 Q 行片段
138     const int warp_KV = 0; // 所有 warp 共享 K (减少跨 warp 通信)
139
140     // Global memory base offsets for this (batch, head)
141     // 这里默认 Q/K/V 共享同一 per-head 基址布局, 对应 self-attention 场景
142     const int Q_gmem_offset = (Nb_id * H * N + Nh_id * N) * kHeadDim;
143     const int K_gmem_offset = Q_gmem_offset;
144     const int V_gmem_offset = Q_gmem_offset;
145     const int O_gmem_offset = Q_gmem_offset;
146
147     // Thread-to-smem mapping for cooperative load
148     int load_smem_Q_Br = tid / (kNumThreads / Br);
149     int load_smem_Q_d = (tid % (kNumThreads / Br)) * (kHeadDim / (kNumThreads / Br));
150     int load_smem_K_Bc = tid / (kNumThreads / Bc);
151     int load_smem_K_d = (tid % (kNumThreads / Bc)) * (kHeadDim / (kNumThreads / Bc));
152     int load_smem_V_Bc = tid / (kNumThreads / Bc);
153     int load_smem_V_d = (tid % (kNumThreads / Bc)) * (kHeadDim / (kNumThreads / Bc));
154
155     int load_gmem_Q_Br = Q_tile_id * Br + load_smem_Q_Br;
156     if (load_gmem_Q_Br >= N)
157         return;
158
159     // Shared memory layout
160     // Q/K/V independently use padded row-major when kPad* > 0 and compact XOR
161     // swizzle when kPad* == 0. Padding and XOR are never combined per operand.
162     // The swizzled physical layout is [col / 16][row][16]; swizzle<16>() selects

```

```

163 // the 0/8 phase inside the final 16-half tile.
164 extern __shared__ half smem[];
165 constexpr int Q_tile_size = Br * (kHeadDim + kPadQ);
166 constexpr int K_tile_size = Bc * (kHeadDim + kPadK);
167 constexpr int kSmemStrideQ = kHeadDim + kPadQ;
168 constexpr int kSmemStrideK = kHeadDim + kPadK;
169 constexpr int kSmemStrideV = kHeadDim + kPadV;
170 half *Q_tile_smem = smem;
171 half *K_tile_smem = Q_tile_smem + Q_tile_size;
172 half *V_tile_smem = K_tile_smem + kStagesK * K_tile_size;
173
174 uint32_t smem_Q_base_ptr = __cvta_generic_to_shared(Q_tile_smem);
175 uint32_t smem_K_base_ptr = __cvta_generic_to_shared(K_tile_smem);
176 uint32_t smem_V_base_ptr = __cvta_generic_to_shared(V_tile_smem);
177
178 // Online Softmax persistent state
179 // lane_block_row_max_old[i][r]: running max for row r of warp tile i
180 // lane_block_row_sum_old[i][r]: running denominator l for row r of warp tile i
181 float lane_block_row_max_old[kValTileSeqLenQ][2];
182 float lane_block_row_sum_old[kValTileSeqLenQ][2];
183 fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_block_row_max_old, -INFINITY);
184 fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_block_row_sum_old, 0.0f);
185
186 uint32_t R_Q[kValTileSeqLenQ][4]; // Q regs
187 uint32_t R_K[kValTileSeqLenK][2]; // K regs
188 uint32_t R_V[kValTileHeadDimV][2]; // V regs
189 // R_S / R_O / R_D 都按 mma.sync.aligned.m16n8k16 的 fragment 约定存储。
190 // f16 acc: 每 tile 2 个 uint32 (4 half); f32 acc: 每 tile 4 个 uint32 (4 f32) 。
191 // 后续 softmax、P@V、online rescale 都直接围绕这组 fragment 布局做寄存器内变换。
192 uint32_t R_S[kValTileSeqLenQ][kValTileSeqLenK][kNRegs]; // S=Q@K^T / P=softmax(S)
193 uint32_t R_O[kValTileSeqLenP][kValTileHeadDimV][kNRegs]; // O for current tile
194 uint32_t R_D[kValTileSeqLenP][kValTileHeadDimV][kNRegs]; // O accumulator (final output)
195
196 fill_3D_regs<uint32_t, kValTileSeqLenQ, kValTileSeqLenK, kNRegs>(R_S, 0);
197 fill_3D_regs<uint32_t, kValTileSeqLenP, kValTileHeadDimV, kNRegs>(R_D, 0);
198
199 // =====
200 // Step 1: 加载 Q[Br, d] 到 shared memory (整个外循环只加载一次)
201 // =====
202 {
203     int load_gmem_Q_addr =
204         Q_gmem_offset + load_gmem_Q_Br * kHeadDim + load_smem_Q_d;
205     if constexpr (kSwizzleQ) {
206 #pragma unroll
207         for (int i = 0; i < (kHeadDim / (kNumThreads / Br)); i += 8) {
208             int col = load_smem_Q_d + i;
209             uint32_t ptr = smem_Q_base_ptr +
210                 ((col / kMmaAtomK) * Br * kMmaAtomK +
211                 load_smem_Q_Br * kMmaAtomK +
212                 swizzle<kMmaAtomK>(load_smem_Q_Br, col % kMmaAtomK)) *
213                 sizeof(half);
214             CP_ASYNC_CG(ptr, &Q[load_gmem_Q_addr + i], 16);
215         }
216     } else {
217         uint32_t load_smem_Q_ptr =
218             smem_Q_base_ptr +
219             (load_smem_Q_Br * kSmemStrideQ + load_smem_Q_d) * sizeof(half);
220 #pragma unroll
221         for (int i = 0; i < (kHeadDim / (kNumThreads / Br)); i += 8) {
222             CP_ASYNC_CG(load_smem_Q_ptr + i * 2, &Q[load_gmem_Q_addr + i], 16);
223         }
224     }
225     CP_ASYNC_COMMIT_GROUP();

```

```

226 }
227
228 // =====
229 // Step 2: 预加载前 (kStagesK-1) 个 K tile (多 stage pipeline 预热)
230 // 注意: Q 由 blockIdx.x 固定到当前 Q tile; 而 K/V 的 seqlen 遍历始终从 tile 0 开始,
231 // 后续在外循环里不断递增至 tile 1/2/3/.../Tc-1。
232 // =====
233 if constexpr (kStagesK > 1) {
234 #pragma unroll
235 for (int stage = 0; stage < (kStagesK - 1); ++stage) {
236     int load_gmem_K_Bc = stage * Bc + load_smem_K_Bc;
237     int load_gmem_K_addr =
238         K_gmem_offset + load_gmem_K_Bc * kHeadDim + load_smem_K_d;
239     if constexpr (kSwizzleK) {
240 #pragma unroll
241         for (int i = 0; i < (kHeadDim / (kNumThreads / Bc)); i += 8) {
242             int col = load_smem_K_d + i;
243             uint32_t ptr = smem_K_base_ptr +
244                 (stage * K_tile_size +
245                  (col / kMmaAtomK) * Bc * kMmaAtomK +
246                  load_smem_K_Bc * kMmaAtomK +
247                  swizzle<kMmaAtomK>(load_smem_K_Bc, col % kMmaAtomK)) *
248                 sizeof(half);
249             CP_ASYNC_CG(ptr, &K[load_gmem_K_addr + i], 16);
250         }
251     } else {
252         uint32_t load_smem_K_ptr =
253             smem_K_base_ptr +
254             (stage * K_tile_size + load_smem_K_Bc * kSmemStrideK +
255              load_smem_K_d) *
256             sizeof(half);
257 #pragma unroll
258         for (int i = 0; i < (kHeadDim / (kNumThreads / Bc)); i += 8) {
259             CP_ASYNC_CG(load_smem_K_ptr + i * 2, &K[load_gmem_K_addr + i], 16);
260         }
261     }
262     CP_ASYNC_COMMIT_GROUP();
263 }
264 CP_ASYNC_WAIT_GROUP(kStagesK - 2);
265 __syncthreads();
266 }
267
268 // =====
269 // Step 3: 外循环 — 沿 K seqlen 迭代 (Tc = ceil(seqlen/Bc))
270 // 每次迭代处理一个 K[Bc,d] + V[Bc,d] tile
271 // =====
272 #pragma unroll 1
273 for (int tile_K_seqlen = 0; tile_K_seqlen < Tc; ++tile_K_seqlen) {
274     int smem_sel = tile_K_seqlen % kStagesK;
275     int smem_sel_next = (tile_K_seqlen + (kStagesK - 1)) % kStagesK;
276
277     // ---- 3a: 异步加载 K/V tile (多 stage pipeline) ----
278     // kStagesK>1: V 加载当前 tile, K 预取下一 tile (pipeline)
279     // kStagesK=1: V 和 K 都加载当前 tile (smem_sel==smem_sel_next==0),
280     // 无 pipeline 预取, 每轮 QK 前必须 wait 当前 K 就绪。
281     // Load current V tile (no pipeline for V — one stage is enough)
282     {
283         int load_gmem_V_Bc = tile_K_seqlen * Bc + load_smem_V_Bc;
284         int load_gmem_V_addr =
285             V_gmem_offset + load_gmem_V_Bc * kHeadDim + load_smem_V_d;
286         if constexpr (kSwizzleV) {
287 #pragma unroll
288             for (int i = 0; i < (kHeadDim / (kNumThreads / Bc)); i += 8) {

```

```

289     int col = load_smem_V_d + i;
290     uint32_t ptr = smem_V_base_ptr +
291         ((col / kMmaAtomK) * Bc * kMmaAtomK +
292         load_smem_V_Bc * kMmaAtomK +
293         swizzle<kMmaAtomK>(load_smem_V_Bc, col % kMmaAtomK)) *
294         sizeof(half);
295     CP_ASYNC_CG(ptr, &V[load_gmem_V_addr + i], 16);
296 }
297 } else {
298     uint32_t load_smem_V_ptr =
299         smem_V_base_ptr +
300         (load_smem_V_Bc * kSmemStrideV + load_smem_V_d) * sizeof(half);
301 #pragma unroll
302     for (int i = 0; i < (kHeadDim / (kNumThreads / Bc)); i += 8) {
303         CP_ASYNC_CG(load_smem_V_ptr + i * 2, &V[load_gmem_V_addr + i], 16);
304     }
305 }
306 CP_ASYNC_COMMIT_GROUP();
307 }
308
309 if constexpr (kStagesK > 1) {
310     // Prefetch next K tile (pipelined)
311     if ((tile_K_seqlen + 1) < Tc) {
312         int load_gmem_K_Bc = (tile_K_seqlen + 1) * Bc + load_smem_K_Bc;
313         int load_gmem_K_addr =
314             K_gmem_offset + load_gmem_K_Bc * kHeadDim + load_smem_K_d;
315         if constexpr (kSwizzleK) {
316 #pragma unroll
317             for (int i = 0; i < (kHeadDim / (kNumThreads / Bc)); i += 8) {
318                 int col = load_smem_K_d + i;
319                 uint32_t ptr = smem_K_base_ptr +
320                     (smem_sel_next * K_tile_size +
321                     (col / kMmaAtomK) * Bc * kMmaAtomK +
322                     load_smem_K_Bc * kMmaAtomK +
323                     swizzle<kMmaAtomK>(load_smem_K_Bc, col % kMmaAtomK)) *
324                     sizeof(half);
325                 CP_ASYNC_CG(ptr, &K[load_gmem_K_addr + i], 16);
326             }
327         } else {
328             uint32_t load_smem_K_ptr =
329                 smem_K_base_ptr +
330                 (smem_sel_next * K_tile_size +
331                 load_smem_K_Bc * kSmemStrideK +
332                 load_smem_K_d) *
333                 sizeof(half);
334 #pragma unroll
335             for (int i = 0; i < (kHeadDim / (kNumThreads / Bc)); i += 8) {
336                 CP_ASYNC_CG(load_smem_K_ptr + i * 2, &K[load_gmem_K_addr + i], 16);
337             }
338         }
339         CP_ASYNC_COMMIT_GROUP();
340     }
341 } else {
342     // kStagesK==1: 加载当前 K tile (smem_sel_next == smem_sel == 0) 。
343     // Step 2 预加载循环 (kStagesK-1=0) 不执行, 所以每轮 3a 必须加载当前 K。
344     {
345         int load_gmem_K_Bc = tile_K_seqlen * Bc + load_smem_K_Bc;
346         int load_gmem_K_addr =
347             K_gmem_offset + load_gmem_K_Bc * kHeadDim + load_smem_K_d;
348         if constexpr (kSwizzleK) {
349 #pragma unroll
350             for (int i = 0; i < (kHeadDim / (kNumThreads / Bc)); i += 8) {
351                 int col = load_smem_K_d + i;

```

```

352     uint32_t ptr = smem_K_base_ptr +
353         (smem_sel * K_tile_size +
354         (col / kMmaAtomK) * Bc * kMmaAtomK +
355         load_smem_K_Bc * kMmaAtomK +
356         swizzle<kMmaAtomK>(load_smem_K_Bc, col % kMmaAtomK)) *
357         sizeof(half);
358     CP_ASYNC.CG(ptr, &K[load_gmem_K_addr + i], 16);
359 }
360 } else {
361     uint32_t load_smem_K_ptr =
362         smem_K_base_ptr +
363         (smem_sel * K_tile_size +
364         load_smem_K_Bc * kSmemStrideK +
365         load_smem_K_d) *
366         sizeof(half);
367 #pragma unroll
368     for (int i = 0; i < (kHeadDim / (kNumThreads / Bc)); i += 8) {
369         CP_ASYNC.CG(load_smem_K_ptr + i * 2, &K[load_gmem_K_addr + i], 16);
370     }
371 }
372 CP_ASYNC.COMMIT_GROUP();
373 }
374 }
375
376 // 3b: Q@K^T = S[Br, Bc] — 沿 head dim (d/kMmaAtomK=16) 内循环
377 // kStagesK==1: 3a 刚提交当前 K 的 cp.async, 必须 wait 后才能 ldmatrix。
378 // kStagesK>1: K 已在上一轮预取并 wait (Step 2 或 3e 末尾的 wait), 无需再 wait。
379 if constexpr (kStagesK == 1) {
380     CP_ASYNC.WAIT_GROUP(0);
381     __syncthreads();
382 }
383 fill_3D_regs<uint32_t, kValTileSeqLenQ, kValTileSeqLenK, kNRegs>(R_S, 0);
384 #pragma unroll
385 for (int tile_K_d = 0; tile_K_d < (kHeadDim / kMmaAtomK); ++tile_K_d) {
386     // ldmatrix.x4: 加载 Q 的 m16k16 片段到 R_Q
387     #pragma unroll
388     for (int i = 0; i < kValTileSeqLenQ; ++i) {
389         int warp_smem_Q_Br =
390             warp_QP * (kMmaAtomM * kValTileSeqLenQ) + i * kMmaAtomM;
391         int lane_smem_Q_Br =
392             warp_smem_Q_Br + lane_id % 16; // ldmatrix uses 16 lanes
393         int lane_smem_Q_d = tile_K_d * kMmaAtomK + (lane_id / 16) * 8; // 0, 8
394         uint32_t lane_smem_Q_ptr;
395         if constexpr (kSwizzleQ) {
396             lane_smem_Q_ptr = smem_Q_base_ptr +
397                 ((lane_smem_Q_d / kMmaAtomK) * Br * kMmaAtomK +
398                 lane_smem_Q_Br * kMmaAtomK +
399                 swizzle<kMmaAtomK>(lane_smem_Q_Br,
400                 lane_smem_Q_d % kMmaAtomK)) *
401                 sizeof(half);
402         } else {
403             lane_smem_Q_ptr = smem_Q_base_ptr +
404                 (lane_smem_Q_Br * kSmemStrideQ + lane_smem_Q_d) * sizeof(half);
405         }
406         LDMATRIX_X4(R_Q[i][0], R_Q[i][1], R_Q[i][2], R_Q[i][3],
407             lane_smem_Q_ptr);
408     }
409
410     // ldmatrix.x2: 加载 K 的 k16n8 片段到 R_K
411     // K[Bc,d] row-major = K^T[d,Bc] col-major (NT 布局的 B 矩阵)
412     #pragma unroll
413     for (int j = 0; j < kValTileSeqLenK; ++j) {
414         int warp_smem_K_Bc =

```

```

415     warp_KV * (kMmaAtomN * kValTileSeqLenK) + j * kMmaAtomN;
416     int lane_smem_K_Bc =
417         warp_smem_K_Bc + lane_id % 8; // ldmatrix B uses 8 lanes
418     int lane_smem_K_d =
419         tile_K_d * kMmaAtomK + ((lane_id / 8) % 2) * 8; // 0, 8
420     uint32_t lane_smem_K_ptr;
421     if constexpr (kSwizzleK) {
422         lane_smem_K_ptr = smem_K_base_ptr +
423             (smem_sel * K_tile_size +
424              (lane_smem_K_d / kMmaAtomK) * Bc * kMmaAtomK +
425              lane_smem_K_Bc * kMmaAtomK +
426              swizzle<kMmaAtomK>(lane_smem_K_Bc,
427                               lane_smem_K_d % kMmaAtomK)) *
428             sizeof(half);
429     } else {
430         lane_smem_K_ptr = smem_K_base_ptr +
431             (smem_sel * K_tile_size + lane_smem_K_Bc * kSmemStrideK +
432              lane_smem_K_d) *
433             sizeof(half);
434     }
435     LDMATRIX_X2(R_K[j][0], R_K[j][1], lane_smem_K_ptr);
436 }
437
438 // MMA: S[tile] += Q[tile] @ K^T[tile]
439 #pragma unroll
440 for (int i = 0; i < kValTileSeqLenQ; ++i) {
441     #pragma unroll
442     for (int j = 0; j < kValTileSeqLenK; ++j) {
443         if constexpr (kMmaAccF32) {
444             MMA16816F32(R_S[i][j][0], R_S[i][j][1], R_S[i][j][2], R_S[i][j][3],
445                        R_Q[i][0], R_Q[i][1], R_Q[i][2], R_Q[i][3],
446                        R_K[j][0], R_K[j][1],
447                        R_S[i][j][0], R_S[i][j][1], R_S[i][j][2], R_S[i][j][3]);
448         } else {
449             MMA16816(R_S[i][j][0], R_S[i][j][1], R_Q[i][0], R_Q[i][1], R_Q[i][2],
450                    R_Q[i][3], R_K[j][0], R_K[j][1], R_S[i][j][0],
451                    R_S[i][j][1]);
452         }
453     }
454 }
455 } // end loop over d
456 __syncthreads();
457
458 // =====
459 // 3c: Online Safe Softmax — row-wise max + exp + sum, then store P back to R_S
460 // =====
461 // MMA C fragment layout for m16n8k16 (PTX ISA 对应的线程寄存器分布):
462 // - R_S[i][j][0] 对应当前 16x8 tile 的 rows 0~7 里的两个 half 值 {c0,c1}
463 // - R_S[i][j][1] 对应 rows 8~15 里的两个 half 值 {c2,c3}
464 // - lane 0~3 持有 row 0 的片段, lane 4~7 持有 row 1, ..., lane 28~31 持有 row 7
465 // - 对于 rows 8~15 也是同样的 lane 分组, 只是读取的是 reg[1]
466 // 这就是为什么后面做 row max / row sum 时, warp 内真正参与同一行归约的是
467 // {0,4,8,...,28} 这一类 4-lane 子组, 而不是整 warp 32 个线程。
468 // Each (i, j) pair = one 16x8 MMA tile; there are kValTileSeqLenQ x
469 // kValTileSeqLenK tiles.
470
471 float lane_row_max_new[kValTileSeqLenQ][2];
472 float lane_row_sum_new[kValTileSeqLenQ][2];
473 fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_row_max_new, -INFINITY);
474 fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_row_sum_new, 0.0f);
475
476 // Pass 1: Thread-level reduce max across all columns (kValTileSeqLenK tiles)
477 #pragma unroll

```

```

478     for (int i = 0; i < kValTileSeqLenQ; ++i) {
479 #pragma unroll
480     for (int j = 0; j < kValTileSeqLenK; ++j) {
481         float tmp_max_0, tmp_max_1;
482         if constexpr (kMmaAccF32) {
483             float *t_fptr_S = reinterpret_cast<float *>(&R_S[i][j][0]);
484             tmp_max_0 = max(t_fptr_S[0], t_fptr_S[1]) * scale;
485             tmp_max_1 = max(t_fptr_S[2], t_fptr_S[3]) * scale;
486         } else {
487             float2 t_reg_S_0 =
488                 __half22float2(HALF2(R_S[i][j][0])); // rows 0~7: {c0, c1}
489             float2 t_reg_S_1 =
490                 __half22float2(HALF2(R_S[i][j][1])); // rows 8~15: {c2, c3}
491             tmp_max_0 = max(t_reg_S_0.x, t_reg_S_0.y) * scale;
492             tmp_max_1 = max(t_reg_S_1.x, t_reg_S_1.y) * scale;
493         }
494         lane_row_max_new[i][0] = max(lane_row_max_new[i][0], tmp_max_0);
495         lane_row_max_new[i][1] = max(lane_row_max_new[i][1], tmp_max_1);
496     }
497     // Warp-level reduce max (kWarpWidth = 4 for Q@K^T — only lanes
498     // {0,4,8,...,28} hold valid data)
499     lane_row_max_new[i][0] =
500         warp_reduce_max<4, float>(lane_row_max_new[i][0]);
501     lane_row_max_new[i][1] =
502         warp_reduce_max<4, float>(lane_row_max_new[i][1]);
503 }
504
505 // Pass 2: Compute P = exp(S*scale - m_new), store back to R_S
506 // 面试关键点: 这里将 P 写回 R_S 寄存器!
507 // 为什么可以? 当前实现依赖 m16n8k16 这一路径下约定好的 fragment 布局,
508 // 使 softmax 后的 P 能继续留在 R_S 中供后面的 P@V 直接消费, 无需额外重组。
509 #pragma unroll
510 for (int i = 0; i < kValTileSeqLenQ; ++i) {
511     // m_new = max(m_old, m_cur)
512     float block_row_max_new_0 =
513         max(lane_block_row_max_old[i][0], lane_row_max_new[i][0]);
514     float block_row_max_new_1 =
515         max(lane_block_row_max_old[i][1], lane_row_max_new[i][1]);
516
517 #pragma unroll
518     for (int j = 0; j < kValTileSeqLenK; ++j) {
519         if constexpr (kMmaAccF32) {
520             float *t_fptr_S = reinterpret_cast<float *>(&R_S[i][j][0]);
521             half *t_hptr_S = reinterpret_cast<half *>(&R_S[i][j][0]);
522             t_fptr_S[0] = __expf(__fmaf_rn(t_fptr_S[0], scale, -block_row_max_new_0));
523             t_fptr_S[1] = __expf(__fmaf_rn(t_fptr_S[1], scale, -block_row_max_new_0));
524             t_fptr_S[2] = __expf(__fmaf_rn(t_fptr_S[2], scale, -block_row_max_new_1));
525             t_fptr_S[3] = __expf(__fmaf_rn(t_fptr_S[3], scale, -block_row_max_new_1));
526             lane_row_sum_new[i][0] += (t_fptr_S[0] + t_fptr_S[1]);
527             lane_row_sum_new[i][1] += (t_fptr_S[2] + t_fptr_S[3]);
528             t_hptr_S[0] = __float2half_rn(t_fptr_S[0]);
529             t_hptr_S[1] = __float2half_rn(t_fptr_S[1]);
530             t_hptr_S[2] = __float2half_rn(t_fptr_S[2]);
531             t_hptr_S[3] = __float2half_rn(t_fptr_S[3]);
532         } else {
533             float2 t_reg_S_0 = __half22float2(HALF2(R_S[i][j][0]));
534             float2 t_reg_S_1 = __half22float2(HALF2(R_S[i][j][1]));
535             t_reg_S_0.x =
536                 __expf(__fmaf_rn(t_reg_S_0.x, scale, -block_row_max_new_0));
537             t_reg_S_0.y =
538                 __expf(__fmaf_rn(t_reg_S_0.y, scale, -block_row_max_new_0));
539             t_reg_S_1.x =
540                 __expf(__fmaf_rn(t_reg_S_1.x, scale, -block_row_max_new_1));

```

```

541     t_reg_S_1.y =
542         __expf(__fmaf_rn(t_reg_S_1.y, scale, -block_row_max_new_1));
543     lane_row_sum_new[i][0] += (t_reg_S_0.x + t_reg_S_0.y);
544     lane_row_sum_new[i][1] += (t_reg_S_1.x + t_reg_S_1.y);
545     HALF2(R_S[i][j][0]) = __float22half2_rn(t_reg_S_0);
546     HALF2(R_S[i][j][1]) = __float22half2_rn(t_reg_S_1);
547 }
548 }
549
550 // Warp-level reduce sum (kWarpWidth = 4, same as max)
551 lane_row_sum_new[i][0] =
552     warp_reduce_sum<4, float>(lane_row_sum_new[i][0]);
553 lane_row_sum_new[i][1] =
554     warp_reduce_sum<4, float>(lane_row_sum_new[i][1]);
555 }
556 __syncthreads();
557
558 // =====
559 // 3d: P@V — P[Br,Bc] @ V[Bc,d] = O[Br,d]
560 // =====
561 // Wait for V to be ready before computing P@V
562 if constexpr (kStagesK > 1) {
563     if ((tile_K_seqLen + 1) < Tc) {
564         CP_ASYNC_WAIT_GROUP(1);
565     } else {
566         CP_ASYNC_WAIT_GROUP(0);
567     }
568 } else {
569     CP_ASYNC_WAIT_GROUP(0);
570 }
571 __syncthreads();
572
573 fill_3D_regs<uint32_t, kValTileSeqLenP, kValTileHeadDimV, kNRegs>(R_0, 0);
574
575 // tile_V_Bc: iterate over chunks of Bc/K=16 columns in P matrix
576 // Bc=kMmaAtomK=16 → 1 iteration for kHeadDim≤64 configurations
577 #pragma unroll
578 for (int tile_V_Bc = 0; tile_V_Bc < (Bc / kMmaAtomK); ++tile_V_Bc) {
579     // ldmatrix.x2.trans: load V[Bc,d] with transposition
580     // V is row-major [Bc,d], but NN matmul needs B matrix in col-major → use
581     // transposed ldmatrix
582 #pragma unroll
583     for (int j = 0; j < kValTileHeadDimV; ++j) {
584         int warp_smem_V_d =
585             warp_KV * (kMmaAtomN * kValTileHeadDimV) + j * kMmaAtomN;
586         int lane_smem_V_Bc = tile_V_Bc * kMmaAtomK + lane_id % 16;
587         int lane_smem_V_d = warp_smem_V_d;
588         uint32_t lane_smem_V_ptr;
589         if constexpr (kSwizzleV) {
590             lane_smem_V_ptr = smem_V_base_ptr +
591                 ((lane_smem_V_d / kMmaAtomK) * Bc * kMmaAtomK +
592                 lane_smem_V_Bc * kMmaAtomK +
593                 swizzle<kMmaAtomK>(lane_smem_V_Bc,
594                 lane_smem_V_d % kMmaAtomK)) *
595                 sizeof(half);
596         } else {
597             lane_smem_V_ptr = smem_V_base_ptr +
598                 (lane_smem_V_Bc * kSmemStrideV + lane_smem_V_d) * sizeof(half);
599         }
600         LDMATRIX_X2_T(R_V[j][0], R_V[j][1], lane_smem_V_ptr);
601     }
602
603     // P matrix layout in R_S[i][j][2]:

```

```

604 // MMA = m16n8k16, Br=16x4=64, Bc=8x8=64, layout: 4 warps
605 // | 64x64 | warp_KV 0 |
606 // | warp_QP 0 | MMA 0 ... MMA 0 (x8) |
607 // | warp_QP 1 | MMA 1 ... MMA 1 (x8) |
608 // | warp_QP 2 | MMA 2 ... MMA 2 (x8) |
609 // | warp_QP 3 | MMA 3 ... MMA 3 (x8) |
610 // tile_V_Bc selects which 16-column slice of P to use:
611 // tile_V_Bc=0 → cols 0:16 → MMA indices 0,1 → w=0
612 // tile_V_Bc=1 → cols 16:32 → MMA indices 2,3 → w=2
613 // tile_V_Bc=2 → cols 32:48 → MMA indices 4,5 → w=4
614 // tile_V_Bc=3 → cols 48:64 → MMA indices 6,7 → w=6
615 // 对应的 MMA A fragment 布局可以这样记:
616 // - rows 0~7: lane 0~3 → {a0,a1} 与 {a4,a5}, lane 4~7 → 下一行, 依次类推
617 // - rows 8~15: lane 0~3 → {a2,a3} 与 {a6,a7}, lane 4~7 → 下一行, 依次类推
618 // 当前实现正是利用这一路径下 A fragment 与前面生成的 P fragment 可以直接对接,
619 // 才能把 R_S 中的 P 直接喂给 HMMA16816 做 P@V; 复习时不要把它背成对所有 MMA
620 // fragment 都无条件成立的通用结论。layout转换逻辑:
621 // C layout: 8 x (16 x 8) layout → A layout: 4 x (16 x 16) layout
622 int w = tile_V_Bc * 2;
623 #pragma unroll
624 for (int i = 0; i < kValTileSeqLenP; ++i) {
625 #pragma unroll
626 for (int j = 0; j < kValTileHeadDimV; ++j) {
627 if constexpr (kMmaAccF32) {
628 HMMA16816F32(R_0[i][j][0], R_0[i][j][1], R_0[i][j][2], R_0[i][j][3],
629 R_S[i][w][0], R_S[i][w][1], R_S[i][w + 1][0], R_S[i][w + 1][1],
630 R_V[j][0], R_V[j][1],
631 R_0[i][j][0], R_0[i][j][1], R_0[i][j][2], R_0[i][j][3]);
632 } else {
633 HMMA16816(R_0[i][j][0], R_0[i][j][1],
634 R_S[i][w][0], R_S[i][w][1], R_S[i][w + 1][0], R_S[i][w + 1][1],
635 R_V[j][0], R_V[j][1],
636 R_0[i][j][0], R_0[i][j][1]);
637 }
638 }
639 }
640 } // end for tile_V_Bc
641 __syncthreads();
642
643 // =====
644 // 3e: Online rescaling — 0_new = exp(m_old - m_new) * 0_old + P@V
645 // =====
646 // 公式来源: FA2 paper Eq.(7-8), 使用 exp(m_old - m_new) 做 0 与 l 的 rescale
647 #pragma unroll
648 for (int i = 0; i < kValTileSeqLenP; ++i) {
649 float block_row_max_new_0 = lane_row_max_new[i][0];
650 float block_row_max_new_1 = lane_row_max_new[i][1];
651 float block_row_sum_new_0 = lane_row_sum_new[i][0];
652 float block_row_sum_new_1 = lane_row_sum_new[i][1];
653
654 float block_row_max_old_0 = lane_block_row_max_old[i][0];
655 float block_row_max_old_1 = lane_block_row_max_old[i][1];
656
657 block_row_max_new_0 = max(block_row_max_old_0, block_row_max_new_0);
658 block_row_max_new_1 = max(block_row_max_old_1, block_row_max_new_1);
659
660 // Handle first iteration: m_old = -inf, need to use m_new directly
661 block_row_max_old_0 =
662 (tile_K_seqlen > 0 ? block_row_max_old_0 : block_row_max_new_0);
663 block_row_max_old_1 =
664 (tile_K_seqlen > 0 ? block_row_max_old_1 : block_row_max_new_1);
665
666 float rescale_o_factor_0 =

```

```

667     __expf(block_row_max_old_0 - block_row_max_new_0);
668     float rescale_o_factor_1 =
669         __expf(block_row_max_old_1 - block_row_max_new_1);
670
671     // Rescale O_old + Add P@V in one fused step
672     #pragma unroll
673     for (int j = 0; j < kValTileHeadDimV; ++j) {
674         if constexpr (kMmaAccF32) {
675             float *t_fptr_0 = reinterpret_cast<float *>(&R_0[i][j][0]);
676             float *t_fptr_D = reinterpret_cast<float *>(&R_D[i][j][0]);
677             t_fptr_D[0] = __fmaf_rn(rescale_o_factor_0, t_fptr_D[0], t_fptr_0[0]);
678             t_fptr_D[1] = __fmaf_rn(rescale_o_factor_0, t_fptr_D[1], t_fptr_0[1]);
679             t_fptr_D[2] = __fmaf_rn(rescale_o_factor_1, t_fptr_D[2], t_fptr_0[2]);
680             t_fptr_D[3] = __fmaf_rn(rescale_o_factor_1, t_fptr_D[3], t_fptr_0[3]);
681         } else {
682             float2 t_reg_0_0 = __half22float2(HALF2(R_0[i][j][0]));
683             float2 t_reg_0_1 = __half22float2(HALF2(R_0[i][j][1]));
684             float2 t_reg_D_0 = __half22float2(HALF2(R_D[i][j][0]));
685             float2 t_reg_D_1 = __half22float2(HALF2(R_D[i][j][1]));
686             t_reg_D_0.x = __fmaf_rn(rescale_o_factor_0, t_reg_D_0.x, t_reg_0_0.x);
687             t_reg_D_0.y = __fmaf_rn(rescale_o_factor_0, t_reg_D_0.y, t_reg_0_0.y);
688             t_reg_D_1.x = __fmaf_rn(rescale_o_factor_1, t_reg_D_1.x, t_reg_0_1.x);
689             t_reg_D_1.y = __fmaf_rn(rescale_o_factor_1, t_reg_D_1.y, t_reg_0_1.y);
690             HALF2(R_D[i][j][0]) = __float22half2_rn(t_reg_D_0);
691             HALF2(R_D[i][j][1]) = __float22half2_rn(t_reg_D_1);
692         }
693     }
694
695     // Update l: l_new = exp(m_old - m_new) * l_old + row_sum(P)
696     float block_row_sum_old_0 = lane_block_row_sum_old[i][0];
697     float block_row_sum_old_1 = lane_block_row_sum_old[i][1];
698     lane_block_row_sum_old[i][0] = __fmaf_rn(
699         rescale_o_factor_0, block_row_sum_old_0, block_row_sum_new_0);
700     lane_block_row_sum_old[i][1] = __fmaf_rn(
701         rescale_o_factor_1, block_row_sum_old_1, block_row_sum_new_1);
702
703     // Update m
704     lane_block_row_max_old[i][0] = block_row_max_new_0;
705     lane_block_row_max_old[i][1] = block_row_max_new_1;
706 }
707
708 // Wait for next K tile to be ready in smem before next iteration
709 if constexpr (kStagesK > 1) {
710     if ((tile_K_seqlen + 1) < Tc) {
711         CP_ASYNC_WAIT_GROUP(0);
712     }
713     __syncthreads();
714 }
715 } // end outer loop over K seqlen
716 __syncthreads();
717
718 // =====
719 // Step 4: 最终 rescale — O_final = (1/l_final) * O_final
720 // =====
721 #pragma unroll
722 for (int i = 0; i < kValTileSeqLenP; ++i) {
723     float rescale_factor_0 = __frcp_rn(lane_block_row_sum_old[i][0]);
724     float rescale_factor_1 = __frcp_rn(lane_block_row_sum_old[i][1]);
725     #pragma unroll
726     for (int j = 0; j < kValTileHeadDimV; ++j) {
727         if constexpr (kMmaAccF32) {
728             float *t_fptr_D = reinterpret_cast<float *>(&R_D[i][j][0]);
729             half *t_hptr_D = reinterpret_cast<half *>(&R_D[i][j][0]);

```

```

730     t_hptr_D[0] = __float2half_rn(rescale_factor_0 * t_fptr_D[0]);
731     t_hptr_D[1] = __float2half_rn(rescale_factor_0 * t_fptr_D[1]);
732     t_hptr_D[2] = __float2half_rn(rescale_factor_1 * t_fptr_D[2]);
733     t_hptr_D[3] = __float2half_rn(rescale_factor_1 * t_fptr_D[3]);
734 } else {
735     float2 t_reg_D_0 = __half2float2(HALF2(R_D[i][j][0]));
736     float2 t_reg_D_1 = __half2float2(HALF2(R_D[i][j][1]));
737     t_reg_D_0.x = rescale_factor_0 * t_reg_D_0.x;
738     t_reg_D_0.y = rescale_factor_0 * t_reg_D_0.y;
739     t_reg_D_1.x = rescale_factor_1 * t_reg_D_1.x;
740     t_reg_D_1.y = rescale_factor_1 * t_reg_D_1.y;
741     HALF2(R_D[i][j][0]) = __float2half2_rn(t_reg_D_0);
742     HALF2(R_D[i][j][1]) = __float2half2_rn(t_reg_D_1);
743 }
744 }
745 }
746
747 // =====
748 // Step 5: Epilogue — Collective store via warp shuffle + 128-bit store
749 // =====
750 // 利用 warp shuffle 将分散在各 lane 的寄存器数据收集到 lane 0~3,
751 // 然后用 LDST128BITS (st.global.v4.f32) 一次性写入 16 bytes
752 #pragma unroll
753 for (int i = 0; i < kValTileSeqLenP; ++i) {
754     #pragma unroll
755     for (int j = 0; j < kValTileHeadDimV; ++j) {
756         uint32_t R_Z[2][4];
757         R_Z[0][0] = R_D[i][j][0];
758         R_Z[1][0] = R_D[i][j][1];
759         R_Z[0][1] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 1, 4);
760         R_Z[0][2] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 2, 4);
761         R_Z[0][3] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 3, 4);
762         R_Z[1][1] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 1, 4);
763         R_Z[1][2] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 2, 4);
764         R_Z[1][3] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 3, 4);
765
766         // st.global.v4.f32: 128-bit store, 4 lanes × 32-bit
767         if (lane_id % 4 == 0) {
768             int store_warp_regs_0_Br =
769                 warp_QP * (kMmaAtomM * kValTileSeqLenP) + i * kMmaAtomM;
770             int store_lane_gmem_0_Br =
771                 Q_tile_id * Br + store_warp_regs_0_Br + lane_id / 4;
772             int store_warp_regs_0_d =
773                 warp_KV * (kMmaAtomN * kValTileHeadDimV) + j * kMmaAtomN;
774             int store_lane_gmem_0_d = store_warp_regs_0_d;
775             int store_gmem_0_addr_0 = 0_gmem_offset +
776                 (store_lane_gmem_0_Br + 0) * kHeadDim +
777                 store_lane_gmem_0_d;
778             int store_gmem_0_addr_1 = 0_gmem_offset +
779                 (store_lane_gmem_0_Br + 8) * kHeadDim +
780                 store_lane_gmem_0_d;
781             // LDST128BITS = reinterpret_cast<float4*>
782             *reinterpret_cast<float4*>(&0[store_gmem_0_addr_0]) =
783                 *reinterpret_cast<float4*>(&R_Z[0][0]);
784             *reinterpret_cast<float4*>(&0[store_gmem_0_addr_1]) =
785                 *reinterpret_cast<float4*>(&R_Z[1][0]);
786         }
787     }
788 }
789 }
790
791 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
792 // Phase 8b: FlashAttention-2 TMA + MMA Warp Specialization (SM120)

```

```

793 // =====
794 // 面试要点 (FA TMA MMA WS vs cp.async FA) :
795 //   - Producer (WG0, 128T) 用 TMA 128B SWIZZLE 搬运 Q/K/V tile 到 smem
796 //   - Consumer (WG1, 256T) 用 ldmatrix + mma.sync.m16n8k16 做 Q@K^T / P@V
797 //   - 与 hgemm_tma_mma_ws_tn 的对比:
798 //     * HGEMM 只做一次 GEMM, K 维迭代; FA 做 QK^T 和 PV 两次 GEMM, KV seqlen 迭代
799 //     * HGEMM 的 A/B 都 staged; FA 中 Q 只 load 一次 (split-Q), K staged, V 单 buffer
800 //
801 // * split-Q 语义 (关键, 避免误解) :
802 //   - grid = (seqlen/Br, B*H), 每个 block 处理一个 **block-level Q tile [Br, d]**
803 //     (不是全局 Q)。对 Q 的 seqlen 做 block-level 并行。
804 //   - 每个 block 遍历**完整的 KV seqlen** (Tc = seqlen/Bc 次外循环)。
805 //   - 因此每个 block 的 Q tile 只需 **load 一次到 smem**, 在整个 KV 遍历中复用
806 //     → 只需 full_Q barrier (无 empty_Q)。
807 //
808 // * Pipeline 同步协议 (arrive_count = 257 = 256 consumer + 1 producer) :
809 //   - Q: full_Q — Producer 发 arrive_tx, Consumer 在主循环前 wait 一次
810 //   - K: full_K[kStagesK] / empty_K[kStagesK] — 多 stage pipeline
811 //   - V: full_V / empty_V — 单 buffer 串行 (QK 前发 TMA → QK 重叠 → PV 前等 → PV 后释放)
812 //
813 // * V ldmatrix.x2.trans 正确性 (已论证无风险) :
814 //   TMA 128B SWIZZLE 是 1-1 映射, ldmatrix 用 swizzle<64>(row,col) 计算的物理地址
815 //   = TMA 写入的物理地址 (smem 1024B 对齐保证 phase=0)。ldmatrix.x2.trans 的转置
816 //   语义在寄存器层面工作, 与 smem 物理布局无关 → 必然正确。
817 //
818 // * Shared Memory 占用分析 (D=64/128, kStagesK=1/2/3/4, kStagesV=1/2) :
819 //   smem = (Q[Br,D] + K[kStagesK,Bc,D] + V[kStagesV,Bc,D]) * sizeof(half)
820 //         = D * (Br + kStagesK*Bc + kStagesV*Bc) * 2
821 //         = D * (128 + (kStagesK+kStagesV)*64) * 2      (Br=128, Bc=64)
822 //
823 //   | kStagesK | kStagesV | D=64 bytes | D=128 bytes |
824 //   |-----|-----|-----|-----|
825 //   | 1 | 1 | 32 KB | 64 KB |
826 //   | 2 | 1 | 40 KB | 80 KB |
827 //   | 2 | 2 | 48 KB | 96 KB |
828 //   | 3 | 1 | 48 KB | 96 KB |
829 //   | 3 | 2 | 56 KB | 112 KB |
830 //   | 4 | 1 | 56 KB | 112 KB |
831 //
832 //   SM120 (RTX PRO 5000) optin smem 上限 ~100KB:
833 //   - D=64: S=1/2/3/4 全部可行 (32-56 KB)
834 //   - D=128: S=1/2/3 可行 (64-96 KB); S=4 (112 KB) 超出 optin 上限, 会被
835 //     check_smem_feasible 兜底判为 SMEM SKIP。D=128 推荐 S=2 (甜点: 80 KB,
836 //     2 blocks/SM, pipeline 深度足够隐藏 TMA 延迟)。
837 //
838 // v1 限制: kHeadDim=64 or 128; seqlen % Br == 0 且 seqlen % Bc == 0 (不处理尾 tile)
839 //
840 // =====
841 // FA TMA WS smem offset helper (D=64/128 通用)
842 // =====
843 // 背景: CU_TENSOR_MAP_SWIZZLE_128B 硬件要求 box innermost dim ≤ 128B = 64 half
844 // (见 /usr/local/cuda/include/cuda.h 的 cuTensorMapEncodeTiled 文档)。因此 D=128
845 // 不能用单个 TMA box=(128, Br) 覆盖整行。
846 //
847 // 方案: D=128 时 box 固定为 (64, Br), 沿 head_dim 方向连续发 kTmaChunks 次 TMA
848 // (minor_coord = c*64), 写入 chunk-major smem 布局 [kTmaChunks, Br, 64]:
849 //   - chunk c 的 smem 起始偏移 = c * Br * 64
850 //   - chunk c 内部按 [Br, 64] row-major + SWIZZLE_128B 存放 (64 half = 128B 周期)
851 //
852 // 本 helper 计算 (row, col) 在该 chunk-major 布局下的 swizzled smem 元素偏移:
853 //   chunk = col / 64      ∈ [0, kTmaChunks)
854 //   col_in = col % 64     ∈ [0, 64)
855 //   offset = chunk * Br * 64 + row * 64 + swizzle<64>(row, col_in)

```

```

856 //
857 // D=64 退化: kTmaChunks=1, chunk=0, offset = row*64 + swizzle<64>(row, col)
858 // 与原公式 (row*64 + swizzle<64>(row, col)) 完全一致 → 无回归。
859 template <int kHeadDim, int Br>
860 __device__ __forceinline__ int swizzle_fa(int row, int col) {
861     static_assert(kHeadDim == 64 || kHeadDim == 128, "D=64 or 128 only");
862     if constexpr (kHeadDim == 64) {
863         return row * 64 + swizzle<64>(row, col);
864     } else { // kHeadDim == 128
865         const int chunk = col >> 6; // col / 64 ∈ {0, 1}
866         const int col_in = col & 63; // col % 64
867         return chunk * (Br * 64) + row * 64 + swizzle<64>(row, col_in);
868     }
869 }
870
871 template <
872     const int kHeadDim, // head dim: v1 only 64
873     const int kMmaAtomM, // 16 (MMA instruction M dim)
874     const int kMmaAtomN, // 8 (MMA instruction N dim)
875     const int kMmaAtomK, // 16 (MMA instruction K dim)
876     const int kMmaAccF32, // 0=f16 acc, 1=f32 acc (mma.sync.f32.f16.f16.f32)
877     const int kMmaTileSeqLenQ, // MMA tiles along Q's M dim, 8 → Br=16*8=128
878     const int kMmaTileSeqLenK, // MMA tiles along K's N dim, 1 → Bc basis=8
879     const int kMmaTileSeqLenP, // MMA tiles for P@V M dim, == kMmaTileSeqLenQ
880     const int kMmaTileHeadDimV, // MMA tiles for P@V N dim (head dim direction)
881     const int kValTileSeqLenQ, // value tiles along Q's M, 1 → Br per warp=16
882     const int kValTileSeqLenK, // value tiles along K's N, 8 → Bc_warp=8*8=64
883     const int kValTileSeqLenP, // value tiles for P@V M dim, 1
884     const int kValTileHeadDimV, // value tiles for P@V N dim, kHeadDim/(8*kMmaTileHeadDimV)
885     const int kStagesK, // K pipeline depth (>=1)
886     const int kStagesV, // V pipeline depth (>=1; 1=single buffer, >=2=pipelined)
887     const int kNumThreads> // 384, 128 producer + 256 consumer
888 __global__ void __launch_bounds__(kNumThreads, 1)
889     flash_attn_tma_mma_ws_stages_split_q(
890         half *Q, half *K, half *V, half *O, int N, int H,
891         const CUtensorMap *__restrict__ tensorMapQ,
892         const CUtensorMap *__restrict__ tensorMapK,
893         const CUtensorMap *__restrict__ tensorMapV) {
894     static_assert(kHeadDim == 64 || kHeadDim == 128,
895         "v1 supports kHeadDim == 64 or 128");
896     static_assert(kNumThreads == 384, "128 producer + 256 consumer");
897     static_assert(kMmaAccF32 == 0 || kMmaAccF32 == 1, "kMmaAccF32 must be 0 or 1");
898     static_assert(kStagesK >= 1, "kStagesK must be >= 1");
899     static_assert(kStagesV >= 1, "kStagesV must be >= 1");
900     constexpr int kNRegs = kMmaAccF32 ? 4 : 2;
901
902     constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ; // 128
903     constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK; // 64
904     // BK for QK: kHeadDim=64, split into kHeadDim/kMmaAtomK=4 MMA K slices
905     // BK for PV: Bc=64, split into Bc/kMmaAtomK=4 MMA K slices (P rows)
906     constexpr int kConsumerThreads = 256; // 8 warps
907     constexpr int kProducerThreads = 128; // 4 warps, only thread 0 issues TMA
908     constexpr int kQTileBytes = Br * kHeadDim * sizeof(half); // 16 KB
909     constexpr int kKTileBytes = Bc * kHeadDim * sizeof(half); // 8 KB
910     constexpr int kVTileBytes = Bc * kHeadDim * sizeof(half); // 8 KB
911     // TMA box innermost 固定 64 half (128B), 满足 CU_TENSOR_MAP_SWIZZLE_128B 的
912     // box innermost ≤ 128B 硬约束。D=128 时沿 head_dim 发 kTmaChunks 次 TMA,
913     // 写入 chunk-major smem 布局 [kTmaChunks, Br, 64]。
914     constexpr int kTmaBoxMinor = 64;
915     constexpr int kTmaChunks = kHeadDim / kTmaBoxMinor;
916
917     // Block indexing — split-Q: blockIdx.x indexes Q tile along seqlen
918     const int Nb_id = blockIdx.y / H;

```

```

919 const int Nh_id = blockIdx.y % H;
920 const int Q_tile_id = blockIdx.x;
921 const int Tc = (N + Bc - 1) / Bc; // number of KV tiles along seqlen
922 const float scale = 1.0f / sqrtf((float)kHeadDim);
923
924 // Per-head gmem base offset (self-attention: Q=K=V share layout)
925 const int QKV_gmem_offset = (Nb_id * H * N + Nh_id * N) * kHeadDim;
926 const int O_gmem_offset = QKV_gmem_offset;
927 // TMA descriptor covers the entire [B*H*N, D] gmem as one 2D matrix; the
928 // producer must offset major_coord by this head/batch base so each block
929 // loads its own (batch, head) Q/K/V tile instead of always reading head 0.
930 const int qkv_major_base = (Nb_id * H + Nh_id) * N;
931
932 // ---- Shared memory layout ----
933 // TMA CU_TENSOR_MAP_SWIZZLE_128B requires 1024B-aligned smem base so the
934 // hardware swizzle phase starts at zero. Consumer swizzle<64>() assumes
935 // zero phase (no base_offset compensation like WGMMMA descriptor).
936 // Q: [Br, kHeadDim] = [128, 64] = 16 KB (1024B aligned ✓)
937 // K: [kStagesK, Bc, kHeadDim] = [kStagesK, 64, 64] (8 KB per stage)
938 // V: [kStagesV, Bc, kHeadDim] = [kStagesV, 64, 64] (8 KB per stage)
939 extern __shared__ __align__(1024) uint8_t smem_fa_tma_ws[];
940 half *Q_smem = reinterpret_cast<half*>(smem_fa_tma_ws);
941 half *K_smem = Q_smem + Br * kHeadDim;
942 half *V_smem = K_smem + kStagesK * Bc * kHeadDim;
943
944 const uint32_t smem_Q_base_ptr = __cvta_generic_to_shared(Q_smem);
945 const uint32_t smem_K_base_ptr = __cvta_generic_to_shared(K_smem);
946 const uint32_t smem_V_base_ptr = __cvta_generic_to_shared(V_smem);
947
948 // ---- Barriers ----
949 // arrive_count = kConsumerThreads + 1 = 257 (256 consumer arrives + 1 producer arrive_tx)
950 #pragma nv_diag_suppress static_var_with_dynamic_init
951 __shared__ cuda::barrier<cuda::thread_scope_block> full_Q;
952 __shared__ cuda::barrier<cuda::thread_scope_block> full_K[kStagesK];
953 __shared__ cuda::barrier<cuda::thread_scope_block> empty_K[kStagesK];
954 __shared__ cuda::barrier<cuda::thread_scope_block> full_V[kStagesV];
955 __shared__ cuda::barrier<cuda::thread_scope_block> empty_V[kStagesV];
956 #pragma nv_diag_default static_var_with_dynamic_init
957
958 // Thread partition: first kProducerThreads (0~127) = Producer WG,
959 // remaining kConsumerThreads (128~383) = Consumer WG.
960 // WARN: Must use kProducerThreads (not kConsumerThreads) as the divisor, otherwise
961 // the split is wrong: 384/256=1.5 would give Producer 256 threads and
962 // Consumer only 128, but barriers expect 256 consumer arrives → deadlock.
963 const int wg_idx = (threadIdx.x < kProducerThreads) ? 0 : 1;
964 const int wg_tid = (threadIdx.x < kProducerThreads) ? threadIdx.x :
965                 (threadIdx.x - kProducerThreads);
966
967 if (threadIdx.x == 0) {
968     init(&full_Q, kConsumerThreads + 1);
969     for (int s = 0; s < kStagesV; ++s) {
970         init(&full_V[s], kConsumerThreads + 1);
971         init(&empty_V[s], kConsumerThreads + 1);
972     }
973     for (int s = 0; s < kStagesK; ++s) {
974         init(&full_K[s], kConsumerThreads + 1);
975         init(&empty_K[s], kConsumerThreads + 1);
976     }
977     tma_fence_proxy_async_shared_cta();
978 }
979 __syncthreads();
980
981 // =====

```

```

982 // Producer Warpgroup (WG0, threadIdx.x 0~127)
983 // Only wg_tid == 0 issues TMA. Split-Q: Q tile loaded once, K staged, V single buffer.
984 // D=128 时每个 tile 沿 head_dim 发 kTmaChunks 次 TMA (box innermost=64 half) ,
985 // 共享同一 barrier: N 次 cp.async.bulk.tensor 各 reduce chunk_bytes 到 tx count,
986 // 1 次 mbarrier_arrive_expect_tx 声明总字节 + producer thread arrive.
987 // =====
988 if (wg_idx == 0) {
989     NOTES_V2_REG_DEALLOC(40);
990     if (wg_tid == 0) {
991         // Step P0: Load block-level Q tile [Br, d] once. D=128 时发 kTmaChunks 次。
992         // minor_coord = c * 64, major_coord = qkv_major_base + Q_tile_id * Br
993         // smem dst = Q_smem + c * Br * 64 (chunk-major 布局)
994         for (int c = 0; c < kTmaChunks; ++c) {
995             tma_load_2d(Q_smem + c * Br * kTmaBoxMinor, tensorMapQ,
996                 c * kTmaBoxMinor, qkv_major_base + Q_tile_id * Br, full_Q);
997         }
998         tma_arrive_expect_tx(full_Q, kQTileBytes);
999
1000         // Step P1: Prefetch first (kStagesK-1) K tiles.
1001         // K tile coords: minor = c*64, major = qkv_major_base + s*Bc
1002         for (int s = 0; s < kStagesK - 1; ++s) {
1003             empty_K[s].wait(empty_K[s].arrive());
1004             for (int c = 0; c < kTmaChunks; ++c) {
1005                 tma_load_2d(K_smem + s * Bc * kHeadDim + c * Bc * kTmaBoxMinor,
1006                     tensorMapK, c * kTmaBoxMinor,
1007                     qkv_major_base + s * Bc, full_K[s]);
1008             }
1009             tma_arrive_expect_tx(full_K[s], kKTileBytes);
1010         }
1011
1012         // Step P1b: Prefetch first (kStagesV-1) V tiles (only if kStagesV > 1).
1013         // kStagesV=1: no prefetch; V[k] loaded in P2 overlaps only QK gemm.
1014         // kStagesV>=2: V prefetched here; in P2 V[k+kStagesV-1] overlaps
1015         // QK+softmax+PV of current iter → true V pipeline.
1016         if constexpr (kStagesV > 1) {
1017             for (int s = 0; s < kStagesV - 1; ++s) {
1018                 empty_V[s].wait(empty_V[s].arrive());
1019                 for (int c = 0; c < kTmaChunks; ++c) {
1020                     tma_load_2d(V_smem + s * Bc * kHeadDim + c * Bc * kTmaBoxMinor,
1021                         tensorMapV, c * kTmaBoxMinor,
1022                         qkv_major_base + s * Bc, full_V[s]);
1023                 }
1024                 tma_arrive_expect_tx(full_V[s], kVTileBytes);
1025             }
1026         }
1027
1028         // Step P2: Main loop over KV seqlen tiles
1029         for (int k = 0; k < Tc; ++k) {
1030             [[maybe_unused]] const int stage = k % kStagesK;
1031
1032             // Issue V[k+kStagesV-1] TMA to stage_next_v (pipelined).
1033             // kStagesV=1: loads V[k] to stage 0, overlaps with QK of same iter.
1034             // kStagesV>=2: loads V[k+kStagesV-1], overlaps with QK+softmax+PV.
1035             {
1036                 const int v_tile = k + kStagesV - 1;
1037                 if (v_tile < Tc) {
1038                     const int stage_next_v = v_tile % kStagesV;
1039                     empty_V[stage_next_v].wait(empty_V[stage_next_v].arrive());
1040                     for (int c = 0; c < kTmaChunks; ++c) {
1041                         tma_load_2d(V_smem + stage_next_v * Bc * kHeadDim +
1042                             c * Bc * kTmaBoxMinor,
1043                             tensorMapV, c * kTmaBoxMinor,
1044                             qkv_major_base + v_tile * Bc,

```

```

1045         full_V[stage_next_v]);
1046     }
1047     tma_arrive_expect_tx(full_V[stage_next_v], kVTileBytes);
1048 }
1049 }
1050
1051 // Prefetch K[k+kStagesK-1] (pipelined).
1052 if (k + kStagesK - 1 < Tc) {
1053     const int stage_next = (k + kStagesK - 1) % kStagesK;
1054     empty_K[stage_next].wait(empty_K[stage_next].arrive());
1055     for (int c = 0; c < kTmaChunks; ++c) {
1056         tma_load_2d(K_smem + stage_next * Bc * kHeadDim +
1057             c * Bc * kTmaBoxMinor,
1058             tensorMapK, c * kTmaBoxMinor,
1059             qkv_major_base + (k + kStagesK - 1) * Bc,
1060             full_K[stage_next]);
1061     }
1062     tma_arrive_expect_tx(full_K[stage_next], kKTileBytes);
1063 }
1064 }
1065 }
1066 }
1067 // =====
1068 // Consumer Warpgroup (WG1, threadIdx.x 128~383, 256 threads = 8 warps)
1069 // All 256 threads participate in ldmatrix + mma.sync.
1070 // =====
1071 else {
1072     const int warp_id = wg_tid / kWarpSize; // 0~7
1073     const int lane_id = wg_tid % kWarpSize; // 0~31
1074     // Split-Q warp layout: 8 warps form 8x1 grid (warp_QP = 0~7, warp_KV = 0)
1075     // Br = 16 * 8 * 1 = 128 → 8 warps each handle 16 rows of Q
1076     const int warp_QP = warp_id;
1077     const int warp_KV = 0;
1078
1079     // Step C0: init — mark all K stages and V stages as "empty" (consumable by producer)
1080     // Consumer register budget per Triton flash_attn_v2 maxnreg strategy on
1081     // Blackwell warp_specialize: D=128 → 168, otherwise → 80.
1082     if constexpr (kHeadDim == 128) {
1083         NOTES_V2_REG_ALLOC(168);
1084     } else {
1085         NOTES_V2_REG_ALLOC(80);
1086     }
1087
1088     for (int s = 0; s < kStagesK; ++s) {
1089         [[maybe_unused]] auto token = empty_K[s].arrive();
1090     }
1091     for (int s = 0; s < kStagesV; ++s) {
1092         [[maybe_unused]] auto token = empty_V[s].arrive();
1093     }
1094
1095     // Wait for Q tile ready once (split-Q: Q smem is reused across all KV iters)
1096     full_Q.wait(full_Q.arrive());
1097
1098     // Online softmax persistent state (per-warp, per-row-pair)
1099     float lane_block_row_max_old[kValTileSeqLenQ][2];
1100     float lane_block_row_sum_old[kValTileSeqLenQ][2];
1101     fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_block_row_max_old, -INFINITY);
1102     fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_block_row_sum_old, 0.0f);
1103
1104     uint32_t R_Q[kValTileSeqLenQ][4];
1105     uint32_t R_K[kValTileSeqLenK][2];
1106     uint32_t R_V[kValTileHeadDimV][2];
1107     uint32_t R_S[kValTileSeqLenQ][kValTileSeqLenK][kNRegs];

```

```

1108     uint32_t R_0[kValTileSeqLenP][kValTileHeadDimV][kNRegs];
1109     uint32_t R_D[kValTileSeqLenP][kValTileHeadDimV][kNRegs];
1110
1111     fill_3D_regs<uint32_t, kValTileSeqLenP, kValTileHeadDimV, kNRegs>(R_D, 0);
1112
1113     // ---- Main loop over KV seqLen ----
1114     for (int tile_K_seqLen = 0; tile_K_seqLen < Tc; ++tile_K_seqLen) {
1115         const int stage = tile_K_seqLen % kStagesK;
1116         const int stage_v = tile_K_seqLen % kStagesV;
1117
1118         // Wait for K[stage] ready
1119         full_K[stage].wait(full_K[stage].arrive());
1120         tma_fence_proxy_async_shared_cta();
1121
1122         // ---- 3b: Q @ K^T = S[Br, Bc] ----
1123         // Q smem reused (split-Q). K from stage `stage`.
1124         // smem addr (TMA 128B swizzle, swizzle<64>)
1125         fill_3D_regs<uint32_t, kValTileSeqLenQ, kValTileSeqLenK, kNRegs>(R_S, 0);
1126 #pragma unroll
1127         for (int tile_K_d = 0; tile_K_d < (kHeadDim / kMmaAtomK); ++tile_K_d) {
1128             // ldmatrix.x4: load Q m16k16 fragment (Q row-major, non-transpose)
1129 #pragma unroll
1130             for (int i = 0; i < kValTileSeqLenQ; ++i) {
1131                 const int warp_smem_Q_Br =
1132                     warp_QP * (kMmaAtomM * kValTileSeqLenQ) + i * kMmaAtomM;
1133                 const int lane_smem_Q_Br = warp_smem_Q_Br + lane_id % 16;
1134                 const int lane_smem_Q_d = tile_K_d * kMmaAtomK + (lane_id / 16) * 8;
1135                 const uint32_t lane_smem_Q_ptr =
1136                     smem_Q_base_ptr +
1137                     swizzle_fa<kHeadDim, Br>(lane_smem_Q_Br, lane_smem_Q_d) *
1138                     sizeof(half);
1139                 LDMATRIX_X4(R_Q[i][0], R_Q[i][1], R_Q[i][2], R_Q[i][3],
1140                             lane_smem_Q_ptr);
1141             }
1142
1143             // ldmatrix.x2: load K k16n8 fragment (K row-major = K^T col-major)
1144 #pragma unroll
1145             for (int j = 0; j < kValTileSeqLenK; ++j) {
1146                 const int warp_smem_K_Bc =
1147                     warp_KV * (kMmaAtomN * kValTileSeqLenK) + j * kMmaAtomN;
1148                 const int lane_smem_K_Bc = warp_smem_K_Bc + lane_id % 8;
1149                 const int lane_smem_K_d =
1150                     tile_K_d * kMmaAtomK + ((lane_id / 8) % 2) * 8;
1151                 const uint32_t lane_smem_K_ptr =
1152                     smem_K_base_ptr +
1153                     (stage * Bc * kHeadDim +
1154                     swizzle_fa<kHeadDim, Bc>(lane_smem_K_Bc, lane_smem_K_d)) *
1155                     sizeof(half);
1156                 LDMATRIX_X2(R_K[j][0], R_K[j][1], lane_smem_K_ptr);
1157             }
1158
1159 #pragma unroll
1160             for (int i = 0; i < kValTileSeqLenQ; ++i) {
1161 #pragma unroll
1162                 for (int j = 0; j < kValTileSeqLenK; ++j) {
1163                     if constexpr (kMmaAccF32) {
1164                         HMMA16816F32(R_S[i][j][0], R_S[i][j][1], R_S[i][j][2], R_S[i][j][3],
1165                                     R_Q[i][0], R_Q[i][1], R_Q[i][2], R_Q[i][3],
1166                                     R_K[j][0], R_K[j][1],
1167                                     R_S[i][j][0], R_S[i][j][1], R_S[i][j][2], R_S[i][j][3]);
1168                     } else {
1169                         HMMA16816(R_S[i][j][0], R_S[i][j][1], R_Q[i][0], R_Q[i][1],
1170                                  R_Q[i][2], R_Q[i][3], R_K[j][0], R_K[j][1],

```

```

1171         R_S[i][j][0], R_S[i][j][1]);
1172     }
1173 }
1174 }
1175 }
1176
1177 // Release K[stage] for producer reuse as early as possible: QK^T GEMM has
1178 // consumed all K smem data; the subsequent softmax (3c) and PV GEMM (3d)
1179 // only touch registers (R_S, R_0) and V smem, so K[stage] is free to be
1180 // overwritten by the producer's next TMA prefetch.
1181 {
1182     [[maybe_unused]] auto token = empty_K[stage].arrive();
1183 }
1184
1185 // ---- 3c: Online Safe Softmax — row max + exp + sum, P back to R_S ----
1186 // Softmax only touches R_S (registers), so V TMA latency can be overlapped
1187 // with this compute. Defer full_V.wait() until just before P@V (3d).
1188 float lane_row_max_new[kValTileSeqLenQ][2];
1189 float lane_row_sum_new[kValTileSeqLenQ][2];
1190 fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_row_max_new, -INFINITY);
1191 fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_row_sum_new, 0.0f);
1192
1193 // Pass 1: row-wise max across kValTileSeqLenK tiles (warp_reduce_max<4>)
1194 #pragma unroll
1195 for (int i = 0; i < kValTileSeqLenQ; ++i) {
1196     #pragma unroll
1197     for (int j = 0; j < kValTileSeqLenK; ++j) {
1198         float tmp_max_0, tmp_max_1;
1199         if constexpr (kMmaAccF32) {
1200             float *t_fptr_S = reinterpret_cast<float *>(&R_S[i][j][0]);
1201             tmp_max_0 = max(t_fptr_S[0], t_fptr_S[1]) * scale;
1202             tmp_max_1 = max(t_fptr_S[2], t_fptr_S[3]) * scale;
1203         } else {
1204             float2 t_reg_S_0 = __half22float2(HALF2(R_S[i][j][0]));
1205             float2 t_reg_S_1 = __half22float2(HALF2(R_S[i][j][1]));
1206             tmp_max_0 = max(t_reg_S_0.x, t_reg_S_0.y) * scale;
1207             tmp_max_1 = max(t_reg_S_1.x, t_reg_S_1.y) * scale;
1208         }
1209         lane_row_max_new[i][0] = max(lane_row_max_new[i][0], tmp_max_0);
1210         lane_row_max_new[i][1] = max(lane_row_max_new[i][1], tmp_max_1);
1211     }
1212     lane_row_max_new[i][0] =
1213         warp_reduce_max<4, float>(lane_row_max_new[i][0]);
1214     lane_row_max_new[i][1] =
1215         warp_reduce_max<4, float>(lane_row_max_new[i][1]);
1216 }
1217
1218 // Pass 2: P = exp(S*scale - m_new), store back to R_S, accumulate row sums
1219 #pragma unroll
1220 for (int i = 0; i < kValTileSeqLenQ; ++i) {
1221     float block_row_max_new_0 =
1222         max(lane_block_row_max_old[i][0], lane_row_max_new[i][0]);
1223     float block_row_max_new_1 =
1224         max(lane_block_row_max_old[i][1], lane_row_max_new[i][1]);
1225
1226     #pragma unroll
1227     for (int j = 0; j < kValTileSeqLenK; ++j) {
1228         if constexpr (kMmaAccF32) {
1229             float *t_fptr_S = reinterpret_cast<float *>(&R_S[i][j][0]);
1230             half *t_hptr_S = reinterpret_cast<half *>(&R_S[i][j][0]);
1231             t_fptr_S[0] = __expf(__fmaf_rn(t_fptr_S[0], scale, -block_row_max_new_0));
1232             t_fptr_S[1] = __expf(__fmaf_rn(t_fptr_S[1], scale, -block_row_max_new_0));
1233             t_fptr_S[2] = __expf(__fmaf_rn(t_fptr_S[2], scale, -block_row_max_new_1));

```

```

1234     t_fptr_S[3] = __expf(__fmaf_rn(t_fptr_S[3], scale, -block_row_max_new_1));
1235     lane_row_sum_new[i][0] += (t_fptr_S[0] + t_fptr_S[1]);
1236     lane_row_sum_new[i][1] += (t_fptr_S[2] + t_fptr_S[3]);
1237     t_hptr_S[0] = __float2half_rn(t_fptr_S[0]);
1238     t_hptr_S[1] = __float2half_rn(t_fptr_S[1]);
1239     t_hptr_S[2] = __float2half_rn(t_fptr_S[2]);
1240     t_hptr_S[3] = __float2half_rn(t_fptr_S[3]);
1241 } else {
1242     float2 t_reg_S_0 = __half22float2(HALF2(R_S[i][j][0]));
1243     float2 t_reg_S_1 = __half22float2(HALF2(R_S[i][j][1]));
1244     t_reg_S_0.x = __expf(__fmaf_rn(t_reg_S_0.x, scale, -block_row_max_new_0));
1245     t_reg_S_0.y = __expf(__fmaf_rn(t_reg_S_0.y, scale, -block_row_max_new_0));
1246     t_reg_S_1.x = __expf(__fmaf_rn(t_reg_S_1.x, scale, -block_row_max_new_1));
1247     t_reg_S_1.y = __expf(__fmaf_rn(t_reg_S_1.y, scale, -block_row_max_new_1));
1248     lane_row_sum_new[i][0] += (t_reg_S_0.x + t_reg_S_0.y);
1249     lane_row_sum_new[i][1] += (t_reg_S_1.x + t_reg_S_1.y);
1250     HALF2(R_S[i][j][0]) = __float22half2_rn(t_reg_S_0);
1251     HALF2(R_S[i][j][1]) = __float22half2_rn(t_reg_S_1);
1252 }
1253 }
1254 lane_row_sum_new[i][0] =
1255     warp_reduce_sum<4, float>(lane_row_sum_new[i][0]);
1256 lane_row_sum_new[i][1] =
1257     warp_reduce_sum<4, float>(lane_row_sum_new[i][1]);
1258 }
1259
1260 // ---- 3d: P @ V = 0[Br, d] ----
1261 // Wait for V[stage_v] ready just before P@V.
1262 // kStagesV=1: V was loaded in same iter, TMA overlaps QK+softmax.
1263 // kStagesV>=2: V was loaded in previous iter, TMA overlaps QK+softmax+PV.
1264 full_V[stage_v].wait(full_V[stage_v].arrive());
1265 tma_fence_proxy_async_shared_cta();
1266
1267 // ldmatrix.x2.trans: V row-major → col-major B fragment for NN matmul
1268 fill_3D_regs<uint32_t, kValTileSeqLenP, kValTileHeadDimV, kNRegs>(R_0, 0);
1269 #pragma unroll
1270 for (int tile_V_Bc = 0; tile_V_Bc < (Bc / kMmaAtomK); ++tile_V_Bc) {
1271 #pragma unroll
1272     for (int j = 0; j < kValTileHeadDimV; ++j) {
1273         const int warp_smem_V_d =
1274             warp_KV * (kMmaAtomN * kValTileHeadDimV) + j * kMmaAtomN;
1275         const int lane_smem_V_Bc = tile_V_Bc * kMmaAtomK + lane_id % 16;
1276         const int lane_smem_V_d = warp_smem_V_d;
1277         const uint32_t lane_smem_V_ptr =
1278             smem_V_base_ptr +
1279             (stage_v * Bc * kHeadDim +
1280              swizzle_fa<kHeadDim, Bc>(lane_smem_V_Bc, lane_smem_V_d)) *
1281             sizeof(half);
1282         LDMATRIX_X2_T(R_V[j][0], R_V[j][1], lane_smem_V_ptr);
1283     }
1284
1285     // P fragment (R_S) directly feeds A of P@V MMA (layout reuse trick)
1286     const int w = tile_V_Bc * 2;
1287 #pragma unroll
1288     for (int i = 0; i < kValTileSeqLenP; ++i) {
1289 #pragma unroll
1290         for (int j = 0; j < kValTileHeadDimV; ++j) {
1291             if constexpr (kMmaAccF32) {
1292                 HMMA16816F32(R_0[i][j][0], R_0[i][j][1], R_0[i][j][2], R_0[i][j][3],
1293                             R_S[i][w][0], R_S[i][w][1], R_S[i][w + 1][0], R_S[i][w + 1][1],
1294                             R_V[j][0], R_V[j][1],
1295                             R_0[i][j][0], R_0[i][j][1], R_0[i][j][2], R_0[i][j][3]);
1296             } else {

```

```

1297         HMMMA16816(R_0[i][j][0], R_0[i][j][1],
1298                   R_S[i][w][0], R_S[i][w][1],
1299                   R_S[i][w + 1][0], R_S[i][w + 1][1],
1300                   R_V[j][0], R_V[j][1],
1301                   R_0[i][j][0], R_0[i][j][1]);
1302     }
1303 }
1304 }
1305 }
1306
1307 // Release V[stage_v] for producer reuse: PV GEMM has consumed all
1308 // V smem data; the subsequent online rescaling (3e) only touches registers
1309 // (R_0, R_D, R_S, lane_block_*), so V[stage_v] is free to be overwritten.
1310 {
1311     [[maybe_unused]] auto token = empty_V[stage_v].arrive();
1312 }
1313
1314 // ---- 3e: Online rescaling — 0_new = exp(m_old - m_new) * 0_old + P@V ----
1315 #pragma unroll
1316 for (int i = 0; i < kValTileSeqLenP; ++i) {
1317     float block_row_max_new_0 = lane_row_max_new[i][0];
1318     float block_row_max_new_1 = lane_row_max_new[i][1];
1319     float block_row_sum_new_0 = lane_row_sum_new[i][0];
1320     float block_row_sum_new_1 = lane_row_sum_new[i][1];
1321     float block_row_max_old_0 = lane_block_row_max_old[i][0];
1322     float block_row_max_old_1 = lane_block_row_max_old[i][1];
1323
1324     block_row_max_new_0 = max(block_row_max_old_0, block_row_max_new_0);
1325     block_row_max_new_1 = max(block_row_max_old_1, block_row_max_new_1);
1326
1327     // First iteration: m_old = -inf, use m_new directly
1328     block_row_max_old_0 = (tile_K_seqlen > 0 ? block_row_max_old_0
1329                                     : block_row_max_new_0);
1330     block_row_max_old_1 = (tile_K_seqlen > 0 ? block_row_max_old_1
1331                                     : block_row_max_new_1);
1332
1333     float rescale_o_factor_0 =
1334         __expf(block_row_max_old_0 - block_row_max_new_0);
1335     float rescale_o_factor_1 =
1336         __expf(block_row_max_old_1 - block_row_max_new_1);
1337
1338 #pragma unroll
1339 for (int j = 0; j < kValTileHeadDimV; ++j) {
1340     if constexpr (kMmaAccF32) {
1341         float *t_fptr_0 = reinterpret_cast<float *>(&R_0[i][j][0]);
1342         float *t_fptr_D = reinterpret_cast<float *>(&R_D[i][j][0]);
1343         t_fptr_D[0] = __fmaf_rn(rescale_o_factor_0, t_fptr_D[0], t_fptr_0[0]);
1344         t_fptr_D[1] = __fmaf_rn(rescale_o_factor_0, t_fptr_D[1], t_fptr_0[1]);
1345         t_fptr_D[2] = __fmaf_rn(rescale_o_factor_1, t_fptr_D[2], t_fptr_0[2]);
1346         t_fptr_D[3] = __fmaf_rn(rescale_o_factor_1, t_fptr_D[3], t_fptr_0[3]);
1347     } else {
1348         float2 t_reg_0_0 = __half22float2(HALF2(R_0[i][j][0]));
1349         float2 t_reg_0_1 = __half22float2(HALF2(R_0[i][j][1]));
1350         float2 t_reg_D_0 = __half22float2(HALF2(R_D[i][j][0]));
1351         float2 t_reg_D_1 = __half22float2(HALF2(R_D[i][j][1]));
1352         t_reg_D_0.x = __fmaf_rn(rescale_o_factor_0, t_reg_D_0.x, t_reg_0_0.x);
1353         t_reg_D_0.y = __fmaf_rn(rescale_o_factor_0, t_reg_D_0.y, t_reg_0_0.y);
1354         t_reg_D_1.x = __fmaf_rn(rescale_o_factor_1, t_reg_D_1.x, t_reg_0_1.x);
1355         t_reg_D_1.y = __fmaf_rn(rescale_o_factor_1, t_reg_D_1.y, t_reg_0_1.y);
1356         HALF2(R_D[i][j][0]) = __float22half2_rn(t_reg_D_0);
1357         HALF2(R_D[i][j][1]) = __float22half2_rn(t_reg_D_1);
1358     }
1359 }

```

```

1360     float block_row_sum_old_0 = lane_block_row_sum_old[i][0];
1361     float block_row_sum_old_1 = lane_block_row_sum_old[i][1];
1362     lane_block_row_sum_old[i][0] =
1363         __fmaf_rn(rescale_o_factor_0, block_row_sum_old_0,
1364                 block_row_sum_new_0);
1365     lane_block_row_sum_old[i][1] =
1366         __fmaf_rn(rescale_o_factor_1, block_row_sum_old_1,
1367                 block_row_sum_new_1);
1368     lane_block_row_max_old[i][0] = block_row_max_new_0;
1369     lane_block_row_max_old[i][1] = block_row_max_new_1;
1370 }
1371 }
1372
1373
1374 // ---- Step 4: Final rescale — 0_final = (1/l_final) * 0_final ----
1375 #pragma unroll
1376 for (int i = 0; i < kValTileSeqLenP; ++i) {
1377     float rescale_factor_0 = __frcp_rn(lane_block_row_sum_old[i][0]);
1378     float rescale_factor_1 = __frcp_rn(lane_block_row_sum_old[i][1]);
1379 #pragma unroll
1380 for (int j = 0; j < kValTileHeadDimV; ++j) {
1381     if constexpr (kMmaAccF32) {
1382         float *t_fptr_D = reinterpret_cast<float *>(&R_D[i][j][0]);
1383         half *t_hptr_D = reinterpret_cast<half *>(&R_D[i][j][0]);
1384         t_hptr_D[0] = __float2half_rn(rescale_factor_0 * t_fptr_D[0]);
1385         t_hptr_D[1] = __float2half_rn(rescale_factor_0 * t_fptr_D[1]);
1386         t_hptr_D[2] = __float2half_rn(rescale_factor_1 * t_fptr_D[2]);
1387         t_hptr_D[3] = __float2half_rn(rescale_factor_1 * t_fptr_D[3]);
1388     } else {
1389         float2 t_reg_D_0 = __half22float2(HALF2(R_D[i][j][0]));
1390         float2 t_reg_D_1 = __half22float2(HALF2(R_D[i][j][1]));
1391         t_reg_D_0.x = rescale_factor_0 * t_reg_D_0.x;
1392         t_reg_D_0.y = rescale_factor_0 * t_reg_D_0.y;
1393         t_reg_D_1.x = rescale_factor_1 * t_reg_D_1.x;
1394         t_reg_D_1.y = rescale_factor_1 * t_reg_D_1.y;
1395         HALF2(R_D[i][j][0]) = __float22half2_rn(t_reg_D_0);
1396         HALF2(R_D[i][j][1]) = __float22half2_rn(t_reg_D_1);
1397     }
1398 }
1399 }
1400
1401 // ---- Step 5: Epilogue — warp shuffle + 128-bit collective store ----
1402 #pragma unroll
1403 for (int i = 0; i < kValTileSeqLenP; ++i) {
1404 #pragma unroll
1405 for (int j = 0; j < kValTileHeadDimV; ++j) {
1406     uint32_t R_Z[2][4];
1407     R_Z[0][0] = R_D[i][j][0];
1408     R_Z[1][0] = R_D[i][j][1];
1409     R_Z[0][1] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 1, 4);
1410     R_Z[0][2] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 2, 4);
1411     R_Z[0][3] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 3, 4);
1412     R_Z[1][1] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 1, 4);
1413     R_Z[1][2] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 2, 4);
1414     R_Z[1][3] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 3, 4);
1415
1416     if (lane_id % 4 == 0) {
1417         const int store_warp_regs_0_Br =
1418             warp_QP * (kMmaAtomM * kValTileSeqLenP) + i * kMmaAtomM;
1419         const int store_lane_gmem_0_Br =
1420             Q_tile_id * Br + store_warp_regs_0_Br + lane_id / 4;
1421         const int store_warp_regs_0_d =
1422             warp_KV * (kMmaAtomN * kValTileHeadDimV) + j * kMmaAtomN;

```

```

1423     const int store_lane_gmem_0_d = store_warp_regs_0_d;
1424     const int store_gmem_0_addr_0 =
1425         0_gmem_offset + store_lane_gmem_0_Br * kHeadDim +
1426         store_lane_gmem_0_d;
1427     const int store_gmem_0_addr_1 =
1428         0_gmem_offset + (store_lane_gmem_0_Br + 8) * kHeadDim +
1429         store_lane_gmem_0_d;
1430     *reinterpret_cast<float4*>(&0[store_gmem_0_addr_0]) =
1431         *reinterpret_cast<float4*>(&R_Z[0][0]);
1432     *reinterpret_cast<float4*>(&0[store_gmem_0_addr_1]) =
1433         *reinterpret_cast<float4*>(&R_Z[1][0]);
1434 }
1435 }
1436 }
1437 }
1438 }
1439
1440 // =====
1441 // Phase 8c: FlashAttention-3-style TMA + MMA Warp Specialization (SM120)
1442 //   Dual 128-thread consumer warpgroups processing disjoint KV tiles,
1443 //   with per-WG independent K pipeline (kStagesK) and single V buffer (kStagesV=1).
1444 //
1445 // 与 flash_attn_tma_mma_ws_stages_split_q (Phase 8b) 的核心区别:
1446 //   - 8b: 1 producer WG (128T) + 1 consumer WG (256T, 8 warps), 全 KV 遍历
1447 //   - 8c: 1 producer WG (128T) + 2 consumer WGs (各128T, 4 warps), KV tile 奇偶拆分
1448 //
1449 // 线程布局 (384 threads):
1450 //   WG0 [0,127]:   TMA producer (仅 thread 0 发 TMA)
1451 //   WG1 [128,255]: consumer_id=0, 处理偶数 KV tile (0,2,4,...)
1452 //   WG2 [256,383]: consumer_id=1, 处理奇数 KV tile (1,3,5,...)
1453 //
1454 // per-WG 独立 stage 语义 (关键改进):
1455 // -----
1456 // 每个 consumer WG 拥有独立的 K stage slots (kStagesK 个) 和 1 个 V buffer。
1457 // K/V smem 按 consumer_id 分 bank: K_smem[cid][stage], V_smem[cid]。
1458 // Barrier 也按 consumer_id 隔离: full_K[cid][s], empty_K[cid][s], full_V[cid], empty_V[cid]。
1459 // 两个 WG 绝不争抢同一 smem slot 或 barrier, producer 通过 cid = tile&1 路由 TMA。
1460 //
1461 // 这样 stage 语义从 "跨 WG 共享的 2 slot (每 WG 实际 depth=1)" 变为
1462 // "每 WG 独立拥有 kStagesK 个 K slot (每 WG 真正 depth=kStagesK)"。
1463 //
1464 // Pipeline 时序 (Sk=1, Sv=1):
1465 //   K[tile] TMA 被 softmax+PV+rescale 隐藏 (Sk=1 复用同 slot, 用完即释放);
1466 //   V[tile] TMA 被 rescale+QK[next]+softmax[next] 隐藏 (Sv=1 同理)。
1467 //
1468 // Pipeline 时序 (Sk=2, Sv=1, D=64 only):
1469 //   K[tile+2] 在 QK[tile] 之前就 prefetch 到另一个 slot, 被整个 tile compute 隐藏;
1470 //   V[tile] TMA 同 Sk=1, 被 QK+softmax 隐藏。
1471 //
1472 // 为什么是奇偶 tile 拆分 (而非前/后半区各自一半)?
1473 // -----
1474 // 关键约束: producer 按 global tile 顺序 0,1,2,3,... 连续 TMA。
1475 // 奇偶拆分让 cid = tile & 1 静态路由: tile 0,2,4,... -> WG1; tile 1,3,5,... -> WG2。
1476 // 每 WG 的下一个 tile (tile+2) 正好是 producer 当前/下一轮要加载的 tile,
1477 // 保证 producer/consumer 流水无停顿。
1478 //
1479 // 若改为前/后半区拆分: 过渡区两个 WG 可能同时需要同一 cid 的 slot,
1480 // 且 WG2 跳跃访问远端 tile 时数据尚未被 producer TMA 到 smem。
1481 //
1482 // Partial state merge (Split-KV / FlashDecoding-style reduction):
1483 // -----
1484 // 每个 WG 各自处理一半 KV tile 序列, 独立维护 "未归一化" 的 online-softmax
1485 // 中间状态 (m, l, 0acc), 合并后等价于全 KV 遍历。

```

```

1486 //
1487 // 稳定归并公式 (避免 exp 溢出):
1488 //   m      = max(m_0, m_1)           // 全局 row max
1489 //   alpha = exp(m_0 - m) (<=1)       // WG1 rescale factor
1490 //   beta  = exp(m_1 - m) (<=1)       // WG2 rescale factor
1491 //   l      = alpha * l_0 + beta * l_1
1492 //   Oacc   = alpha * Oacc_0 + beta * Oacc_1
1493 //   0      = Oacc / l                // 最终归一化
1494 //
1495 // Tc==1 退化: WG2 无 tile, m_1=-inf, l_1=0, Oacc_1=0;
1496 //   beta = exp(-inf) = 0, 退化为 WG1 结果。
1497 //
1498 // 限制 (首版):
1499 //   - kStagesV == 1 (V 只需 1 buffer, TMA 被 QK+softmax 隐藏)
1500 //   - kStagesK >= 1 (K pipeline depth; D=128 在 SM120 只支持 Sk=1)
1501 //   - D=128 时 Sk=1 (Sk=2 需 112KB > 101KB optin 上限)
1502 //   - Br=64, Bc=64, D=64/128, aligned seqlen (N % Br == 0 && N % Bc == 0)
1503 //   - 仅 self-attention, 无 causal/varlen/GQA
1504 // =====
1505 template <
1506     const int kHeadDim,
1507     const int kMmaAtomM, const int kMmaAtomN, const int kMmaAtomK,
1508     const int kMmaAccF32,
1509     const int kMmaTileSeqLenQ, const int kMmaTileSeqLenK,
1510     const int kMmaTileSeqLenP, const int kMmaTileHeadDimV,
1511     const int kValTileSeqLenQ, const int kValTileSeqLenK,
1512     const int kValTileSeqLenP, const int kValTileHeadDimV,
1513     const int kStagesK, const int kStagesV,
1514     const int kNumThreads>
1515 __global__ void __launch_bounds__(kNumThreads, 1)
1516     flash_attn_3_tma_ws_stages_split_q(
1517         half *Q, half *K, half *V, half *O, int N, int H,
1518         const CUtensorMap *__restrict__ tensorMapQ,
1519         const CUtensorMap *__restrict__ tensorMapK,
1520         const CUtensorMap *__restrict__ tensorMapV) {
1521     static_assert(kHeadDim == 64 || kHeadDim == 128,
1522         "v1 supports kHeadDim == 64 or 128");
1523     static_assert(kNumThreads == 384, "128 producer + 256 consumer (2 WGs)");
1524     static_assert(kMmaAccF32 == 0 || kMmaAccF32 == 1, "kMmaAccF32 must be 0 or 1");
1525     static_assert(kStagesV == 1, "per-WG V pipeline depth must be 1");
1526     static_assert(kStagesK >= 1, "kStagesK must be >= 1");
1527     constexpr int kNRegs = kMmaAccF32 ? 4 : 2;
1528
1529     constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ; // 64
1530     constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK; // 64
1531     constexpr int kConsumerThreadsPerWG = 128;
1532     constexpr int kProducerThreads = 128;
1533     constexpr int kNumConsumerWGs = 2;
1534     constexpr int kQTileBytes = Br * kHeadDim * sizeof(half);
1535     constexpr int kKTileBytes = Bc * kHeadDim * sizeof(half);
1536     constexpr int kVTileBytes = Bc * kHeadDim * sizeof(half);
1537     constexpr int kTmaBoxMinor = 64;
1538     constexpr int kTmaChunks = kHeadDim / kTmaBoxMinor;
1539
1540     const int Nb_id = blockIdx.y / H;
1541     const int Nh_id = blockIdx.y % H;
1542     const int Q_tile_id = blockIdx.x;
1543     const int Tc = (N + Bc - 1) / Bc;
1544     const float scale = 1.0f / sqrtf((float)kHeadDim);
1545
1546     const int QKV_gmem_offset = (Nb_id * H * N + Nh_id * N) * kHeadDim;
1547     const int O_gmem_offset = QKV_gmem_offset;
1548     const int qkv_major_base = (Nb_id * H + Nh_id) * N;

```

```

1549 // ---- Shared memory layout (per-WG independent K/V banks) ----
1550 // [Q_shared: Br*D]
1551 // [K[cid=0][s=0..Sk-1]: Sk*Bc*D] [K[cid=1][s=0..Sk-1]]
1552 // [V[cid=0]: Bc*D] [V[cid=1]: Bc*D]
1553 extern __shared__ __align__(1024) uint8_t smem_fa3_tma_ws[];
1554 half *Q_smem = reinterpret_cast<half *>(smem_fa3_tma_ws);
1555 half *K_smem_base = Q_smem + Br * kHeadDim;
1556 // K: [numConsumerWGs][kStagesK][Bc*kHeadDim], linearized
1557 // K_smem_base + cid * (kStagesK * Bc * kHeadDim) + stg * (Bc * kHeadDim)
1558 half *V_smem_base = K_smem_base + kNumConsumerWGs * kStagesK * Bc * kHeadDim;
1559 // V: [numConsumerWGs][Bc*kHeadDim]
1560 // V_smem_base + cid * (Bc * kHeadDim)
1561
1562 const uint32_t smem_Q_base_ptr = __cvta_generic_to_shared(Q_smem);
1563 const uint32_t smem_K_base_ptr = __cvta_generic_to_shared(K_smem_base);
1564 const uint32_t smem_V_base_ptr = __cvta_generic_to_shared(V_smem_base);
1565
1566 #pragma nv_diag_suppress static_var_with_dynamic_init
1567 __shared__ cuda::barrier<cuda::thread_scope_block> full_Q;
1568 // per-WG K barriers: [cid][stage]
1569 __shared__ cuda::barrier<cuda::thread_scope_block> full_K[kNumConsumerWGs][kStagesK];
1570 __shared__ cuda::barrier<cuda::thread_scope_block> empty_K[kNumConsumerWGs][kStagesK];
1571 // per-WG V barriers: [cid] (kStagesV=1)
1572 __shared__ cuda::barrier<cuda::thread_scope_block> full_V[kNumConsumerWGs];
1573 __shared__ cuda::barrier<cuda::thread_scope_block> empty_V[kNumConsumerWGs];
1574 #pragma nv_diag_default static_var_with_dynamic_init
1575
1576 const bool is_producer = threadIdx.x < kProducerThreads;
1577 const int consumer_id = is_producer ? 0
1578 : (threadIdx.x - kProducerThreads) / kConsumerThreadsPerWG;
1579 const int wg_tid = is_producer ? threadIdx.x
1580 : (threadIdx.x - kProducerThreads) % kConsumerThreadsPerWG;
1581
1582 if (threadIdx.x == 0) {
1583     init(&full_Q, 256 + 1); // 2 consumer WGs * 128 + 1 producer
1584     for (int cid = 0; cid < kNumConsumerWGs; ++cid) {
1585         for (int s = 0; s < kStagesK; ++s) {
1586             init(&full_K[cid][s], kConsumerThreadsPerWG + 1);
1587             init(&empty_K[cid][s], kConsumerThreadsPerWG + 1);
1588         }
1589         init(&full_V[cid], kConsumerThreadsPerWG + 1);
1590         init(&empty_V[cid], kConsumerThreadsPerWG + 1);
1591     }
1592     tma_fence_proxy_async_shared_cta();
1593 }
1594 __syncthreads();
1595
1596 // Helper: K_smem ptr for (cid, stg)
1597 // K layout: K_smem_base + cid*(Sk*Bc*D) + stg*(Bc*D)
1598 // D=128 chunk-major: each tile's 2 chunks at stg*Bc*D + c*Bc*64
1599
1600 // =====
1601 // Producer Warpgroup (WG0, threadIdx.x 0~127)
1602 // Single global tile loop over all KV tiles (0..Tc-1).
1603 // cid = tile & 1 routes TMA to the owning consumer WG's smem bank.
1604 //
1605 // Per-tile load order: V first, then K (prefetch next).
1606 // - V[tile] TMA: blocked until consumer releases V[cid] (after PV).
1607 // But K[tile] signal already sent in warmup/prev iter, so consumer
1608 // can do QK[tile] while producer waits for V release.
1609 // - K[tile+2] prefetch (Sk>=2): blocked until consumer releases K[cid][stg_next]
1610 // (after QK). QK is first compute step, always done before producer reaches here.

```

```

1612 // =====
1613 if (is_producer) {
1614     NOTES_V2_REG_DEALLOC(40);
1615     if (wg_tid == 0) {
1616         // P0: Load Q tile once
1617         for (int c = 0; c < kTmaChunks; ++c) {
1618             tma_load_2d(Q_smem + c * Br * kTmaBoxMinor, tensorMapQ,
1619                 c * kTmaBoxMinor, qkv_major_base + Q_tile_id * Br, full_Q);
1620         }
1621         tma_arrive_expect_tx(full_Q, kQTileBytes);
1622
1623         // P1: Warmup - prefetch first K tile for each consumer WG (Sk-1 tiles)
1624         // tile 0 -> cid=0, tile 1 -> cid=1
1625         if constexpr (kStagesK > 1) {
1626             for (int cid = 0; cid < kNumConsumerWGs; ++cid) {
1627                 const int tile0 = cid; // first tile for this WG
1628                 if (tile0 < Tc) {
1629                     const int stg = 0; // first slot
1630                     empty_K[cid][stg].wait(empty_K[cid][stg].arrive());
1631                     for (int c = 0; c < kTmaChunks; ++c) {
1632                         tma_load_2d(K_smem_base + cid * (kStagesK * Bc * kHeadDim) +
1633                             stg * Bc * kHeadDim + c * Bc * kTmaBoxMinor,
1634                             tensorMapK, c * kTmaBoxMinor,
1635                             qkv_major_base + tile0 * Bc, full_K[cid][stg]);
1636                     }
1637                     tma_arrive_expect_tx(full_K[cid][stg], kKTileBytes);
1638                 }
1639             }
1640         }
1641
1642         // P2: Main loop over all global KV tiles
1643         // K 优先于 V: K 的 release 依赖 QK (tile 第一步 compute, 早完成),
1644         // V 的 release 依赖 PV (tile 最后一步 compute, 晚完成)。
1645         // 先发 K 让 consumer 尽早开始 QK, V 的 TMA 在 K 之后发出,
1646         // 被 QK+softmax 的 compute 时间隐藏。
1647         for (int tile = 0; tile < Tc; ++tile) {
1648             const int cid = tile & 1;
1649
1650             // Step 1: Load/prefetch K first (release depends on QK, which is early)
1651             if constexpr (kStagesK == 1) {
1652                 // Sk=1: load K[tile] to the single slot (reuse after consumer QK release)
1653                 empty_K[cid][0].wait(empty_K[cid][0].arrive());
1654                 for (int c = 0; c < kTmaChunks; ++c) {
1655                     tma_load_2d(K_smem_base + cid * (kStagesK * Bc * kHeadDim) +
1656                         c * Bc * kTmaBoxMinor,
1657                         tensorMapK, c * kTmaBoxMinor,
1658                         qkv_major_base + tile * Bc, full_K[cid][0]);
1659                 }
1660                 tma_arrive_expect_tx(full_K[cid][0], kKTileBytes);
1661             } else {
1662                 // Sk>=2: prefetch K[tile+2] to next slot (true ping-pong pipeline!)
1663                 const int local_tile = (tile - cid) / 2;
1664                 const int stg_next = (local_tile + 1) % kStagesK;
1665                 const int next_tile = tile + 2;
1666                 if (next_tile < Tc) {
1667                     empty_K[cid][stg_next].wait(empty_K[cid][stg_next].arrive());
1668                     for (int c = 0; c < kTmaChunks; ++c) {
1669                         tma_load_2d(K_smem_base + cid * (kStagesK * Bc * kHeadDim) +
1670                             stg_next * Bc * kHeadDim + c * Bc * kTmaBoxMinor,
1671                             tensorMapK, c * kTmaBoxMinor,
1672                             qkv_major_base + next_tile * Bc,
1673                             full_K[cid][stg_next]);
1674                 }
1675             }
1676         }
1677     }
1678 }

```

```

1675         tma_arrive_expect_tx(full_K[cid][stg_next], kKTileBytes);
1676     }
1677 }
1678
1679 // Step 2: Load V[tile] (release depends on PV, which is late)
1680 empty_V[cid].wait(empty_V[cid].arrive());
1681 for (int c = 0; c < kTmaChunks; ++c) {
1682     tma_load_2d(V_smem_base + cid * Bc * kHeadDim + c * Bc * kTmaBoxMinor,
1683               tensorMapV, c * kTmaBoxMinor,
1684               qkv_major_base + tile * Bc, full_V[cid]);
1685 }
1686 tma_arrive_expect_tx(full_V[cid], kVTileBytes);
1687 }
1688 }
1689 }
1690 // =====
1691 // Consumer Warpgroups (WG1 consumer_id=0, WG2 consumer_id=1)
1692 // Each WG processes every other KV tile (consumer_id, consumer_id+2, ...).
1693 // K stage index: local_iter % kStagesK (Sk=1: always 0; Sk=2: 0,1,0,1,...)
1694 // V stage index: always consumer_id (only 1 V buffer per WG)
1695 // =====
1696 else {
1697     const int warp_id = wg_tid / kWarpSize; // 0~3
1698     const int lane_id = wg_tid % kWarpSize; // 0~31
1699     const int warp_QP = warp_id;
1700     const int warp_KV = 0;
1701
1702     if constexpr (kHeadDim == 128) {
1703         NOTES_V2_REG_ALLOC(168);
1704     } else {
1705         NOTES_V2_REG_ALLOC(80);
1706     }
1707
1708     // C0: init - mark OWN K/V slots as empty (consumer arrives, producer is 129th)
1709     for (int s = 0; s < kStagesK; ++s) {
1710         [[maybe_unused]] auto tk = empty_K[consumer_id][s].arrive();
1711     }
1712     {
1713         [[maybe_unused]] auto tv = empty_V[consumer_id].arrive();
1714     }
1715
1716     // Wait for Q tile (shared by both WGs)
1717     full_Q.wait(full_Q.arrive());
1718
1719     // Partial online-softmax state (per-WG, independent)
1720     float lane_block_row_max_old[kValTileSeqLenQ][2];
1721     float lane_block_row_sum_old[kValTileSeqLenQ][2];
1722     fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_block_row_max_old, -INFINITY);
1723     fill_2D_regs<float, kValTileSeqLenQ, 2>(lane_block_row_sum_old, 0.0f);
1724
1725     uint32_t R_Q[kValTileSeqLenQ][4];
1726     uint32_t R_K[kValTileSeqLenK][2];
1727     uint32_t R_V[kValTileHeadDimV][2];
1728     uint32_t R_S[kValTileSeqLenQ][kValTileSeqLenK][kNRegs];
1729     uint32_t R_O[kValTileSeqLenP][kValTileHeadDimV][kNRegs];
1730     uint32_t R_D[kValTileSeqLenP][kValTileHeadDimV][kNRegs];
1731     fill_3D_regs<uint32_t, kValTileSeqLenP, kValTileHeadDimV, kNRegs>(R_D, 0);
1732
1733     // K smem offset for this WG: base + cid*(Sk*Bc*D)
1734     // stg within: local_iter % kStagesK
1735     // V smem offset for this WG: V_smem_base + cid*(Bc*D)
1736     const uint32_t smem_K_wg_base = smem_K_base_ptr +
1737         consumer_id * (kStagesK * Bc * kHeadDim) * sizeof(half);

```

```

1738 const uint32_t smem_V_wg_base = smem_V_base_ptr +
1739     consumer_id * (Bc * kHeadDim) * sizeof(half);
1740
1741 int local_iter = 0;
1742 for (int tile = consumer_id; tile < Tc; tile += 2) {
1743     const int k_stg = local_iter % kStagesK;
1744
1745     // K smem offset for this stage: wg_base + stg*(Bc*D)
1746     const uint32_t smem_K_stage_ptr = smem_K_wg_base +
1747         k_stg * Bc * kHeadDim * sizeof(half);
1748
1749     // Wait for K[consumer_id][k_stg] ready
1750     full_K[consumer_id][k_stg].wait(full_K[consumer_id][k_stg].arrive());
1751     tma_fence_proxy_async_shared_cta();
1752
1753     // ---- Q @ K^T = S[Br, Bc] ----
1754     // K early release: 在最后一个 tile_K_d 的 ldmatrix_K 完成后立即 release K,
1755     // 不等整个 QK MMA 循环结束。K 数据已全部读入 R_K 寄存器, smem 可被
1756     // producer 复用。这对 Sk=1 尤其重要: producer 可更早开始下一个 K TMA。
1757     fill_3D_regs<uint32_t, kValTileSeqLenQ, kValTileSeqLenK, kNRegs>(R_S, 0);
1758 #pragma unroll
1759     for (int tile_K_d = 0; tile_K_d < (kHeadDim / kMmaAtomK); ++tile_K_d) {
1760 #pragma unroll
1761         for (int i = 0; i < kValTileSeqLenQ; ++i) {
1762             const int warp_smem_Q_Br =
1763                 warp_QP * (kMmaAtomM * kValTileSeqLenQ) + i * kMmaAtomM;
1764             const int lane_smem_Q_Br = warp_smem_Q_Br + lane_id % 16;
1765             const int lane_smem_Q_d = tile_K_d * kMmaAtomK + (lane_id / 16) * 8;
1766             const uint32_t lane_smem_Q_ptr =
1767                 smem_Q_base_ptr +
1768                 swizzle_fa<kHeadDim, Br>(lane_smem_Q_Br, lane_smem_Q_d) *
1769                     sizeof(half);
1770             LDMATRIX_X4(R_Q[i][0], R_Q[i][1], R_Q[i][2], R_Q[i][3],
1771                 lane_smem_Q_ptr);
1772         }
1773
1774 #pragma unroll
1775         for (int j = 0; j < kValTileSeqLenK; ++j) {
1776             const int warp_smem_K_Bc =
1777                 warp_KV * (kMmaAtomN * kValTileSeqLenK) + j * kMmaAtomN;
1778             const int lane_smem_K_Bc = warp_smem_K_Bc + lane_id % 8;
1779             const int lane_smem_K_d =
1780                 tile_K_d * kMmaAtomK + ((lane_id / 8) % 2) * 8;
1781             // K smem: stage_ptr + swizzle_fa offset (no stage*Bc*D since stage_ptr already includes it)
1782             const uint32_t lane_smem_K_ptr =
1783                 smem_K_stage_ptr +
1784                 swizzle_fa<kHeadDim, Bc>(lane_smem_K_Bc, lane_smem_K_d) *
1785                     sizeof(half);
1786             LDMATRIX_X2(R_K[j][0], R_K[j][1], lane_smem_K_ptr);
1787         }
1788
1789         // Early release K after last ldmatrix_K: K data now fully in registers
1790         if (tile_K_d == (kHeadDim / kMmaAtomK) - 1) {
1791             [[maybe_unused]] auto token = empty_K[consumer_id][k_stg].arrive();
1792         }
1793
1794 #pragma unroll
1795         for (int i = 0; i < kValTileSeqLenQ; ++i) {
1796 #pragma unroll
1797             for (int j = 0; j < kValTileSeqLenK; ++j) {
1798                 if constexpr (kMmaAccF32) {
1799                     HMMMA16816F32(R_S[i][j][0], R_S[i][j][1], R_S[i][j][2], R_S[i][j][3],
1800                         R_Q[i][0], R_Q[i][1], R_Q[i][2], R_Q[i][3],

```

```

1801         R_K[j][0], R_K[j][1],
1802         R_S[i][j][0], R_S[i][j][1], R_S[i][j][2], R_S[i][j][3]);
1803     } else {
1804         MMA16816(R_S[i][j][0], R_S[i][j][1], R_Q[i][0], R_Q[i][1],
1805         R_Q[i][2], R_Q[i][3], R_K[j][0], R_K[j][1],
1806         R_S[i][j][0], R_S[i][j][1]);
1807     }
1808 }
1809 }
1810 }
1811
1812 // K already released above (after last ldmatrix_K)
1813
1814 // ---- Online Safe Softmax ----
1815 // This compute overlaps with producer's V[tile] TMA (which waits for
1816 // empty_V from previous PV, or is already in-flight)
1817 float lane_row_max_new[kValTileSeqLenQ][2];
1818 float lane_row_sum_new[kValTileSeqLenQ][2];
1819 fill_2D_regs<float>(kValTileSeqLenQ, 2>(lane_row_max_new, -INFINITY);
1820 fill_2D_regs<float>(kValTileSeqLenQ, 2>(lane_row_sum_new, 0.0f);
1821
1822 #pragma unroll
1823 for (int i = 0; i < kValTileSeqLenQ; ++i) {
1824 #pragma unroll
1825     for (int j = 0; j < kValTileSeqLenK; ++j) {
1826         float tmp_max_0, tmp_max_1;
1827         if constexpr (kMmaAccF32) {
1828             float *t_fptr_S = reinterpret_cast<float *>(&R_S[i][j][0]);
1829             tmp_max_0 = max(t_fptr_S[0], t_fptr_S[1]) * scale;
1830             tmp_max_1 = max(t_fptr_S[2], t_fptr_S[3]) * scale;
1831         } else {
1832             float2 t_reg_S_0 = __half22float2(HALF2(R_S[i][j][0]));
1833             float2 t_reg_S_1 = __half22float2(HALF2(R_S[i][j][1]));
1834             tmp_max_0 = max(t_reg_S_0.x, t_reg_S_0.y) * scale;
1835             tmp_max_1 = max(t_reg_S_1.x, t_reg_S_1.y) * scale;
1836         }
1837         lane_row_max_new[i][0] = max(lane_row_max_new[i][0], tmp_max_0);
1838         lane_row_max_new[i][1] = max(lane_row_max_new[i][1], tmp_max_1);
1839     }
1840     lane_row_max_new[i][0] =
1841         warp_reduce_max<4, float>(lane_row_max_new[i][0]);
1842     lane_row_max_new[i][1] =
1843         warp_reduce_max<4, float>(lane_row_max_new[i][1]);
1844 }
1845
1846 #pragma unroll
1847 for (int i = 0; i < kValTileSeqLenQ; ++i) {
1848     float block_row_max_new_0 =
1849         max(lane_block_row_max_old[i][0], lane_row_max_new[i][0]);
1850     float block_row_max_new_1 =
1851         max(lane_block_row_max_old[i][1], lane_row_max_new[i][1]);
1852
1853 #pragma unroll
1854     for (int j = 0; j < kValTileSeqLenK; ++j) {
1855         if constexpr (kMmaAccF32) {
1856             float *t_fptr_S = reinterpret_cast<float *>(&R_S[i][j][0]);
1857             half *t_hptr_S = reinterpret_cast<half *>(&R_S[i][j][0]);
1858             t_fptr_S[0] = __expf(__fmaf_rn(t_fptr_S[0], scale, -block_row_max_new_0));
1859             t_fptr_S[1] = __expf(__fmaf_rn(t_fptr_S[1], scale, -block_row_max_new_0));
1860             t_fptr_S[2] = __expf(__fmaf_rn(t_fptr_S[2], scale, -block_row_max_new_1));
1861             t_fptr_S[3] = __expf(__fmaf_rn(t_fptr_S[3], scale, -block_row_max_new_1));
1862             lane_row_sum_new[i][0] += (t_fptr_S[0] + t_fptr_S[1]);
1863             lane_row_sum_new[i][1] += (t_fptr_S[2] + t_fptr_S[3]);

```

```

1864     t_hptr_S[0] = __float2half_rn(t_fptr_S[0]);
1865     t_hptr_S[1] = __float2half_rn(t_fptr_S[1]);
1866     t_hptr_S[2] = __float2half_rn(t_fptr_S[2]);
1867     t_hptr_S[3] = __float2half_rn(t_fptr_S[3]);
1868 } else {
1869     float2 t_reg_S_0 = __half2float2(HALF2(R_S[i][j][0]));
1870     float2 t_reg_S_1 = __half2float2(HALF2(R_S[i][j][1]));
1871     t_reg_S_0.x = __expf(__fmaf_rn(t_reg_S_0.x, scale, -block_row_max_new_0));
1872     t_reg_S_0.y = __expf(__fmaf_rn(t_reg_S_0.y, scale, -block_row_max_new_0));
1873     t_reg_S_1.x = __expf(__fmaf_rn(t_reg_S_1.x, scale, -block_row_max_new_1));
1874     t_reg_S_1.y = __expf(__fmaf_rn(t_reg_S_1.y, scale, -block_row_max_new_1));
1875     lane_row_sum_new[i][0] += (t_reg_S_0.x + t_reg_S_0.y);
1876     lane_row_sum_new[i][1] += (t_reg_S_1.x + t_reg_S_1.y);
1877     HALF2(R_S[i][j][0]) = __float2half2_rn(t_reg_S_0);
1878     HALF2(R_S[i][j][1]) = __float2half2_rn(t_reg_S_1);
1879 }
1880 }
1881 lane_row_sum_new[i][0] =
1882     warp_reduce_sum<4, float>(lane_row_sum_new[i][0]);
1883 lane_row_sum_new[i][1] =
1884     warp_reduce_sum<4, float>(lane_row_sum_new[i][1]);
1885 }
1886
1887 // ---- P @ V = 0[Br, d] ----
1888 // Wait for V[consumer_id] ready
1889 full_V[consumer_id].wait(full_V[consumer_id].arrive());
1890 tma_fence_proxy_async_shared_cta();
1891
1892 fill_3D_regs<uint32_t, kValTileSeqLenP, kValTileHeadDimV, kNRegs>(R_0, 0);
1893 #pragma unroll
1894 for (int tile_V_Bc = 0; tile_V_Bc < (Bc / kMmaAtomK); ++tile_V_Bc) {
1895 #pragma unroll
1896     for (int j = 0; j < kValTileHeadDimV; ++j) {
1897         const int warp_smem_V_d =
1898             warp_KV * (kMmaAtomN * kValTileHeadDimV) + j * kMmaAtomN;
1899         const int lane_smem_V_Bc = tile_V_Bc * kMmaAtomK + lane_id % 16;
1900         const int lane_smem_V_d = warp_smem_V_d;
1901         // V smem: V_wg_base + swizzle_fa offset
1902         const uint32_t lane_smem_V_ptr =
1903             smem_V_wg_base +
1904             swizzle_fa<kHeadDim, Bc>(lane_smem_V_Bc, lane_smem_V_d) *
1905                 sizeof(half);
1906         LDMATRIX_X2_T(R_V[j][0], R_V[j][1], lane_smem_V_ptr);
1907     }
1908
1909     // Early release V after last ldmatrix_V: V data now fully in registers.
1910     // This lets producer start next V TMA during the remaining PV MMA +
1911     // online rescaling, giving more overlap window than releasing after PV.
1912     if (tile_V_Bc == (Bc / kMmaAtomK) - 1) {
1913         [[maybe_unused]] auto token = empty_V[consumer_id].arrive();
1914     }
1915
1916     const int w = tile_V_Bc * 2;
1917 #pragma unroll
1918     for (int i = 0; i < kValTileSeqLenP; ++i) {
1919 #pragma unroll
1920         for (int j = 0; j < kValTileHeadDimV; ++j) {
1921             if constexpr (kMmaAccF32) {
1922                 HMMA16816F32(R_0[i][j][0], R_0[i][j][1], R_0[i][j][2], R_0[i][j][3],
1923                     R_S[i][w][0], R_S[i][w][1], R_S[i][w + 1][0], R_S[i][w + 1][1],
1924                     R_V[j][0], R_V[j][1],
1925                     R_0[i][j][0], R_0[i][j][1], R_0[i][j][2], R_0[i][j][3]);
1926             } else {

```

```

1927         HMMAl6816(R_0[i][j][0], R_0[i][j][1],
1928                 R_S[i][w][0], R_S[i][w][1],
1929                 R_S[i][w + 1][0], R_S[i][w + 1][1],
1930                 R_V[j][0], R_V[j][1],
1931                 R_0[i][j][0], R_0[i][j][1]);
1932     }
1933 }
1934 }
1935 }
1936
1937 // V already released above (after last ldmatrix_V)
1938
1939 // ---- Online rescaling - O_new = exp(m_old - m_new) * O_old + P@V ----
1940 #pragma unroll
1941 for (int i = 0; i < kValTileSeqLenP; ++i) {
1942     float block_row_max_new_0 = lane_row_max_new[i][0];
1943     float block_row_max_new_1 = lane_row_max_new[i][1];
1944     float block_row_sum_new_0 = lane_row_sum_new[i][0];
1945     float block_row_sum_new_1 = lane_row_sum_new[i][1];
1946     float block_row_max_old_0 = lane_block_row_max_old[i][0];
1947     float block_row_max_old_1 = lane_block_row_max_old[i][1];
1948
1949     block_row_max_new_0 = max(block_row_max_old_0, block_row_max_new_0);
1950     block_row_max_new_1 = max(block_row_max_old_1, block_row_max_new_1);
1951
1952     // First iteration per WG: m_old = -inf, use m_new directly
1953     block_row_max_old_0 = (local_iter > 0 ? block_row_max_old_0
1954                                         : block_row_max_new_0);
1955     block_row_max_old_1 = (local_iter > 0 ? block_row_max_old_1
1956                                         : block_row_max_new_1);
1957
1958     float rescale_o_factor_0 =
1959         __expf(block_row_max_old_0 - block_row_max_new_0);
1960     float rescale_o_factor_1 =
1961         __expf(block_row_max_old_1 - block_row_max_new_1);
1962
1963 #pragma unroll
1964 for (int j = 0; j < kValTileHeadDimV; ++j) {
1965     if constexpr (kMmaAccF32) {
1966         float *t_fptr_0 = reinterpret_cast<float *>(&R_0[i][j][0]);
1967         float *t_fptr_D = reinterpret_cast<float *>(&R_D[i][j][0]);
1968         t_fptr_D[0] = __fmaf_rn(rescale_o_factor_0, t_fptr_D[0], t_fptr_0[0]);
1969         t_fptr_D[1] = __fmaf_rn(rescale_o_factor_0, t_fptr_D[1], t_fptr_0[1]);
1970         t_fptr_D[2] = __fmaf_rn(rescale_o_factor_1, t_fptr_D[2], t_fptr_0[2]);
1971         t_fptr_D[3] = __fmaf_rn(rescale_o_factor_1, t_fptr_D[3], t_fptr_0[3]);
1972     } else {
1973         float2 t_reg_0_0 = __half22float2(HALF2(R_0[i][j][0]));
1974         float2 t_reg_0_1 = __half22float2(HALF2(R_0[i][j][1]));
1975         float2 t_reg_D_0 = __half22float2(HALF2(R_D[i][j][0]));
1976         float2 t_reg_D_1 = __half22float2(HALF2(R_D[i][j][1]));
1977         t_reg_D_0.x = __fmaf_rn(rescale_o_factor_0, t_reg_D_0.x, t_reg_0_0.x);
1978         t_reg_D_0.y = __fmaf_rn(rescale_o_factor_0, t_reg_D_0.y, t_reg_0_0.y);
1979         t_reg_D_1.x = __fmaf_rn(rescale_o_factor_1, t_reg_D_1.x, t_reg_0_1.x);
1980         t_reg_D_1.y = __fmaf_rn(rescale_o_factor_1, t_reg_D_1.y, t_reg_0_1.y);
1981         HALF2(R_D[i][j][0]) = __float22half2_rn(t_reg_D_0);
1982         HALF2(R_D[i][j][1]) = __float22half2_rn(t_reg_D_1);
1983     }
1984 }
1985
1986 float block_row_sum_old_0 = lane_block_row_sum_old[i][0];
1987 float block_row_sum_old_1 = lane_block_row_sum_old[i][1];
1988 lane_block_row_sum_old[i][0] =
1989     __fmaf_rn(rescale_o_factor_0, block_row_sum_old_0,

```

```

1990         block_row_sum_new_0);
1991     lane_block_row_sum_old[i][1] =
1992         __fmaf_rn(rescale_o_factor_1, block_row_sum_old_1,
1993                 block_row_sum_new_1);
1994     lane_block_row_max_old[i][0] = block_row_max_new_0;
1995     lane_block_row_max_old[i][1] = block_row_max_new_1;
1996 }
1997 ++local_iter;
1998 }
1999
2000 // =====
2001 // Final merge of two WG partial states (split-KV reduction)
2002 //
2003 // 安全 alias dynamic smem 的前提:
2004 // 1) producer 已退出 mainloop (所有 TMA 已 arrive_expect_tx);
2005 // 2) 两个 consumer 都已 release 自己的 K/V stage (QK 后 release K,
2006 //    PV 后 release V), 且 mainloop 结束后不再访问任何 K/V smem;
2007 // 3) Q_smem 在 mainloop 内只读, 此后也不再访问。
2008 // 故此时整个 dynamic smem (Q/K/V 区) 生命周期结束, 可安全重命名为
2009 // merge scratch, 不增加 launch smem_bytes。第一次 __syncthreads
2010 // 保证所有 TMA/ldmatrix 完成后才开始写 scratch。
2011 // 完整的 split-KV merge 数学推导见 kernel 头部注释。
2012 // =====
2013 __syncthreads();
2014
2015 // Merge scratch 布局 (按 wg_tid 索引, 每 WG 128 thread):
2016 // [merge_rd]: 128 * kRdPerThread 个 uint32 <- WG2 未归一化 R_D fragment
2017 // [merge_ml]: 128 * kValTileSeqLenQ 个 float4 <- WG2 的 [m0,m1,l0,l1] interleaved
2018 // WG1 用自己的 R_D/m/l 作为 (Oacc_0, m_0, l_0), 读取 WG2 写出的作为
2019 // (Oacc_1, m_1, l_1), 按 stable merge 公式归并后由 WG1 写最终 O。
2020 // m/l interleaved 为 float4 布局, 写入/读取各一次 128-bit store/load。
2021 constexpr int kRdPerThread = kValTileSeqLenP * kValTileHeadDimV * kNRegs;
2022 constexpr int kRdPackPerThread = kRdPerThread / 4; // float4 个数 (kNRegs 始终偶数)
2023 constexpr int kMlPackPerThread = kValTileSeqLenQ; // 每 thread 的 float4 个数
2024 uint32_t *merge_rd = reinterpret_cast<uint32_t*>(smem_fa3_tma_ws);
2025 float4 *merge_rd_pack = reinterpret_cast<float4*>(merge_rd);
2026 float4 *merge_ml_pack = merge_rd_pack + 128 * kRdPackPerThread;
2027
2028 // WG2 写出未归一化 partial (R_D, m, l) 到 scratch, 按 wg_tid 索引。
2029 // R_D: 以 float4 (128-bit) 连续写入, kRdPackPerThread 次。
2030 // m/l: interleaved 为 float4 [m0, m1, l0, l1], 每次 128-bit store。
2031 // 按 wg_tid 索引保证 WG1 的同一 lane 读到对应的 WG2 partial:
2032 // R_D[i][j][k] fragment 对应的 Q rows 与 lane_block_row_max/sum[i][0/1]
2033 // 的两组 rows (rows 0-7 / rows 8-15) 严格对齐, 故 WG1/WG2 同一
2034 // wg_tid 的 partial 覆盖相同 Q 行, 可直接逐元素归并。
2035 // Tc==1 时 WG2 未进入循环, m=-inf/l=0/R_D=0, merge 自然退化。
2036 // 写完后第二次 __syncthreads 让 WG1 看到 WG2 的全部写入。
2037 if (consumer_id == 1) {
2038     // R_D: 128-bit stores
2039     float4 *rd_dst = merge_rd_pack + wg_tid * kRdPackPerThread;
2040 #pragma unroll
2041     for (int idx = 0; idx < kRdPackPerThread; ++idx)
2042         rd_dst[idx] = reinterpret_cast<float4*>(&R_D[0][0][0])[idx];
2043     // m/l interleaved: [m0, m1, l0, l1] per i, 128-bit store
2044 #pragma unroll
2045     for (int i = 0; i < kValTileSeqLenQ; ++i) {
2046         float4 ml;
2047         ml.x = lane_block_row_max_old[i][0];
2048         ml.y = lane_block_row_max_old[i][1];
2049         ml.z = lane_block_row_sum_old[i][0];
2050         ml.w = lane_block_row_sum_old[i][1];
2051         merge_ml_pack[wg_tid * kMlPackPerThread + i] = ml;
2052     }

```

```

2053 }
2054 __syncthreads();
2055
2056 // WG1 merges WG2's partial into its own, normalizes, and writes 0
2057 if (consumer_id == 0) {
2058     // 把 WG2 的未归一化 R_D (Oacc_1) 载入 R_0 寄存器复用为临时 buffer。
2059     // R_D: 以 float4 (128-bit) 连续读取。
2060     // 后续 merge 用 R_D[i][j] 作为 Oacc_0, R_0[i][j] 作为 Oacc_1, 逐元素加权后
2061     // 结果写回 R_D, 再做最终 O=Oacc/l 归一化。
2062     float4 *rd_src = merge_rd_pack + wg_tid * kRdPackPerThread;
2063 #pragma unroll
2064     for (int idx = 0; idx < kRdPackPerThread; ++idx)
2065         reinterpret_cast<float4 *>(&R_0[0][0][0])[idx] = rd_src[idx];
2066
2067     // 稳定归并 (实现层面, 完整数学见头部注释):
2068     // - m0/l0 = WG1 自己的 partial (寄存器), m1/l1 = WG2 的 partial (scratch)
2069     // - [i][0] 对应 Q rows [i*16 .. i*16+7], [i][1] 对应 rows [i*16+8 .. +15]
2070     // - alpha=exp(m0-m)<=1, beta=exp(m1-m)<=1: 用全局 max 做 pivot 避免 exp 溢出
2071     // - F32Acc: 直接对 float fragment 逐元素加权求和
2072     // - F16Acc: 把 packed half2 提升到 float2 计算, 再 pack 回 half2
2073     //   (避免 half 精度下 alpha/beta 加权累加的精度损失)
2074     // - Tc==1: m1=-inf -> beta=exp(-inf)=0, Oacc/Oacc_1 项归零, 退化为 WG1 结果
2075     // m=max(m0,m1), a=exp(m0-m), b=exp(m1-m),
2076     // l=a*l0+b*l1, Oacc=a*Oacc0+b*Oacc1
2077 #pragma unroll
2078     for (int i = 0; i < kValTileSeqLenQ; ++i) {
2079         // 128-bit load: [m1_0, m1_1, l1_0, l1_1] interleaved
2080         float4 m1l = merge_ml_pack[wg_tid * kMlPackPerThread + i];
2081         float m1_0 = m1l.x;
2082         float m1_1 = m1l.y;
2083         float l1_0 = m1l.z;
2084         float l1_1 = m1l.w;
2085
2086         float m0_0 = lane_block_row_max_old[i][0];
2087         float m0_1 = lane_block_row_max_old[i][1];
2088         float l0_0 = lane_block_row_sum_old[i][0];
2089         float l0_1 = lane_block_row_sum_old[i][1];
2090
2091         float m_0 = fmaxf(m0_0, m1_0);
2092         float m_1 = fmaxf(m0_1, m1_1);
2093         float alpha_0 = __expf(m0_0 - m_0);
2094         float alpha_1 = __expf(m0_1 - m_1);
2095         float beta_0 = __expf(m1_0 - m_0);
2096         float beta_1 = __expf(m1_1 - m_1);
2097
2098         lane_block_row_max_old[i][0] = m_0;
2099         lane_block_row_max_old[i][1] = m_1;
2100         lane_block_row_sum_old[i][0] = alpha_0 * l0_0 + beta_0 * l1_0;
2101         lane_block_row_sum_old[i][1] = alpha_1 * l0_1 + beta_1 * l1_1;
2102
2103 #pragma unroll
2104         for (int j = 0; j < kValTileHeadDimV; ++j) {
2105             if constexpr (kMmaAccF32) {
2106                 float *d0 = reinterpret_cast<float *>(&R_D[i][j][0]);
2107                 float *d1 = reinterpret_cast<float *>(&R_0[i][j][0]);
2108                 d0[0] = alpha_0 * d0[0] + beta_0 * d1[0];
2109                 d0[1] = alpha_0 * d0[1] + beta_0 * d1[1];
2110                 d0[2] = alpha_1 * d0[2] + beta_1 * d1[2];
2111                 d0[3] = alpha_1 * d0[3] + beta_1 * d1[3];
2112             } else {
2113                 float2 d0_0 = __half22float2(HALF2(R_D[i][j][0]));
2114                 float2 d0_1 = __half22float2(HALF2(R_D[i][j][1]));
2115                 float2 d1_0 = __half22float2(HALF2(R_0[i][j][0]));

```

```

2116         float2 d1_1 = __half22float2(HALF2(R_0[i][j][1]));
2117         d0_0.x = alpha_0 * d0_0.x + beta_0 * d1_0.x;
2118         d0_0.y = alpha_0 * d0_0.y + beta_0 * d1_0.y;
2119         d0_1.x = alpha_1 * d0_1.x + beta_1 * d1_1.x;
2120         d0_1.y = alpha_1 * d0_1.y + beta_1 * d1_1.y;
2121         HALF2(R_D[i][j][0]) = __float22half2_rn(d0_0);
2122         HALF2(R_D[i][j][1]) = __float22half2_rn(d0_1);
2123     }
2124 }
2125 }
2126
2127 // ---- Final rescale: 0 = 0acc / l ----
2128 #pragma unroll
2129 for (int i = 0; i < kValTileSeqLenP; ++i) {
2130     float rescale_factor_0 = __frcp_rn(lane_block_row_sum_old[i][0]);
2131     float rescale_factor_1 = __frcp_rn(lane_block_row_sum_old[i][1]);
2132 #pragma unroll
2133     for (int j = 0; j < kValTileHeadDimV; ++j) {
2134         if constexpr (kMmaAccF32) {
2135             float *t_fptr_D = reinterpret_cast<float *>(&R_D[i][j][0]);
2136             half *t_hptr_D = reinterpret_cast<half *>(&R_D[i][j][0]);
2137             t_hptr_D[0] = __float2half_rn(rescale_factor_0 * t_fptr_D[0]);
2138             t_hptr_D[1] = __float2half_rn(rescale_factor_0 * t_fptr_D[1]);
2139             t_hptr_D[2] = __float2half_rn(rescale_factor_1 * t_fptr_D[2]);
2140             t_hptr_D[3] = __float2half_rn(rescale_factor_1 * t_fptr_D[3]);
2141         } else {
2142             float2 t_reg_D_0 = __half22float2(HALF2(R_D[i][j][0]));
2143             float2 t_reg_D_1 = __half22float2(HALF2(R_D[i][j][1]));
2144             t_reg_D_0.x = rescale_factor_0 * t_reg_D_0.x;
2145             t_reg_D_0.y = rescale_factor_0 * t_reg_D_0.y;
2146             t_reg_D_1.x = rescale_factor_1 * t_reg_D_1.x;
2147             t_reg_D_1.y = rescale_factor_1 * t_reg_D_1.y;
2148             HALF2(R_D[i][j][0]) = __float22half2_rn(t_reg_D_0);
2149             HALF2(R_D[i][j][1]) = __float22half2_rn(t_reg_D_1);
2150         }
2151     }
2152 }
2153
2154 // ---- Epilogue: warp shuffle + 128-bit collective store ----
2155 #pragma unroll
2156 for (int i = 0; i < kValTileSeqLenP; ++i) {
2157 #pragma unroll
2158     for (int j = 0; j < kValTileHeadDimV; ++j) {
2159         uint32_t R_Z[2][4];
2160         R_Z[0][0] = R_D[i][j][0];
2161         R_Z[1][0] = R_D[i][j][1];
2162         R_Z[0][1] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 1, 4);
2163         R_Z[0][2] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 2, 4);
2164         R_Z[0][3] = __shfl_sync(0xffffffff, R_D[i][j][0], lane_id + 3, 4);
2165         R_Z[1][1] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 1, 4);
2166         R_Z[1][2] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 2, 4);
2167         R_Z[1][3] = __shfl_sync(0xffffffff, R_D[i][j][1], lane_id + 3, 4);
2168
2169         if (lane_id % 4 == 0) {
2170             const int store_warp_regs_0_Br =
2171                 warp_QP * (kMmaAtomM * kValTileSeqLenP) + i * kMmaAtomM;
2172             const int store_lane_gmem_0_Br =
2173                 Q_tile_id * Br + store_warp_regs_0_Br + lane_id / 4;
2174             const int store_warp_regs_0_d =
2175                 warp_KV * (kMmaAtomN * kValTileHeadDimV) + j * kMmaAtomN;
2176             const int store_lane_gmem_0_d = store_warp_regs_0_d;
2177             const int store_gmem_0_addr_0 =
2178                 0_gmem_offset + store_lane_gmem_0_Br * kHeadDim +

```

```

2179         store_lane_gmem_0_d;
2180         const int store_gmem_0_addr_1 =
2181             0_gmem_offset + (store_lane_gmem_0_Br + 8) * kHeadDim +
2182             store_lane_gmem_0_d;
2183         *reinterpret_cast<float4*>(&O[store_gmem_0_addr_0]) =
2184             *reinterpret_cast<float4*>(&R_Z[0][0]);
2185         *reinterpret_cast<float4*>(&O[store_gmem_0_addr_1]) =
2186             *reinterpret_cast<float4*>(&R_Z[1][0]);
2187     }
2188 }
2189 }
2190 }
2191 }
2192 }
2193
2194 #if defined(NOTES_V2_ENABLE_CUTE)
2195 namespace fa_cute {
2196 using namespace cute;
2197
2198 // FlashAttn2CuTeTraits: FA2-style CuTe kernel 配置 (Br=128, Bc=64, 单 consumer WG)。
2199 // 与 FlashAttn3CuTeTraits 的核心区别:
2200 //   - Br=128 -> 8 warps along M (vs FA3 的 4), Q tile [128, D] (vs [64, D])
2201 //   - 单 consumer WG 全 KV 遍历, 无 split-KV merge
2202 //   - K/V tile 仍为 [64, D], SmemLayoutAtom/MmaAtom/SmemCopyAtom 与 FA3 一致
2203 //     (设计原因见 FlashAttn3CuTeTraits 头注释)
2204 //
2205 // 设计差异详述:
2206 // — SmemLayout (Q 更大, K/V 不变) —
2207 //   SmemLayoutQ = tile_to_shape(atom, (128, D)): Q tile [128, D] (Br=128),
2208 //   是 FA3 Q tile [64, D] 的 2 倍行数。8x8 atom 按 128 行重复, swizzle pattern 不变。
2209 //   SmemLayoutKV = tile_to_shape(atom, (64, D)): K/V tile [64, D] (Bc=64),
2210 //   与 FA3 的 SmemLayoutQKV 完全一致。
2211 //   SmemLayoutVt: 同 FA3, composition 叠加 col-major (D, 64) 得 V^T 转置视图。
2212 //
2213 // — MMA (8 warps, 单 WG 覆盖 Br=128) —
2214 //   TiledMma = TiledMMA(atom, EURepeat<8,1,1>, Tile<128,16,16>):
2215 //   - EURepeat<8,1,1>: 8 warp 沿 M 重复 MMA atom
2216 //   -> 8 warps × 32 threads = 256 threads = 1 consumer WG (FA3 为 128 threads)
2217 //   - Tile<128,16,16>: 逻辑 MMA tile
2218 //     M=128 = 8 warps × 16 rows = Br (单 WG 覆盖 Br=128, FA3 为 64)
2219 //     N=16, K=16: 与 FA3 一致
2220 //   MmaAtom / SmemCopyAtom / SmemCopyAtomTransposed 与 FA3 完全相同,
2221 //   不再重复说明 (见 FlashAttn3CuTeTraits 头注释)。
2222 // 复用 fa_cute 的 gemm_ss / gemm_rs / convert_layout_acc_* / convert_type。
2223 template <int kHeadDim>
2224 struct FlashAttn2CuTeTraits {
2225     static_assert(kHeadDim == 64 || kHeadDim == 128);
2226
2227     using Element = cutlass::half_t;
2228     using SmemLayoutAtom = GMMMA::Layout_K_SW128_Atom<Element>;
2229     // Q tile: [Br=128, D], K/V tile: [Bc=64, D]
2230     using SmemLayoutQ = decltype(tile_to_shape(
2231         SmemLayoutAtom{}, Shape<_128, Int<kHeadDim>>{}));
2232     using SmemLayoutKV = decltype(tile_to_shape(
2233         SmemLayoutAtom{}, Shape<_64, Int<kHeadDim>>{}));
2234     using SmemLayoutVt = decltype(composition(
2235         SmemLayoutKV{},
2236         make_layout(Shape<Int<kHeadDim>, _64>{}, GenRowMajor{})));
2237
2238     // 8 warps along M (Br=128 = 8*16), ValTile N=2 (16 cols = 2*8)
2239     using MmaAtom = MMA_Atom<SM80_16x8x16_F32F16F16F32_TN>;
2240     using TiledMma = TiledMMA<
2241         MmaAtom, Layout<Shape<_8, _1, _1>>,

```

```

2242     Tile<_128, _16, _16>;
2243     using SmemCopyAtom = Copy_Atom<SM75_U32x4_LDSM_N, Element>;
2244     using SmemCopyAtomTransposed = Copy_Atom<SM75_U16x8_LDSM_T, Element>;
2245 };
2246
2247 // FlashAttn3CuTeTraits: FA3-style CuTe kernel 编译期配置 (Br=Bc=64, 双 consumer WG)。
2248 // 所有 layout/atom/tiler 都是编译期类型, kernel body 只接收实例化后的类型。
2249 //
2250 // — SmemLayout 设计 (与 TMA 128B swizzle 匹配, 保证 ldmatrix 无 bank conflict) —
2251 //   SmemLayoutAtom = GMMMA::Layout_K_SW128_Atom<Element>:
2252 //     - GMMMA 128B swizzle atom: (8, 8) layout + Swizzle<3,4,3>
2253 //     - 一个 atom 覆盖 8 行 × 8 half = 128B, 恰好一个 swizzle 周期
2254 //     - 与 TMA CU_TENSOR_MAP_SWIZZLE_128B 写入 pattern 完全一致:
2255 //       producer 用 TMA 按 128B 粒度写入 swizzle 地址, consumer 用 cute::copy
2256 //       读取时由同款 swizzle layout 推导物理地址, 读写两侧 swizzle 一致 -> 无 bank conflict
2257 //   SmemLayoutQKV = tile_to_shape(atom, (64, D)):
2258 //     - Q/K/V tile 都是 [Br=Bc=64, D] (FA3 中 Br=Bc=64)
2259 //     - tile_to_shape 把 8×8 atom 按 (64, D) 重复, 保持 swizzle pattern
2260 //   SmemLayoutVt = composition(QKV, col-major (D, 64)):
2261 //     - V 转置视图: P@V 时 V 作为 MMA 的 B 矩阵, mma.sync.row.col 要求
2262 //       B 为 col-major。V 物理存储是 row-major [64, D], 通过 composition
2263 //       叠加 col-major (D, 64) 逻辑视图, 得到 V^T[D, 64] 访问接口。
2264 //     - composition 不搬数据, 只改 layout 解读, 物理 swizzle 不变。
2265 //
2266 // — MMA 设计 (Ampere m16n8k16, f32 累加保证 softmax 精度) —
2267 //   MmaAtom = SM80_16x8x16_F32F16F16F32_TN:
2268 //     - f16 输入 + f32 累加。FA 的 softmax 对精度敏感 (exp/sum),
2269 //       f16 累加会导致 row sum 溢出/下溢, 必须用 f32。
2270 //     - TN 布局: A(row-major) × B(col-major), 与 smem 物理布局天然匹配。
2271 //   TiledMma = TiledMMA(atom, EURepeat<4,1,1>, Tile<64,16,16>):
2272 //     - EURepeat<4,1,1>: 4 warp 沿 M 重复 MMA atom
2273 //     -> 4 warps × 32 threads = 128 threads = 1 consumer WG
2274 //     - Tile<64,16,16>: 逻辑 MMA tile
2275 //       M=64 = 4 warps × 16 rows = Br (单 WG 覆盖 Br=64)
2276 //       N=16 = 2 × kMmaN(8) -> 一次覆盖 16 列
2277 //       K=16 = kMmaK (单次 MMA K slice)
2278 //
2279 // — SmemCopy 设计 (S->R ldmatrix, 与 smem swizzle layout 配合) —
2280 //   SmemCopyAtom = SM75_U32x4_LDSM_N:
2281 //     - 对应 ldmatrix.sync.aligned.x4.m8n8.shared.b16 (非转置)
2282 //     - 一次加载 4 个 8×8 half 片段到 4 条 uint32 寄存器
2283 //     - 用于 Q 和 K: Q/K 在 smem 是 row-major, ldmatrix.x4 非转置加载
2284 //       得到 col-major fragment, 匹配 mma.row.col 的 A/B 输入约定
2285 //   SmemCopyAtomTransposed = SM75_U16x8_LDSM_T:
2286 //     - 对应 ldmatrix.sync.aligned.x2.trans.m8n8.shared.b16 (转置)
2287 //     - 用于 V: V 在 smem 是 row-major [Bc, D], P@V 需把 V 当 col-major
2288 //       B 矩阵。ldmatrix.x2.trans 转置加载为 col-major fragment, 直接匹配
2289 //       mma.row.col 的 B 输入, 无需软件转置 V smem。
2290 //     - 用 x2 (非 x4): V 的 K 维 (Bc=64) 按 kMmaK=16 切片, 每片 16×8
2291 //       = 1 个 ldmatrix.x2 (2 个 8×8 fragment)
2292 template <int kHeadDim>
2293 struct FlashAttn3CuTeTraits {
2294     static_assert(kHeadDim == 64 || kHeadDim == 128);
2295
2296     using Element = cutlass::half_t;
2297     using SmemLayoutAtom = GMMMA::Layout_K_SW128_Atom<Element>;
2298     using SmemLayoutQKV = decltype(tile_to_shape(
2299         SmemLayoutAtom{}, Shape<_64, Int<kHeadDim>>{}));
2300     using SmemLayoutVt = decltype(composition(
2301         SmemLayoutQKV{},
2302         make_layout(Shape<Int<kHeadDim>, _64>{}, GenRowMajor{})));
2303
2304     using MmaAtom = MMA_Atom<SM80_16x8x16_F32F16F16F32_TN>;

```

```

2305 using TiledMma = TiledMMA<
2306     MmaAtom, Layout<Shape<_4, _1, _1>>,
2307     Tile<_64, _16, _16>>;
2308 using SmemCopyAtom = Copy_Atom<SM75_U32x4_LDSM_N, Element>;
2309 using SmemCopyAtomTransposed = Copy_Atom<SM75_U16x8_LDSM_T, Element>;
2310 };
2311
2312 // convert_layout_acc_rowcol: 把 MMA accumulator 的 (MMA=4, MMA_M, MMA_N)
2313 // layout 转换为 ((2, MMA_M), (2, MMA_N)) 的 (nrow, ncol) 二维布局。
2314 //
2315 // 为什么需要转换? FA 的 online softmax 需要按"行"操作 score 矩阵 S:
2316 //   - 求 row max、exp(S - max)、row sum 都要求同一行的元素聚集在同一线程。
2317 //   - 但 m16n8k16 的 MMA C fragment 的 4 个寄存器按 (MMA=4, MMA_M, MMA_N)
2318 //     排布, lane 0/4/8/.. 持有同一行片段但散布在 MMA mode 0/1。
2319 //   - logical_divide(_, Shape<_2>{}) 把 MMA=4 拆成 (2, 2):
2320 //     (2, 2, MMA_M, MMA_N) -> 重组为 ((2, MMA_M), (2, MMA_N))
2321 //     外层 (2, MMA_M) = nrow: 前 2 个 MMA mode × MMA_M 组成"行块"
2322 //     内层 (2, MMA_N) = ncol: 后 2 个 MMA mode × MMA_N 组成"列块"
2323 //   - 转换后 scores(r, c) 的 r 索引行、c 索引列, online softmax 可直接遍历
2324 //     size<1>(scores) 做行内 reduce max/sum, 再用 __shfl_xor_sync 在 4-lane
2325 //     子组内归约(因为 m16n8k16 下每 4 个 lane 共享同一行的不同列片段)。
2326 //
2327 // 出处: flash-attention/csrc/flash_attn/src/utils.h:188
2328 //       flash-attention/csrc/flash_attn/src/softmax.h:139 (调用点)
2329 //       用于 tCrS (QK score) 和 tCr0 (PV accumulator) 的 rowcol 视图。
2330 template <typename Layout>
2331 CUTE_DEVICE auto convert_layout_acc_rowcol(Layout acc_layout) {
2332     auto divided = logical_divide(acc_layout, Shape<_2>{});
2333     return make_layout(
2334         make_layout(get<0, 1>(divided), get<1>(divided)),
2335         make_layout(get<0, 0>(divided), get<2>(divided)));
2336 }
2337
2338 // convert_layout_acc_Aregs: 把 MMA accumulator (QK 的 score S) 转换为
2339 // P@V 所需的 A-register (左矩阵 P) layout, 使 softmax 后的 P 能直接喂给
2340 // 下一个 MMA 做 P@V, 无需额外数据搬运。
2341 //
2342 // 为什么需要转换? QK 和 PV 用同一个 TiledMma, 但 C fragment 和 A fragment
2343 // 的寄存器排布不同:
2344 //   - QK:  $S = Q @ K^T$ , S 是 C fragment, layout = (MMA=4, MMA_M, MMA_N)
2345 //   - PV:  $O = P @ V$ , P 是 A fragment, layout = (MMA, MMA_M, MMA_K)
2346 //   - m16n8k16 的 MMA=4 对应 4 个寄存器, C 的 4 个按 (2行, 2列) 排布,
2347 //     A 的 4 个按 (2行, 2K-slice) 排布, 直接用 C layout 做 A 会导致
2348 //     寄存器对齐错误。
2349 //   - logical_divide(_, Shape<X, X, _2>{}) 把 MMA_N 拆成 (2, MMA_N/2):
2350 //     (MMA=4, MMA_M, (2, MMA_N/2)) -> 重组为 ((4, 2), MMA_M, MMA_N/2)
2351 //     外层 (4, 2) = 新的 MMA mode, 把 C 的列方向 2 个 half 配对到
2352 //     A 的 K 方向连续 2 个元素, 匹配 m16n8k16 的 K=16 内部 2-element 步长。
2353 //   - 转换后 tCrPv 可直接作为 gemm_rs 的 tCrA 参数, P 的数据仍在原寄存器中,
2354 //     只是逻辑坐标重解释, 零拷贝。
2355 //
2356 // ★ 这就是 FA "寄存器复用" 的核心技巧: softmax 后的 P 不写回 smem 再
2357 //   ldmatrix, 而是原地重解释为 A fragment, 省一次 smem 往返。
2358 //   但注意这依赖 m16n8k16 的特定 fragment 约定, 不是通用性质。
2359 //
2360 // 出处: flash-attention/csrc/flash_attn/src/utils.h:200
2361 //       flash-attention/csrc/flash_attn/src/flash_fwd_kernel.h:365 (调用点)
2362 //       注释 "Reshape rP from (MMA=4, MMA_M, MMA_N) to ((4, 2), MMA_M, MMA_N / 2)"
2363 template <typename TiledMma, typename Layout>
2364 CUTE_DEVICE auto convert_layout_acc_Aregs(Layout acc_layout) {
2365     using X = Underscore;
2366     auto divided = logical_divide(acc_layout, Shape<X, X, _2>{});
2367     return make_layout(

```

```

2368     make_layout(get<0>(divided), get<2, 0>(divided)),
2369     get<1>(divided), get<2, 1>(divided));
2370 }
2371
2372 // convert_type: 在寄存器层面做 dtype 转换, 不搬数据只改元素类型解释。
2373 //
2374 // 为什么需要? MMA accumulator 是 f32 (F32F16F16F32_TN), 但下游消费需要 f16:
2375 //   - tCrP = convert_type<Element>(tCrS): softmax 后的 score S (f32) 转 P (f16),
2376 //     供 P@V 的 A fragment (mma 要求 f16 输入)
2377 //   - tCr0_half = convert_type<Element>(tCr0): 最终 0 (f32 acc) 转 f16, 供 store
2378 //
2379 // 实现: NumericArrayConverter 一次性批量转换整个 fragment (kElements 个元素),
2380 // 返回的 tensor 共享原 layout, 只是 value_type 从 From 变为 To。
2381 // 零拷贝, 纯寄存器内转换 (make_rmem_ptr 标记为 register memory)。
2382 //
2383 // 出处: flash-attention/csrc/flash_attn/src/epilogue/epilogue.hpp (同类实现)
2384 template <typename To, typename Engine, typename Layout>
2385 CUTE_DEVICE auto convert_type(Tensor<Engine, Layout> const &tensor) {
2386     using From = typename Engine::value_type;
2387     constexpr int kElements = decltype(size(tensor))::value;
2388     cutlass::NumericArrayConverter<To, From, kElements> convert;
2389     auto fragment = convert(
2390         *reinterpret_cast<cutlass::Array<From, kElements> const *>(tensor.data()));
2391     return make_tensor(make_rmem_ptr<To>(&fragment), tensor.layout());
2392 }
2393
2394 // gemm_ss: Shared-Shared GEMM, A 和 B 都从 smem 加载到寄存器再做 MMA。
2395 // 用于 FA 的 Q@K^T 步骤: Q 和 K 都在 smem 中 (Q 由 TMA/cp.async 预加载,
2396 // K 由 TMA 加载到当前 stage), 需要 ldmatrix 同时搬运 A(Q) 和 B(K)。
2397 //
2398 // 流程 (software-pipelined ldmatrix + mma):
2399 //   1. retile_D: 把 A/B 的 register fragment 按 TiledCopy 的 source 视图重排,
2400 //     使 copy() 能正确写入后续 MMA 消费的寄存器位置。
2401 //   2. 预加载 tile_k=0: copy A[0] 和 B[0] 到寄存器。
2402 //   3. for tile_k = 0..MMA_K-1:
2403 //       a. 若 tile_k+1 存在, 预加载 A[tile_k+1] 和 B[tile_k+1] (与当前 MMA 重叠)
2404 //       b. gemm(mma, A[tile_k], B[tile_k], acc) - 发射 mma.sync 指令
2405 //   每个 K slice 的 ldmatrix 与前一个 slice 的 MMA 重叠执行, 隐藏 smem->reg 延迟。
2406 //
2407 // 参数:
2408 //   acc: (MMA, MMA_M, MMA_N) 累加器, 输入输出
2409 //   fragment_a/fragment_b: A/B 寄存器 fragment, 会被 retile 后复用
2410 //   shared_a/shared_b: smem 中的 A/B tile (带 swizzle layout)
2411 //   tiled_mma: TiledMMA 类型实例
2412 //   tiled_copy_a/b: A/B 的 S->R TiledCopy (make_tiled_copy_A/B 生成)
2413 //   thread_copy_a/b: A/B 的线程级 copy slice
2414 //
2415 // 出处: flash-attention/csrc/flash_attn/src/utils.h:166 (gemm_rs 的对照)
2416 //       FlashMLA/csrc/sm90/helpers.h:97 (gemm_ss 的同类实现)
2417 //       本 notes 的 QK GEMM 调用: fa_cute::gemm_ss(tCrS, tCrQ, tCrK, ...)
2418 template <typename TensorC, typename TensorA, typename TensorB,
2419          typename TensorSA, typename TensorSB, typename TiledMma,
2420          typename TiledCopyA, typename TiledCopyB,
2421          typename ThreadCopyA, typename ThreadCopyB>
2422 CUTE_DEVICE void gemm_ss(
2423     TensorC &acc, TensorA &fragment_a, TensorB &fragment_b,
2424     TensorSA const &shared_a, TensorSB const &shared_b,
2425     TiledMma tiled_mma, TiledCopyA tiled_copy_a, TiledCopyB tiled_copy_b,
2426     ThreadCopyA thread_copy_a, ThreadCopyB thread_copy_b) {
2427     auto copy_view_a = thread_copy_a.retile_D(fragment_a);
2428     auto copy_view_b = thread_copy_b.retile_D(fragment_b);
2429     copy(tiled_copy_a, shared_a(_, _, _0{}), copy_view_a(_, _, _0{}));
2430     copy(tiled_copy_b, shared_b(_, _, _0{}), copy_view_b(_, _, _0{}));

```

```

2431 #pragma unroll
2432 for (int tile_k = 0; tile_k < size<2>(fragment_a); ++tile_k) {
2433     if (tile_k + 1 < size<2>(fragment_a)) {
2434         copy(tiled_copy_a, shared_a(_, _, tile_k + 1),
2435             copy_view_a(_, _, tile_k + 1));
2436         copy(tiled_copy_b, shared_b(_, _, tile_k + 1),
2437             copy_view_b(_, _, tile_k + 1));
2438     }
2439     gemm(tiled_mma, fragment_a(_, _, tile_k),
2440         fragment_b(_, _, tile_k), acc);
2441 }
2442 }
2443
2444 // gemm_rs: Register-Shared GEMM, A 在寄存器中 (已是 fragment), B 从 smem 加载。
2445 // 用于 FA 的 P@V 步骤: P 已经在寄存器中 (softmax 后的 tCrS, 经 convert_layout_acc_Aregs
2446 // 重解释为 A fragment), V 在 smem 中 (TMA 加载), 只需 ldmatrix 搬运 B(V)。
2447 //
2448 // 与 gemm_ss 的区别: A 不从 smem 加载 (P 已在寄存器), 省 A 的 ldmatrix,
2449 // 这就是 FA "寄存器复用" 的性能优势: softmax 后的 P 不回写 smem, 直接做 A。
2450 //
2451 // 流程 (与 gemm_ss 对称, 但只 pipelined B):
2452 // 1. retile_D: B 的 register fragment 按 TiledCopy source 视图重排。
2453 // 2. 预加载 tile_k=0: copy B[0] 到寄存器。
2454 // 3. for tile_k = 0..MMA_K-1:
2455 //     a. 若 tile_k+1 存在, 预加载 B[tile_k+1]
2456 //     b. gemm(mma, A[tile_k], B[tile_k], acc) - A 是 P (寄存器), B 是 V (smem)
2457 //
2458 // 参数:
2459 // acc: (MMA, MMA_M, MMA_N) 累加器, 输出 0
2460 // fragment_a: P 寄存器 fragment (softmax 后的 score, 已 Aregs 转换)
2461 // fragment_b: V 寄存器 fragment (会被 retile 后复用)
2462 // shared_b: smem 中的 V tile (带 transposed swizzle layout)
2463 //
2464 // 出处: flash-attention/csrc/flash_attn/src/utils.h:166
2465 // flash-attention/csrc/flash_attn/src/flash_fwd_kernel.h:367 (调用点)
2466 // 本 notes 的 PV GEMM 调用: fa_cute::gemm_rs(tCr0, tCrPv, tCrV, ...)
2467 template <typename TensorC, typename TensorA, typename TensorB,
2468           typename TensorSB, typename TiledMma, typename TiledCopyB,
2469           typename ThreadCopyB>
2470 CUTE_DEVICE void gemm_rs(
2471     TensorC &acc, TensorA &fragment_a, TensorB &fragment_b,
2472     TensorSB const &shared_b, TiledMma tiled_mma,
2473     TiledCopyB tiled_copy_b, ThreadCopyB thread_copy_b) {
2474     auto copy_view_b = thread_copy_b.retile_D(fragment_b);
2475     copy(tiled_copy_b, shared_b(_, _, _0{}), copy_view_b(_, _, _0{}));
2476 #pragma unroll
2477 for (int tile_k = 0; tile_k < size<2>(fragment_a); ++tile_k) {
2478     if (tile_k + 1 < size<2>(fragment_a)) {
2479         copy(tiled_copy_b, shared_b(_, _, tile_k + 1),
2480             copy_view_b(_, _, tile_k + 1));
2481     }
2482     gemm(tiled_mma, fragment_a(_, _, tile_k),
2483         fragment_b(_, _, tile_k), acc);
2484 }
2485 }
2486
2487 } // namespace fa_cute
2488
2489 // =====
2490 // FA2 CuTe cp.async MMA Stages Split-Q (no TMA, no Warp Specialization)
2491 // =====
2492 // 是 flash_attn_tma_mma_ws_split_q_cute 的简化版: 去掉 TMA producer/consumer WS,
2493 // 改用 cp.async + 统一 pipeline (参考 hgemm_mma_stages_tn_cute 的 cute::copy 用法

```

```

2494 // + flash_attn_mma_stages_split_q 手写版本的 V/K group 管理策略)。
2495 //
2496 // ★ 核心设计:
2497 //   - 只支持 kStagesK (V 固定单 buffer, 无 kStagesV)
2498 //   - V 在 QK 之前发起 cp.async, 通过 QK+softmax 隐藏 V 延迟
2499 //   - V/K 分开 commit_group, 利用 cp.async group 的 LIFO 栈特性选择性 wait
2500 //
2501 // ★ cp.async group 管理策略 (对齐手写版 L4323-4765):
2502 //   提交顺序 (每轮 tile):
2503 //     1. copy V[tile] -> sV; cp_async_fence() (group V_tile, 栈底)
2504 //     2. copy K[tile+kStagesK-1] -> sK[write]; cp_async_fence() (group K_next, 栈顶)
2505 //   等待策略:
2506 //     QK 前: kStagesK==1 -> wait<0>+sync (V+K 都是当前 tile, 全等)
2507 //           kStagesK>1 -> 无 wait (K[tile] 已在上一轮循环末尾 wait<0> 完成)
2508 //     PV 前: kStagesK>1 且非最后 tile -> wait<1>+sync (栈顶 K_next 未完成, V_tile 完成)
2509 //           否则 -> wait<0>+sync
2510 //     循环末尾: kStagesK>1 且非最后 tile -> wait<0>+sync (确保 K_next 完成,
2511 //               下一轮 QK 安全; 同时保护 V/K smem 不被提前覆盖)
2512 //
2513 // ★ 同步链 (参考手写版 6 个同步点):
2514 //   1. Q load: copy + fence + wait<0> + sync
2515 //   2. PREFETCH K[0..Sk-2]: copy + fence, 然后 wait<Sk-2> + sync
2516 //   3. 每轮 tile:
2517 //       3a: V[tile] fence + K_next fence (kStagesK>1) 或 K[tile] fence (kStagesK==1)
2518 //       3b: QK 前 wait (kStagesK==1: wait<0>+sync; kStagesK>1: 无)
2519 //           QK gemm_ss
2520 //       3c: softmax (纯寄存器, 无 sync)
2521 //       3d: PV 前 wait (见上) + sync
2522 //           PV gemm_rs
2523 //       3e: rescale (纯寄存器, 无 sync)
2524 //     循环末尾: kStagesK>1 且非最后 -> wait<0> + sync
2525 //
2526 // 限制: 仅 self-attention, 无 causal/varlen/GQA (与 TMA FA2 v1 一致)
2527 template <int kHeadDim, int kStagesK = 2>
2528 __global__ void __launch_bounds__(256)
2529 flash_attn_mma_stages_split_q_cute(
2530     cutlass::half_t *Q, cutlass::half_t *K, cutlass::half_t *V,
2531     cutlass::half_t *output, int rows, int seqlen) {
2532     // rows: B * H * seqlen, seqlen: N, kHeadDim: D
2533     using namespace cute;
2534     using Traits = fa_cute::FlashAttn2CuTeTraits<kHeadDim>;
2535     using Element = typename Traits::Element;
2536     using SmemLayoutQ = typename Traits::SmemLayoutQ;
2537     using SmemLayoutKV = typename Traits::SmemLayoutKV;
2538     static_assert(kHeadDim == 64 || kHeadDim == 128, "D=64 or 128 only");
2539     static_assert(kStagesK >= 1, "kStagesK must be >= 1");
2540
2541     constexpr int kBr = 128;
2542     constexpr int kBc = 64;
2543     constexpr int kQTileElements = size(SmemLayoutQ{});
2544     constexpr int kKVTileElements = size(SmemLayoutKV{});
2545
2546     // Shared memory: Q[128,D] + K[kStagesK,64,D] + V[1,64,D]
2547     // __align__(1024): swizzle phase=0, cp.async 写 GMMA swizzle layout 同样要求
2548     extern __shared__ __align__(1024) Element shm[];
2549     auto sQ = make_tensor(make_smem_ptr(shm), SmemLayoutQ{});
2550     // K smem 带 stage mode: (64, D, kStagesK)
2551     using SmemLayoutKStage = decltype(tile_to_shape(
2552         typename Traits::SmemLayoutAtom{},
2553         make_shape(_64{}, Int<kHeadDim>{}, Int<kStagesK>{})));
2554     auto sK = make_tensor(make_smem_ptr(shm + kQTileElements), SmemLayoutKStage{});
2555     // V 单 buffer: (64, D), 无 stage mode
2556     Element *v_base = shm + kQTileElements + kStagesK * kKVTileElements;

```

```

2557 auto sV = make_tensor(make_smem_ptr(v_base), SmemLayoutKV{});
2558
2559 int tid = threadIdx.x;
2560 int q_tile = blockIdx.y * (seqLen / kBr) + blockIdx.x;
2561 int kv_tiles = seqLen / kBr;
2562 // * 多 head K/V offset: local_tile coord 需加 blockIdx.y * kv_tiles 偏移,
2563 // 否则所有 head 都读 head 0 的 K/V (参考 TMA 版 L6624 注释)
2564 int kv_base = blockIdx.y * kv_tiles;
2565
2566 // Global memory tensors: [rows=B*H*N, D] row-major
2567 auto mQ = make_tensor(make_gmem_ptr(Q), make_shape(rows, Int<kHeadDim>{}),
2568                       make_stride(Int<kHeadDim>{}, _1{}));
2569 auto mK = make_tensor(make_gmem_ptr(K), make_shape(rows, Int<kHeadDim>{}),
2570                       make_stride(Int<kHeadDim>{}, _1{}));
2571 auto mV = make_tensor(make_gmem_ptr(V), make_shape(rows, Int<kHeadDim>{}),
2572                       make_stride(Int<kHeadDim>{}, _1{}));
2573
2574 auto gQ = local_tile(mQ, Shape<_128, Int<kHeadDim>>{}, make_coord(q_tile, _0{}));
2575 // K/V 不在此处做 local_tile (无 remainder mode), main loop 内按 tile 做 local_tile + partition_S
2576 // (参考 TMA 版 producer L6624: 每次 local_tile 传入 blockIdx.y * kv_tiles + tile)
2577
2578 // G2S TiledCopy: 128-bit cp.async, 256 threads, 每线程 8 half = 128b
2579 using g2s_copy_op = SM80_CP_ASYNC_CACHEGLOBAL<cute::uint128_t>;
2580 using g2s_copy_atom = CopyAtom<CopyTraits<g2s_copy_op>, Element>;
2581 using G2SCopy = decltype(make_tiled_copy(
2582     g2s_copy_atom{},
2583     make_layout(make_shape(Int<32>{}, Int<8>{}), // ThrLayout: 32*8=256 threads
2584                 make_stride(Int<8>{}, Int<1>{})),
2585     make_layout(make_shape(Int<1>{}, Int<8>{}))); // ValLayout: 1*8=8 half=128b
2586 G2SCopy g2s_copy;
2587 auto g2s_thr = g2s_copy.get_slice(tid);
2588
2589 // G2S partition: Q source(gmem) 和 destination(smem)
2590 auto tQgQ = g2s_thr.partition_S(gQ);
2591 auto tQsQ = g2s_thr.partition_D(sQ);
2592 // K/V 的 smem destination (source 在 main loop 内每次 partition_S)
2593 auto tKsK = g2s_thr.partition_D(sK); // (CPY, CPY_M, CPY_K, kStagesK)
2594 auto tVsV = g2s_thr.partition_D(sV); // (CPY, CPY_M, CPY_K) 无 stage
2595
2596 // TiledMMA + S2R partition (完全复用 TMA 版 consumer 逻辑 L6696-6737)
2597 typename Traits::TiledMma tiled_mma;
2598 auto thr_mma = tiled_mma.get_thread_slice(tid);
2599 auto tCrQ = thr_mma.partition_fragment_A(sQ);
2600 // V non-swizzle 视图用于推导寄存器 layout (不能与 swizzle composition 冲突)
2601 auto sVt0 = make_tensor(sV.data(), typename Traits::SmemLayoutVt{});
2602 auto sVt0_ns = make_tensor(
2603     sV.data(), get_nonswizzle_portion(typename Traits::SmemLayoutVt{}));
2604 auto tCrV_layout = thr_mma.partition_fragment_B(sVt0_ns).layout();
2605 auto tCr0 = partition_fragment_C(tiled_mma, Shape<_128, Int<kHeadDim>>{});
2606 clear(tCr0);
2607
2608 auto s2r_copy_q = make_tiled_copy_A(typename Traits::SmemCopyAtom{}, tiled_mma);
2609 auto s2r_thr_q = s2r_copy_q.get_thread_slice(tid);
2610 auto tQsQ_s2r = s2r_thr_q.partition_S(sQ);
2611 auto s2r_copy_k = make_tiled_copy_B(typename Traits::SmemCopyAtom{}, tiled_mma);
2612 auto s2r_thr_k = s2r_copy_k.get_thread_slice(tid);
2613 auto s2r_copy_v = make_tiled_copy_B(
2614     typename Traits::SmemCopyAtomTransposed{}, tiled_mma);
2615 auto s2r_thr_v = s2r_copy_v.get_thread_slice(tid);
2616
2617 // Online softmax state
2618 auto tCr0_rc = make_tensor(
2619     tCr0.data(), fa_cute::convert_layout_acc_rowcol(tCr0.layout()));

```

```

2620 constexpr int kRows = decltype(size<0>(tCr0_rc))::value;
2621 float row_max[kRows], row_sum[kRows];
2622 #pragma unroll
2623 for (int r = 0; r < kRows; ++r) {
2624     row_max[r] = -INFINITY;
2625     row_sum[r] = 0.0f;
2626 }
2627
2628 float scale = rsqrtf(static_cast<float>(kHeadDim));
2629
2630 // ===== Step 1: Q 一次性 cp.async 加载 (split-Q 核心) =====
2631 cute::copy(g2s_copy, tQgQ, tQsQ);
2632 cp_async_fence();
2633 cp_async_wait<0>();
2634 __syncthreads();
2635
2636 // ===== Step 2: PREFETCH K 前 kStagesK-1 个 tile (仅 K, 不预取 V) =====
2637 int ismem_read = 0;
2638 int ismem_write = 0;
2639 #pragma unroll
2640 for (int s = 0; s < kStagesK - 1; ++s) {
2641     if (s < kv_tiles) {
2642         int kv_coord = kv_base + s;
2643         auto gK_tile = local_tile(mK, Shape<_64, Int<kHeadDim>>{}, make_coord(kv_coord, _0{}));
2644         auto tKgK_tile = g2s_thr.partition_S(gK_tile);
2645         cute::copy(g2s_copy, tKgK_tile, tKsK(_, _, _, s));
2646         cp_async_fence();
2647         ++ismem_write;
2648     }
2649 }
2650 if constexpr (kStagesK > 1) {
2651     cp_async_wait<kStagesK - 2>();
2652     __syncthreads();
2653 }
2654
2655 // ===== Step 3: Main loop over KV tiles =====
2656 for (int tile = 0; tile < kv_tiles; ++tile) {
2657     // 3a: 提交 V[tile] (group V) + K_next (group K prefetch)
2658     // V 单 buffer, 每轮重写 sV; V 在 QK 之前发起, 通过 QK+softmax 隐藏延迟
2659     {
2660         int kv_coord = kv_base + tile;
2661         auto gV_tile = local_tile(mV, Shape<_64, Int<kHeadDim>>{}, make_coord(kv_coord, _0{}));
2662         auto tVgV_tile = g2s_thr.partition_S(gV_tile);
2663         cute::copy(g2s_copy, tVgV_tile, tVsV);
2664     }
2665     cp_async_fence(); // group V_tile (栈底)
2666
2667     if constexpr (kStagesK > 1) {
2668         // Prefetch K[tile+kStagesK-1] -> sK[ismem_write] (group K_next, 栈顶)
2669         int k_next = tile + kStagesK - 1;
2670         if (k_next < kv_tiles) {
2671             int kv_coord = kv_base + k_next;
2672             auto gK_tile = local_tile(mK, Shape<_64, Int<kHeadDim>>{}, make_coord(kv_coord, _0{}));
2673             auto tKgK_tile = g2s_thr.partition_S(gK_tile);
2674             cute::copy(g2s_copy, tKgK_tile, tKsK(_, _, _, ismem_write));
2675             cp_async_fence(); // group K_next (栈顶)
2676         }
2677     } else {
2678         // kStagesK==1: 加载当前 K[tile] (smem_sel_next == smem_sel == 0)
2679         int kv_coord = kv_base + tile;
2680         auto gK_tile = local_tile(mK, Shape<_64, Int<kHeadDim>>{}, make_coord(kv_coord, _0{}));
2681         auto tKgK_tile = g2s_thr.partition_S(gK_tile);
2682         cute::copy(g2s_copy, tKgK_tile, tKsK(_, _, _, 0));

```

```

2683     cp_async_fence(); // group K_current (栈顶)
2684 }
2685
2686 // 3b: QK 前 wait K[tile] 就绪
2687 if constexpr (kStagesK == 1) {
2688     // kStagesK==1: V+K 都是当前 tile, wait<0> 等全部完成
2689     // (kStagesK==1 是退化路径, V 延迟无法隐藏)
2690     cp_async_wait<0>();
2691     __syncthreads();
2692 }
2693 // kStagesK>1: K[tile] 已在上一轮循环末尾 wait<0> 完成, 无需 wait
2694
2695 // Per-stage K smem (剥离 stage mode)
2696 auto sK_stg = make_tensor(
2697     make_smem_ptr(shm + kQTileElements + ismem_read * kKVTileElements),
2698     SmemLayoutKV{});
2699 auto tCrK = thr_mma.partition_fragment_B(sK_stg);
2700 CUTE_STATIC_ASSERT_V(size(tCrK) == size(tCrV_layout));
2701 auto tCrV = make_tensor(tCrK.data(), tCrV_layout); // K 和 V 共享寄存器存储
2702 auto tKsK_s2r = s2r_thr_k.partition_S(sK_stg);
2703
2704 // QK GEMM: S = Q @ K^T
2705 auto tCrS = partition_fragment_C(tiled_mma, Shape<_128, _64>{});
2706 clear(tCrS);
2707 fa_cute::gemm_ss(tCrS, tCrQ, tCrK, tQsQ_s2r, tKsK_s2r,
2708     tiled_mma, s2r_copy_q, s2r_copy_k, s2r_thr_q, s2r_thr_k);
2709
2710 // 3c: Online softmax (纯寄存器操作, V[tile] 在后台加载)
2711 auto scores = make_tensor(
2712     tCrS.data(), fa_cute::convert_layout_acc_rowcol(tCrS.layout()));
2713 #pragma unroll
2714 for (int r = 0; r < kRows; ++r) {
2715     float tile_max = -INFINITY;
2716 #pragma unroll
2717 for (int c = 0; c < size<1>(scores); ++c)
2718     tile_max = fmaxf(tile_max, scores(r, c) * scale);
2719 tile_max = fmaxf(tile_max, __shfl_xor_sync(0xffffffff, tile_max, 1));
2720 tile_max = fmaxf(tile_max, __shfl_xor_sync(0xffffffff, tile_max, 2));
2721 float nxt = fmaxf(row_max[r], tile_max);
2722 float rs = __expf(row_max[r] - nxt);
2723 #pragma unroll
2724 for (int c = 0; c < size<1>(tCr0_rc); ++c)
2725     tCr0_rc(r, c) *= rs;
2726 float ts = 0.0f;
2727 #pragma unroll
2728 for (int c = 0; c < size<1>(scores); ++c) {
2729     float p = __expf(scores(r, c) * scale - nxt);
2730     scores(r, c) = p;
2731     ts += p;
2732 }
2733 ts += __shfl_xor_sync(0xffffffff, ts, 1);
2734 ts += __shfl_xor_sync(0xffffffff, ts, 2);
2735 row_sum[r] = row_sum[r] * rs + ts;
2736 row_max[r] = nxt;
2737 }
2738
2739 // 3d: PV 前 wait V[tile] 就绪
2740 // kStagesK>1: K_next 提交条件是 tile + kStagesK - 1 < kv_tiles。
2741 // 已提交时栈顶有 K_next, wait<1> 只等 V_tile;
2742 // 未提交 (尾部 Sk-1 个 tile) 时栈只有 V_tile, 必须 wait<0>。
2743 // tile=0 必然已提交 (PREFETCH 保证 kv_tiles >= kStagesK-1, 且循环能跑说明 kv_tiles >= 1)。
2744 if constexpr (kStagesK > 1) {
2745     if (tile + kStagesK - 1 < kv_tiles) {

```

```

2746     cp_async_wait<1>();
2747 } else {
2748     cp_async_wait<0>();
2749 }
2750 }
2751 // kStagesK==1: 3b 已 wait<0>, V 已就绪
2752 __syncthreads();
2753
2754 // PV GEMM: 0 = P @ V
2755 // V smem 带 swizzle: partition_S 推导 smem 源地址映射, copy() 正确应用 swizzle
2756 auto sVt_stg = make_tensor(sV.data(), typename Traits::SmemLayoutVt{});
2757 auto tVsVt_s2r = s2r_thr_v.partition_S(sVt_stg);
2758 auto tCrP = fa_cute::convert_type<Element>(tCrS);
2759 auto tCrPv = make_tensor(
2760     tCrP.data(),
2761     fa_cute::convert_layout_acc_Aregs<typename Traits::TiledMma>(
2762         tCrP.layout()));
2763 fa_cute::gemm_rs(tCr0, tCrPv, tCrV, tVsVt_s2r,
2764     tiled_mma, s2r_copy_v, s2r_thr_v);
2765 __syncthreads(); // 确保 ldmatrix V 完成后才能覆盖 V smem
2766
2767 // 3e: rescale 已合并到 3c softmax pass
2768 // 循环末尾: kStagesK>1 且非最后 tile -> wait<0> 确保 K_next 完成
2769 // (下一轮 QK 安全; 同时保护 V/K smem 不被提前覆盖)
2770 if constexpr (kStagesK > 1) {
2771     if (tile + 1 < kv_tiles) {
2772         cp_async_wait<0>();
2773         __syncthreads();
2774     }
2775 }
2776
2777 // 更新 pipeline 指针
2778 ismem_read = (ismem_read + 1) % kStagesK;
2779 ismem_write = (ismem_write + 1) % kStagesK;
2780 }
2781
2782 // ===== Step 4: Final normalize 0 = 0acc / l =====
2783 #pragma unroll
2784 for (int r = 0; r < kRows; ++r) {
2785     float inv_sum = 1.0f / row_sum[r];
2786 #pragma unroll
2787 for (int c = 0; c < size<1>(tCr0_rc); ++c)
2788     tCr0_rc(r, c) *= inv_sum;
2789 }
2790
2791 // ===== Step 5: Epilogue - direct R->G store =====
2792 auto tCr0_half = fa_cute::convert_type<Element>(tCr0);
2793 auto m0 = make_tensor(make_gmem_ptr(output),
2794     make_shape(rows, Int<kHeadDim>{}),
2795     make_stride(Int<kHeadDim>{}, _1{}));
2796 auto g0 = local_tile(m0, Shape<_128, Int<kHeadDim>>{}, make_coord(q_tile, _0{}));
2797 auto tCg0 = thr_mma.partition_C(g0);
2798 copy(tCr0_half, tCg0);
2799 }
2800
2801 template <int kHeadDim, typename TmaQ, typename TmaK, typename TmaV,
2802     int kStagesK = 1, int kStagesV = 1>
2803 __global__ void __launch_bounds__(384, 1)
2804 flash_attn_tma_mma_ws_split_q_cute(
2805     CUTLASS_GRID_CONSTANT TmaQ const tma_q,
2806     CUTLASS_GRID_CONSTANT TmaK const tma_k,
2807     CUTLASS_GRID_CONSTANT TmaV const tma_v,
2808     cutlass::half_t *output, int rows, int seqlen) {

```

```

2809 // rows: B * H * seqlen, seqlen: N, kHeadDim: D
2810 using namespace cute;
2811 using Traits = fa_cute::FlashAttn2CuTeTraits<kHeadDim>;
2812 using Element = typename Traits::Element;
2813 using SmemLayoutQ = typename Traits::SmemLayoutQ;
2814 using SmemLayoutKV = typename Traits::SmemLayoutKV;
2815 using TmaBarrier = cutlass::arch::ClusterTransactionBarrier;
2816 using CtaBarrier = cutlass::arch::ClusterBarrier;
2817 constexpr int kBr = 128;
2818 constexpr int kBc = 64;
2819 constexpr int kConsumerThreads = 256;
2820 constexpr int kProducerThreads = 128;
2821 constexpr int kQTileElements = cosize(SmemLayoutQ{});
2822 constexpr int kKVTileElements = cosize(SmemLayoutKV{});
2823
2824 extern __shared__ __align__(1024) Element shm[];
2825 auto sQ = make_tensor(make_smem_ptr(shm), SmemLayoutQ{});
2826 Element *k_base = shm + kQTileElements;
2827 Element *v_base = k_base + kStagesK * kKVTileElements;
2828
2829 __shared__ uint64_t q_full;
2830 __shared__ uint64_t k_full[kStagesK];
2831 __shared__ uint64_t k_empty[kStagesK];
2832 __shared__ uint64_t v_full[kStagesV];
2833 __shared__ uint64_t v_empty[kStagesV];
2834
2835 const bool is_producer = threadIdx.x < kProducerThreads;
2836 const int wg_tid = is_producer ? threadIdx.x
2837 : threadIdx.x - kProducerThreads;
2838
2839 if (threadIdx.x == 0) {
2840     TmaBarrier::init(&q_full, 1);
2841     for (int s = 0; s < kStagesK; ++s) {
2842         TmaBarrier::init(&k_full[s], 1);
2843         CtaBarrier::init(&k_empty[s], kConsumerThreads);
2844     }
2845     for (int s = 0; s < kStagesV; ++s) {
2846         TmaBarrier::init(&v_full[s], 1);
2847         CtaBarrier::init(&v_empty[s], kConsumerThreads);
2848     }
2849 }
2850 __syncthreads();
2851
2852 int q_tile = blockIdx.y * (seqlen / kBr) + blockIdx.x;
2853 int kv_tiles = seqlen / kBc;
2854
2855 // =====
2856 // Producer Warpgroup (WG0, threadIdx.x 0~127)
2857 // Only wg_tid == 0 issues TMA. Load Q once, then V-first K-after KV loop.
2858 // P1: prefetch K[0..Sk-2] + V[0..Sv-2] (if Sk/Sv > 1).
2859 // P2: V[tile+Sv-1] then K[tile+Sk-1] (prefetch ahead Sk-1/Sv-1, 对齐 8b).
2860 // =====
2861 if (is_producer) {
2862     if (wg_tid == 0) {
2863         auto mQ = tma_q.get_tma_tensor(make_shape(rows, Int<kHeadDim>{}));
2864         auto gQ = local_tile(
2865             mQ, Shape<_128, Int<kHeadDim>>{}, make_coord(q_tile, _0{}));
2866         auto q_slice = tma_q.get_slice(_0{});
2867         auto tQgQ = q_slice.partition_S(gQ);
2868         auto tQsQ = q_slice.partition_D(sQ);
2869         TmaBarrier::arrive_and_expect_tx(
2870             &q_full, sizeof(Element) * size(sQ));
2871         copy(tma_q.with(q_full), tQgQ, tQsQ);

```

```

2872 auto mK = tma_k.get_tma_tensor(make_shape(rows, Int<kHeadDim>{}));
2873 auto mV = tma_v.get_tma_tensor(make_shape(rows, Int<kHeadDim>{}));
2874 auto k_slice = tma_k.get_slice(_0{});
2875 auto v_slice = tma_v.get_slice(_0{});
2876
2877
2878 // * 易错点 (CuTe TMA multi-head): K/V local_tile 的 coord 是相对于 TMA
2879 // tensor row 维的全局 tile 索引 (TMA tensor shape = [B*H*N, D]), 不是
2880 // per-head tile 索引。多 head 场景必须加 `blockIdx.y * kv_tiles` 偏移,
2881 // 否则所有 head 都读 head 0 的 K/V -> correctness test (H=1) 正常但
2882 // bench (H>1) 误差飙升 2 个量纲。Q 的 q_tile 已含 blockIdx.y 无需修复。
2883 // 参考: /memories/repo/leetcuda-tma-fa-multihead.md
2884 // P1: Prefetch first (kStagesK-1) K tiles
2885 for (int s = 0; s < kStagesK - 1; ++s) {
2886     if (s < kv_tiles) {
2887         auto sK = make_tensor(
2888             make_smem_ptr(k_base + s * kKVTileElements), SmemLayoutKV{});
2889         CtaBarrier::wait(&k_empty[s], 0);
2890         auto gK = local_tile(
2891             mK, Shape<_64, Int<kHeadDim>>{},
2892             make_coord(blockIdx.y * kv_tiles + s, _0{}));
2893         auto tKgK = k_slice.partition_S(gK);
2894         auto tKsK = k_slice.partition_D(sK);
2895         TmaBarrier::arrive_and_expect_tx(
2896             &k_full[s], sizeof(Element) * size(sK));
2897         copy(tma_k.with(k_full[s]), tKgK, tKsK);
2898     }
2899 }
2900
2901 // P1b: Prefetch first (kStagesV-1) V tiles
2902 for (int s = 0; s < kStagesV - 1; ++s) {
2903     if (s < kv_tiles) {
2904         auto sV = make_tensor(
2905             make_smem_ptr(v_base + s * kKVTileElements), SmemLayoutKV{});
2906         CtaBarrier::wait(&v_empty[s], 0);
2907         auto gV = local_tile(
2908             mV, Shape<_64, Int<kHeadDim>>{},
2909             make_coord(blockIdx.y * kv_tiles + s, _0{}));
2910         auto tVgV = v_slice.partition_S(gV);
2911         auto tVsV = v_slice.partition_D(sV);
2912         TmaBarrier::arrive_and_expect_tx(
2913             &v_full[s], sizeof(Element) * size(sV));
2914         copy(tma_v.with(v_full[s]), tVgV, tVsV);
2915     }
2916 }
2917
2918 // P2: Main loop - V-first then K-after (对齐 8b producer 顺序)
2919 for (int tile = 0; tile < kv_tiles; ++tile) {
2920     // V: load V[tile+Sv-1] (prefetch ahead Sv-1)
2921     {
2922         int v_tile = tile + kStagesV - 1;
2923         if (v_tile < kv_tiles) {
2924             int stage_v = v_tile % kStagesV;
2925             int phase_v = (v_tile / kStagesV) & 1;
2926             auto sV = make_tensor(
2927                 make_smem_ptr(v_base + stage_v * kKVTileElements),
2928                 SmemLayoutKV{});
2929             CtaBarrier::wait(&v_empty[stage_v], phase_v);
2930             auto gV = local_tile(
2931                 mV, Shape<_64, Int<kHeadDim>>{},
2932                 make_coord(blockIdx.y * kv_tiles + v_tile, _0{}));
2933             auto tVgV = v_slice.partition_S(gV);
2934             auto tVsV = v_slice.partition_D(sV);

```

```

2935         TmaBarrier::arrive_and_expect_tx(
2936             &v_full[stage_v], sizeof(Element) * size(sV));
2937         copy(tma_v.with(v_full[stage_v]), tVgV, tVsV);
2938     }
2939 }
2940 // K: prefetch K[tile+Sk-1] (prefetch ahead Sk-1)
2941 {
2942     int k_tile = tile + kStagesK - 1;
2943     if (k_tile < kv_tiles) {
2944         int stage_k = k_tile % kStagesK;
2945         int phase_k = (k_tile / kStagesK) & 1;
2946         auto sK = make_tensor(
2947             make_smem_ptr(k_base + stage_k * kKVTileElements),
2948             SmemLayoutKV{});
2949         CtaBarrier::wait(&k_empty[stage_k], phase_k);
2950         auto gK = local_tile(
2951             mK, Shape<_64, Int<kHeadDim>>{},
2952             make_coord(blockIdx.y * kv_tiles + k_tile, _0{}));
2953         auto tKgK = k_slice.partition_S(gK);
2954         auto tKsK = k_slice.partition_D(sK);
2955         TmaBarrier::arrive_and_expect_tx(
2956             &k_full[stage_k], sizeof(Element) * size(sK));
2957         copy(tma_k.with(k_full[stage_k]), tKgK, tKsK);
2958     }
2959 }
2960 }
2961 }
2962 }
2963 // =====
2964 // Consumer Warpgroup (WG1, threadIdx.x 128~383, 256 threads = 8 warps)
2965 // Single consumer: full KV traversal, no split-KV merge.
2966 // Flow: wait K -> QK -> release K -> softmax -> wait V -> PV -> release V.
2967 // =====
2968 else {
2969     TmaBarrier::wait(&q_full, 0);
2970
2971     // V layout 从 stage 0 推导 (所有 stage layout 相同)
2972     auto sV0 = make_tensor(make_smem_ptr(v_base), SmemLayoutKV{});
2973     auto sVt0 = make_tensor(sV0.data(), typename Traits::SmemLayoutVt{});
2974     // * get_nonswizzle_portion 只用于 partition_fragment_B 推导寄存器侧 B
2975     // fragment layout (每线程持有哪些 V 元素), 寄存器布局与 smem swizzle 无关。
2976     // V 的 smem bank conflict 由 TMA 写入 (SmemLayoutKV 带 swizzle) +
2977     // ldmatrix 读取 (主循环内 partition_S(sVt_stg) 带 swizzle) 共同解决。
2978     // 这里不能用带 swizzle 的 sVt0 做 partition_fragment_B, 否则 CuTe 推导
2979     // ldmatrix.trans 的线程-数据映射时会与 swizzle composition 冲突。
2980     // 参考: flash-attention/csrc/flash_attn/src/kernel_traits.h
2981     // SmemLayoutVtransposedNoSwizzle 的同样用法。
2982     auto sVt0_ns = make_tensor(
2983         sV0.data(), get_nonswizzle_portion(typename Traits::SmemLayoutVt{}));
2984
2985     typename Traits::TiledMma tiled_mma;
2986     auto thr_mma = tiled_mma.get_thread_slice(wg_tid);
2987     auto tCrQ = thr_mma.partition_fragment_A(sQ);
2988     auto tCrV_layout = thr_mma.partition_fragment_B(sVt0_ns).layout();
2989     auto tCr0 = partition_fragment_C(
2990         tiled_mma, Shape<_128, Int<kHeadDim>>{});
2991     clear(tCr0);
2992
2993     auto s2r_copy_q = make_tiled_copy_A(
2994         typename Traits::SmemCopyAtom{}, tiled_mma);
2995     auto s2r_thr_q = s2r_copy_q.get_thread_slice(wg_tid);
2996     auto tQsQ_s2r = s2r_thr_q.partition_S(sQ);
2997     auto s2r_copy_k = make_tiled_copy_B(

```

```

2998     typename Traits::SmemCopyAtom{}, tiled_mma);
2999     auto s2r_thr_k = s2r_copy_k.get_thread_slice(wg_tid);
3000     auto s2r_copy_v = make_tiled_copy_B(
3001         typename Traits::SmemCopyAtomTransposed{}, tiled_mma);
3002     auto s2r_thr_v = s2r_copy_v.get_thread_slice(wg_tid);
3003
3004     auto tCr0_rc = make_tensor(
3005         tCr0.data(),
3006         fa_cute::convert_layout_acc_rowcol(tCr0.layout()));
3007     constexpr int kRows = decltype(size<0>(tCr0_rc))::value;
3008     float row_max[kRows], row_sum[kRows];
3009     #pragma unroll
3010     for (int r = 0; r < kRows; ++r) {
3011         row_max[r] = -INFINITY;
3012         row_sum[r] = 0.0f;
3013     }
3014
3015     // Init: mark all K/V slots as empty (256 consumer arrivals each)
3016     for (int s = 0; s < kStagesK; ++s)
3017         CtaBarrier::arrive(&k_empty[s]);
3018     for (int s = 0; s < kStagesV; ++s)
3019         CtaBarrier::arrive(&v_empty[s]);
3020
3021     float scale = rsqrtf(static_cast<float>(kHeadDim));
3022     for (int tile = 0; tile < kv_tiles; ++tile) {
3023         int k_stg = tile % kStagesK;
3024         int k_phase = (tile / kStagesK) & 1;
3025         int v_stg = tile % kStagesV;
3026         int v_phase = (tile / kStagesV) & 1;
3027
3028         // Per-stage K smem and S2R partition
3029         auto sK_stg = make_tensor(
3030             make_smem_ptr(k_base + k_stg * kKVTileElements), SmemLayoutKV{});
3031         auto tCrK = thr_mma.partition_fragment_B(sK_stg);
3032         CUTE_STATIC_ASSERT_V(size(tCrK) == size(tCrV_layout));
3033         auto tCrV = make_tensor(tCrK.data(), tCrV_layout);
3034         auto tKsK_s2r = s2r_thr_k.partition_S(sK_stg);
3035
3036         // 1) Wait K, QK gemm, release K early
3037         TmaBarrier::wait(&k_full[k_stg], k_phase);
3038         auto tCrS = partition_fragment_C(tiled_mma, Shape<_128, _64>{});
3039         clear(tCrS);
3040         fa_cute::gemm_ss(
3041             tCrS, tCrQ, tCrK, tQsQ_s2r, tKsK_s2r,
3042             tiled_mma, s2r_copy_q, s2r_copy_k, s2r_thr_q, s2r_thr_k);
3043         { CtaBarrier::arrive(&k_empty[k_stg]); }
3044
3045         // 2) Online softmax (overlaps with producer's V TMA)
3046         auto scores = make_tensor(
3047             tCrS.data(),
3048             fa_cute::convert_layout_acc_rowcol(tCrS.layout()));
3049     #pragma unroll
3050     for (int r = 0; r < kRows; ++r) {
3051         float tile_max = -INFINITY;
3052     #pragma unroll
3053     for (int c = 0; c < size<1>(scores); ++c)
3054         tile_max = fmaxf(tile_max, scores(r, c) * scale);
3055     tile_max = fmaxf(tile_max, __shfl_xor_sync(0xffffffff, tile_max, 1));
3056     tile_max = fmaxf(tile_max, __shfl_xor_sync(0xffffffff, tile_max, 2));
3057     float nxt = fmaxf(row_max[r], tile_max);
3058     float rs = __expf(row_max[r] - nxt);
3059     #pragma unroll
3060     for (int c = 0; c < size<1>(tCr0_rc); ++c)

```

```

3061         tCr0_rc(r, c) *= rs;
3062         float ts = 0.0f;
3063 #pragma unroll
3064         for (int c = 0; c < size<1>(scores); ++c) {
3065             float p = __expf(scores(r, c) * scale - nxt);
3066             scores(r, c) = p;
3067             ts += p;
3068         }
3069         ts += __shfl_xor_sync(0xffffffff, ts, 1);
3070         ts += __shfl_xor_sync(0xffffffff, ts, 2);
3071         row_sum[r] = row_sum[r] * rs + ts;
3072         row_max[r] = nxt;
3073     }
3074
3075     // 3) Wait V, PV gemm, release V early
3076     // * sVt_stg 带 swizzle: partition_S 推导 smem 源地址映射, copy() 会
3077     // 正确应用 swizzle 物理地址, 与 TMA 写入 (SmemLayoutKV 带 swizzle) 匹配,
3078     // 解决 V 的 smem bank conflict。这是 V bank-conflict-free 的关键路径。
3079     auto sV_stg = make_tensor(
3080         make_smem_ptr(v_base + v_stg * kKVTileElements), SmemLayoutKV{});
3081     auto sVt_stg = make_tensor(sV_stg.data(), typename Traits::SmemLayoutVt{});
3082     auto tVsVt_s2r = s2r_thr_v.partition_S(sVt_stg);
3083
3084     TmaBarrier::wait(&v_full[v_stg], v_phase);
3085     auto tCrP = fa_cute::convert_type<Element>(tCrS);
3086     auto tCrPv = make_tensor(
3087         tCrP.data(),
3088         fa_cute::convert_layout_acc_Aregs<typename Traits::TiledMma>(
3089             tCrP.layout()));
3090     fa_cute::gemm_rs(
3091         tCr0, tCrPv, tCrV, tVsVt_s2r,
3092         tiled_mma, s2r_copy_v, s2r_thr_v);
3093     { CtaBarrier::arrive(&v_empty[v_stg]); }
3094 }
3095
3096     // ---- Final normalize: 0 = 0acc / l (single consumer, no merge) ----
3097 #pragma unroll
3098     for (int r = 0; r < kRows; ++r) {
3099         float inv_sum = 1.0f / row_sum[r];
3100 #pragma unroll
3101         for (int c = 0; c < size<1>(tCr0_rc); ++c)
3102             tCr0_rc(r, c) *= inv_sum;
3103     }
3104
3105     auto tCr0_half = fa_cute::convert_type<Element>(tCr0);
3106     auto m0 = make_tensor(
3107         make_gmem_ptr(output),
3108         make_shape(rows, Int<kHeadDim>{}),
3109         make_stride(Int<kHeadDim>{}, _1{}));
3110     auto g0 = local_tile(
3111         m0, Shape<_128, Int<kHeadDim>>{}, make_coord(q_tile, _0{}));
3112     auto tCg0 = thr_mma.partition_C(g0);
3113     copy(tCr0_half, tCg0);
3114 }
3115 }
3116
3117 template <int kHeadDim, typename TmaQ, typename TmaK, typename TmaV,
3118         int kStagesK = 1>
3119 __global__ void __launch_bounds__(384, 1)
3120 flash_attn_3_tma_mma_ws_split_q_cute(
3121     CUTLASS_GRID_CONSTANT TmaQ const tma_q,
3122     CUTLASS_GRID_CONSTANT TmaK const tma_k,
3123     CUTLASS_GRID_CONSTANT TmaV const tma_v,

```

```

3124     cutlass::half_t *output, int rows, int seqlen) {
3125 // rows: B * H * seqlen, seqlen: N, kHeadDim: D
3126 using namespace cute;
3127 using Traits = fa_cute::FlashAttn3CuTeTraits<kHeadDim>;
3128 using Element = typename Traits::Element;
3129 using SmemLayout = typename Traits::SmemLayoutQKV;
3130 using TmaBarrier = cutlass::arch::ClusterTransactionBarrier;
3131 using CtaBarrier = cutlass::arch::ClusterBarrier;
3132 constexpr int kTile = 64;
3133 constexpr int kNumConsumers = 2;
3134 constexpr int kConsumerThreads = 128;
3135 constexpr int kProducerThreads = 128;
3136 constexpr int kTileElements = cosize(SmemLayout{});
3137
3138 extern __shared__ __align__(1024) Element shm[];
3139 auto sQ = make_tensor(make_smem_ptr(shm), SmemLayout{});
3140 Element *k_base = shm + kTileElements;
3141 Element *v_base = k_base + kNumConsumers * kStagesK * kTileElements;
3142
3143 __shared__ uint64_t q_full;
3144 __shared__ uint64_t k_full[kNumConsumers][kStagesK];
3145 __shared__ uint64_t k_empty[kNumConsumers][kStagesK];
3146 __shared__ uint64_t v_full[kNumConsumers];
3147 __shared__ uint64_t v_empty[kNumConsumers];
3148
3149 const bool is_producer = threadIdx.x < kProducerThreads;
3150 const int consumer_id = is_producer ? 0
3151 : (threadIdx.x - kProducerThreads) / kConsumerThreads;
3152 const int wg_tid = is_producer ? threadIdx.x
3153 : (threadIdx.x - kProducerThreads) % kConsumerThreads;
3154
3155 if (threadIdx.x == 0) {
3156     TmaBarrier::init(&q_full, 1);
3157     for (int cid = 0; cid < kNumConsumers; ++cid) {
3158         for (int s = 0; s < kStagesK; ++s) {
3159             TmaBarrier::init(&k_full[cid][s], 1);
3160             CtaBarrier::init(&k_empty[cid][s], kConsumerThreads);
3161         }
3162         TmaBarrier::init(&v_full[cid], 1);
3163         CtaBarrier::init(&v_empty[cid], kConsumerThreads);
3164     }
3165 }
3166 __syncthreads();
3167
3168 int q_tile = blockIdx.y * (seqlen / kTile) + blockIdx.x;
3169 int kv_tiles = seqlen / kTile;
3170
3171 // =====
3172 // Producer Warpgroup (WG0, threadIdx.x 0~127)
3173 // Only wg_tid == 0 issues TMA. Load Q once, then K-first KV loop.
3174 // Sk=1: load K[tile] + V[tile] per iteration.
3175 // Sk>=2: warmup prefetch K[cid][0], then V[tile] + prefetch K[tile+2].
3176 // =====
3177 if (is_producer) {
3178     if (wg_tid == 0) {
3179         auto mQ = tma_q.get_tma_tensor(make_shape(rows, Int<kHeadDim>{}));
3180         auto gQ = local_tile(
3181             mQ, Shape<64, Int<kHeadDim>>{}, make_coord(q_tile, _0{}));
3182         auto q_slice = tma_q.get_slice(_0{});
3183         auto tQgQ = q_slice.partition_S(gQ);
3184         auto tQsQ = q_slice.partition_D(sQ);
3185         TmaBarrier::arrive_and_expect_tx(
3186             &q_full, sizeof(Element) * size(sQ));

```

```

3187     copy(tma_q.with(q_full), tQgQ, tQsQ);
3188
3189     auto mK = tma_k.get_tma_tensor(make_shape(rows, Int<kHeadDim>{}));
3190     auto mV = tma_v.get_tma_tensor(make_shape(rows, Int<kHeadDim>{}));
3191     auto k_slice = tma_k.get_slice(_0{});
3192     auto v_slice = tma_v.get_slice(_0{});
3193
3194     // Warmup: prefetch first K tile per consumer (Sk>=2 only)
3195     if constexpr (kStagesK > 1) {
3196         for (int cid = 0; cid < kNumConsumers; ++cid) {
3197             if (cid < kv_tiles) {
3198                 auto sK = make_tensor(
3199                     make_smem_ptr(k_base + cid * kStagesK * kTileElements),
3200                     SmemLayout{});
3201                 CtaBarrier::wait(&k_empty[cid][0], 0);
3202                 auto gK = local_tile(
3203                     mK, Shape<_64, Int<kHeadDim>>{},
3204                     make_coord(blockIdx.y * kv_tiles + cid, _0{}));
3205                 auto tKgK = k_slice.partition_S(gK);
3206                 auto tKsK = k_slice.partition_D(sK);
3207                 TmaBarrier::arrive_and_expect_tx(
3208                     &k_full[cid][0], sizeof(Element) * size(sK));
3209                 copy(tma_k.with(k_full[cid][0]), tKgK, tKsK);
3210             }
3211         }
3212     }
3213
3214     for (int tile = 0; tile < kv_tiles; ++tile) {
3215         int cid = tile & 1;
3216         int local = (tile - cid) / kNumConsumers;
3217         int kv_tile = blockIdx.y * kv_tiles + tile;
3218
3219         // Step 1: K — load current (Sk=1) or prefetch next (Sk>=2)
3220         if constexpr (kStagesK == 1) {
3221             auto sK = make_tensor(
3222                 make_smem_ptr(k_base + cid * kStagesK * kTileElements),
3223                 SmemLayout{});
3224             CtaBarrier::wait(&k_empty[cid][0], local & 1);
3225             auto gK = local_tile(
3226                 mK, Shape<_64, Int<kHeadDim>>{}, make_coord(kv_tile, _0{}));
3227             auto tKgK = k_slice.partition_S(gK);
3228             auto tKsK = k_slice.partition_D(sK);
3229             TmaBarrier::arrive_and_expect_tx(
3230                 &k_full[cid][0], sizeof(Element) * size(sK));
3231             copy(tma_k.with(k_full[cid][0]), tKgK, tKsK);
3232         } else {
3233             int stg_next = (local + 1) % kStagesK;
3234             int next_tile = tile + kNumConsumers;
3235             if (next_tile < kv_tiles) {
3236                 int next_local = local + 1;
3237                 int next_phase = (next_local / kStagesK) & 1;
3238                 auto sK = make_tensor(
3239                     make_smem_ptr(k_base +
3240                         (cid * kStagesK + stg_next) * kTileElements),
3241                     SmemLayout{});
3242                 CtaBarrier::wait(&k_empty[cid][stg_next], next_phase);
3243                 auto gK = local_tile(
3244                     mK, Shape<_64, Int<kHeadDim>>{},
3245                     make_coord(blockIdx.y * kv_tiles + next_tile, _0{}));
3246                 auto tKgK = k_slice.partition_S(gK);
3247                 auto tKsK = k_slice.partition_D(sK);
3248                 TmaBarrier::arrive_and_expect_tx(
3249                     &k_full[cid][stg_next], sizeof(Element) * size(sK));

```

```

3250         copy(tma_k.with(k_full[cid][stg_next]), tKgK, tKsK);
3251     }
3252 }
3253
3254 // Step 2: V — single buffer, release depends on PV (late step).
3255 // V TMA latency is hidden by consumer QK+softmax compute.
3256 auto sV = make_tensor(
3257     make_smem_ptr(v_base + cid * kTileElements), SmemLayout{});
3258 CtaBarrier::wait(&v_empty[cid], local & 1);
3259 auto gV = local_tile(
3260     mV, Shape<_64, Int<kHeadDim>>{}, make_coord(kv_tile, _0{}));
3261 auto tVgV = v_slice.partition_S(gV);
3262 auto tVsV = v_slice.partition_D(sV);
3263 TmaBarrier::arrive_and_expect_tx(
3264     &v_full[cid], sizeof(Element) * size(sV));
3265 copy(tma_v.with(v_full[cid]), tVgV, tVsV);
3266 }
3267 }
3268 }
3269 // =====
3270 // Consumer Warpgroups (WG1 consumer_id=0, WG2 consumer_id=1)
3271 // Each WG processes every other KV tile independently.
3272 // K staged (kStagesK depth), V single buffer (always 1).
3273 // Flow: wait K → QK → release K → softmax → wait V → PV → release V → rescale.
3274 // =====
3275 else {
3276     TmaBarrier::wait(&q_full, 0);
3277
3278     auto sV = make_tensor(
3279         make_smem_ptr(v_base + consumer_id * kTileElements), SmemLayout{});
3280     auto sVt = make_tensor(sV.data(), typename Traits::SmemLayoutVt{});
3281     auto sVt_ns = make_tensor(
3282         sV.data(), get_nonswizzle_portion(typename Traits::SmemLayoutVt{}));
3283
3284     typename Traits::TiledMma tiled_mma;
3285     auto thr_mma = tiled_mma.get_thread_slice(wg_tid);
3286     auto tCrQ = thr_mma.partition_fragment_A(sQ);
3287     auto tCrV_layout = thr_mma.partition_fragment_B(sVt_ns).layout();
3288     auto tCr0 = partition_fragment_C(
3289         tiled_mma, Shape<_64, Int<kHeadDim>>{});
3290     clear(tCr0);
3291
3292     auto s2r_copy_q = make_tiled_copy_A(
3293         typename Traits::SmemCopyAtom{}, tiled_mma);
3294     auto s2r_thr_q = s2r_copy_q.get_thread_slice(wg_tid);
3295     auto tQsQ_s2r = s2r_thr_q.partition_S(sQ);
3296     auto s2r_copy_k = make_tiled_copy_B(
3297         typename Traits::SmemCopyAtom{}, tiled_mma);
3298     auto s2r_thr_k = s2r_copy_k.get_thread_slice(wg_tid);
3299     auto s2r_copy_v = make_tiled_copy_B(
3300         typename Traits::SmemCopyAtomTransposed{}, tiled_mma);
3301     auto s2r_thr_v = s2r_copy_v.get_thread_slice(wg_tid);
3302     auto tVsVt_s2r = s2r_thr_v.partition_S(sVt);
3303
3304     auto tCr0_rc = make_tensor(
3305         tCr0.data(),
3306         fa_cute::convert_layout_acc_rowcol(tCr0.layout()));
3307     constexpr int kRows = decltype(size<0>(tCr0_rc))::value;
3308     float row_max[kRows], row_sum[kRows];
3309 #pragma unroll
3310     for (int r = 0; r < kRows; ++r) {
3311         row_max[r] = -INFINITY;
3312         row_sum[r] = 0.0f;

```

```

3313 }
3314
3315 // Init: mark own K/V slots as empty
3316 for (int s = 0; s < kStagesK; ++s)
3317     CtaBarrier::arrive(&k_empty[consumer_id][s]);
3318 CtaBarrier::arrive(&v_empty[consumer_id]);
3319
3320 float scale = rsqrtf(static_cast<float>(kHeadDim));
3321 int local_iter = 0;
3322 for (int tile = consumer_id; tile < kv_tiles;
3323      tile += kNumConsumers, ++local_iter) {
3324     int k_stg = local_iter % kStagesK;
3325     int k_phase = (local_iter / kStagesK) & 1;
3326
3327     // Per-stage K smem and S2R partition
3328     auto sK_stg = make_tensor(
3329         make_smem_ptr(k_base +
3330             (consumer_id * kStagesK + k_stg) * kTileElements),
3331         SmemLayout{});
3332     auto tCrK = thr_mma.partition_fragment_B(sK_stg);
3333     CUTE_STATIC_ASSERT_V(size(tCrK) == size(tCrV_layout));
3334     auto tCrV = make_tensor(tCrK.data(), tCrV_layout);
3335     auto tKsK_s2r = s2r_thr_k.partition_S(sK_stg);
3336
3337     // 1) Wait K, QK gemm, release K early
3338     TmaBarrier::wait(&k_full[consumer_id][k_stg], k_phase);
3339     auto tCrS = partition_fragment_C(tiled_mma, Shape<_64, _64>{});
3340     clear(tCrS);
3341     fa_cute::gemm_ss(
3342         tCrS, tCrQ, tCrK, tQsQ_s2r, tKsK_s2r,
3343         tiled_mma, s2r_copy_q, s2r_copy_k, s2r_thr_q, s2r_thr_k);
3344     { CtaBarrier::arrive(&k_empty[consumer_id][k_stg]); }
3345
3346     // 2) Online softmax (overlaps with producer's V TMA)
3347     auto scores = make_tensor(
3348         tCrS.data(),
3349         fa_cute::convert_layout_acc_rowcol(tCrS.layout()));
3350 #pragma unroll
3351     for (int r = 0; r < kRows; ++r) {
3352         float tile_max = -INFINITY;
3353 #pragma unroll
3354         for (int c = 0; c < size<1>(scores); ++c)
3355             tile_max = fmaxf(tile_max, scores(r, c) * scale);
3356         tile_max = fmaxf(tile_max, __shfl_xor_sync(0xffffffff, tile_max, 1));
3357         tile_max = fmaxf(tile_max, __shfl_xor_sync(0xffffffff, tile_max, 2));
3358         float nxt = fmaxf(row_max[r], tile_max);
3359         float rs = __expf(row_max[r] - nxt);
3360 #pragma unroll
3361         for (int c = 0; c < size<1>(tCrO_rc); ++c)
3362             tCrO_rc(r, c) *= rs;
3363         float ts = 0.0f;
3364 #pragma unroll
3365         for (int c = 0; c < size<1>(scores); ++c) {
3366             float p = __expf(scores(r, c) * scale - nxt);
3367             scores(r, c) = p;
3368             ts += p;
3369         }
3370         ts += __shfl_xor_sync(0xffffffff, ts, 1);
3371         ts += __shfl_xor_sync(0xffffffff, ts, 2);
3372         row_sum[r] = row_sum[r] * rs + ts;
3373         row_max[r] = nxt;
3374     }
3375

```

```

3376 // 3) Wait V, PV gemm, release V early
3377 TmaBarrier::wait(&v_full[consumer_id], local_iter & 1);
3378 auto tCrP = fa_cute::convert_type<Element>(tCrS);
3379 auto tCrPv = make_tensor(
3380     tCrP.data(),
3381     fa_cute::convert_layout_acc_Aregs<typename Traits::TiledMma>(
3382         tCrP.layout()));
3383 fa_cute::gemm_rs(
3384     tCr0, tCrPv, tCrV, tVsVt_s2r,
3385     tiled_mma, s2r_copy_v, s2r_thr_v);
3386 { CtaBarrier::arrive(&v_empty[consumer_id]); }
3387 }
3388
3389 // Split-KV merge
3390 __syncthreads();
3391 float *merge_o = reinterpret_cast<float *>(shm);
3392 float4 *merge_stats = reinterpret_cast<float4 *>(
3393     merge_o + 128 * size(tCr0));
3394
3395 if (consumer_id == 1) {
3396 #pragma unroll
3397     for (int idx = 0; idx < size(tCr0); ++idx)
3398         merge_o[idx * 128 + wg_tid] = tCr0(idx);
3399     merge_stats[wg_tid] = make_float4(
3400         row_max[0], row_max[1], row_sum[0], row_sum[1]);
3401 }
3402 __syncthreads();
3403
3404 if (consumer_id == 0) {
3405     float4 st = merge_stats[wg_tid];
3406     float o_max[kRows] = {st.x, st.y};
3407     float o_sum[kRows] = {st.z, st.w};
3408 #pragma unroll
3409     for (int r = 0; r < kRows; ++r) {
3410         float mg = fmaxf(row_max[r], o_max[r]);
3411         float lhs = __expf(row_max[r] - mg);
3412         float rhs = __expf(o_max[r] - mg);
3413         float ms = lhs * row_sum[r] + rhs * o_sum[r];
3414 #pragma unroll
3415         for (int c = 0; c < size<1>(tCr0_rc); ++c) {
3416             int idx = tCr0_rc.layout()(make_coord(r, c));
3417             float ov = merge_o[idx * 128 + wg_tid];
3418             tCr0_rc(r, c) = (lhs * tCr0_rc(r, c) + rhs * ov) / ms;
3419         }
3420     }
3421
3422     auto tCr0_half = fa_cute::convert_type<Element>(tCr0);
3423     auto m0 = make_tensor(
3424         make_gmem_ptr(output),
3425         make_shape(rows, Int<kHeadDim>{}),
3426         make_stride(Int<kHeadDim>{}, _1{}));
3427     auto g0 = local_tile(
3428         m0, Shape<_64, Int<kHeadDim>>{}, make_coord(q_tile, _0{}));
3429     auto tCg0 = thr_mma.partition_C(g0);
3430     copy(tCr0_half, tCg0);
3431 }
3432 }
3433 }
3434
3435
3436 template <int kHeadDim, typename TmaQ>
3437 __global__ void flash_attn_3_cute_tma_copy_smoke(
3438     CUTLASS_GRID_CONSTANT TmaQ const tma_q,

```

```

3439 cutlass::half_t *output, int rows) {
3440 using namespace cute;
3441 using Traits = fa_cute::FlashAttn3CuTeTraits<kHeadDim>;
3442 using Element = typename Traits::Element;
3443 using SmemLayout = typename Traits::SmemLayoutQKV;
3444 using TransactionBarrier = cutlass::arch::ClusterTransactionBarrier;
3445
3446 extern __shared__ __align__(1024) Element shared_storage[];
3447 auto sQ = make_tensor(make_smem_ptr(shared_storage), SmemLayout{});
3448 __shared__ uint64_t tma_barrier;
3449
3450 if (threadIdx.x == 0) {
3451     TransactionBarrier::init(&tma_barrier, 1);
3452 }
3453 __syncthreads();
3454
3455 auto mQ = tma_q.get_tma_tensor(make_shape(rows, Int<kHeadDim>{}));
3456 auto gQ = local_tile(
3457     mQ, Shape<_64, Int<kHeadDim>>{},
3458     make_coord(blockIdx.x, _0{}));
3459 auto cta_tma_q = tma_q.get_slice(_0{});
3460 auto tQgQ = cta_tma_q.partition_S(gQ);
3461 auto tQsQ = cta_tma_q.partition_D(sQ);
3462
3463 if (threadIdx.x == 0) {
3464     TransactionBarrier::arrive_and_expect_tx(
3465         &tma_barrier, sizeof(half) * size(sQ));
3466     copy(tma_q.with(tma_barrier), tQgQ, tQsQ);
3467 }
3468 TransactionBarrier::wait(&tma_barrier, 0);
3469
3470 for (int linear = threadIdx.x; linear < size(sQ); linear += blockDim.x) {
3471     int row = linear / kHeadDim;
3472     int col = linear % kHeadDim;
3473     output[(blockIdx.x * 64 + row) * kHeadDim + col] = sQ(row, col);
3474 }
3475 }
3476 #endif
3477 #endif // END NOTES_V2_ENABLE_TMA_MMA_WS

```

```

1 // =====
2 // notes-v2.cu — CUDA Kernel 面试背题笔记
3 // =====
4 //
5 // 整理自 LeetCUDA 项目 (https://github.com/xlite-dev/LeetCUDA) , 涵盖:
6 //   - 面试高频 CUDA kernel 的完整实现 (~30 个 kernel)
7 //   - 每类 kernel 附带详细的面试要点注释 (WHY + HOW)
8 //   - 优化技术的递进式讲解 (naive → tiling → vectorize → tensor core → ws)
9 //   - BLAS 语义: N=col-major(Normal), T=row-major(Transposed)
10 //
11 // 10 个 Phase 覆盖:
12 //   Phase 0 — 面试框架速查 (GPU 架构 / Memory Hierarchy / Roofline / 优化清单)
13 //   Phase 1 — 基础原语: Warp Reduce / Block Reduce / Dot Product (含 broadcast 增强版)
14 //   Phase 2 — Elementwise: ReLU / Elementwise Add / Histogram (基础 + float4 向量化 + atomic)
15 //   Phase 3 — Softmax: naive → safe → online + RMS/Layer Norm
16 //   Phase 4 — RoPE: 旋转位置编码 (Llama 风格 theta=10000)
17 //   Phase 5 — Mat Transpose: 基础版 + BCF merge_write 最佳版 (Bank Conflict专题)
18 //   Phase 6 — GEMV: SGEMV K32/K128/K16 (warp-per-row)
19 //   Phase 7 — GEMM *: SGEMM → HGEMM → MMA m16n8k16(TN布局) → WGMMMA m64n128k16
20 //   Phase 8 — FlashAttention-2split_q (FA-2, 含 online softmax + P@V 寄存器复用)
21 //
22 // =====
23 #include "base.cuh"

```

```

24 #include "sgemv.cuh"
25 #include "sgemm.cuh"
26 #include "hgemm.cuh"
27 #include "flash_attn.cuh"
28
29 // =====
30 // 以下是测试代码，验证 Phase 1 - Phase 8 的kernel的正确性，不评估性能。
31 // =====
32
33 // Bench / test mode globals (placed early so all functions can reference them)
34 static bool g_debug = false;
35 static bool g_bench_hgemm = false;
36 static bool g_bench_hgemm_all = false;
37 static bool g_bench_fa = false;
38 static bool g_bench_fa3_cute_only = false;
39 static bool g_bench_all = false;
40 static bool g_fa_skip_check = false;
41 static bool g_swizzle_eq_check = false;
42 enum class FALayout {
43     All, Pad, SwizzleQ, SwizzleK, SwizzleV, SwizzleQK, SwizzleQV, SwizzleKV, Swizzle
44 };
45 static FALayout g_fa_layout = FALayout::Pad;
46 static int g_bench_M = 4096, g_bench_N = 4096, g_bench_K = 4096;
47 static int g_bench_B = 1, g_bench_H = 48, g_bench_Nfa = 8192, g_bench_D = 128;
48 static int g_warmup = 2, g_repeat = 3;
49 static float g_fa_f16_max_tflops = 0.0f;
50 static float g_fa_f32_max_tflops = 0.0f;
51 static bool g_verbose = false;
52
53 // Decide whether to print a FA TFLOPS line. When --verbose/--debug is off,
54 // only print when the current TFLOPS exceeds the running max for its
55 // accumulator category (f16/f32). Correctness failures always print so they
56 // are never silently dropped (callers gate FAIL paths themselves).
57 static bool should_print_fa_tflops(int acc_f32, float tflops) {
58     if (g_verbose || g_debug) return true;
59     float &max_tflops = acc_f32 ? g_fa_f32_max_tflops : g_fa_f16_max_tflops;
60     if (tflops > max_tflops) {
61         max_tflops = tflops;
62         return true;
63     }
64     return false;
65 }
66
67 static bool check_smem_feasible(const void *kernel_func, size_t dyn_smem_bytes) {
68     int device = 0;
69     int max_smem = 0;
70     cudaFuncAttributes attrs{};
71     cudaGetDevice(&device);
72     cudaDeviceGetAttribute(&max_smem, cudaDevAttrMaxSharedMemoryPerBlockOptin, device);
73     cudaFuncGetAttributes(&attrs, kernel_func);
74     return dyn_smem_bytes + attrs.sharedSizeBytes <= (size_t)max_smem;
75 }
76
77 static inline void check(cudaError_t err, const char *msg) {
78     if (err != cudaSuccess) {
79         fprintf(stderr, "[ERROR] %s: %s\n", msg, cudaGetErrorString(err));
80         exit(EXIT_FAILURE);
81     }
82 }
83
84 #if defined(NOTES_V2_ENABLE_CUTE) && defined(NOTES_V2_ENABLE_TMA_MMA_WS)
85 template <int kHeadDim>
86 static void test_flash_attn_3_cute_tma_copy_smoke() {

```

```

87 using namespace cute;
88 using Traits = fa_cute::FlashAttn3CuTeTraits<kHeadDim>;
89 using SmemLayout = typename Traits::SmemLayoutQKV;
90 constexpr int kRows = 128;
91 constexpr int kCount = kRows * kHeadDim;
92
93 half *h_input = (half *)malloc(kCount * sizeof(half));
94 half *h_output = (half *)malloc(kCount * sizeof(half));
95 for (int idx = 0; idx < kCount; ++idx) {
96     h_input[idx] = __float2half((idx % 251) * 0.00390625f);
97 }
98
99 half *d_input;
100 half *d_output;
101 check(cudaMalloc(&d_input, kCount * sizeof(half)), "cute tma smoke alloc input");
102 check(cudaMalloc(&d_output, kCount * sizeof(half)), "cute tma smoke alloc output");
103 check(cudaMemcpy(d_input, h_input, kCount * sizeof(half), cudaMemcpyHostToDevice),
104     "cute tma smoke H2D");
105
106 auto mQ = make_tensor(
107     make_gmem_ptr(reinterpret_cast<cutlass::half_t *>(d_input)),
108     make_shape(kRows, Int<kHeadDim>{}),
109     make_stride(Int<kHeadDim>{}, _1{}));
110 auto tma_q = make_tma_copy(
111     SM90_TMA_LOAD{}, mQ, SmemLayout{},
112     Shape<_64, Int<kHeadDim>>{}, _1{});
113 auto kernel = flash_attn_3_cute_tma_copy_smoke<kHeadDim, decltype(tma_q)>;
114 constexpr int kSmemBytes = cosize(SmemLayout{}) * sizeof(cutlass::half_t);
115 check(cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
116     kSmemBytes),
117     "cute tma smoke set smem");
118 kernel<<<2, 128, kSmemBytes>>>(&tma_q, reinterpret_cast<cutlass::half_t *>(d_output), kRows);
119 check(cudaGetLastError(), "cute tma smoke launch");
120 check(cudaDeviceSynchronize(), "cute tma smoke sync");
121 check(cudaMemcpy(h_output, d_output, kCount * sizeof(half), cudaMemcpyDeviceToHost),
122     "cute tma smoke D2H");
123
124
125 float max_err = 0.0f;
126 for (int idx = 0; idx < kCount; ++idx) {
127     max_err = max(max_err, fabsf(__half2float(h_input[idx]) -
128         __half2float(h_output[idx])));
129 }
130 char label[64];
131 snprintf(label, sizeof(label), "CuTe TMA copy smoke (D=%d)", kHeadDim);
132 printf("| %-56s | %.3e |\n", label, max_err);
133
134 free(h_input);
135 free(h_output);
136 cudaFree(d_input);
137 cudaFree(d_output);
138 }
139
140 template <int kHeadDim>
141 static void test_flash_attn_3_tma_mma_ws_split_q_cute() {
142     using namespace cute;
143     using Traits = fa_cute::FlashAttn3CuTeTraits<kHeadDim>;
144     using SmemLayout = typename Traits::SmemLayoutQKV;
145     constexpr int kSeqLen = 128;
146     constexpr int kCount = kSeqLen * kHeadDim;
147
148     half *h_q = (half *)malloc(kCount * sizeof(half));
149     half *h_k = (half *)malloc(kCount * sizeof(half));

```

```

150  half *h_v = (half *)malloc(kCount * sizeof(half));
151  half *h_o = (half *)malloc(kCount * sizeof(half));
152  float *ref_o = (float *)malloc(kCount * sizeof(float));
153  srand(42 + kHeadDim);
154  for (int idx = 0; idx < kCount; ++idx) {
155      h_q[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
156      h_k[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
157      h_v[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
158  }
159
160  float scale = 1.0f / sqrtf((float)kHeadDim);
161  for (int row = 0; row < kSeqlen; ++row) {
162      float scores[kSeqlen];
163      float row_max = -INFINITY;
164      for (int col = 0; col < kSeqlen; ++col) {
165          float score = 0.0f;
166          for (int dim = 0; dim < kHeadDim; ++dim) {
167              score += __half2float(h_q[row * kHeadDim + dim]) *
168                      __half2float(h_k[col * kHeadDim + dim]);
169          }
170          scores[col] = score * scale;
171          row_max = max(row_max, scores[col]);
172      }
173      float row_sum = 0.0f;
174      for (int col = 0; col < kSeqlen; ++col) {
175          scores[col] = expf(scores[col] - row_max);
176          row_sum += scores[col];
177      }
178      for (int dim = 0; dim < kHeadDim; ++dim) {
179          float output = 0.0f;
180          for (int col = 0; col < kSeqlen; ++col) {
181              output += scores[col] * __half2float(h_v[col * kHeadDim + dim]);
182          }
183          ref_o[row * kHeadDim + dim] = output / row_sum;
184      }
185  }
186
187  half *d_q;
188  half *d_k;
189  half *d_v;
190  half *d_o;
191  check(cudaMalloc(&d_q, kCount * sizeof(half)), "cute fa3 alloc Q");
192  check(cudaMalloc(&d_k, kCount * sizeof(half)), "cute fa3 alloc K");
193  check(cudaMalloc(&d_v, kCount * sizeof(half)), "cute fa3 alloc V");
194  check(cudaMalloc(&d_o, kCount * sizeof(half)), "cute fa3 alloc O");
195  check(cudaMemcpy(d_q, h_q, kCount * sizeof(half), cudaMemcpyHostToDevice),
196        "cute fa3 H2D Q");
197  check(cudaMemcpy(d_k, h_k, kCount * sizeof(half), cudaMemcpyHostToDevice),
198        "cute fa3 H2D K");
199  check(cudaMemcpy(d_v, h_v, kCount * sizeof(half), cudaMemcpyHostToDevice),
200        "cute fa3 H2D V");
201
202  auto make_tma = [](half *pointer) {
203      auto tensor = make_tensor(
204          make_gmem_ptr(reinterpret_cast<cutlass::half_t *>(pointer)),
205          make_shape(kSeqlen, Int<kHeadDim>{}),
206          make_stride(Int<kHeadDim>{}, _1{}));
207      return make_tma_copy(
208          SM90_TMA_LOAD{}, tensor, SmemLayout{},
209          Shape<_64, Int<kHeadDim>>{}, _1{});
210  };
211  auto tma_q = make_tma(d_q);
212  auto tma_k = make_tma(d_k);

```

```

213 auto tma_v = make_tma(d_v);
214 auto kernel = flash_attn_3_tma_mma_ws_split_q_cute<
215     kHeadDim, decltpe(tma_q), decltpe(tma_k), decltpe(tma_v)>;
216 auto acc_o = partition_fragment_C(
217     typename Traits::TiledMma{}, Shape<_64, Int<kHeadDim>>{});
218 constexpr int kNumConsumers = 2;
219 constexpr int kStagesK = 1;
220 constexpr int kTiles = 1 + kNumConsumers * kStagesK + kNumConsumers;
221 constexpr int kTilesBytes =
222     kTiles * cosize(SmemLayout{}) * sizeof(cutlass::half_t);
223 int merge_bytes = 128 * size(acc_o) * sizeof(float) + 128 * sizeof(float4);
224 int smem_bytes = max(kTilesBytes, merge_bytes);
225 check(cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
226     smem_bytes),
227     "cute fa3 set smem");
228 kernel<<<dim3(2, 1), 384, smem_bytes>>>(
229     tma_q, tma_k, tma_v,
230     reinterpret_cast<cutlass::half_t*>(d_o), kSeqlen, kSeqlen);
231 check(cudaGetLastError(), "cute fa3 launch");
232 check(cudaDeviceSynchronize(), "cute fa3 sync");
233 check(cudaMemcpy(h_o, d_o, kCount * sizeof(half), cudaMemcpyDeviceToHost),
234     "cute fa3 D2H");
235
236 float max_err = 0.0f;
237 for (int idx = 0; idx < kCount; ++idx) {
238     max_err = max(max_err, fabsf(__half2float(h_o[idx]) - ref_o[idx]));
239 }
240 char label[80];
241 snprintf(label, sizeof(label), "FA3 CuTe TMA MMA WS (2 Consumer WG) (Sk=1, Sv=1, F32Acc, D=%d)", kHeadDim);
242 printf("| %-56s | %.3e |\n", label, max_err);
243
244 free(h_q);
245 free(h_k);
246 free(h_v);
247 free(h_o);
248 free(ref_o);
249 cudaFree(d_q);
250 cudaFree(d_k);
251 cudaFree(d_v);
252 cudaFree(d_o);
253 }
254 #endif
255
256 #if defined(NOTES_V2_ENABLE_CUTE) && defined(NOTES_V2_ENABLE_TMA_MMA_WS)
257 // FA2-style CuTe cp.async test: single consumer, Br=128, no TMA/WS.
258 // kSeqlen=256 -> 2 Q tiles, covers multi-Q-tile path.
259 // kStagesK=2 验证 cp.async pipeline + V 单 buffer + V/K group 策略正确性。
260 template <int kHeadDim>
261 static void test_flash_attn_mma_stages_split_q_cute() {
262     using namespace cute;
263     using Traits = fa_cute::FlashAttn2CuTeTraits<kHeadDim>;
264     using SmemLayoutQ = typename Traits::SmemLayoutQ;
265     using SmemLayoutKV = typename Traits::SmemLayoutKV;
266     constexpr int kBr = 128;
267     constexpr int kStagesK = 2;
268     constexpr int kSeqlen = 256;
269     constexpr int kCount = kSeqlen * kHeadDim;
270
271     half *h_q = (half *)malloc(kCount * sizeof(half));
272     half *h_k = (half *)malloc(kCount * sizeof(half));
273     half *h_v = (half *)malloc(kCount * sizeof(half));
274     half *h_o = (half *)malloc(kCount * sizeof(half));

```

```

275 float *ref_o = (float *)malloc(kCount * sizeof(float));
276 srand(42 + kHeadDim);
277 for (int idx = 0; idx < kCount; ++idx) {
278     h_q[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
279     h_k[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
280     h_v[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
281 }
282
283 // CPU FP32 naive attention reference
284 float scale = 1.0f / sqrtf((float)kHeadDim);
285 for (int row = 0; row < kSeqLen; ++row) {
286     float scores[kSeqLen];
287     float row_max = -INFINITY;
288     for (int col = 0; col < kSeqLen; ++col) {
289         float score = 0.0f;
290         for (int dim = 0; dim < kHeadDim; ++dim) {
291             score += __half2float(h_q[row * kHeadDim + dim]) *
292                     __half2float(h_k[col * kHeadDim + dim]);
293         }
294         scores[col] = score * scale;
295         row_max = max(row_max, scores[col]);
296     }
297     float row_sum = 0.0f;
298     for (int col = 0; col < kSeqLen; ++col) {
299         scores[col] = expf(scores[col] - row_max);
300         row_sum += scores[col];
301     }
302     for (int dim = 0; dim < kHeadDim; ++dim) {
303         float output = 0.0f;
304         for (int col = 0; col < kSeqLen; ++col) {
305             output += scores[col] * __half2float(h_v[col * kHeadDim + dim]);
306         }
307         ref_o[row * kHeadDim + dim] = output / row_sum;
308     }
309 }
310
311 half *d_q;
312 half *d_k;
313 half *d_v;
314 half *d_o;
315 check(cudaMalloc(&d_q, kCount * sizeof(half)), "cute fa2 cpasync alloc Q");
316 check(cudaMalloc(&d_k, kCount * sizeof(half)), "cute fa2 cpasync alloc K");
317 check(cudaMalloc(&d_v, kCount * sizeof(half)), "cute fa2 cpasync alloc V");
318 check(cudaMalloc(&d_o, kCount * sizeof(half)), "cute fa2 cpasync alloc O");
319 check(cudaMemcpy(d_q, h_q, kCount * sizeof(half), cudaMemcpyHostToDevice),
320        "cute fa2 cpasync H2D Q");
321 check(cudaMemcpy(d_k, h_k, kCount * sizeof(half), cudaMemcpyHostToDevice),
322        "cute fa2 cpasync H2D K");
323 check(cudaMemcpy(d_v, h_v, kCount * sizeof(half), cudaMemcpyHostToDevice),
324        "cute fa2 cpasync H2D V");
325
326 // 无需 TMA descriptor, 直接传指针
327 auto kernel = flash_attn_mma_stages_split_q_cute<kHeadDim, kStagesK>;
328 // smem = Q[128*D] + K[Sk*64*D] + V[1*64*D]
329 constexpr int kSmemBytes =
330     (size(SmemLayoutQ{}) +
331      kStagesK * size(SmemLayoutKV{}) +
332      1 * size(SmemLayoutKV{})) * sizeof(cutlass::half_t);
333 check(cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
334                             kSmemBytes),
335        "cute fa2 cpasync set smem");
336 kernel<<<dim3(kSeqLen / kBr, 1), 256, kSmemBytes>>>(
337     reinterpret_cast<cutlass::half_t*>(d_q),

```

```

338     reinterpret_cast<cutlass::half_t *>(d_k),
339     reinterpret_cast<cutlass::half_t *>(d_v),
340     reinterpret_cast<cutlass::half_t *>(d_o), kSeqlen, kSeqlen);
341 check(cudaGetLastError(), "cute fa2 cpasync launch");
342 check(cudaDeviceSynchronize(), "cute fa2 cpasync sync");
343 check(cudaMemcpy(h_o, d_o, kCount * sizeof(half), cudaMemcpyDeviceToHost),
344        "cute fa2 cpasync D2H");
345
346 float max_err = 0.0f;
347 for (int idx = 0; idx < kCount; ++idx) {
348     max_err = max(max_err, fabsf(__half2float(h_o[idx]) - ref_o[idx]));
349 }
350 char label[96];
351 snprintf(label, sizeof(label),
352          "FA2 CuTe MMA Stages (Sk=%d, F32Acc, D=%d)", kStagesK, kHeadDim);
353 printf("| %-56s | %.3e |\n", label, max_err);
354
355 free(h_q);
356 free(h_k);
357 free(h_v);
358 free(h_o);
359 free(ref_o);
360 cudaFree(d_q);
361 cudaFree(d_k);
362 cudaFree(d_v);
363 cudaFree(d_o);
364 }
365 #endif
366
367 #if defined(NOTES_V2_ENABLE_CUTE) && defined(NOTES_V2_ENABLE_TMA_MMA_WS)
368 // FA2-style CuTe test: single consumer, Br=128, no merge.
369 // kSeqlen=256 -> 2 Q tiles, covers multi-Q-tile path.
370 template <int kHeadDim>
371 static void test_flash_attn_tma_mma_ws_split_q_cute() {
372     using namespace cute;
373     using Traits = fa_cute::FlashAttn2CuTeTraits<kHeadDim>;
374     using SmemLayoutQ = typename Traits::SmemLayoutQ;
375     using SmemLayoutKV = typename Traits::SmemLayoutKV;
376     constexpr int kBr = 128;
377     constexpr int kSeqlen = 256;
378     constexpr int kCount = kSeqlen * kHeadDim;
379
380     half *h_q = (half *)malloc(kCount * sizeof(half));
381     half *h_k = (half *)malloc(kCount * sizeof(half));
382     half *h_v = (half *)malloc(kCount * sizeof(half));
383     half *h_o = (half *)malloc(kCount * sizeof(half));
384     float *ref_o = (float *)malloc(kCount * sizeof(float));
385     srand(42 + kHeadDim);
386     for (int idx = 0; idx < kCount; ++idx) {
387         h_q[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
388         h_k[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
389         h_v[idx] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
390     }
391
392     float scale = 1.0f / sqrtf((float)kHeadDim);
393     for (int row = 0; row < kSeqlen; ++row) {
394         float scores[kSeqlen];
395         float row_max = -INFINITY;
396         for (int col = 0; col < kSeqlen; ++col) {
397             float score = 0.0f;
398             for (int dim = 0; dim < kHeadDim; ++dim) {
399                 score += __half2float(h_q[row * kHeadDim + dim]) *
400                     __half2float(h_k[col * kHeadDim + dim]);

```

```

401     }
402     scores[col] = score * scale;
403     row_max = max(row_max, scores[col]);
404 }
405 float row_sum = 0.0f;
406 for (int col = 0; col < kSeqlen; ++col) {
407     scores[col] = expf(scores[col] - row_max);
408     row_sum += scores[col];
409 }
410 for (int dim = 0; dim < kHeadDim; ++dim) {
411     float output = 0.0f;
412     for (int col = 0; col < kSeqlen; ++col) {
413         output += scores[col] * __half2float(h_v[col * kHeadDim + dim]);
414     }
415     ref_o[row * kHeadDim + dim] = output / row_sum;
416 }
417 }
418
419 half *d_q;
420 half *d_k;
421 half *d_v;
422 half *d_o;
423 check(cudaMalloc(&d_q, kCount * sizeof(half)), "cute fa2 alloc Q");
424 check(cudaMalloc(&d_k, kCount * sizeof(half)), "cute fa2 alloc K");
425 check(cudaMalloc(&d_v, kCount * sizeof(half)), "cute fa2 alloc V");
426 check(cudaMalloc(&d_o, kCount * sizeof(half)), "cute fa2 alloc O");
427 check(cudaMemcpy(d_q, h_q, kCount * sizeof(half), cudaMemcpyHostToDevice),
428         "cute fa2 H2D Q");
429 check(cudaMemcpy(d_k, h_k, kCount * sizeof(half), cudaMemcpyHostToDevice),
430         "cute fa2 H2D K");
431 check(cudaMemcpy(d_v, h_v, kCount * sizeof(half), cudaMemcpyHostToDevice),
432         "cute fa2 H2D V");
433
434 // Q tile: [128, D], K/V tile: [64, D] (both use GMMMA::Layout_K_SW128_Atom)
435 auto make_tma_q = [=]() {
436     auto tensor = make_tensor(
437         make_gmem_ptr(reinterpret_cast<cutlass::half_t *>(d_q)),
438         make_shape(kSeqlen, Int<kHeadDim>{}),
439         make_stride(Int<kHeadDim>{}, _1{}));
440     return make_tma_copy(
441         SM90_TMA_LOAD{}, tensor, SmemLayoutQ{},
442         Shape<_128, Int<kHeadDim>>{}, _1{});
443 };
444 auto make_tma_kv = [=](half *pointer) {
445     auto tensor = make_tensor(
446         make_gmem_ptr(reinterpret_cast<cutlass::half_t *>(pointer)),
447         make_shape(kSeqlen, Int<kHeadDim>{}),
448         make_stride(Int<kHeadDim>{}, _1{}));
449     return make_tma_copy(
450         SM90_TMA_LOAD{}, tensor, SmemLayoutKV{},
451         Shape<_64, Int<kHeadDim>>{}, _1{});
452 };
453 auto tma_q = make_tma_q();
454 auto tma_k = make_tma_kv(d_k);
455 auto tma_v = make_tma_kv(d_v);
456 auto kernel = flash_attn_tma_mma_ws_split_q_cute<
457     kHeadDim, decltype(tma_q), decltype(tma_k), decltype(tma_v)>;
458 // smem = Q[128*D] + K[Sk*64*D] + V[Sv*64*D] (Sk=1, Sv=1 default)
459 constexpr int kSmemBytes =
460     (cosize(SmemLayoutQ{}) +
461      1 * cosize(SmemLayoutKV{}) +
462      1 * cosize(SmemLayoutKV{})) * sizeof(cutlass::half_t);
463 check(cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,

```

```

464         kSmemBytes),
465         "cute fa2 set smem");
466 kernel<<<dim3(kSeqlen / kBr, 1), 384, kSmemBytes>>>(
467     tma_q, tma_k, tma_v,
468     reinterpret_cast<cutlass::half_t *>(d_o), kSeqlen, kSeqlen);
469 check(cudaGetLastError(), "cute fa2 launch");
470 check(cudaDeviceSynchronize(), "cute fa2 sync");
471 check(cudaMemcpy(h_o, d_o, kCount * sizeof(half), cudaMemcpyDeviceToHost),
472     "cute fa2 D2H");
473
474 float max_err = 0.0f;
475 for (int idx = 0; idx < kCount; ++idx) {
476     max_err = max(max_err, fabsf(__half2float(h_o[idx]) - ref_o[idx]));
477 }
478 char label[80];
479 snprintf(label, sizeof(label), "FA2 CuTe TMA MMA WS (1 Consumer WG) (Sk=1, Sv=1, F32Acc, D=%d)", kHeadDim);
480 printf("| %-56s | %.3e |\n", label, max_err);
481
482 free(h_q);
483 free(h_k);
484 free(h_v);
485 free(h_o);
486 free(ref_o);
487 cudaFree(d_q);
488 cudaFree(d_k);
489 cudaFree(d_v);
490 cudaFree(d_o);
491 }
492 #endif
493
494 static void test_block_reduce(int N) {
495
496     srand(42);
497     float *h_a = (float *)malloc((size_t)N * sizeof(float));
498     for (int i = 0; i < N; i++)
499         h_a[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
500
501     // CPU reference: sum of all elements
502     double ref = 0.0;
503     for (int i = 0; i < N; i++) ref += (double)h_a[i];
504
505     float *d_a, *d_y;
506     check(cudaMalloc(&d_a, (size_t)N * sizeof(float)), "blockreduce alloc A");
507     check(cudaMalloc(&d_y, sizeof(float)), "blockreduce alloc Y");
508
509     check(cudaMemcpy(d_a, h_a, (size_t)N * sizeof(float), cudaMemcpyHostToDevice), "blockreduce H2D A");
510     check(cudaMemset(d_y, 0, sizeof(float)), "blockreduce zero Y");
511
512     dim3 block(128);
513     dim3 grid((N + 127) / 128);
514     block_reduce_all<128><<<grid, block>>>(d_a, d_y, N);
515     check(cudaGetLastError(), "blockreduce launch");
516     check(cudaDeviceSynchronize(), "blockreduce sync");
517
518     float result;
519     check(cudaMemcpy(&result, d_y, sizeof(float), cudaMemcpyDeviceToHost), "blockreduce D2H");
520
521     float err = fabsf(result - (float)ref);
522     printf("| %-56s | %.3e |\n", "BlockReduce", err);
523
524     free(h_a);
525     cudaFree(d_a); cudaFree(d_y);

```

```

526 }
527
528
529 static void test_dot(int N) {
530
531     srand(42);
532     float *h_a = (float *)malloc((size_t)N * sizeof(float));
533     float *h_b = (float *)malloc((size_t)N * sizeof(float));
534     for (int i = 0; i < N; i++) {
535         h_a[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
536         h_b[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
537     }
538
539     // CPU reference
540     double ref = 0.0;
541     for (int i = 0; i < N; i++) ref += (double)h_a[i] * (double)h_b[i];
542
543     float *d_a, *d_b, *d_y;
544     check(cudaMalloc(&d_a, (size_t)N * sizeof(float)), "dot alloc A");
545     check(cudaMalloc(&d_b, (size_t)N * sizeof(float)), "dot alloc B");
546     check(cudaMalloc(&d_y, sizeof(float)), "dot alloc Y");
547
548     check(cudaMemcpy(d_a, h_a, (size_t)N * sizeof(float), cudaMemcpyHostToDevice), "dot H2D A");
549     check(cudaMemcpy(d_b, h_b, (size_t)N * sizeof(float), cudaMemcpyHostToDevice), "dot H2D B");
550     check(cudaMemset(d_y, 0, sizeof(float)), "dot zero Y");
551
552     dim3 block(128);
553     dim3 grid((N + 127) / 128);
554     dot<128><<<grid, block>>>(d_a, d_b, d_y, N);
555     check(cudaGetLastError(), "dot launch");
556     check(cudaDeviceSynchronize(), "dot sync");
557
558     float result;
559     check(cudaMemcpy(&result, d_y, sizeof(float), cudaMemcpyDeviceToHost), "dot D2H");
560
561     float err = fabsf(result - (float)ref);
562     printf("| %-56s | %.3e |\n", "Dot", err);
563
564     // ---- Dot Vec4 ----
565     check(cudaMemset(d_y, 0, sizeof(float)), "dot_vec4 zero Y");
566     dim3 block_v4(32);
567     dot_vec4<32><<<grid, block_v4>>>(d_a, d_b, d_y, N);
568     check(cudaGetLastError(), "dot_vec4 launch");
569     check(cudaDeviceSynchronize(), "dot_vec4 sync");
570
571     check(cudaMemcpy(&result, d_y, sizeof(float), cudaMemcpyDeviceToHost), "dot_vec4 D2H");
572     float err_v4 = fabsf(result - (float)ref);
573     printf("| %-56s | %.3e |\n", "Dot-Vec4", err_v4);
574
575     free(h_a); free(h_b);
576     cudaFree(d_a); cudaFree(d_b); cudaFree(d_y);
577 }
578
579
580 static void test_relu(int N) {
581     srand(42);
582     float *h_x = (float *)malloc((size_t)N * sizeof(float));
583     float *h_y = (float *)malloc((size_t)N * sizeof(float));
584     for (int i = 0; i < N; i++)
585         h_x[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
586
587     float *d_x, *d_y;
588     check(cudaMalloc(&d_x, (size_t)N * sizeof(float)), "relu alloc X");

```

```

589 check(cudaMalloc(&d_y, (size_t)N * sizeof(float)), "relu alloc Y");
590 check(cudaMemcpy(d_x, h_x, (size_t)N * sizeof(float), cudaMemcpyHostToDevice), "relu H2D");
591
592 for (int i = 0; i < N; i++) h_y[i] = fmaxf(0.0f, h_x[i]);
593 dim3 block256(256);
594 dim3 grid256((N + 255) / 256);
595 relu<<<grid256, block256>>>(d_x, d_y, N);
596 check(cudaGetLastError(), "relu launch");
597 check(cudaDeviceSynchronize(), "relu sync");
598 check(cudaMemcpy(h_y, d_y, (size_t)N * sizeof(float), cudaMemcpyDeviceToHost), "relu D2H");
599 float max_err = 0.0f;
600 for (int i = 0; i < N; i++) {
601     float expected = fmaxf(0.0f, h_x[i]);
602     float err = fabsf(h_y[i] - expected);
603     if (err > max_err) max_err = err;
604 }
605 printf("| %-56s | %.3e |\n", "ReLU", max_err);
606
607 dim3 block64(64);
608 relu_vec4<<<grid256, block64>>>(d_x, d_y, N);
609 check(cudaGetLastError(), "relu_vec4 launch");
610 check(cudaDeviceSynchronize(), "relu_vec4 sync");
611 check(cudaMemcpy(h_y, d_y, (size_t)N * sizeof(float), cudaMemcpyDeviceToHost), "relu_vec4 D2H");
612 max_err = 0.0f;
613 for (int i = 0; i < N; i++) {
614     float expected = fmaxf(0.0f, h_x[i]);
615     float err = fabsf(h_y[i] - expected);
616     if (err > max_err) max_err = err;
617 }
618 printf("| %-56s | %.3e |\n", "ReLU-Vec4", max_err);
619
620 free(h_x); free(h_y);
621 cudaFree(d_x); cudaFree(d_y);
622 }
623
624
625 static void test_elementwise(int N) {
626     srand(42);
627     float *h_a = (float *)malloc((size_t)N * sizeof(float));
628     float *h_b = (float *)malloc((size_t)N * sizeof(float));
629     for (int i = 0; i < N; i++) {
630         h_a[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
631         h_b[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
632     }
633
634     float *d_a, *d_b, *d_c;
635     check(cudaMalloc(&d_a, (size_t)N * sizeof(float)), "eadd alloc A");
636     check(cudaMalloc(&d_b, (size_t)N * sizeof(float)), "eadd alloc B");
637     check(cudaMalloc(&d_c, (size_t)N * sizeof(float)), "eadd alloc C");
638     check(cudaMemcpy(d_a, h_a, (size_t)N * sizeof(float), cudaMemcpyHostToDevice), "eadd H2D A");
639     check(cudaMemcpy(d_b, h_b, (size_t)N * sizeof(float), cudaMemcpyHostToDevice), "eadd H2D B");
640
641     dim3 block256(256);
642     dim3 grid256((N + 255) / 256);
643     elementwise_add<<<grid256, block256>>>(d_a, d_b, d_c, N);
644     check(cudaGetLastError(), "eadd launch");
645     check(cudaDeviceSynchronize(), "eadd sync");
646     float *h_c = (float *)malloc((size_t)N * sizeof(float));
647     check(cudaMemcpy(h_c, d_c, (size_t)N * sizeof(float), cudaMemcpyDeviceToHost), "eadd D2H");
648     float max_err = 0.0f;
649     for (int i = 0; i < N; i++) {
650         float err = fabsf(h_c[i] - (h_a[i] + h_b[i]));
651         if (err > max_err) max_err = err;

```

```

652 }
653 printf("| %-56s | %.3e |\n", "ElemwiseAdd", max_err);
654
655 dim3 block64(64);
656 check(cudaMemset(d_c, 0, (size_t)N * sizeof(float)), "eadd_vec4 zero C");
657 elementwise_add_vec4<<<grid256, block64>>>(d_a, d_b, d_c, N);
658 check(cudaGetLastError(), "eadd_vec4 launch");
659 check(cudaDeviceSynchronize(), "eadd_vec4 sync");
660 check(cudaMemcpy(h_c, d_c, (size_t)N * sizeof(float), cudaMemcpyDeviceToHost), "eadd_vec4 D2H");
661 max_err = 0.0f;
662 for (int i = 0; i < N; i++) {
663     float err = fabsf(h_c[i] - (h_a[i] + h_b[i]));
664     if (err > max_err) max_err = err;
665 }
666 printf("| %-56s | %.3e |\n", "ElemwiseAdd-Vec4", max_err);
667
668 free(h_a); free(h_b); free(h_c);
669 cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
670 }
671
672
673 static void test_histogram(int N) {
674     srand(42);
675     int BINS = 16;
676     int *h_hist = (int *)calloc(BINS, sizeof(int));
677     int *h_hist_ref = (int *)calloc(BINS, sizeof(int));
678     int *h_idx = (int *)malloc((size_t)N * sizeof(int));
679     for (int i = 0; i < N; i++) h_idx[i] = rand() % BINS;
680     for (int i = 0; i < N; i++) h_hist_ref[h_idx[i]]++;
681
682     int *d_idx, *d_hist;
683     check(cudaMalloc(&d_idx, (size_t)N * sizeof(int)), "hist alloc idx");
684     check(cudaMalloc(&d_hist, BINS * sizeof(int)), "hist alloc hist");
685     check(cudaMemcpy(d_idx, h_idx, (size_t)N * sizeof(int), cudaMemcpyHostToDevice), "hist H2D idx");
686     check(cudaMemset(d_hist, 0, BINS * sizeof(int)), "hist zero");
687
688     dim3 block256(256);
689     dim3 grid256((N + 255) / 256);
690     histogram<<<grid256, block256>>>(d_idx, d_hist, N);
691     check(cudaGetLastError(), "histogram launch");
692     check(cudaDeviceSynchronize(), "histogram sync");
693     check(cudaMemcpy(h_hist, d_hist, BINS * sizeof(int), cudaMemcpyDeviceToHost), "hist D2H");
694
695     float max_err = 0.0f;
696     for (int i = 0; i < BINS; i++) {
697         float err = fabsf((float)(h_hist[i] - h_hist_ref[i]));
698         if (err > max_err) max_err = err;
699     }
700     printf("| %-56s | %.3e |\n", "Histogram", max_err);
701
702     free(h_hist); free(h_hist_ref); free(h_idx);
703     cudaFree(d_idx); cudaFree(d_hist);
704 }
705
706
707 static void test_merge_attn_states(int num_tokens, int num_heads,
708                                   int head_size) {
709
710     size_t out_size = (size_t)num_tokens * num_heads * head_size * sizeof(float);
711     size_t lse_size = (size_t)num_heads * num_tokens * sizeof(float);
712
713     float *h_prefix_out = (float *)malloc(out_size);
714     float *h_suffix_out = (float *)malloc(out_size);

```

```

715 float *h_prefix_lse = (float *)malloc(lse_size);
716 float *h_suffix_lse = (float *)malloc(lse_size);
717 float *h_output_ref = (float *)malloc(out_size);
718
719 srand(42);
720 for (int i = 0; i < num_tokens * num_heads * head_size; i++) {
721     h_prefix_out[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
722     h_suffix_out[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
723 }
724 for (int i = 0; i < num_heads * num_tokens; i++) {
725     h_prefix_lse[i] = ((float)rand() / RAND_MAX) * 20.0f - 10.0f;
726     h_suffix_lse[i] = ((float)rand() / RAND_MAX) * 20.0f - 10.0f;
727 }
728
729 // CPU reference: 与 kernel 完全相同的浮点计算
730 for (int t = 0; t < num_tokens; t++) {
731     for (int h = 0; h < num_heads; h++) {
732         float p_lse = h_prefix_lse[h * num_tokens + t];
733         float s_lse = h_suffix_lse[h * num_tokens + t];
734         p_lse = isinf(p_lse) ? -INFINITY : p_lse;
735         s_lse = isinf(s_lse) ? -INFINITY : s_lse;
736
737         float max_lse = fmaxf(p_lse, s_lse);
738         p_lse -= max_lse;
739         s_lse -= max_lse;
740         float p_se = expf(p_lse);
741         float s_se = expf(s_lse);
742         float p_scale = p_se / (p_se + s_se);
743         float s_scale = s_se / (p_se + s_se);
744
745         int head_off = t * num_heads * head_size + h * head_size;
746         for (int d = 0; d < head_size; d++) {
747             h_output_ref[head_off + d] =
748                 h_prefix_out[head_off + d] * p_scale +
749                 h_suffix_out[head_off + d] * s_scale;
750         }
751     }
752 }
753
754 float *d_prefix_out, *d_suffix_out, *d_prefix_lse, *d_suffix_lse, *d_output;
755 check(cudaMalloc(&d_prefix_out, out_size), "merge_attn alloc p_out");
756 check(cudaMalloc(&d_suffix_out, out_size), "merge_attn alloc s_out");
757 check(cudaMalloc(&d_prefix_lse, lse_size), "merge_attn alloc p_lse");
758 check(cudaMalloc(&d_suffix_lse, lse_size), "merge_attn alloc s_lse");
759 check(cudaMalloc(&d_output, out_size), "merge_attn alloc output");
760
761 check(cudaMemcpy(d_prefix_out, h_prefix_out, out_size,
762                 cudaMemcpyHostToDevice),
763        "merge_attn H2D p_out");
764 check(cudaMemcpy(d_suffix_out, h_suffix_out, out_size,
765                 cudaMemcpyHostToDevice),
766        "merge_attn H2D s_out");
767 check(cudaMemcpy(d_prefix_lse, h_prefix_lse, lse_size,
768                 cudaMemcpyHostToDevice),
769        "merge_attn H2D p_lse");
770 check(cudaMemcpy(d_suffix_lse, h_suffix_lse, lse_size,
771                 cudaMemcpyHostToDevice),
772        "merge_attn H2D s_lse");
773
774 int threads_per_head = head_size / 4;
775 int total_threads = num_tokens * num_heads * threads_per_head;
776 dim3 block(128);
777 dim3 grid((total_threads + 127) / 128);

```

```

778 merge_attn_states<<<grid, block>>>(  

779     d_output, d_prefix_out, d_prefix_lse, d_suffix_out, d_suffix_lse,  

780     num_tokens, num_heads, head_size);  

781 check(cudaGetLastError(), "merge_attn launch");  

782 check(cudaDeviceSynchronize(), "merge_attn sync");  

783  

784 float *h_output = (float *)malloc(out_size);  

785 check(cudaMemcpy(h_output, d_output, out_size, cudaMemcpyDeviceToHost),  

786     "merge_attn D2H");  

787  

788 float max_err = 0.0f;  

789 for (int i = 0; i < num_tokens * num_heads * head_size; i++) {  

790     float err = fabsf(h_output[i] - h_output_ref[i]);  

791     if (err > max_err) max_err = err;  

792 }  

793 printf("| %-56s | %.3e |\n", "MergeAttnStates", max_err);  

794  

795 // 边界测试: +inf LSE → 权重退化为 0 (空 attention 段)  

796 h_prefix_lse[0] = INFINITY; // head 0, token 0 的 LSE = +inf  

797 h_suffix_lse[0] = 0.0f;  

798 check(cudaMemcpy(d_prefix_lse, h_prefix_lse, lse_size,  

799     cudaMemcpyHostToDevice),  

800     "merge_attn H2D inf lse");  

801 merge_attn_states<<<grid, block>>>(  

802     d_output, d_prefix_out, d_prefix_lse, d_suffix_out, d_suffix_lse,  

803     num_tokens, num_heads, head_size);  

804 check(cudaGetLastError(), "merge_attn inf launch");  

805 check(cudaDeviceSynchronize(), "merge_attn inf sync");  

806 check(cudaMemcpy(h_output, d_output, out_size, cudaMemcpyDeviceToHost),  

807     "merge_attn inf D2H");  

808  

809 // token 0, head 0 的所有元素应等于 suffix_output (prefix 权重  $\alpha=0$ )  

810 float inf_err = 0.0f;  

811 for (int d = 0; d < head_size; d++) {  

812     float err = fabsf(h_output[d] - h_suffix_out[d]);  

813     if (err > inf_err) inf_err = err;  

814 }  

815 printf("| %-56s | %.3e |\n", "MergeAttnStates-inf", inf_err);  

816  

817 free(h_prefix_out);  

818 free(h_suffix_out);  

819 free(h_prefix_lse);  

820 free(h_suffix_lse);  

821 free(h_output_ref);  

822 free(h_output);  

823 cudaFree(d_prefix_out);  

824 cudaFree(d_suffix_out);  

825 cudaFree(d_prefix_lse);  

826 cudaFree(d_suffix_lse);  

827 cudaFree(d_output);  

828 }  

829  

830  

831 static void test_softmax(int N) {  

832     // Use N = blockDim.x so one block processes all elements as one token.  

833     constexpr int kNumThreads = 256;  

834     if (N != kNumThreads) {  

835         printf("  OnlineSafeSoftmax: N must be %d (got %d), skipping.\n", kNumThreads, N);  

836         return;  

837     }  

838  

839     srand(42);  

840     float *h_x = (float *)malloc((size_t)N * sizeof(float));

```

```

841 float *h_y_ref = (float *)malloc((size_t)N * sizeof(float));
842 for (int i = 0; i < N; i++)
843     h_x[i] = ((float)rand() / RAND_MAX) * 10.0f - 5.0f; // [-5, 5]
844
845 // CPU reference: softmax(x_i) = exp(x_i - max) / sum(exp(x_j - max))
846 float max_val = -FLT_MAX;
847 for (int i = 0; i < N; i++) if (h_x[i] > max_val) max_val = h_x[i];
848 double sum_exp = 0.0;
849 for (int i = 0; i < N; i++) sum_exp += (double)expf(h_x[i] - max_val);
850 for (int i = 0; i < N; i++)
851     h_y_ref[i] = expf(h_x[i] - max_val) / (float)sum_exp;
852
853 float *d_x, *d_y;
854 check(cudaMalloc(&d_x, (size_t)N * sizeof(float)), "softmax alloc X");
855 check(cudaMalloc(&d_y, (size_t)N * sizeof(float)), "softmax alloc Y");
856
857 check(cudaMemcpy(d_x, h_x, (size_t)N * sizeof(float), cudaMemcpyHostToDevice), "softmax H2D X");
858
859 dim3 block(256);
860 dim3 grid(1); // one token covering all N elements
861 online_safe_softmax_per_token<<<grid, block>>>(d_x, d_y, N);
862 check(cudaGetLastError(), "softmax launch");
863 check(cudaDeviceSynchronize(), "softmax sync");
864
865 float *h_y = (float *)malloc((size_t)N * sizeof(float));
866 check(cudaMemcpy(h_y, d_y, (size_t)N * sizeof(float), cudaMemcpyDeviceToHost), "softmax D2H");
867
868 float max_err = 0.0f;
869 for (int i = 0; i < N; i++) {
870     float err = fabsf(h_y[i] - h_y_ref[i]);
871     if (err > max_err) max_err = err;
872 }
873 printf("| %-56s | %.3e |\n", "OnlineSafeSoftmax", max_err);
874
875 // ---- Safe Softmax ----
876 safe_softmax_per_token<<<grid, block>>>(d_x, d_y, N);
877 check(cudaGetLastError(), "safe_softmax launch");
878 check(cudaDeviceSynchronize(), "safe_softmax sync");
879 check(cudaMemcpy(h_y, d_y, (size_t)N * sizeof(float), cudaMemcpyDeviceToHost), "safe_softmax D2H");
880 max_err = 0.0f;
881 for (int i = 0; i < N; i++) {
882     float err = fabsf(h_y[i] - h_y_ref[i]);
883     if (err > max_err) max_err = err;
884 }
885 printf("| %-56s | %.3e |\n", "SafeSoftmax", max_err);
886
887 // ---- Naive Softmax ----
888 softmax_per_token<<<grid, block>>>(d_x, d_y, N);
889 check(cudaGetLastError(), "naive_softmax launch");
890 check(cudaDeviceSynchronize(), "naive_softmax sync");
891 check(cudaMemcpy(h_y, d_y, (size_t)N * sizeof(float), cudaMemcpyDeviceToHost), "naive_softmax D2H");
892 max_err = 0.0f;
893 for (int i = 0; i < N; i++) {
894     float err = fabsf(h_y[i] - h_y_ref[i]);
895     if (err > max_err) max_err = err;
896 }
897 printf("| %-56s | %.3e |\n", "NaiveSoftmax", max_err);
898
899 free(h_x); free(h_y); free(h_y_ref);
900 cudaFree(d_x); cudaFree(d_y);
901 }
902
903

```

```

904 static void test_rms_norm(int N, int K) {
905
906     srand(42);
907     float *h_x = (float *)malloc((size_t)N * K * sizeof(float));
908     float *h_y_ref = (float *)malloc((size_t)N * K * sizeof(float));
909     for (int i = 0; i < N * K; i++)
910         h_x[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
911     float g = 1.5f; // gain
912
913     // CPU reference: y = (x / rms(x)) * g
914     float epsilon = 1e-5f;
915     for (int n = 0; n < N; n++) {
916         double sum_sq = 0.0;
917         for (int k = 0; k < K; k++) sum_sq += (double)h_x[n * K + k] * (double)h_x[n * K + k];
918         float rms = sqrtf((float)sum_sq / (float)K + epsilon);
919         for (int k = 0; k < K; k++)
920             h_y_ref[n * K + k] = (h_x[n * K + k] / rms) * g;
921     }
922
923     float *d_x, *d_y;
924     check(cudaMalloc(&d_x, (size_t)N * K * sizeof(float)), "rmsnorm alloc X");
925     check(cudaMalloc(&d_y, (size_t)N * K * sizeof(float)), "rmsnorm alloc Y");
926     check(cudaMemcpy(d_x, h_x, (size_t)N * K * sizeof(float), cudaMemcpyHostToDevice), "rmsnorm H2D X");
927
928     dim3 block(128);
929     dim3 grid(N);
930     rms_norm<<<grid, block>>>(d_x, d_y, g, N, K);
931     check(cudaGetLastError(), "rmsnorm launch");
932     check(cudaDeviceSynchronize(), "rmsnorm sync");
933
934     float *h_y = (float *)malloc((size_t)N * K * sizeof(float));
935     check(cudaMemcpy(h_y, d_y, (size_t)N * K * sizeof(float), cudaMemcpyDeviceToHost), "rmsnorm D2H");
936
937     float max_err = 0.0f;
938     for (int i = 0; i < N * K; i++) {
939         float err = fabsf(h_y[i] - h_y_ref[i]);
940         if (err > max_err) max_err = err;
941     }
942     printf("| %-56s | %.3e |\n", "RMSNorm", max_err);
943
944     // ---- RMS Norm Vec4 ----
945     dim3 block_rv4(32);
946     rms_norm_vec4<<<grid, block_rv4>>>(d_x, d_y, g, N, K);
947     check(cudaGetLastError(), "rmsnorm_vec4 launch");
948     check(cudaDeviceSynchronize(), "rmsnorm_vec4 sync");
949     check(cudaMemcpy(h_y, d_y, (size_t)N * K * sizeof(float), cudaMemcpyDeviceToHost), "rmsnorm_vec4 D2H");
950     max_err = 0.0f;
951     for (int i = 0; i < N * K; i++) {
952         float err = fabsf(h_y[i] - h_y_ref[i]);
953         if (err > max_err) max_err = err;
954     }
955     printf("| %-56s | %.3e |\n", "RMSNorm-Vec4", max_err);
956
957     free(h_x); free(h_y); free(h_y_ref);
958     cudaFree(d_x); cudaFree(d_y);
959 }
960
961
962 static void test_layer_norm(int N, int K) {
963
964     srand(42);
965     float *h_x = (float *)malloc((size_t)N * K * sizeof(float));
966     float *h_y_ref = (float *)malloc((size_t)N * K * sizeof(float));

```

```

967 for (int i = 0; i < N * K; i++)
968     h_x[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
969 float g = 1.5f, b = 0.3f; // gain and bias
970
971 // CPU reference: y = ((x - mean) / std) * g + b
972 float epsilon = 1e-5f;
973 for (int n = 0; n < N; n++) {
974     double sum = 0.0;
975     for (int k = 0; k < K; k++) sum += (double)h_x[n * K + k];
976     float mean = (float)sum / (float)K;
977     double sum_sq = 0.0;
978     for (int k = 0; k < K; k++) {
979         float diff = h_x[n * K + k] - mean;
980         sum_sq += (double)diff * (double)diff;
981     }
982     float std = sqrtf((float)sum_sq / (float)K + epsilon);
983     for (int k = 0; k < K; k++)
984         h_y_ref[n * K + k] = ((h_x[n * K + k] - mean) / std) * g + b;
985 }
986
987 float *d_x, *d_y;
988 check(cudaMalloc(&d_x, (size_t)N * K * sizeof(float)), "layernorm alloc X");
989 check(cudaMalloc(&d_y, (size_t)N * K * sizeof(float)), "layernorm alloc Y");
990 check(cudaMemcpy(d_x, h_x, (size_t)N * K * sizeof(float), cudaMemcpyHostToDevice), "layernorm H2D X");
991
992 dim3 block(128);
993 dim3 grid(N);
994 layer_norm<<<grid, block>>>(d_x, d_y, g, b, N, K);
995 check(cudaGetLastError(), "layernorm launch");
996 check(cudaDeviceSynchronize(), "layernorm sync");
997
998 float *h_y = (float *)malloc((size_t)N * K * sizeof(float));
999 check(cudaMemcpy(h_y, d_y, (size_t)N * K * sizeof(float), cudaMemcpyDeviceToHost), "layernorm D2H");
1000
1001 float max_err = 0.0f;
1002 for (int i = 0; i < N * K; i++) {
1003     float err = fabsf(h_y[i] - h_y_ref[i]);
1004     if (err > max_err) max_err = err;
1005 }
1006 printf("| %-56s | %.3e |\n", "LayerNorm", max_err);
1007
1008 // ---- Layer Norm Vec4 ----
1009 dim3 block_lv4(32);
1010 layer_norm_vec4<<<grid, block_lv4>>>(d_x, d_y, g, b, N, K);
1011 check(cudaGetLastError(), "layernorm_vec4 launch");
1012 check(cudaDeviceSynchronize(), "layernorm_vec4 sync");
1013 check(cudaMemcpy(h_y, d_y, (size_t)N * K * sizeof(float), cudaMemcpyDeviceToHost), "layernorm_vec4 D2H");
1014 max_err = 0.0f;
1015 for (int i = 0; i < N * K; i++) {
1016     float err = fabsf(h_y[i] - h_y_ref[i]);
1017     if (err > max_err) max_err = err;
1018 }
1019 printf("| %-56s | %.3e |\n", "LayerNorm-Vec4", max_err);
1020
1021 free(h_x); free(h_y); free(h_y_ref);
1022 cudaFree(d_x); cudaFree(d_y);
1023 }
1024
1025
1026 static void test_rope(int seq_len, int N) {
1027
1028     int total_pairs = seq_len * N;
1029     int total_elems = total_pairs * 2;

```

```

1030 size_t size = (size_t)total_elems * sizeof(float);
1031
1032 float *h_x = (float *)malloc(size);
1033 float *h_y_ref = (float *)malloc(size);
1034
1035 srand(42);
1036 for (int i = 0; i < total_elems; i++)
1037     h_x[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
1038
1039 // CPU reference: 2D rotation for each pair
1040 for (int idx = 0; idx < total_pairs; idx++) {
1041     int token_pos = idx / N;
1042     int token_idx = idx % N;
1043     float x1 = h_x[idx * 2];
1044     float x2 = h_x[idx * 2 + 1];
1045     float theta = 1.0f / powf(10000.0f, 2.0f * token_idx / (N * 2.0f));
1046     float angle = (float)token_pos * theta;
1047     float cos_v = cosf(angle);
1048     float sin_v = sinf(angle);
1049     h_y_ref[idx * 2] = x1 * cos_v - x2 * sin_v;
1050     h_y_ref[idx * 2 + 1] = x1 * sin_v + x2 * cos_v;
1051 }
1052
1053 float *d_x, *d_y;
1054 check(cudaMalloc(&d_x, size), "rope alloc X");
1055 check(cudaMalloc(&d_y, size), "rope alloc Y");
1056 check(cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice), "rope H2D");
1057
1058 dim3 block(256);
1059 dim3 grid((total_pairs + 255) / 256);
1060 rope<<<grid, block>>>(d_x, d_y, seq_len, N);
1061 check(cudaGetLastError(), "rope launch");
1062 check(cudaDeviceSynchronize(), "rope sync");
1063
1064 float *h_y = (float *)malloc(size);
1065 check(cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost), "rope D2H");
1066
1067 float max_err = 0.0f;
1068 for (int i = 0; i < total_elems; i++) {
1069     float err = fabsf(h_y[i] - h_y_ref[i]);
1070     if (err > max_err) max_err = err;
1071 }
1072 printf("| %-56s | %.3e |\n", "RoPE", max_err);
1073
1074 free(h_x);
1075 free(h_y);
1076 free(h_y_ref);
1077 cudaFree(d_x);
1078 cudaFree(d_y);
1079 }
1080
1081
1082 static void test_mat_transpose(int row, int col) {
1083
1084     size_t size_in = (size_t)row * col * sizeof(float);
1085     size_t size_out = (size_t)col * row * sizeof(float);
1086
1087     float *h_x = (float *)malloc(size_in);
1088     float *h_y_ref = (float *)malloc(size_out);
1089
1090     srand(42);
1091     for (int i = 0; i < row * col; i++)
1092         h_x[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;

```

```

1093 // CPU reference: y[j][i] = x[i][j] (row-major)
1094 for (int i = 0; i < row; i++)
1095     for (int j = 0; j < col; j++)
1096         h_y_ref[j * row + i] = h_x[i * col + j];
1097
1098 float *d_x, *d_y;
1099 check(cudaMalloc(&d_x, size_in), "mattrans alloc X");
1100 check(cudaMalloc(&d_y, size_out), "mattrans alloc Y");
1101 check(cudaMemcpy(d_x, h_x, size_in, cudaMemcpyHostToDevice), "mattrans H2D");
1102
1103 dim3 block(16, 16);
1104 dim3 grid((col + 15) / 16, (row + 15) / 16);
1105 mat_transpose<<<grid, block>>>(d_x, d_y, row, col);
1106 check(cudaGetLastError(), "mattrans launch");
1107 check(cudaDeviceSynchronize(), "mattrans sync");
1108
1109 float *h_y = (float *)malloc(size_out);
1110 check(cudaMemcpy(h_y, d_y, size_out, cudaMemcpyDeviceToHost), "mattrans D2H");
1111
1112 float max_err = 0.0f;
1113 for (int i = 0; i < col * row; i++) {
1114     float err = fabsf(h_y[i] - h_y_ref[i]);
1115     if (err > max_err) max_err = err;
1116 }
1117 printf("| %-56s | %.3e |\n", "MatTranspose", max_err);
1118
1119 free(h_x);
1120 free(h_y);
1121 free(h_y_ref);
1122 cudaFree(d_x);
1123 cudaFree(d_y);
1124 }
1125
1126
1127 static void test_mat_transpose_padded(int row, int col) {
1128
1129     size_t size_in = (size_t)row * col * sizeof(float);
1130     size_t size_out = (size_t)col * row * sizeof(float);
1131
1132     float *h_x = (float *)malloc(size_in);
1133     float *h_y_ref = (float *)malloc(size_out);
1134
1135     srand(42);
1136     for (int i = 0; i < row * col; i++)
1137         h_x[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
1138
1139     // CPU reference: y[j][i] = x[i][j] (row-major)
1140     for (int i = 0; i < row; i++)
1141         for (int j = 0; j < col; j++)
1142             h_y_ref[j * row + i] = h_x[i * col + j];
1143
1144     float *d_x, *d_y;
1145     check(cudaMalloc(&d_x, size_in), "mattrans_padded alloc X");
1146     check(cudaMalloc(&d_y, size_out), "mattrans_padded alloc Y");
1147     check(cudaMemcpy(d_x, h_x, size_in, cudaMemcpyHostToDevice), "mattrans_padded H2D");
1148
1149     dim3 block(16, 16);
1150     dim3 grid((col + 15) / 16, (row + 63) / 64);
1151     mat_transpose_padded<<<grid, block>>>(d_x, d_y, row, col);
1152     check(cudaGetLastError(), "mattrans_padded launch");
1153     check(cudaDeviceSynchronize(), "mattrans_padded sync");
1154
1155

```

```

1156 float *h_y = (float *)malloc(size_out);
1157 check(cudaMemcpy(h_y, d_y, size_out, cudaMemcpyDeviceToHost), "mattrans_padded D2H");
1158
1159 float max_err = 0.0f;
1160 for (int i = 0; i < col * row; i++) {
1161     float err = fabsf(h_y[i] - h_y_ref[i]);
1162     if (err > max_err) max_err = err;
1163 }
1164 printf("| %-56s | %.3e |\n", "MatTransposePadded", max_err);
1165
1166 free(h_x);
1167 free(h_y);
1168 free(h_y_ref);
1169 cudaFree(d_x);
1170 cudaFree(d_y);
1171 }
1172
1173
1174 static void test_sgemv(int M, int K) {
1175
1176     srand(42);
1177     float *h_a = (float *)malloc((size_t)M * K * sizeof(float));
1178     float *h_x = (float *)malloc((size_t)K * sizeof(float));
1179     float *h_y_ref = (float *)malloc((size_t)M * sizeof(float));
1180     for (int i = 0; i < M * K; i++) h_a[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
1181     for (int i = 0; i < K; i++) h_x[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
1182
1183     // CPU reference: y[m] = sum_k A[m,k] * x[k]
1184     for (int m = 0; m < M; m++) {
1185         double sum = 0.0;
1186         for (int k = 0; k < K; k++) sum += (double)h_a[m * K + k] * (double)h_x[k];
1187         h_y_ref[m] = (float)sum;
1188     }
1189
1190     float *d_a, *d_x, *d_y;
1191     check(cudaMalloc(&d_a, (size_t)M * K * sizeof(float)), "sgemv alloc A");
1192     check(cudaMalloc(&d_x, (size_t)K * sizeof(float)), "sgemv alloc X");
1193     check(cudaMalloc(&d_y, (size_t)M * sizeof(float)), "sgemv alloc Y");
1194
1195     check(cudaMemcpy(d_a, h_a, (size_t)M * K * sizeof(float), cudaMemcpyHostToDevice), "sgemv H2D A");
1196     check(cudaMemcpy(d_x, h_x, (size_t)K * sizeof(float), cudaMemcpyHostToDevice), "sgemv H2D X");
1197
1198     dim3 block(32, 4);
1199     dim3 grid((M + 3) / 4);
1200     sgemv_k128<<<grid, block>>>(d_a, d_x, d_y, M, K);
1201     check(cudaGetLastError(), "sgemv launch");
1202     check(cudaDeviceSynchronize(), "sgemv sync");
1203
1204     float *h_y = (float *)malloc((size_t)M * sizeof(float));
1205     check(cudaMemcpy(h_y, d_y, (size_t)M * sizeof(float), cudaMemcpyDeviceToHost), "sgemv D2H");
1206
1207     float max_err = 0.0f;
1208     for (int m = 0; m < M; m++) {
1209         float err = fabsf(h_y[m] - h_y_ref[m]);
1210         if (err > max_err) max_err = err;
1211     }
1212     printf("| %-56s | %.3e |\n", "SGEMV-K128", max_err);
1213
1214     // ---- SGEMV K32 ----
1215     check(cudaMemset(d_y, 0, M * sizeof(float)), "sgemv_k32 zero Y");
1216     sgemv_k32<<<grid, block>>>(d_a, d_x, d_y, M, K);
1217     check(cudaGetLastError(), "sgemv_k32 launch");
1218     check(cudaDeviceSynchronize(), "sgemv_k32 sync");

```

```

1219
1220 check(cudaMemcpy(h_y, d_y, (size_t)M * sizeof(float), cudaMemcpyDeviceToHost), "sgemv_k32 D2H");
1221
1222 max_err = 0.0f;
1223 for (int m = 0; m < M; m++) {
1224     float err = fabsf(h_y[m] - h_y_ref[m]);
1225     if (err > max_err) max_err = err;
1226 }
1227 printf("| %-56s | %.3e |\n", "SGEMV-K32", max_err);
1228
1229 // ---- SGEMV K16 ----
1230 free(h_a); free(h_x); free(h_y); free(h_y_ref);
1231 cudaFree(d_a); cudaFree(d_x); cudaFree(d_y);
1232
1233 int K16 = 16;
1234 h_a = (float *)malloc((size_t)M * K16 * sizeof(float));
1235 h_x = (float *)malloc((size_t)K16 * sizeof(float));
1236 h_y_ref = (float *)malloc((size_t)M * sizeof(float));
1237 for (int i = 0; i < M * K16; i++) h_a[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
1238 for (int i = 0; i < K16; i++) h_x[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
1239
1240 for (int m = 0; m < M; m++) {
1241     double sum = 0.0;
1242     for (int k = 0; k < K16; k++) sum += (double)h_a[m * K16 + k] * (double)h_x[k];
1243     h_y_ref[m] = (float)sum;
1244 }
1245
1246 check(cudaMalloc(&d_a, (size_t)M * K16 * sizeof(float)), "sgemv_k16 alloc A");
1247 check(cudaMalloc(&d_x, (size_t)K16 * sizeof(float)), "sgemv_k16 alloc X");
1248 check(cudaMalloc(&d_y, (size_t)M * sizeof(float)), "sgemv_k16 alloc Y");
1249
1250 check(cudaMemcpy(d_a, h_a, (size_t)M * K16 * sizeof(float), cudaMemcpyHostToDevice), "sgemv_k16 H2D A");
1251 check(cudaMemcpy(d_x, h_x, (size_t)K16 * sizeof(float), cudaMemcpyHostToDevice), "sgemv_k16 H2D X");
1252
1253 dim3 grid_k16((M + 7) / 8);
1254 sgemv_k16<2><<<grid_k16, block>>>(d_a, d_x, d_y, M, K16);
1255 check(cudaGetLastError(), "sgemv_k16 launch");
1256 check(cudaDeviceSynchronize(), "sgemv_k16 sync");
1257
1258 h_y = (float *)malloc((size_t)M * sizeof(float));
1259 check(cudaMemcpy(h_y, d_y, (size_t)M * sizeof(float), cudaMemcpyDeviceToHost), "sgemv_k16 D2H");
1260
1261 max_err = 0.0f;
1262 for (int m = 0; m < M; m++) {
1263     float err = fabsf(h_y[m] - h_y_ref[m]);
1264     if (err > max_err) max_err = err;
1265 }
1266 printf("| %-56s | %.3e |\n", "SGEMV-K16", max_err);
1267
1268 free(h_a); free(h_x); free(h_y); free(h_y_ref);
1269 cudaFree(d_a); cudaFree(d_x); cudaFree(d_y);
1270 }
1271
1272
1273 static void test_sgemm(int M, int N, int K) {
1274
1275     size_t size_a = (size_t)M * K * sizeof(float);
1276     size_t size_b = (size_t)K * N * sizeof(float);
1277     size_t size_c = (size_t)M * N * sizeof(float);
1278
1279     float *h_a = (float *)malloc(size_a);
1280     float *h_b = (float *)malloc(size_b);
1281     float *h_c_ref = (float *)malloc(size_c);

```

```

1282 srand(42);
1283 for (int i = 0; i < M * K; i++) h_a[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
1284 for (int i = 0; i < K * N; i++) h_b[i] = ((float)rand() / RAND_MAX) * 2.0f - 1.0f;
1285
1286 float *d_a, *d_b, *d_c;
1287 check(cudaMalloc(&d_a, size_a), "sgemm alloc A");
1288 check(cudaMalloc(&d_b, size_b), "sgemm alloc B");
1289 check(cudaMalloc(&d_c, size_c), "sgemm alloc C");
1290
1291
1292 check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "sgemm H2D A");
1293 check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "sgemm H2D B");
1294
1295 // cuBLAS reference (row-major idiom: swap M/N, swap A/B)
1296 cublasHandle_t handle;
1297 cublasCreate(&handle);
1298 float alpha = 1.0f, beta = 0.0f;
1299 cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K,
1300             &alpha, d_b, N, d_a, K, &beta, d_c, N);
1301 check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost), "sgemm D2H ref");
1302
1303 // Kernel
1304 cudaEvent_t start, stop;
1305 cudaEventCreate(&start);
1306 cudaEventCreate(&stop);
1307
1308 dim3 block(32, 32);
1309 dim3 grid((N + 31) / 32, (M + 31) / 32);
1310 sgemm<<<grid, block>>>(d_a, d_b, d_c, M, N, K);
1311 check(cudaGetLastError(), "sgemm launch");
1312 check(cudaDeviceSynchronize(), "sgemm sync");
1313
1314 float *h_c = (float *)malloc(size_c);
1315 check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "sgemm D2H");
1316
1317 // Verify
1318 float max_err = 0.0f;
1319 for (int i = 0; i < M * N; i++) {
1320     float err = fabsf(h_c[i] - h_c_ref[i]);
1321     if (err > max_err) max_err = err;
1322 }
1323 printf("| %-56s | %.3e |\n", "SGEMM", max_err);
1324
1325 // ---- SGEMM Vec4 (128x128 tile, 4x4 thread tile) ----
1326
1327 dim3 grid_vec4((N + 127) / 128, (M + 127) / 128);
1328 check(cudaMemset(d_c, 0, size_c), "sgemm_vec4 zero C");
1329 sgemm_vec4<<<grid_vec4, block>>>(d_a, d_b, d_c, M, N, K);
1330 check(cudaGetLastError(), "sgemm_vec4 launch");
1331 check(cudaDeviceSynchronize(), "sgemm_vec4 sync");
1332
1333 check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "sgemm_vec4 D2H");
1334
1335 max_err = 0.0f;
1336 for (int i = 0; i < M * N; i++) {
1337     float err = fabsf(h_c[i] - h_c_ref[i]);
1338     if (err > max_err) max_err = err;
1339 }
1340 printf("| %-56s | %.3e |\n", "SGEMM-Vec4", max_err);
1341
1342 free(h_a); free(h_b); free(h_c); free(h_c_ref);
1343 cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
1344 cublasDestroy(handle);

```

```

1345     cudaEventDestroy(start);
1346     cudaEventDestroy(stop);
1347 }
1348
1349
1350 static void test_hgemm_mma(int M, int N, int K) {
1351
1352     size_t size_a = (size_t)M * K * sizeof(half);
1353     size_t size_b = (size_t)K * N * sizeof(half);
1354     size_t size_c = (size_t)M * N * sizeof(half);
1355
1356     half *h_a = (half *)malloc(size_a);
1357     half *h_b = (half *)malloc(size_b);
1358     half *h_c_ref = (half *)malloc(size_c);
1359
1360     srand(42);
1361     for (int i = 0; i < M * K; i++) h_a[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1362     for (int i = 0; i < K * N; i++) h_b[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1363
1364     // Kernel expects B^T [N×K] row-major layout (TN layout convention).
1365     // We store h_b as B [K×N] row-major for cuBLAS, then create B^T for kernel.
1366     size_t size_b_t = (size_t)N * K * sizeof(half);
1367     half *h_b_t = (half *)malloc(size_b_t);
1368     for (int n = 0; n < N; n++)
1369         for (int k = 0; k < K; k++)
1370             h_b_t[n * K + k] = h_b[k * N + n];
1371
1372     half *d_a, *d_b, *d_b_t, *d_c;
1373     check(cudaMalloc(&d_a, size_a), "hgemm alloc A");
1374     check(cudaMalloc(&d_b, size_b), "hgemm alloc B (cuBLAS)");
1375     check(cudaMalloc(&d_b_t, size_b_t), "hgemm alloc B_t (kernel)");
1376     check(cudaMalloc(&d_c, size_c), "hgemm alloc C");
1377
1378     check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "hgemm H2D A");
1379     check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "hgemm H2D B (cuBLAS)");
1380     check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice), "hgemm H2D B_t (kernel)");
1381
1382     // cuBLAS FP16 reference (row-major idiom: swap M/N, swap A/B)
1383     // Note: use CUBLAS_COMPUTE_16F to match the kernel's f16.f16.f16.f16 accumulation.
1384     cublasHandle_t handle;
1385     cublasCreate(&handle);
1386     half alpha_h = __float2half(1.0f), beta_h = __float2half(0.0f);
1387     cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K,
1388                 &alpha_h, d_b, CUDA_R_16F, N, d_a, CUDA_R_16F, K,
1389                 &beta_h, d_c, CUDA_R_16F, N,
1390                 CUBLAS_COMPUTE_16F, CUBLAS_GEMM_DEFAULT);
1391     check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost), "hgemm D2H ref");
1392
1393     // MMA kernel (TN layout: A row-major, B^T row-major = B col-major)
1394     cudaEvent_t start, stop;
1395     cudaEventCreate(&start);
1396     cudaEventCreate(&stop);
1397
1398     constexpr int BM = 128, BN = 128, BK = 16, kStages = 3;
1399     size_t smem_bytes = kStages * (BM * BK + BN * BK) * sizeof(half); // 24576
1400     dim3 block(256);
1401     dim3 grid((N + BN - 1) / BN, (M + BM - 1) / BM);
1402     hgemm_mma_stages_tn<<<grid, block, smem_bytes>>>(d_a, d_b_t, d_c, M, N, K);
1403     check(cudaGetLastError(), "hgemm launch");
1404     check(cudaDeviceSynchronize(), "hgemm sync");
1405
1406     half *h_c = (half *)malloc(size_c);
1407     check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "hgemm D2H");

```

```

1408
1409 // Verify
1410 float max_err = 0.0f;
1411 for (int i = 0; i < M * N; i++) {
1412     float err = fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i]));
1413     if (err > max_err) max_err = err;
1414 }
1415 printf("| %-56s | %.3e |\n", "HGEMM MMA", max_err);
1416
1417 free(h_a); free(h_b); free(h_b_t); free(h_c); free(h_c_ref);
1418 cudaFree(d_a); cudaFree(d_b); cudaFree(d_b_t); cudaFree(d_c);
1419 cublasDestroy(handle);
1420 cudaEventDestroy(start);
1421 cudaEventDestroy(stop);
1422 }
1423
1424
1425 static void test_hgemm_swizzle(int M, int N, int K) {
1426     // HGEMM MMA Swizzle — m16n8k16 + multistage pipeline + TN 布局 + XOR swizzle
1427     // + Register Double Buffering (kValTileK=4, BK=64)
1428     // TN layout: C[M×N] = A[M×K] × BT[N×K]
1429     // Kernel: hgemm_mma_stages_tn_swizzle with default template params
1430     // (kValTileK=4, kStages=2, BK=64)
1431     // smem: kStages × (BM×BK + BN×BK) halves
1432
1433     size_t size_a = (size_t)M * K * sizeof(half);
1434     size_t size_b = (size_t)K * N * sizeof(half);
1435     size_t size_c = (size_t)M * N * sizeof(half);
1436
1437     half *h_a = (half *)malloc(size_a);
1438     half *h_b = (half *)malloc(size_b);
1439     half *h_c_ref = (half *)malloc(size_c);
1440
1441     srand(42);
1442     for (int i = 0; i < M * K; i++) h_a[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1443     for (int i = 0; i < K * N; i++) h_b[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1444
1445     // Kernel expects BT [N×K] row-major layout (TN layout convention).
1446     size_t size_b_t = (size_t)N * K * sizeof(half);
1447     half *h_b_t = (half *)malloc(size_b_t);
1448     for (int n = 0; n < N; n++)
1449         for (int k = 0; k < K; k++)
1450             h_b_t[n * K + k] = h_b[k * N + n];
1451
1452     half *d_a, *d_b, *d_b_t, *d_c;
1453     check(cudaMalloc(&d_a, size_a), "hgemm_swizzle alloc A");
1454     check(cudaMalloc(&d_b, size_b), "hgemm_swizzle alloc B (cuBLAS)");
1455     check(cudaMalloc(&d_b_t, size_b_t), "hgemm_swizzle alloc B_t (kernel)");
1456     check(cudaMalloc(&d_c, size_c), "hgemm_swizzle alloc C");
1457
1458     check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "hgemm_swizzle H2D A");
1459     check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "hgemm_swizzle H2D B (cuBLAS)");
1460     check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice), "hgemm_swizzle H2D B_t (kernel)");
1461
1462     // cuBLAS FP16 reference
1463     cublasHandle_t handle;
1464     cublasCreate(&handle);
1465     half alpha_h = __float2half(1.0f), beta_h = __float2half(0.0f);
1466     cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K,
1467                 &alpha_h, d_b, CUDA_R_16F, N, d_a, CUDA_R_16F, K,
1468                 &beta_h, d_c, CUDA_R_16F, N,
1469                 CUBLAS_COMPUTE_16F, CUBLAS_GEMM_DEFAULT);
1470     check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost), "hgemm_swizzle D2H ref");

```

```

1471 // MMA swizzle kernel (default params: kStages=2, kValTileK=4, BK=64)
1472 constexpr int BM = 128, BN = 128, BK = 64, K_STAGE_S = 2;
1473 size_t smem_bytes = K_STAGE_S * (BM * BK + BN * BK) * sizeof(half);
1474 cudaFuncSetAttribute(
1475     (const void *)hgemm_mma_stages_tn_swizzle<16, 8, 16, 2, 4, 4, 4, 4, K_STAGE_S, 0>,
1476     cudaFuncAttributeMaxDynamicSharedMemorySize, smem_bytes);
1477 dim3 block(256);
1478 dim3 grid((N + BN - 1) / BN, (M + BM - 1) / BM);
1479 hgemm_mma_stages_tn_swizzle<<<grid, block, smem_bytes>>>(d_a, d_b_t, d_c, M, N, K);
1480 check(cudaGetLastError(), "hgemm_swizzle launch");
1481 check(cudaDeviceSynchronize(), "hgemm_swizzle sync");
1482
1483 half *h_c = (half *)malloc(size_c);
1484 check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "hgemm_swizzle D2H");
1485
1486 // Verify
1487 float max_err = 0.0f;
1488 for (int i = 0; i < M * N; i++) {
1489     float err = fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i]));
1490     if (err > max_err) max_err = err;
1491 }
1492 printf("| %-56s | %.3e |\n", "HGEMM Swizzle + Reg2x", max_err);
1493
1494 free(h_a); free(h_b); free(h_b_t); free(h_c); free(h_c_ref);
1495 cudaFree(d_a); cudaFree(d_b); cudaFree(d_b_t); cudaFree(d_c);
1496 cublasDestroy(handle);
1497 }
1498
1499
1500
1501 #if defined(NOTES_V2_ENABLE_CUTE)
1502 static void test_hgemm_cute(int M, int N, int K) {
1503     // HGEMM CuTe — SM80_16x8x16_F16F16F16F16_TN + Swizzle<3,3,3> + kStage=2
1504     // TN layout: C[MxN] = A[MxK] × B^T[NxK]
1505     // Kernel: hgemm_mma_stages_tn_cute via launch_hgemm_mma_stages_tn_cute
1506     // Tile: BM=128, BN=256, BK=32, 128 threads/block
1507
1508     size_t size_a = (size_t)M * K * sizeof(half);
1509     size_t size_b = (size_t)K * N * sizeof(half);
1510     size_t size_c = (size_t)M * N * sizeof(half);
1511
1512     half *h_a = (half *)malloc(size_a);
1513     half *h_b = (half *)malloc(size_b);
1514     half *h_c_ref = (half *)malloc(size_c);
1515
1516     srand(42);
1517     for (int i = 0; i < M * K; i++)
1518         h_a[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1519     for (int i = 0; i < K * N; i++)
1520         h_b[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1521
1522     // CuTe kernel expects B^T [NxK] row-major (TN layout).
1523     size_t size_b_t = (size_t)N * K * sizeof(half);
1524     half *h_b_t = (half *)malloc(size_b_t);
1525     for (int n = 0; n < N; n++)
1526         for (int k = 0; k < K; k++)
1527             h_b_t[n * K + k] = h_b[k * N + n];
1528
1529     half *d_a, *d_b, *d_b_t, *d_c;
1530     check(cudaMalloc(&d_a, size_a), "hgemm_cute alloc A");
1531     check(cudaMalloc(&d_b, size_b), "hgemm_cute alloc B (cuBLAS)");
1532     check(cudaMalloc(&d_b_t, size_b_t), "hgemm_cute alloc B_t (kernel)");
1533     check(cudaMalloc(&d_c, size_c), "hgemm_cute alloc C");

```

```

1534 check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "hgemm_cute H2D A");
1535 check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "hgemm_cute H2D B (cuBLAS)");
1536 check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice), "hgemm_cute H2D B_t (kernel)");
1537
1538
1539 // cuBLAS FP16 reference (row-major idiom: swap M/N, swap A/B)
1540 cublasHandle_t handle;
1541 cublasCreate(&handle);
1542 half alpha_h = __float2half(1.0f), beta_h = __float2half(0.0f);
1543 cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha_h, d_b,
1544             CUDA_R_16F, N, d_a, CUDA_R_16F, K, &beta_h, d_c, CUDA_R_16F, N,
1545             CUBLAS_COMPUTE_16F, CUBLAS_GEMM_DEFAULT);
1546 check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost), "hgemm_cute D2H ref");
1547
1548 // CuTe kernel (Stages=2, TN layout)
1549 launch_hgemm_mma_stages_tn_cute<half, 2, 0>(d_a, d_b_t, d_c, M, N, K);
1550 check(cudaGetLastError(), "hgemm_cute launch");
1551 check(cudaDeviceSynchronize(), "hgemm_cute sync");
1552
1553 half *h_c = (half *)malloc(size_c);
1554 check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "hgemm_cute D2H");
1555
1556 // Verify
1557 float max_err = 0.0f;
1558 for (int i = 0; i < M * N; i++) {
1559     float err = fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i]));
1560     if (err > max_err) max_err = err;
1561 }
1562 printf("| %-56s | %.3e |\n", "HGEMM CuTe Swizzle + Reg2x",
1563       max_err);
1564
1565 free(h_a); free(h_b); free(h_b_t); free(h_c); free(h_c_ref);
1566 cudaFree(d_a); cudaFree(d_b); cudaFree(d_b_t); cudaFree(d_c);
1567 cublasDestroy(handle);
1568 }
1569 #endif /* NOTES_V2_ENABLE_CUTE */
1570
1571
1572 #if defined(NOTES_V2_ENABLE_WGMMA)
1573 static void test_hgemm_wgmma(int M, int N, int K) {
1574     // HGEMM WGMMA — m64n128k16 + TMA + Warp Specialization (Hopper SM90+)
1575     // TN layout: C[M×N] = A[M×K] × B^T[N×K]
1576     // Kernel: hgemm_wgmma_stages_tn with default template params
1577
1578     constexpr int BM = 128, BN = 128, BK = 64, kStages = 3, kNumThreads = 256;
1579
1580     // M, K must be divisible by tile dims
1581     if (M % BM != 0 || N % BN != 0 || K % BK != 0) {
1582         printf("| %-56s | %-9s |\n", "HGEMM WGMMA", "SKIP");
1583         return;
1584     }
1585
1586     size_t size_a = (size_t)M * K * sizeof(half);
1587     size_t size_b = (size_t)K * N * sizeof(half);
1588     size_t size_c = (size_t)M * N * sizeof(half);
1589
1590     half *h_a = (half *)malloc(size_a);
1591     half *h_b = (half *)malloc(size_b);
1592     half *h_c_ref = (half *)malloc(size_c);
1593
1594     srand(42);
1595     for (int i = 0; i < M * K; i++)
1596         h_a[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);

```

```

1597 for (int i = 0; i < K * N; i++)
1598     h_b[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1599
1600 // B^T [N×K] row-major for TN layout (same as hgemm_mma kernel)
1601 size_t size_b_t = (size_t)N * K * sizeof(half);
1602 half *h_b_t = (half *)malloc(size_b_t);
1603 for (int n = 0; n < N; n++)
1604     for (int k = 0; k < K; k++)
1605         h_b_t[n * K + k] = h_b[k * N + n];
1606
1607 half *d_a, *d_b, *d_b_t, *d_c;
1608 check(cudaMalloc(&d_a, size_a), "wgmma alloc A");
1609 check(cudaMalloc(&d_b, size_b), "wgmma alloc B (cuBLAS)");
1610 check(cudaMalloc(&d_b_t, size_b_t), "wgmma alloc B_t (kernel)");
1611 check(cudaMalloc(&d_c, size_c), "wgmma alloc C");
1612
1613 check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "wgmma H2D A");
1614 check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice),
1615         "wgmma H2D B (cuBLAS)");
1616 check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice),
1617         "wgmma H2D B_t (kernel)");
1618
1619 // cuBLAS FP16 reference (row-major idiom, same as hgemm_mma)
1620 cublasHandle_t handle;
1621 cublasCreate(&handle);
1622 half alpha_h = __float2half(1.0f), beta_h = __float2half(0.0f);
1623 cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha_h, d_b,
1624              CUDA_R_16F, N, d_a, CUDA_R_16F, K, &beta_h, d_c, CUDA_R_16F, N,
1625              CUBLAS_COMPUTE_16F, CUBLAS_GEMM_DEFAULT);
1626 check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost),
1627         "wgmma D2H ref");
1628
1629 // Create TMA tensor maps for A and B^T
1630 // A[M×K] row-major: TMA box=(BK=64, BM=128), global shape=(K, M)
1631 // B^T[N×K] row-major: TMA box=(BK=64, BN=128), global shape=(K, N)
1632 CUTensorMap *tma_a =
1633     allocate_and_create_tensor_map(d_a, M / BM, K / BK);
1634 CUTensorMap *tma_b =
1635     allocate_and_create_tensor_map(d_b_t, N / BN, K / BK);
1636
1637 // Launch WGMMMA kernel
1638 // kBlockSwizzle=false → 2D grid, no swizzle
1639 size_t smem_bytes =
1640     kStages * (BM * BK + BN * BK) * sizeof(half); // 3*16384*2 = 96KB
1641 cudaFuncSetAttribute(
1642     hgemm_wgmma_stages_tn<64, 128, 16, BM, BN, BK, kNumThreads, kStages,
1643     false>,
1644     cudaFuncAttributeMaxDynamicSharedMemorySize, smem_bytes);
1645
1646 dim3 block(kNumThreads);
1647 dim3 grid((N + BN - 1) / BN, (M + BM - 1) / BM);
1648 hgemm_wgmma_stages_tn<64, 128, 16, BM, BN, BK, kNumThreads, kStages, false>
1649     <<<grid, block, smem_bytes>>>(M, N, K, d_c, tma_a, tma_b);
1650 check(cudaGetLastError(), "wgmma launch");
1651 check(cudaDeviceSynchronize(), "wgmma sync");
1652
1653 half *h_c = (half *)malloc(size_c);
1654 check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "wgmma D2H");
1655
1656 // Verify
1657 float max_err = 0.0f;
1658 for (int i = 0; i < M * N; i++) {
1659     float err = fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i]));

```

```

1660     if (err > max_err)
1661         max_err = err;
1662 }
1663 printf("| %-56s | %.3e |\n", "HGEMM TMA WGMMMA WS (3-stage)", max_err);
1664
1665 free(h_a);
1666 free(h_b);
1667 free(h_b_t);
1668 free(h_c);
1669 free(h_c_ref);
1670 cudaFree(d_a);
1671 cudaFree(d_b);
1672 cudaFree(d_b_t);
1673 cudaFree(d_c);
1674 cudaFree(tma_a);
1675 cudaFree(tma_b);
1676 cublasDestroy(handle);
1677 }
1678 #endif /* NOTES_V2_ENABLE_WGMMMA */
1679
1680 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
1681 template <int kStages, int kBlockSwizzle = 0>
1682 static bool launch_hgemm_tma_mma_ws(int M, int N, int K, half *d_a,
1683                                     half *d_b_t, half *d_c,
1684                                     CUTensorMap *tma_a, CUTensorMap *tma_b) {
1685     constexpr int kMmaM = 16, kMmaN = 8, kMmaK = 16;
1686     constexpr int kMmaTileM = 2, kMmaTileN = 2;
1687     constexpr int kValTileM = 4, kValTileN = 8, kValTileK = 4;
1688     constexpr int BM = kMmaM * kMmaTileM * kValTileM;
1689     constexpr int BN = kMmaN * kMmaTileN * kValTileN;
1690     constexpr int BK = kMmaK * kValTileK;
1691     constexpr int kNumThreads = 256;
1692     constexpr size_t payload_bytes =
1693         kStages * (BM * BK + BN * BK) * sizeof(half);
1694     constexpr size_t smem_bytes = payload_bytes;
1695     using Kernel = void (*)(int, int, int, half *, const CUTensorMap *,
1696                             const CUTensorMap *);
1697     Kernel kernel = hgemm_tma_mma_ws_tn<
1698         kMmaM, kMmaN, kMmaK, kMmaTileM, kMmaTileN, kValTileM, kValTileN,
1699         kValTileK, kStages, kNumThreads, kBlockSwizzle>;
1700
1701     int device = 0;
1702     int max_smem = 0;
1703     cudaFuncAttributes attributes{};
1704     check(cudaGetDevice(&device), "tma_mma_ws get device");
1705     check(cudaDeviceGetAttribute(&max_smem,
1706                                 cudaDevAttrMaxSharedMemoryPerBlockOptin,
1707                                 device),
1708           "tma_mma_ws max shared memory");
1709     check(cudaFuncGetAttributes(&attributes, kernel),
1710           "tma_mma_ws function attributes");
1711     if (smem_bytes + attributes.sharedSizeBytes > size_t(max_smem)) {
1712         if (g_debug)
1713             printf("| %-56s | %-9s |\n",
1714                   kStages == 2 ? "HGEMM TMA MMA WS (S=2, BLK_SW=0)"
1715                               : "HGEMM TMA MMA WS (S=3, BLK_SW=0)",
1716                   "SMEM SKIP");
1717         return false;
1718     }
1719
1720     check(cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
1721                               smem_bytes),
1722           "tma_mma_ws set dynamic shared memory");

```

```

1723 dim3 block(kNumThreads);
1724 constexpr int kSwizzleN = 16;
1725 const int n_tiles = N / BN;
1726 const int grid_x = kBlockSwizzle ? div_ceil(n_tiles, kSwizzleN) : n_tiles;
1727 const int grid_z = kBlockSwizzle ? kSwizzleN : 1;
1728 dim3 grid(grid_x, M / BM, grid_z);
1729 kernel<<<grid, block, smem_bytes>>>(M, N, K, d_c, tma_a, tma_b);
1730 check(cudaGetLastError(), "tma_mma_ws launch");
1731 check(cudaDeviceSynchronize(), "tma_mma_ws sync");
1732 return true;
1733 }
1734
1735 static void test_hgemm_tma_mma_ws(int M, int N, int K) {
1736     constexpr int BM = 128, BN = 128, BK = 64;
1737     if (M % BM != 0 || N % BN != 0 || K % BK != 0) {
1738         printf("| %-5s | %-9s |\n", "HGEMM TMA MMA WS", "SKIP");
1739         return;
1740     }
1741
1742     const size_t size_a = size_t(M) * K * sizeof(half);
1743     const size_t size_b = size_t(K) * N * sizeof(half);
1744     const size_t size_b_t = size_t(N) * K * sizeof(half);
1745     const size_t size_c = size_t(M) * N * sizeof(half);
1746     half *h_a = static_cast<half *>(malloc(size_a));
1747     half *h_b = static_cast<half *>(malloc(size_b));
1748     half *h_b_t = static_cast<half *>(malloc(size_b_t));
1749     half *h_c = static_cast<half *>(malloc(size_c));
1750     half *h_c_ref = static_cast<half *>(malloc(size_c));
1751     srand(42);
1752     for (int i = 0; i < M * K; ++i)
1753         h_a[i] = __float2half((float(rand()) / RAND_MAX) * 2.0f - 1.0f);
1754     for (int i = 0; i < K * N; ++i)
1755         h_b[i] = __float2half((float(rand()) / RAND_MAX) * 2.0f - 1.0f);
1756     for (int n = 0; n < N; ++n)
1757         for (int k = 0; k < K; ++k)
1758             h_b_t[n * K + k] = h_b[k * N + n];
1759
1760     half *d_a, *d_b, *d_b_t, *d_c;
1761     check(cudaMalloc(&d_a, size_a), "tma_mma_ws alloc A");
1762     check(cudaMalloc(&d_b, size_b), "tma_mma_ws alloc B");
1763     check(cudaMalloc(&d_b_t, size_b_t), "tma_mma_ws alloc B_t");
1764     check(cudaMalloc(&d_c, size_c), "tma_mma_ws alloc C");
1765     check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "tma_mma_ws H2D A");
1766     check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "tma_mma_ws H2D B");
1767     check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice),
1768           "tma_mma_ws H2D B_t");
1769
1770     cublasHandle_t handle;
1771     cublasCreate(&handle);
1772     half alpha = __float2half(1.0f), beta = __float2half(0.0f);
1773     cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha, d_b,
1774                  CUDA_R_16F, N, d_a, CUDA_R_16F, K, &beta, d_c, CUDA_R_16F, N,
1775                  CUBLAS_COMPUTE_16F, CUBLAS_GEMM_DEFAULT);
1776     check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost),
1777           "tma_mma_ws D2H reference");
1778
1779     CUTensorMap *tma_a = allocate_and_create_tensor_map(d_a, M / BM, K / BK);
1780     CUTensorMap *tma_b = allocate_and_create_tensor_map(d_b_t, N / BN, K / BK);
1781     for (int block_swizzle : {0, 1}) {
1782         for (int stages : {2, 3}) {
1783             const bool launched = stages == 2
1784                 ? (block_swizzle
1785                   ? launch_hgemm_tma_mma_ws<2, 1>(M, N, K, d_a, d_b_t, d_c,

```

```

1786         tma_a, tma_b)
1787     : launch_hgemmm_tma_mma_ws<2>(M, N, K, d_a, d_b_t, d_c,
1788         tma_a, tma_b))
1789     : (block_swizzle
1790         ? launch_hgemmm_tma_mma_ws<3, 1>(M, N, K, d_a, d_b_t, d_c,
1791             tma_a, tma_b)
1792         : launch_hgemmm_tma_mma_ws<3>(M, N, K, d_a, d_b_t, d_c,
1793             tma_a, tma_b));
1794     if (!launched)
1795         continue;
1796     check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost),
1797         "tma_mma_ws D2H");
1798     float max_err = 0.0f;
1799     for (int i = 0; i < M * N; ++i)
1800         max_err = fmaxf(max_err, fabsf(__half2float(h_c[i]) -
1801             __half2float(h_c_ref[i])));
1802     printf("| %-56s | %.3e |\n",
1803         block_swizzle
1804             ? (stages == 2 ? "HGEMM TMA MMA WS (S=2, BLK_SW=1)"
1805                 : "HGEMM TMA MMA WS (S=3, BLK_SW=1)")
1806             : (stages == 2 ? "HGEMM TMA MMA WS (S=2, BLK_SW=0)"
1807                 : "HGEMM TMA MMA WS (S=3, BLK_SW=0)"),
1808         max_err);
1809 }
1810 }
1811
1812 free(h_a); free(h_b); free(h_b_t); free(h_c); free(h_c_ref);
1813 cudaFree(d_a); cudaFree(d_b); cudaFree(d_b_t); cudaFree(d_c);
1814 cudaFree(tma_a); cudaFree(tma_b);
1815 cublasDestroy(handle);
1816 }
1817 #endif /* NOTES_V2_ENABLE_TMA_MMA_WS */
1818
1819
1820 static void test_flash_attn(int seqlen, int head_dim) {
1821     // FlashAttention-2 with split-Q, MMA m16n8k16
1822     int B = 1, H = 8;
1823
1824     size_t sz = (size_t)B * H * seqlen * head_dim * sizeof(half);
1825
1826     srand(42);
1827     half *h_q = (half *)malloc(sz);
1828     half *h_k = (half *)malloc(sz);
1829     half *h_v = (half *)malloc(sz);
1830     for (int i = 0; i < B * H * seqlen * head_dim; i++) {
1831         h_q[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1832         h_k[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1833         h_v[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1834     }
1835
1836     // CPU reference in FP32: 0 = softmax(Q @ K^T / sqrt(d)) @ V
1837     float *ref_q = (float *)malloc(sz * 4 / sizeof(half)); // 4x for float
1838     float *ref_k = (float *)malloc(sz * 4 / sizeof(half));
1839     float *ref_v = (float *)malloc(sz * 4 / sizeof(half));
1840     float *ref_o = (float *)malloc(sz * 4 / sizeof(half));
1841     int count = B * H * seqlen * head_dim;
1842     for (int i = 0; i < count; i++) {
1843         ref_q[i] = __half2float(h_q[i]);
1844         ref_k[i] = __half2float(h_k[i]);
1845         ref_v[i] = __half2float(h_v[i]);
1846     }
1847
1848     float scale = 1.0f / sqrtf((float)head_dim);

```

```

1849 for (int bi = 0; bi < B * H; bi++) {
1850     for (int qi = 0; qi < seqlen; qi++) {
1851         // S[qi, kj] = Q[qi,:] @ K[kj,:]^T * scale
1852         float smax = -INFINITY;
1853         float *S = (float *)malloc((size_t)seqlen * sizeof(float));
1854         for (int kj = 0; kj < seqlen; kj++) {
1855             float s = 0.0f;
1856             for (int d = 0; d < head_dim; d++)
1857                 s += ref_q[bi * seqlen * head_dim + qi * head_dim + d] *
1858                     ref_k[bi * seqlen * head_dim + kj * head_dim + d];
1859             S[kj] = s * scale;
1860             if (S[kj] > smax) smax = S[kj];
1861         }
1862         // softmax
1863         double sum_exp = 0.0;
1864         for (int kj = 0; kj < seqlen; kj++) sum_exp += (double)expf(S[kj] - smax);
1865         float inv_sum = 1.0f / (float)sum_exp;
1866         // O[qi, :] = sum_kj P[qi, kj] * V[kj, :]
1867         for (int d = 0; d < head_dim; d++) {
1868             double o_acc = 0.0;
1869             for (int kj = 0; kj < seqlen; kj++)
1870                 o_acc += (double)(expf(S[kj] - smax) * inv_sum) *
1871                     ref_v[bi * seqlen * head_dim + kj * head_dim + d];
1872             ref_o[bi * seqlen * head_dim + qi * head_dim + d] = (float)o_acc;
1873         }
1874         free(S);
1875     }
1876 }
1877
1878 half *d_q, *d_k, *d_v, *d_o;
1879 check(cudaMalloc(&d_q, sz), "fa alloc Q");
1880 check(cudaMalloc(&d_k, sz), "fa alloc K");
1881 check(cudaMalloc(&d_v, sz), "fa alloc V");
1882 check(cudaMalloc(&d_o, sz), "fa alloc O");
1883 check(cudaMemcpy(d_q, h_q, sz, cudaMemcpyHostToDevice), "fa H2D Q");
1884 check(cudaMemcpy(d_k, h_k, sz, cudaMemcpyHostToDevice), "fa H2D K");
1885 check(cudaMemcpy(d_v, h_v, sz, cudaMemcpyHostToDevice), "fa H2D V");
1886
1887 // Template params for kHeadDim=64, kStagesK=2
1888 constexpr int kHeadDim = 64;
1889 constexpr int kStagesK = 2;
1890 constexpr int kPadQ = 8;
1891 constexpr int kPadK = 8;
1892 constexpr int kPadV = 8;
1893 constexpr int kMmaAtomM = 16;
1894 constexpr int kMmaAtomN = 8;
1895 constexpr int kMmaAtomK = 16;
1896 constexpr int kMmaTileSeqLenQ = 8;
1897 constexpr int kMmaTileSeqLenK = 1;
1898 constexpr int kMmaTileSeqLenP = 8;
1899 constexpr int kMmaTileHeadDimV = 1;
1900 constexpr int kValTileSeqLenQ = 1;
1901 constexpr int kValTileSeqLenK = 8;
1902 constexpr int kValTileSeqLenP = 1;
1903 constexpr int kValTileHeadDimV = kHeadDim / (8 * kMmaTileHeadDimV);
1904
1905 constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ;
1906 constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK;
1907 if (seqlen < Br) return; // kernel requires seqlen >= tile size
1908 size_t smem_bytes =
1909     (Br * (kHeadDim + kPadQ) +
1910      kStagesK * Bc * (kHeadDim + kPadK) +
1911      Bc * (kHeadDim + kPadV)) * sizeof(half);

```

```

1912
1913 dim3 block(256);
1914 dim3 grid((seqlen + Br - 1) / Br, B * H);
1915
1916 // Test both accumulator variants: f16 (kMmaAccF32=0) and f32 (kMmaAccF32=1)
1917 for (int acc = 0; acc <= 1; ++acc) {
1918     const int kMmaAcc = acc;
1919     half *h_o = (half *)malloc(sz);
1920
1921     if (kMmaAcc == 0) {
1922         using FAKernel = void (*)(half *, half *, half *, half *, int, int);
1923         FAKernel fa_k = flash_attn_mma_stages_split_q<kHeadDim, kMmaAtomM, kMmaAtomN,
1924             kMmaAtomK, 0, kMmaTileSeqLenQ, kMmaTileSeqLenK, kMmaTileSeqLenP,
1925             kMmaTileHeadDimV, kValTileSeqLenQ, kValTileSeqLenK, kValTileSeqLenP,
1926             kValTileHeadDimV, kStagesK, kPadQ, kPadK, kPadV>;
1927         cudaFuncSetAttribute(fa_k, cudaFuncAttributeMaxDynamicSharedMemorySize, smem_bytes);
1928         fa_k<<<grid, block, smem_bytes>>>(d_q, d_k, d_v, d_o, seqlen, H);
1929     } else {
1930         using FAKernel = void (*)(half *, half *, half *, half *, int, int);
1931         FAKernel fa_k = flash_attn_mma_stages_split_q<kHeadDim, kMmaAtomM, kMmaAtomN,
1932             kMmaAtomK, 1, kMmaTileSeqLenQ, kMmaTileSeqLenK, kMmaTileSeqLenP,
1933             kMmaTileHeadDimV, kValTileSeqLenQ, kValTileSeqLenK, kValTileSeqLenP,
1934             kValTileHeadDimV, kStagesK, kPadQ, kPadK, kPadV>;
1935         cudaFuncSetAttribute(fa_k, cudaFuncAttributeMaxDynamicSharedMemorySize, smem_bytes);
1936         fa_k<<<grid, block, smem_bytes>>>(d_q, d_k, d_v, d_o, seqlen, H);
1937     }
1938     check(cudaGetLastError(), "fa launch");
1939     check(cudaDeviceSynchronize(), "fa sync");
1940
1941     check(cudaMemcpy(h_o, d_o, sz, cudaMemcpyDeviceToHost), "fa D2H");
1942
1943     float max_err = 0.0f;
1944     for (int i = 0; i < count; i++) {
1945         float err = fabsf(__half2float(h_o[i]) - ref_o[i]);
1946         if (err > max_err) max_err = err;
1947     }
1948     const char *acc_label = kMmaAcc ? "F32Acc" : "F16Acc";
1949     char label[64];
1950     snprintf(label, sizeof(label), "FA2 (kStagesK=2, Pad, %s)", acc_label);
1951     printf("| %-56s | %.3e |\n", label,
1952         max_err);
1953     free(h_o);
1954 }
1955
1956 free(h_q); free(h_k); free(h_v);
1957 free(ref_q); free(ref_k); free(ref_v); free(ref_o);
1958 cudaFree(d_q); cudaFree(d_k); cudaFree(d_v); cudaFree(d_o);
1959 }
1960
1961
1962 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
1963 // Test for flash_attn_tma_mma_ws_stages_split_q (SM120, D=64/128)
1964 // Reference: cuDNN SDPA (half) if available, else CPU FP32 (float) fallback.
1965 template <int kHeadDim>
1966 static void test_flash_attn_tma_mma_ws_impl(int seqlen, int head_dim) {
1967     int B = 1, H = 8;
1968     constexpr int kStagesK = 2;
1969     constexpr int kStagesV = 1;
1970     constexpr int kMmaAtomM = 16, kMmaAtomN = 8, kMmaAtomK = 16;
1971     constexpr int kMmaTileSeqLenQ = 8, kMmaTileSeqLenK = 1;
1972     constexpr int kMmaTileSeqLenP = 8, kMmaTileHeadDimV = 1;
1973     constexpr int kValTileSeqLenQ = 1, kValTileSeqLenK = 8;
1974     constexpr int kValTileSeqLenP = 1;

```

```

1975 constexpr int kValTileHeadDimV = kHeadDim / (8 * kMmaTileHeadDimV);
1976 constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ; // 128
1977 constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK; // 64
1978 constexpr int kNumThreads = 384;
1979
1980 if (seqlen % Br != 0 || seqlen % Bc != 0 || seqlen < Br) {
1981     printf("| %-56s | %-9s |\n",
1982         "FA2 TMA MMA WS (1 Consumer WG) (unaligned)", "SKIP");
1983     return;
1984 }
1985
1986 size_t sz = (size_t)B * H * seqlen * head_dim * sizeof(half);
1987 srand(42);
1988 half *h_q = (half *)malloc(sz);
1989 half *h_k = (half *)malloc(sz);
1990 half *h_v = (half *)malloc(sz);
1991 for (int i = 0; i < B * H * seqlen * head_dim; ++i) {
1992     h_q[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1993     h_k[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1994     h_v[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
1995 }
1996
1997 half *d_q, *d_k, *d_v, *d_o;
1998 check(cudaMalloc(&d_q, sz), "fa_tma_ws alloc Q");
1999 check(cudaMalloc(&d_k, sz), "fa_tma_ws alloc K");
2000 check(cudaMalloc(&d_v, sz), "fa_tma_ws alloc V");
2001 check(cudaMalloc(&d_o, sz), "fa_tma_ws alloc O");
2002 check(cudaMemcpy(d_q, h_q, sz, cudaMemcpyHostToDevice), "fa_tma_ws H2D Q");
2003 check(cudaMemcpy(d_k, h_k, sz, cudaMemcpyHostToDevice), "fa_tma_ws H2D K");
2004 check(cudaMemcpy(d_v, h_v, sz, cudaMemcpyHostToDevice), "fa_tma_ws H2D V");
2005
2006 // Reference output: either cuDNN (half) or CPU (float)
2007 half *h_o_ref = nullptr;
2008 float *ref_o = nullptr;
2009 int count = B * H * seqlen * head_dim;
2010
2011 #if defined(NOTES_V2_ENABLE_CUDNN)
2012 {
2013     // Try cuDNN SDPA first; fall back to CPU if unsupported on this SM
2014     bool cudnn_ok = false;
2015     half *d_o_ref;
2016     check(cudaMalloc(&d_o_ref, sz), "fa_tma_ws alloc O_ref");
2017     {
2018         cudnnHandle_t cudnn_handle;
2019         cudnnCreate(&cudnn_handle);
2020         auto graph = std::make_shared<fe::graph::Graph>();
2021         graph->set_io_data_type(fe::DataType_t::HALF)
2022             .set_intermediate_data_type(fe::DataType_t::FLOAT)
2023             .set_compute_data_type(fe::DataType_t::FLOAT);
2024
2025         auto Q = graph->tensor(fe::graph::Tensor_attributes()
2026             .set_uid(1).set_dim({B, H, seqlen, head_dim})
2027             .set_stride({H * seqlen * head_dim, seqlen * head_dim, head_dim, 1}));
2028         auto K = graph->tensor(fe::graph::Tensor_attributes()
2029             .set_uid(2).set_dim({B, H, seqlen, head_dim})
2030             .set_stride({H * seqlen * head_dim, seqlen * head_dim, head_dim, 1}));
2031         auto V = graph->tensor(fe::graph::Tensor_attributes()
2032             .set_uid(3).set_dim({B, H, seqlen, head_dim})
2033             .set_stride({H * seqlen * head_dim, seqlen * head_dim, head_dim, 1}));
2034
2035         auto [O_sdpa, Stats] = graph->sdpa(Q, K, V,
2036             fe::graph::SDPA_attributes()
2037                 .set_name("sdpa_ref")

```

```

2038         .set_attn_scale(1.0f / sqrtf((float)head_dim)));
2039
2040     O_sdpa->set_output(true).set_uid(4)
2041         .set_dim({B, H, seqlen, head_dim})
2042         .set_stride({H * seqlen * head_dim, seqlen * head_dim, head_dim, 1});
2043
2044     auto build_status = graph->build(cudnn_handle, {fe::HeurMode_t::A, fe::HeurMode_t::FALLBACK});
2045     if (build_status.is_good()) {
2046         std::unordered_map<fe::graph::Tensor_attributes::uid_t, void*> vp = {
2047             {1, d_q}, {2, d_k}, {3, d_v}, {4, d_o_ref}};
2048         int64_t ws_size = 0;
2049         if (graph->get_workspace_size(ws_size).is_good()) {
2050             int8_t *d_ws = nullptr;
2051             if (ws_size > 0) check(cudaMalloc(&d_ws, ws_size), "fa_tma_ws workspace");
2052             if (graph->execute(cudnn_handle, vp, d_ws).is_good()) {
2053                 check(cudaDeviceSynchronize(), "fa_tma_ws cudnn sync");
2054                 cudnn_ok = true;
2055             }
2056             if (d_ws) cudaFree(d_ws);
2057         }
2058     }
2059     cudnnDestroy(cudnn_handle);
2060 }
2061
2062 if (cudnn_ok) {
2063     h_o_ref = (half *)malloc(sz);
2064     check(cudaMemcpy(h_o_ref, d_o_ref, sz, cudaMemcpyDeviceToHost), "fa_tma_ws D2H ref");
2065 } else {
2066     fprintf(stderr, "cudnn SDPA unsupported on this SM, using CPU ref\n");
2067 }
2068 cudaFree(d_o_ref);
2069 }
2070 #endif
2071
2072 // CPU reference fallback
2073 if (!h_o_ref) {
2074     float *ref_q = (float *)malloc(sz * 4 / sizeof(half));
2075     float *ref_k = (float *)malloc(sz * 4 / sizeof(half));
2076     float *ref_v = (float *)malloc(sz * 4 / sizeof(half));
2077     ref_o = (float *)malloc(sz * 4 / sizeof(half));
2078     for (int i = 0; i < count; ++i) {
2079         ref_q[i] = __half2float(h_q[i]);
2080         ref_k[i] = __half2float(h_k[i]);
2081         ref_v[i] = __half2float(h_v[i]);
2082     }
2083     float scale = 1.0f / sqrtf((float)head_dim);
2084     for (int bi = 0; bi < B * H; ++bi) {
2085         for (int qi = 0; qi < seqlen; ++qi) {
2086             float smax = -INFINITY;
2087             float *S = (float *)malloc((size_t)seqlen * sizeof(float));
2088             for (int kj = 0; kj < seqlen; ++kj) {
2089                 float s = 0.0f;
2090                 for (int d = 0; d < head_dim; ++d)
2091                     s += ref_q[bi * seqlen * head_dim + qi * head_dim + d] *
2092                        ref_k[bi * seqlen * head_dim + kj * head_dim + d];
2093                 S[kj] = s * scale;
2094                 if (S[kj] > smax) smax = S[kj];
2095             }
2096             double sum_exp = 0.0;
2097             for (int kj = 0; kj < seqlen; ++kj)
2098                 sum_exp += (double)expf(S[kj] - smax);
2099             float inv_sum = 1.0f / (float)sum_exp;
2100             for (int d = 0; d < head_dim; ++d) {

```

```

2101     double o_acc = 0.0;
2102     for (int kj = 0; kj < seqlen; ++kj)
2103         o_acc += (double)(expf(S[kj] - smax) * inv_sum) *
2104             ref_v[bi * seqlen * head_dim + kj * head_dim + d];
2105     ref_o[bi * seqlen * head_dim + qi * head_dim + d] = (float)o_acc;
2106 }
2107 free(S);
2108 }
2109 }
2110 free(ref_q);
2111 free(ref_k);
2112 free(ref_v);
2113 }
2114
2115 // TMA descriptors: box innermost 固定 64 half (128B), 满足
2116 // CU_TENSOR_MAP_SWIZZLE_128B 对 box innermost ≤ 128B 的硬约束。
2117 // D=64: blocks_width=1, 单次 TMA 覆盖整行。
2118 // D=128: blocks_width=2, kernel producer 沿 head_dim 连续发 2 次 TMA,
2119 //         minor_coord = 0/64, 写入 chunk-major smem 布局 [2, Br, 64]。
2120 // Q/K/V gmem 是 [B, H, seqlen, head_dim] row-major, 作为 2D
2121 // [B*H*seqlen, head_dim] 矩阵描述。blocks_height = B*H*seqlen/tile_major。
2122 // kernel 用 (Nb_id*H+Nh_id)*N 偏移 major_coord。
2123 constexpr int kTmaBoxMinor = 64; // box innermost = 64 half = 128B
2124 constexpr int kTmaChunksQ = kHeadDim / kTmaBoxMinor;
2125 constexpr int kTmaChunksKV = kHeadDim / kTmaBoxMinor;
2126 CUTensorMap *tma_q = allocate_and_create_tensor_map<Br, kTmaBoxMinor>(
2127     d_q, B * H * seqlen / Br, kTmaChunksQ);
2128 CUTensorMap *tma_k = allocate_and_create_tensor_map<Bc, kTmaBoxMinor>(
2129     d_k, B * H * seqlen / Bc, kTmaChunksKV);
2130 CUTensorMap *tma_v = allocate_and_create_tensor_map<Bc, kTmaBoxMinor>(
2131     d_v, B * H * seqlen / Bc, kTmaChunksKV);
2132
2133 using FAKernel = void (*)(half *, half *, half *, half *, int, int,
2134     const CUTensorMap *, const CUTensorMap *,
2135     const CUTensorMap *);
2136 FAKernel fa_k = flash_attn_tma_mma_ws_stages_split_q<
2137     kHeadDim, kMmaAtomM, kMmaAtomN, kMmaAtomK, 0, kMmaTileSeqLenQ,
2138     kMmaTileSeqLenK, kMmaTileSeqLenP, kMmaTileHeadDimV, kValTileSeqLenQ,
2139     kValTileSeqLenK, kValTileSeqLenP, kValTileHeadDimV, kStagesK, kStagesV,
2140     kNumThreads>;
2141
2142 // smem = Q[Br*d] + K[kStagesK*Bc*d] + V[kStagesV*Bc*d]
2143 size_t smem_bytes = (Br * kHeadDim + kStagesK * Bc * kHeadDim +
2144     kStagesV * Bc * kHeadDim) * sizeof(half);
2145 int device = 0, max_smem = 0;
2146 cudaFuncAttributes attributes{};
2147 cudaGetDevice(&device);
2148 cudaDeviceGetAttribute(&max_smem, cudaDevAttrMaxSharedMemoryPerBlockOptin,
2149     device);
2150 cudaFuncGetAttributes(&attributes, fa_k);
2151 bool smem_ok =
2152     (smem_bytes + attributes.sharedSizeBytes <= (size_t)max_smem);
2153 if (!smem_ok) {
2154     if (g_debug)
2155         printf("| %-56s | %-9s |\n",
2156             "FA2 TMA MMA WS (1 Consumer WG) (SMEM SKIP)", "SMEM SKIP");
2157 } else {
2158     dim3 block(kNumThreads);
2159     dim3 grid((seqlen + Br - 1) / Br, B * H);
2160     // Test both accumulator variants
2161     for (int acc = 0; acc <= 1; ++acc) {
2162         const int kMmaAcc = acc;
2163         half *h_o = (half *)malloc(sz);

```

```

2164
2165 if (kMmaAcc == 0) {
2166     using FAK = void (*)(half *, half *, half *, half *, int, int,
2167                          const CUTensorMap *, const CUTensorMap *,
2168                          const CUTensorMap *);
2169     FAK fk = flash_attn_tma_mma_ws_stages_split_q<
2170         kHeadDim, kMmaAtomM, kMmaAtomN, kMmaAtomK, 0, kMmaTileSeqLenQ,
2171         kMmaTileSeqLenK, kMmaTileSeqLenP, kMmaTileHeadDimV, kValTileSeqLenQ,
2172         kValTileSeqLenK, kValTileSeqLenP, kValTileHeadDimV, kStagesK,
2173         kStagesV, kNumThreads>;
2174     cudaFuncSetAttribute(fk, cudaFuncAttributeMaxDynamicSharedMemorySize,
2175                          smem_bytes);
2176     fk<<<grid, block, smem_bytes>>>(d_q, d_k, d_v, d_o, seqlen, H, tma_q,
2177                                     tma_k, tma_v);
2178 } else {
2179     using FAK = void (*)(half *, half *, half *, half *, int, int,
2180                          const CUTensorMap *, const CUTensorMap *,
2181                          const CUTensorMap *);
2182     FAK fk = flash_attn_tma_mma_ws_stages_split_q<
2183         kHeadDim, kMmaAtomM, kMmaAtomN, kMmaAtomK, 1, kMmaTileSeqLenQ,
2184         kMmaTileSeqLenK, kMmaTileSeqLenP, kMmaTileHeadDimV, kValTileSeqLenQ,
2185         kValTileSeqLenK, kValTileSeqLenP, kValTileHeadDimV, kStagesK,
2186         kStagesV, kNumThreads>;
2187     cudaFuncSetAttribute(fk, cudaFuncAttributeMaxDynamicSharedMemorySize,
2188                          smem_bytes);
2189     fk<<<grid, block, smem_bytes>>>(d_q, d_k, d_v, d_o, seqlen, H, tma_q,
2190                                     tma_k, tma_v);
2191 }
2192 check(cudaGetLastError(), "fa_tma_ws launch");
2193 check(cudaDeviceSynchronize(), "fa_tma_ws sync");
2194
2195 check(cudaMemcpy(h_o, d_o, sz, cudaMemcpyDeviceToHost), "fa_tma_ws D2H");
2196 float max_err = 0.0f;
2197 bool checked = h_o_ref || ref_o;
2198 if (checked) {
2199     for (int i = 0; i < count; ++i) {
2200         float ref_val = h_o_ref ? __half2float(h_o_ref[i]) : ref_o[i];
2201         float err = fabsf(__half2float(h_o[i]) - ref_val);
2202         if (err > max_err) max_err = err;
2203     }
2204 }
2205 const char *acc_label = kMmaAcc ? "F32Acc" : "F16Acc";
2206 char label[64];
2207 snprintf(label, sizeof(label),
2208          "FA2 TMA MMA WS (1 Consumer WG) (%s)",
2209          acc_label);
2210 printf("| %-56s | %.3e |\n",
2211        label, max_err);
2212 free(h_o);
2213 }
2214 }
2215
2216 free(h_q); free(h_k); free(h_v);
2217 free(h_o_ref); free(ref_o);
2218 cudaFree(d_q); cudaFree(d_k); cudaFree(d_v); cudaFree(d_o);
2219 cudaFree(tma_q); cudaFree(tma_k); cudaFree(tma_v);
2220 }
2221
2222 // D=64/128 dispatch wrapper
2223 static void test_flash_attn_tma_mma_ws(int seqlen, int head_dim) {
2224     if (head_dim == 64) {
2225         test_flash_attn_tma_mma_ws_impl<64>(seqlen, head_dim);
2226     } else if (head_dim == 128) {

```

```

2227     test_flash_attn_tma_mma_ws_impl<128>(seqlen, head_dim);
2228 } else {
2229     printf("| %-56s | %-9s |\n",
2230           "FA2 TMA MMA WS (1 Consumer WG) (D!=64/128)", "SKIP");
2231 }
2232 }
2233
2234 // FA3-style dual-consumer correctness test (Br=64, Sk=Sv=2)
2235 // Tests both F16Acc and F32Acc against cuDNN SDPA reference.
2236 #if defined(NOTES_V2_ENABLE_CUDNN)
2237 namespace fe = cudnn_frontend;
2238 static float bench_cudnn_sdpa_tflops(half *d_q, half *d_k, half *d_v,
2239                                     half *d_o_ref, int B, int H, int seqlen,
2240                                     int head_dim, fe::DataType_t compute_type);
2241 #endif
2242 template <int kHeadDim, int kStagesK>
2243 static void test_flash_attn_3_tma_ws_impl(int seqlen, int head_dim) {
2244     int B = 1, H = 8;
2245     constexpr int kMmaAtomM = 16, kMmaAtomN = 8, kMmaAtomK = 16;
2246     constexpr int kMmaTileSeqLenQ = 4, kMmaTileSeqLenK = 1;
2247     constexpr int kMmaTileSeqLenP = 4, kMmaTileHeadDimV = 1;
2248     constexpr int kValTileSeqLenQ = 1, kValTileSeqLenK = 8;
2249     constexpr int kValTileSeqLenP = 1;
2250     constexpr int kValTileHeadDimV = kHeadDim / (8 * kMmaTileHeadDimV);
2251     constexpr int kStagesV = 1; // per-WG V: always 1 buffer
2252     constexpr int kNumConsumerWGs = 2;
2253     constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ; // 64
2254     constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK; // 64
2255     constexpr int kNumThreads = 384;
2256
2257     if (seqlen % Br != 0 || seqlen % Bc != 0 || seqlen < Br) {
2258         printf("| %-56s | %-9s |\n",
2259               "FA3 TMA MMA WS (2 Consumer WG) (unaligned)", "SKIP");
2260         return;
2261     }
2262
2263     size_t sz = (size_t)B * H * seqlen * head_dim * sizeof(half);
2264     srand(42);
2265     half *h_q = (half *)malloc(sz);
2266     half *h_k = (half *)malloc(sz);
2267     half *h_v = (half *)malloc(sz);
2268     for (int i = 0; i < B * H * seqlen * head_dim; ++i) {
2269         h_q[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2270         h_k[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2271         h_v[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2272     }
2273
2274     half *d_q, *d_k, *d_v, *d_o;
2275     check(cudaMalloc(&d_q, sz), "fa3 alloc Q");
2276     check(cudaMalloc(&d_k, sz), "fa3 alloc K");
2277     check(cudaMalloc(&d_v, sz), "fa3 alloc V");
2278     check(cudaMalloc(&d_o, sz), "fa3 alloc O");
2279     check(cudaMemcpy(d_q, h_q, sz, cudaMemcpyHostToDevice), "fa3 H2D Q");
2280     check(cudaMemcpy(d_k, h_k, sz, cudaMemcpyHostToDevice), "fa3 H2D K");
2281     check(cudaMemcpy(d_v, h_v, sz, cudaMemcpyHostToDevice), "fa3 H2D V");
2282
2283     // Reference: cuDNN SDPA (half output)
2284     half *h_o_ref = nullptr;
2285     int count = B * H * seqlen * head_dim;
2286     #if defined(NOTES_V2_ENABLE_CUDNN)
2287     {
2288         half *d_o_ref;
2289         check(cudaMalloc(&d_o_ref, sz), "fa3 alloc O_ref");

```

```

2290     bench_cudnn_sdpa_tflops(d_q, d_k, d_v, d_o_ref, B, H, seqlen, head_dim,
2291                             fe::DataType_t::FLOAT);
2292     h_o_ref = (half *)malloc(sz);
2293     check(cudaMemcpy(h_o_ref, d_o_ref, sz, cudaMemcpyDeviceToHost), "fa3 ref D2H");
2294     cudaFree(d_o_ref);
2295 }
2296 #endif
2297
2298 // TMA descriptors (Br=64 for Q, Bc=64 for K/V)
2299 constexpr int kTmaBoxMinor = 64;
2300 constexpr int kTmaChunks = kHeadDim / kTmaBoxMinor;
2301 CUTensorMap *tma_q = allocate_and_create_tensor_map<Br, kTmaBoxMinor>(
2302     d_q, B * H * seqlen / Br, kTmaChunks);
2303 CUTensorMap *tma_k = allocate_and_create_tensor_map<Bc, kTmaBoxMinor>(
2304     d_k, B * H * seqlen / Bc, kTmaChunks);
2305 CUTensorMap *tma_v = allocate_and_create_tensor_map<Bc, kTmaBoxMinor>(
2306     d_v, B * H * seqlen / Bc, kTmaChunks);
2307
2308 using FAKernel = void (*)(half *, half *, half *, half *, int, int,
2309                           const CUTensorMap *, const CUTensorMap *,
2310                           const CUTensorMap *);
2311
2312 // smem = Q + K[2WGs * Sk * Bc*D] + V[2WGs * 1 * Bc*D]
2313 size_t smem_bytes = (Br * kHeadDim +
2314                     kNumConsumerWGs * kStagesK * Bc * kHeadDim +
2315                     kNumConsumerWGs * kStagesV * Bc * kHeadDim) * sizeof(half);
2316 dim3 block(kNumThreads);
2317 dim3 grid((seqlen + Br - 1) / Br, B * H);
2318
2319 for (int acc = 0; acc <= 1; ++acc) {
2320     half *h_o = (half *)malloc(sz);
2321     FAKernel fk;
2322     if (acc == 0) {
2323         fk = flash_attn_3_tma_ws_stages_split_q<
2324             kHeadDim, kMmaAtomM, kMmaAtomN, kMmaAtomK, 0, kMmaTileSeqLenQ,
2325             kMmaTileSeqLenK, kMmaTileSeqLenP, kMmaTileHeadDimV, kValTileSeqLenQ,
2326             kValTileSeqLenK, kValTileSeqLenP, kValTileHeadDimV, kStagesK, kStagesV,
2327             kNumThreads>;
2328     } else {
2329         fk = flash_attn_3_tma_ws_stages_split_q<
2330             kHeadDim, kMmaAtomM, kMmaAtomN, kMmaAtomK, 1, kMmaTileSeqLenQ,
2331             kMmaTileSeqLenK, kMmaTileSeqLenP, kMmaTileHeadDimV, kValTileSeqLenQ,
2332             kValTileSeqLenK, kValTileSeqLenP, kValTileHeadDimV, kStagesK, kStagesV,
2333             kNumThreads>;
2334     }
2335     bool smem_ok = check_smem_feasible((const void *)fk, smem_bytes);
2336     if (!smem_ok) {
2337         const char *acc_label = acc ? "F32Acc" : "F16Acc";
2338         char label[64];
2339         snprintf(label, sizeof(label), "FA3 TMA MMA WS (2 Consumer WG) (%s)",
2340                 acc_label);
2341         if (g_debug)
2342             printf("| %-56s | %-9s |\n", label, "SMEM too large");
2343         free(h_o);
2344         continue;
2345     }
2346     cudaFuncSetAttribute(fk, cudaFuncAttributeMaxDynamicSharedMemorySize,
2347                         smem_bytes);
2348     fk<<<grid, block, smem_bytes>>>(d_q, d_k, d_v, d_o, seqlen, H, tma_q,
2349                                     tma_k, tma_v);
2350     check(cudaGetLastError(), "fa3 launch");
2351     check(cudaDeviceSynchronize(), "fa3 sync");
2352

```

```

2353     check(cudaMemcpy(h_o, d_o, sz, cudaMemcpyDeviceToHost), "fa3 D2H");
2354     float max_err = 0.0f;
2355     bool checked = h_o_ref != nullptr;
2356     if (checked) {
2357         for (int i = 0; i < count; ++i) {
2358             float err = fabsf(__half2float(h_o[i]) - __half2float(h_o_ref[i]));
2359             if (err > max_err) max_err = err;
2360         }
2361     }
2362     const char *acc_label = acc ? "F32Acc" : "F16Acc";
2363     char label[64];
2364     snprintf(label, sizeof(label), "FA3 TMA MMA WS (2 Consumer WG) (%s)",
2365              acc_label);
2366     printf("| %-56s | %.3e |\n", label, max_err);
2367     free(h_o);
2368 }
2369
2370 free(h_q); free(h_k); free(h_v); free(h_o_ref);
2371 cudaFree(d_q); cudaFree(d_k); cudaFree(d_v); cudaFree(d_o);
2372 cudaFree(tma_q); cudaFree(tma_k); cudaFree(tma_v);
2373 }
2374
2375 static void test_flash_attn_3_tma_ws(int seqlen, int head_dim) {
2376     // Try stages up to 4; test_flash_attn_3_tma_ws_impl internally checks
2377     // cudaDevAttrMaxSharedMemoryPerBlockOptin and skips oversized configs.
2378     if (head_dim == 64) {
2379         test_flash_attn_3_tma_ws_impl<64, 1>(seqlen, head_dim);
2380         test_flash_attn_3_tma_ws_impl<64, 2>(seqlen, head_dim);
2381         test_flash_attn_3_tma_ws_impl<64, 3>(seqlen, head_dim);
2382         test_flash_attn_3_tma_ws_impl<64, 4>(seqlen, head_dim);
2383     } else if (head_dim == 128) {
2384         test_flash_attn_3_tma_ws_impl<128, 1>(seqlen, head_dim);
2385         test_flash_attn_3_tma_ws_impl<128, 2>(seqlen, head_dim);
2386         test_flash_attn_3_tma_ws_impl<128, 3>(seqlen, head_dim);
2387         test_flash_attn_3_tma_ws_impl<128, 4>(seqlen, head_dim);
2388     } else {
2389         printf("| %-56s | %-9s |\n",
2390                "FA3 TMA MMA WS (2 Consumer WG) (D!=64/128)", "SKIP");
2391     }
2392 }
2393 #endif /* NOTES_V2_ENABLE_TMA_MMA_WS */
2394
2395 static float bench_hgemm_tflops(int M, int N, int K, float time_ms) {
2396     double flops = 2.0 * M * N * K;
2397     return (float)(flops / (double)time_ms / 1e9);
2398 }
2399
2400 static float bench_fa_tflops(int B, int H, int N, int D, float time_ms) {
2401     double flops = 4.0 * B * H * N * N * D;
2402     return (float)(flops / (double)time_ms / 1e9);
2403 }
2404
2405 static float bench_cublas_hgemm_tflops(cublasHandle_t handle, int M, int N, int K,
2406                                         half *d_a, half *d_b, half *d_c) {
2407     half alpha = __float2half(1.0f), beta = __float2half(0.0f);
2408     cudaStream_t stream;
2409     cudaStreamCreate(&stream);
2410     cublasSetStream(handle, stream);
2411     // warmup
2412     for (int w = 0; w < g_warmup; ++w)
2413         cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha,
2414                      d_b, CUDA_R_16F, N, d_a, CUDA_R_16F, K, &beta,
2415                      d_c, CUDA_R_16F, N, CUBLAS_COMPUTE_16F, CUBLAS_GEMM_DEFAULT);

```

```

2416     cudaStreamSynchronize(stream);
2417     // timed repeat
2418     cudaEvent_t start, stop;
2419     cudaEventCreate(&start);
2420     cudaEventCreate(&stop);
2421     cudaEventRecord(start, stream);
2422     for (int r = 0; r < g_repeat; ++r)
2423         cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha,
2424                     d_b, CUDA_R_16F, N, d_a, CUDA_R_16F, K, &beta,
2425                     d_c, CUDA_R_16F, N, CUBLAS_COMPUTE_16F, CUBLAS_GEMM_DEFAULT);
2426     cudaEventRecord(stop, stream);
2427     cudaEventSynchronize(stop);
2428     float time_ms = 0;
2429     cudaEventElapsedTime(&time_ms, start, stop);
2430     time_ms /= g_repeat;
2431     cudaEventDestroy(start);
2432     cudaEventDestroy(stop);
2433     cublasSetStream(handle, nullptr);
2434     cudaStreamDestroy(stream);
2435     return bench_hgemm_tflops(M, N, K, time_ms);
2436 }
2437
2438 // =====
2439 // Bench: HGEMM MMA (basic m16n8k16 + multistage pipeline)
2440 // =====
2441 template <int kStages, int kBlockSwizzle>
2442 static bool launch_timed_hgemm_mma(
2443     half *d_a, half *d_b_t, half *d_c, half *h_c,
2444     half *h_c_ref, int M, int N, int K, size_t size_c,
2445     cudaEvent_t start, cudaEvent_t stop, float &max_err,
2446     float &time_ms
2447 ) {
2448     constexpr int BM = 128, BN = 128, BK = 16;
2449     using Kernel = void (*)(half *, half *, half *, int, int, int);
2450     Kernel k = hgemm_mma_stages_tn<16, 8, 16, 2, 4, 4, 4, kStages, kBlockSwizzle>;
2451     size_t smem = kStages * (BM * BK + BN * BK) * sizeof(half);
2452     if (!check_smem_feasible((const void *)k, smem)) return false;
2453     dim3 block(256);
2454     const int tiles_n = (N + BN - 1) / BN;
2455     constexpr int kSwizzleN = 16;
2456     int gx = kBlockSwizzle ? div_ceil(tiles_n, kSwizzleN) : tiles_n;
2457     int gz = kBlockSwizzle ? kSwizzleN : 1;
2458     dim3 grid(gx, (M + BM - 1) / BM, gz);
2459     for (int w = 0; w < g_warmup; w++)
2460         k<<<grid, block, smem>>>(d_a, d_b_t, d_c, M, N, K);
2461     cudaDeviceSynchronize();
2462     cudaEventRecord(start);
2463     for (int r = 0; r < g_repeat; r++)
2464         k<<<grid, block, smem>>>(d_a, d_b_t, d_c, M, N, K);
2465     cudaEventRecord(stop);
2466     cudaEventSynchronize(stop);
2467     cudaEventElapsedTime(&time_ms, start, stop);
2468     time_ms /= g_repeat;
2469     check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "bench mma D2H");
2470     max_err = 0.0f;
2471     for (int i = 0; i < M * N; i++) {
2472         float err = fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i]));
2473         if (err > max_err) max_err = err;
2474     }
2475     return true;
2476 }
2477
2478 static void bench_hgemm_mma(int M, int N, int K) {

```

```

2479 size_t size_a = (size_t)M * K * sizeof(half);
2480 size_t size_b = (size_t)K * N * sizeof(half);
2481 size_t size_c = (size_t)M * N * sizeof(half);
2482 half *h_a = (half *)malloc(size_a);
2483 half *h_b = (half *)malloc(size_b);
2484 half *h_c_ref = (half *)malloc(size_c);
2485 srand(42);
2486 for (int i = 0; i < M * K; i++)
2487     h_a[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2488 for (int i = 0; i < K * N; i++)
2489     h_b[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2490 size_t size_b_t = (size_t)N * K * sizeof(half);
2491 half *h_b_t = (half *)malloc(size_b_t);
2492 for (int n = 0; n < N; n++)
2493     for (int k = 0; k < K; k++)
2494         h_b_t[n * K + k] = h_b[k * N + n];
2495 half *d_a, *d_b, *d_b_t, *d_c;
2496 check(cudaMalloc(&d_a, size_a), "bench mma alloc A");
2497 check(cudaMalloc(&d_b, size_b), "bench mma alloc B");
2498 check(cudaMalloc(&d_b_t, size_b_t), "bench mma alloc B_t");
2499 check(cudaMalloc(&d_c, size_c), "bench mma alloc C");
2500 check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "bench mma H2D A");
2501 check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "bench mma H2D B");
2502 check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice), "bench mma H2D B_t");
2503 cublasHandle_t handle;
2504 cublasCreate(&handle);
2505 half alpha_h = __float2half(1.0f), beta_h = __float2half(0.0f);
2506 cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha_h, d_b, CUDA_R_16F, N,
2507             d_a, CUDA_R_16F, K, &beta_h, d_c, CUDA_R_16F, N, CUBLAS_COMPUTE_16F,
2508             CUBLAS_GEMM_DEFAULT);
2509 check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost), "bench mma D2H ref");
2510 half *h_c = (half *)malloc(size_c);
2511 float cublas_tflops = bench_cublas_hgemm_tflops(handle, M, N, K, d_a, d_b, d_c);
2512 cudaEvent_t start, stop;
2513 cudaEventCreate(&start);
2514 cudaEventCreate(&stop);
2515 for (int stages : {2, 3}) {
2516     for (int swizzle : {0, 1}) {
2517         float max_err = 0, time_ms = 0;
2518         bool ok = false;
2519         if (stages == 2)
2520             ok = swizzle ? launch_timed_hgemm_mma<2, 1>(d_a, d_b_t, d_c, h_c, h_c_ref, M, N,
2521                 K, size_c, start, stop, max_err, time_ms)
2522                 : launch_timed_hgemm_mma<2, 0>(d_a, d_b_t, d_c, h_c, h_c_ref, M, N,
2523                 K, size_c, start, stop, max_err, time_ms);
2524         else
2525             ok = swizzle ? launch_timed_hgemm_mma<3, 1>(d_a, d_b_t, d_c, h_c, h_c_ref, M, N,
2526                 K, size_c, start, stop, max_err, time_ms)
2527                 : launch_timed_hgemm_mma<3, 0>(d_a, d_b_t, d_c, h_c, h_c_ref, M, N,
2528                 K, size_c, start, stop, max_err, time_ms);
2529         char label[64];
2530         snprintf(label, sizeof(label), "HGEMM MMA (S=%d, BLK_SW=%d)", stages, swizzle);
2531         if (!ok) {
2532             if (g_debug)
2533                 printf("| %-56s | %-9s | %-19s |\n", label, "SMEM SKIP", "None");
2534         } else {
2535             float tflops = bench_hgemm_tflops(M, N, K, time_ms);
2536             char tflops_str[32];
2537             snprintf(tflops_str, sizeof(tflops_str), "%.1f/%.1f (%.2fx)",
2538                 tflops, cublas_tflops, tflops / cublas_tflops);
2539             printf("| %-56s | %.3e | %-19s |\n", label, max_err,
2540                 tflops_str);
2541         }

```

```

2542     }
2543 }
2544 cudaEventDestroy(start);
2545 cudaEventDestroy(stop);
2546 free(h_a);
2547 free(h_b);
2548 free(h_b_t);
2549 free(h_c);
2550 free(h_c_ref);
2551 cudaFree(d_a);
2552 cudaFree(d_b);
2553 cudaFree(d_b_t);
2554 cudaFree(d_c);
2555 cublasDestroy(handle);
2556 }
2557
2558 // =====
2559 // Bench: HGEMM MMA Swizzle + Register Double Buffering (kValTileK=4, BK=64)
2560 // =====
2561 template <int kStages, int kBlockSwizzle>
2562 static bool launch_timed_hgemm_swizzle(
2563     half *d_a, half *d_b_t, half *d_c, half *h_c,
2564     half *h_c_ref, int M, int N, int K, size_t size_c,
2565     cudaEvent_t start, cudaEvent_t stop, float &max_err,
2566     float &time_ms
2567 ) {
2568     constexpr int BM = 128, BN = 128, BK = 64;
2569     using Kernel = void (*)(half *, half *, half *, int, int, int);
2570     Kernel k = hgemm_mma_stages_tn_swizzle<16, 8, 16, 2, 4, 4, 4, 4, kStages, kBlockSwizzle>;
2571     size_t smem = kStages * (BM * BK + BN * BK) * sizeof(half);
2572     if (!check_smem_feasible((const void *)k, smem)) return false;
2573     cudaFuncSetAttribute((const void *)k, cudaFuncAttributeMaxDynamicSharedMemorySize, smem);
2574     dim3 block(256);
2575     const int tiles_n = (N + BN - 1) / BN;
2576     constexpr int kSwizzleN = 16;
2577     int gx = kBlockSwizzle ? div_ceil(tiles_n, kSwizzleN) : tiles_n;
2578     int gz = kBlockSwizzle ? kSwizzleN : 1;
2579     dim3 grid(gx, (M + BM - 1) / BM, gz);
2580     for (int w = 0; w < g_warmup; w++)
2581         k<<<grid, block, smem>>>(d_a, d_b_t, d_c, M, N, K);
2582     cudaDeviceSynchronize();
2583     cudaEventRecord(start);
2584     for (int r = 0; r < g_repeat; r++)
2585         k<<<grid, block, smem>>>(d_a, d_b_t, d_c, M, N, K);
2586     cudaEventRecord(stop);
2587     cudaEventSynchronize(stop);
2588     cudaEventElapsedTime(&time_ms, start, stop);
2589     time_ms /= g_repeat;
2590     check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "bench swizzle D2H");
2591     max_err = 0.0f;
2592     for (int i = 0; i < M * N; i++) {
2593         float err = fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i]));
2594         if (err > max_err) max_err = err;
2595     }
2596     return true;
2597 }
2598
2599 static void bench_hgemm_swizzle(int M, int N, int K) {
2600     size_t size_a = (size_t)M * K * sizeof(half);
2601     size_t size_b = (size_t)K * N * sizeof(half);
2602     size_t size_c = (size_t)M * N * sizeof(half);
2603     half *h_a = (half *)malloc(size_a);
2604     half *h_b = (half *)malloc(size_b);

```

```

2605     half *h_c_ref = (half *)malloc(size_c);
2606     srand(42);
2607     for (int i = 0; i < M * K; i++)
2608         h_a[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2609     for (int i = 0; i < K * N; i++)
2610         h_b[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2611     size_t size_b_t = (size_t)N * K * sizeof(half);
2612     half *h_b_t = (half *)malloc(size_b_t);
2613     for (int n = 0; n < N; n++)
2614         for (int k = 0; k < K; k++)
2615             h_b_t[n * K + k] = h_b[k * N + n];
2616     half *d_a, *d_b, *d_b_t, *d_c;
2617     check(cudaMalloc(&d_a, size_a), "bench swizzle alloc A");
2618     check(cudaMalloc(&d_b, size_b), "bench swizzle alloc B");
2619     check(cudaMalloc(&d_b_t, size_b_t), "bench swizzle alloc B_t");
2620     check(cudaMalloc(&d_c, size_c), "bench swizzle alloc C");
2621     check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "bench swizzle H2D A");
2622     check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "bench swizzle H2D B");
2623     check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice), "bench swizzle H2D B_t");
2624     cublasHandle_t handle;
2625     cublasCreate(&handle);
2626     half alpha_h = __float2half(1.0f), beta_h = __float2half(0.0f);
2627     cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha_h, d_b, CUDA_R_16F, N,
2628         d_a, CUDA_R_16F, K, &beta_h, d_c, CUDA_R_16F, N, CUBLAS_COMPUTE_16F,
2629         CUBLAS_GEMM_DEFAULT);
2630     check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost), "bench swizzle D2H ref");
2631     half *h_c = (half *)malloc(size_c);
2632     float cublas_tflops = bench_cublas_hgemm_tflops(handle, M, N, K, d_a, d_b, d_c);
2633     cudaEvent_t start, stop;
2634     cudaEventCreate(&start);
2635     cudaEventCreate(&stop);
2636     for (int stages : {2, 3}) {
2637         for (int swizzle : {0, 1}) {
2638             float max_err = 0, time_ms = 0;
2639             bool ok = false;
2640             if (stages == 2)
2641                 ok = swizzle ? launch_timed_hgemm_swizzle<2, 1>(d_a, d_b_t, d_c, h_c, h_c_ref, M,
2642                     N, K, size_c, start, stop, max_err, time_ms)
2643                     : launch_timed_hgemm_swizzle<2, 0>(d_a, d_b_t, d_c, h_c, h_c_ref, M,
2644                     N, K, size_c, start, stop, max_err, time_ms);
2645             else
2646                 ok = swizzle ? launch_timed_hgemm_swizzle<3, 1>(d_a, d_b_t, d_c, h_c, h_c_ref, M,
2647                     N, K, size_c, start, stop, max_err, time_ms)
2648                     : launch_timed_hgemm_swizzle<3, 0>(d_a, d_b_t, d_c, h_c, h_c_ref, M,
2649                     N, K, size_c, start, stop, max_err, time_ms);
2650             char label[64];
2651             snprintf(label, sizeof(label), "HGEMM Swizzle+Reg2x (S=%d, BLK_SW=%d)", stages, swizzle);
2652             if (!ok) {
2653                 if (g_debug)
2654                     printf("| %-56s | %-9s | %-19s |\n", label, "SMEM SKIP", "None");
2655             } else {
2656                 float tflops = bench_hgemm_tflops(M, N, K, time_ms);
2657                 char tflops_str[32];
2658                 snprintf(tflops_str, sizeof(tflops_str), "%.1f/%.1f (%.2fx)",
2659                     tflops, cublas_tflops, tflops / cublas_tflops);
2660                 printf("| %-56s | %-3e | %-19s |\n", label, max_err,
2661                     tflops_str);
2662             }
2663         }
2664     }
2665     cudaEventDestroy(start);
2666     cudaEventDestroy(stop);
2667     free(h_a);

```

```

2668     free(h_b);
2669     free(h_b_t);
2670     free(h_c);
2671     free(h_c_ref);
2672     cudaFree(d_a);
2673     cudaFree(d_b);
2674     cudaFree(d_b_t);
2675     cudaFree(d_c);
2676     cublasDestroy(handle);
2677 }
2678
2679 #if defined(NOTES_V2_ENABLE_CUTE)
2680 // =====
2681 // Bench: HGEMM CuTe
2682 // =====
2683 static void bench_hgemm_cute(int M, int N, int K) {
2684     size_t size_a = (size_t)M * K * sizeof(half);
2685     size_t size_b = (size_t)K * N * sizeof(half);
2686     size_t size_c = (size_t)M * N * sizeof(half);
2687     half *h_a = (half *)malloc(size_a);
2688     half *h_b = (half *)malloc(size_b);
2689     half *h_c_ref = (half *)malloc(size_c);
2690     srand(42);
2691     for (int i = 0; i < M * K; i++)
2692         h_a[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2693     for (int i = 0; i < K * N; i++)
2694         h_b[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2695     size_t size_b_t = (size_t)N * K * sizeof(half);
2696     half *h_b_t = (half *)malloc(size_b_t);
2697     for (int n = 0; n < N; n++)
2698         for (int k = 0; k < K; k++)
2699             h_b_t[n * K + k] = h_b[k * N + n];
2700     half *d_a, *d_b, *d_b_t, *d_c;
2701     check(cudaMalloc(&d_a, size_a), "bench cute alloc A");
2702     check(cudaMalloc(&d_b, size_b), "bench cute alloc B");
2703     check(cudaMalloc(&d_b_t, size_b_t), "bench cute alloc B_t");
2704     check(cudaMalloc(&d_c, size_c), "bench cute alloc C");
2705     check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "bench cute H2D A");
2706     check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "bench cute H2D B");
2707     check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice), "bench cute H2D B_t");
2708     cublasHandle_t handle;
2709     cublasCreate(&handle);
2710     half alpha_h = __float2half(1.0f), beta_h = __float2half(0.0f);
2711     cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha_h, d_b, CUDA_R_16F, N,
2712                 d_a, CUDA_R_16F, K, &beta_h, d_c, CUDA_R_16F, N, CUBLAS_COMPUTE_16F,
2713                 CUBLAS_GEMM_DEFAULT);
2714     check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost), "bench cute D2H ref");
2715     half *h_c = (half *)malloc(size_c);
2716     float cublas_tflops = bench_cublas_hgemm_tflops(handle, M, N, K, d_a, d_b, d_c);
2717     cudaEvent_t start, stop;
2718     cudaEventCreate(&start);
2719     cudaEventCreate(&stop);
2720     for (int stages : {2, 3}) {
2721         for (int swizzle : {0, 1}) {
2722             char label[64];
2723             snprintf(label, sizeof(label), "HGEMM CuTe Swizzle (S=%d, BLK_SW=%d)", stages, swizzle);
2724             bool ok = false;
2725             float time_ms = 0, max_err = 0;
2726             if (stages == 2) {
2727                 auto k = swizzle ? launch_hgemm_mma_stages_tn_cute<half, 2, 1>
2728                                     : launch_hgemm_mma_stages_tn_cute<half, 2, 0>;
2729                 for (int w = 0; w < g_warmup; w++) k(d_a, d_b_t, d_c, M, N, K);
2730                 cudaDeviceSynchronize();

```

```

2731     cudaEventRecord(start);
2732     for (int r = 0; r < g_repeat; r++) k(d_a, d_b_t, d_c, M, N, K);
2733     cudaEventRecord(stop);
2734     cudaEventSynchronize(stop);
2735     cudaEventElapsedTime(&time_ms, start, stop);
2736     time_ms /= g_repeat;
2737     ok = true;
2738 } else {
2739     auto k = swizzle ? launch_hgemm_mma_stages_tn_cute<half, 3, 1>
2740                     : launch_hgemm_mma_stages_tn_cute<half, 3, 0>;
2741     for (int w = 0; w < g_warmup; w++) k(d_a, d_b_t, d_c, M, N, K);
2742     cudaDeviceSynchronize();
2743     cudaEventRecord(start);
2744     for (int r = 0; r < g_repeat; r++) k(d_a, d_b_t, d_c, M, N, K);
2745     cudaEventRecord(stop);
2746     cudaEventSynchronize(stop);
2747     cudaEventElapsedTime(&time_ms, start, stop);
2748     time_ms /= g_repeat;
2749     ok = true;
2750 }
2751 if (ok) {
2752     check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "bench cute D2H");
2753     max_err = 0.0f;
2754     for (int i = 0; i < M * N; i++) {
2755         float err = fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i]));
2756         if (err > max_err) max_err = err;
2757     }
2758     float tflops = bench_hgemm_tflops(M, N, K, time_ms);
2759     char tflops_str[32];
2760     snprintf(tflops_str, sizeof(tflops_str), "%.1f/%.1f (%.2fx)",
2761              tflops, cublas_tflops, tflops / cublas_tflops);
2762     printf("| %-56s | %.3e | %-19s |\n", label, max_err,
2763          tflops_str);
2764 } else {
2765     printf("| %-56s | %-9s | %-19s |\n", label, "LAUNCH ERR", "None");
2766 }
2767 }
2768 }
2769 cudaEventDestroy(start);
2770 cudaEventDestroy(stop);
2771 free(h_a);
2772 free(h_b);
2773 free(h_b_t);
2774 free(h_c);
2775 free(h_c_ref);
2776 cudaFree(d_a);
2777 cudaFree(d_b);
2778 cudaFree(d_b_t);
2779 cudaFree(d_c);
2780 cublasDestroy(handle);
2781 }
2782 #endif
2783
2784 #if defined(NOTES_V2_ENABLE_WGMMMA)
2785 // =====
2786 // Bench: HGEMM WGMMMA (m64n128k16 + TMA + Warp Specialization, SM90+)
2787 // =====
2788 template <int kStages, int kBlockSwizzle>
2789 static bool launch_timed_hgemm_wgmma(half *d_c, half *h_c, half *h_c_ref,
2790                                     CUTensorMap *tma_a, CUTensorMap *tma_b,
2791                                     int M, int N, int K, size_t size_c,
2792                                     cudaEvent_t start, cudaEvent_t stop,
2793                                     float &max_err, float &time_ms) {

```

```

2794 constexpr int BM = 128, BN = 128, BK = 64, kNumThreads = 256;
2795 using Kernel = void (*)(int, int, int, half *, const CUTensorMap *,
2796     const CUTensorMap *);
2797 Kernel k = hgemm_wgmma_stages_tn<64, 128, 16, BM, BN, BK, kNumThreads, kStages,
2798     kBlockSwizzle>;
2799 size_t smem = kStages * (BM * BK + BN * BK) * sizeof(half);
2800 if (!check_smem_feasible((const void *)k, smem)) return false;
2801 cudaFuncSetAttribute(k, cudaFuncAttributeMaxDynamicSharedMemorySize, smem);
2802 dim3 block(kNumThreads);
2803 const int tiles_n = (N + BN - 1) / BN;
2804 constexpr int kSwizzleN = 16;
2805 int gx = kBlockSwizzle ? div_ceil(tiles_n, kSwizzleN) : tiles_n;
2806 int gz = kBlockSwizzle ? kSwizzleN : 1;
2807 dim3 grid(gx, (M + BM - 1) / BM, gz);
2808 for (int w = 0; w < g_warmup; w++)
2809     k<<<grid, block, smem>>>(M, N, K, d_c, tma_a, tma_b);
2810 cudaDeviceSynchronize();
2811 cudaEventRecord(start);
2812 for (int r = 0; r < g_repeat; r++)
2813     k<<<grid, block, smem>>>(M, N, K, d_c, tma_a, tma_b);
2814 cudaEventRecord(stop);
2815 cudaEventSynchronize(stop);
2816 cudaEventElapsedTime(&time_ms, start, stop);
2817 time_ms /= g_repeat;
2818 check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "bench wgmma D2H");
2819 max_err = 0.0f;
2820 for (int i = 0; i < M * N; i++) {
2821     float err = fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i]));
2822     if (err > max_err) max_err = err;
2823 }
2824 return true;
2825 }
2826
2827 static void bench_hgemm_wgmma(int M, int N, int K) {
2828     constexpr int BM = 128, BN = 128, BK = 64;
2829     if (M % BM != 0 || N % BN != 0 || K % BK != 0) {
2830         printf("| %-56s | %-9s | %-19s |\n", "HGEMM WGMMMA (unaligned)", "SKIP", "None");
2831         return;
2832     }
2833     size_t size_a = (size_t)M * K * sizeof(half);
2834     size_t size_b = (size_t)K * N * sizeof(half);
2835     size_t size_c = (size_t)M * N * sizeof(half);
2836     half *h_a = (half *)malloc(size_a);
2837     half *h_b = (half *)malloc(size_b);
2838     half *h_c_ref = (half *)malloc(size_c);
2839     srand(42);
2840     for (int i = 0; i < M * K; i++)
2841         h_a[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2842     for (int i = 0; i < K * N; i++)
2843         h_b[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
2844     size_t size_b_t = (size_t)N * K * sizeof(half);
2845     half *h_b_t = (half *)malloc(size_b_t);
2846     for (int n = 0; n < N; n++)
2847         for (int k = 0; k < K; k++)
2848             h_b_t[n * K + k] = h_b[k * N + n];
2849     half *d_a, *d_b, *d_b_t, *d_c;
2850     check(cudaMalloc(&d_a, size_a), "bench wgmma alloc A");
2851     check(cudaMalloc(&d_b, size_b), "bench wgmma alloc B");
2852     check(cudaMalloc(&d_b_t, size_b_t), "bench wgmma alloc B_t");
2853     check(cudaMalloc(&d_c, size_c), "bench wgmma alloc C");
2854     check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "bench wgmma H2D A");
2855     check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "bench wgmma H2D B");
2856     check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice), "bench wgmma H2D B_t");

```

```

2857 cublasHandle_t handle;
2858 cublasCreate(&handle);
2859 half alpha_h = __float2half(1.0f), beta_h = __float2half(0.0f);
2860 cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha_h, d_b, CUDA_R_16F, N,
2861             d_a, CUDA_R_16F, K, &beta_h, d_c, CUDA_R_16F, N, CUBLAS_COMPUTE_16F,
2862             CUBLAS_GEMM_DEFAULT);
2863 check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost), "bench wgmma D2H ref");
2864 CUsensorMap *tma_a = allocate_and_create_tensor_map(d_a, M / BM, K / BK);
2865 CUsensorMap *tma_b = allocate_and_create_tensor_map(d_b_t, N / BN, K / BK);
2866 half *h_c = (half *)malloc(size_c);
2867 float cublas_tflops = bench_cublas_hgemm_tflops(handle, M, N, K, d_a, d_b, d_c);
2868 cudaEvent_t start, stop;
2869 cudaEventCreate(&start);
2870 cudaEventCreate(&stop);
2871 for (int stages : {1, 2, 3}) {
2872     for (int swizzle : {0, 1}) {
2873         float max_err = 0, time_ms = 0;
2874         bool ok = false;
2875         if (stages == 2)
2876             ok = swizzle
2877                 ? launch_timed_hgemm_wgmma<2, 1>(d_c, h_c, h_c_ref, tma_a, tma_b, M, N,
2878                 K, size_c, start, stop, max_err, time_ms)
2879                 : launch_timed_hgemm_wgmma<2, 0>(d_c, h_c, h_c_ref, tma_a, tma_b, M, N, K,
2880                 size_c, start, stop, max_err, time_ms);
2881         else
2882             ok = swizzle
2883                 ? launch_timed_hgemm_wgmma<3, 1>(d_c, h_c, h_c_ref, tma_a, tma_b, M, N,
2884                 K, size_c, start, stop, max_err, time_ms)
2885                 : launch_timed_hgemm_wgmma<3, 0>(d_c, h_c, h_c_ref, tma_a, tma_b, M, N, K,
2886                 size_c, start, stop, max_err, time_ms);
2887         char label[64];
2888         snprintf(label, sizeof(label), "HGEMM TMA WGMMA WS (S=%d, BLK_SW=%d)", stages, swizzle);
2889         if (!ok) {
2890             if (g_debug)
2891                 printf("| %-56s | %-9s | %-19s |\n", label, "SMEM SKIP", "None");
2892         } else {
2893             float tflops = bench_hgemm_tflops(M, N, K, time_ms);
2894             char tflops_str[32];
2895             snprintf(tflops_str, sizeof(tflops_str), "%.1f/%.1f (%.2fx)", tflops,
2896                     cublas_tflops, tflops / cublas_tflops);
2897             printf("| %-56s | %.3e | %-19s |\n", label, max_err,
2898                     tflops_str);
2899         }
2900     }
2901 }
2902 cudaEventDestroy(start);
2903 cudaEventDestroy(stop);
2904 free(h_a);
2905 free(h_b);
2906 free(h_b_t);
2907 free(h_c);
2908 free(h_c_ref);
2909 cudaFree(d_a);
2910 cudaFree(d_b);
2911 cudaFree(d_b_t);
2912 cudaFree(d_c);
2913 cudaFree(tma_a);
2914 cudaFree(tma_b);
2915 cublasDestroy(handle);
2916 }
2917 #endif
2918
2919 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)

```

```

2920 // =====
2921 // Bench: HGEMM TMA MMA WS (mma.sync + TMA + Warp Specialization, SM120)
2922 // =====
2923 template <int kStages, int kBlockSwizzle>
2924 static bool bench_launch_tma_mma_ws(
2925     int M, int N, int K, half *d_a, half *d_b_t,
2926     half *d_c, half *h_c, half *h_c_ref, size_t size_c,
2927     CUTensorMap *tma_a, CUTensorMap *tma_b,
2928     cudaEvent_t start, cudaEvent_t stop,
2929     float &max_err, float &time_ms) {
2930     constexpr int BM = 128, BN = 128, BK = 64, kNumThreads = 256;
2931     constexpr size_t payload_bytes = kStages * (BM * BK + BN * BK) * sizeof(half);
2932     using Kernel = void (*)(int, int, int, half *, const CUTensorMap *,
2933         const CUTensorMap *);
2934     Kernel kernel = hgemm_tma_mma_ws_tn<16, 8, 16, 2, 2, 4, 8, 4, kStages,
2935         kNumThreads, kBlockSwizzle>;
2936     if (!check_smem_feasible((const void *)kernel, payload_bytes)) return false;
2937     cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
2938         payload_bytes);
2939     dim3 block(kNumThreads);
2940     constexpr int kSwizzleN = 16;
2941     const int n_tiles = N / BN;
2942     const int grid_x = kBlockSwizzle ? div_ceil(n_tiles, kSwizzleN) : n_tiles;
2943     const int grid_z = kBlockSwizzle ? kSwizzleN : 1;
2944     dim3 grid(grid_x, M / BM, grid_z);
2945     for (int w = 0; w < g_warmup; w++)
2946         kernel<<<grid, block, payload_bytes>>>(M, N, K, d_c, tma_a, tma_b);
2947     cudaDeviceSynchronize();
2948     cudaEventRecord(start);
2949     for (int r = 0; r < g_repeat; r++)
2950         kernel<<<grid, block, payload_bytes>>>(M, N, K, d_c, tma_a, tma_b);
2951     cudaEventRecord(stop);
2952     cudaEventSynchronize(stop);
2953     cudaEventElapsedTime(&time_ms, start, stop);
2954     time_ms /= g_repeat;
2955     check(cudaMemcpy(h_c, d_c, size_c, cudaMemcpyDeviceToHost), "bench tma mma ws D2H");
2956     max_err = 0.0f;
2957     for (int i = 0; i < M * N; ++i)
2958         max_err = fmaxf(max_err, fabsf(__half2float(h_c[i]) - __half2float(h_c_ref[i])));
2959     return true;
2960 }
2961
2962 static void bench_hgemm_tma_mma_ws(int M, int N, int K) {
2963     constexpr int BM = 128, BN = 128, BK = 64;
2964     if (M % BM != 0 || N % BN != 0 || K % BK != 0) {
2965         printf("| %-56s | %-9s | %-19s |\n", "HGEMM TMA MMA WS (unaligned)", "SKIP", "None");
2966         return;
2967     }
2968     const size_t size_a = size_t(M) * K * sizeof(half);
2969     const size_t size_b = size_t(K) * N * sizeof(half);
2970     const size_t size_b_t = size_t(N) * K * sizeof(half);
2971     const size_t size_c = size_t(M) * N * sizeof(half);
2972     half *h_a = static_cast<half *>(malloc(size_a));
2973     half *h_b = static_cast<half *>(malloc(size_b));
2974     half *h_b_t = static_cast<half *>(malloc(size_b_t));
2975     half *h_c = static_cast<half *>(malloc(size_c));
2976     half *h_c_ref = static_cast<half *>(malloc(size_c));
2977     srand(42);
2978     for (int i = 0; i < M * K; ++i)
2979         h_a[i] = __float2half((float(rand()) / RAND_MAX) * 2.0f - 1.0f);
2980     for (int i = 0; i < K * N; ++i)
2981         h_b[i] = __float2half((float(rand()) / RAND_MAX) * 2.0f - 1.0f);
2982     for (int n = 0; n < N; ++n)

```

```

2983     for (int k = 0; k < K; ++k)
2984         h_b_t[n * K + k] = h_b[k * N + n];
2985     half *d_a, *d_b, *d_b_t, *d_c;
2986     check(cudaMalloc(&d_a, size_a), "bench tma mma ws alloc A");
2987     check(cudaMalloc(&d_b, size_b), "bench tma mma ws alloc B");
2988     check(cudaMalloc(&d_b_t, size_b_t), "bench tma mma ws alloc B_t");
2989     check(cudaMalloc(&d_c, size_c), "bench tma mma ws alloc C");
2990     check(cudaMemcpy(d_a, h_a, size_a, cudaMemcpyHostToDevice), "bench tma mma ws H2D A");
2991     check(cudaMemcpy(d_b, h_b, size_b, cudaMemcpyHostToDevice), "bench tma mma ws H2D B");
2992     check(cudaMemcpy(d_b_t, h_b_t, size_b_t, cudaMemcpyHostToDevice),
2993         "bench tma mma ws H2D B_t");
2994     cublasHandle_t handle;
2995     cublasCreate(&handle);
2996     half alpha = __float2half(1.0f), beta = __float2half(0.0f);
2997     cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K, &alpha, d_b, CUDA_R_16F, N,
2998         d_a, CUDA_R_16F, K, &beta, d_c, CUDA_R_16F, N, CUBLAS_COMPUTE_16F,
2999         CUBLAS_GEMM_DEFAULT);
3000     check(cudaMemcpy(h_c_ref, d_c, size_c, cudaMemcpyDeviceToHost),
3001         "bench tma mma ws D2H reference");
3002     CUTensorMap *tma_a = allocate_and_create_tensor_map(d_a, M / BM, K / BK);
3003     CUTensorMap *tma_b = allocate_and_create_tensor_map(d_b_t, N / BN, K / BK);
3004     float cublas_tflops = bench_cublas_hgemm_tflops(handle, M, N, K, d_a, d_b, d_c);
3005     cudaEvent_t start, stop;
3006     cudaEventCreate(&start);
3007     cudaEventCreate(&stop);
3008     for (int stages : {1, 2, 3}) {
3009         for (int swizzle : {0, 1}) {
3010             float max_err = 0, time_ms = 0;
3011             bool ok = false;
3012             if (stages == 2)
3013                 ok = swizzle ? bench_launch_tma_mma_ws<2, 1>(M, N, K, d_a, d_b_t, d_c, h_c,
3014                     h_c_ref, size_c, tma_a, tma_b,
3015                     start, stop, max_err, time_ms)
3016                     : bench_launch_tma_mma_ws<2, 0>(M, N, K, d_a, d_b_t, d_c, h_c,
3017                     h_c_ref, size_c, tma_a, tma_b,
3018                     start, stop, max_err, time_ms);
3019             else
3020                 ok = swizzle ? bench_launch_tma_mma_ws<3, 1>(M, N, K, d_a, d_b_t, d_c, h_c,
3021                     h_c_ref, size_c, tma_a, tma_b,
3022                     start, stop, max_err, time_ms)
3023                     : bench_launch_tma_mma_ws<3, 0>(M, N, K, d_a, d_b_t, d_c, h_c,
3024                     h_c_ref, size_c, tma_a, tma_b,
3025                     start, stop, max_err, time_ms);
3026             char label[64];
3027             snprintf(label, sizeof(label), "HGEMM TMA MMA WS (S=%d, BLK_SW=%d)", stages, swizzle);
3028             if (!ok) {
3029                 if (g_debug)
3030                     printf("| %-56s | %-9s | %-19s |\n", label, "SMEM SKIP", "None");
3031             } else {
3032                 float tflops = bench_hgemm_tflops(M, N, K, time_ms);
3033                 char tflops_str[32];
3034                 snprintf(tflops_str, sizeof(tflops_str), "%.1f/%.1f (%.2fx)", tflops,
3035                     cublas_tflops, tflops / cublas_tflops);
3036                 printf("| %-56s | %-3e | %-19s |\n", label, max_err,
3037                     tflops_str);
3038             }
3039         }
3040     }
3041     cudaEventDestroy(start);
3042     cudaEventDestroy(stop);
3043     free(h_a);
3044     free(h_b);
3045     free(h_b_t);

```

```

3046 free(h_c);
3047 free(h_c_ref);
3048 cudaFree(d_a);
3049 cudaFree(d_b);
3050 cudaFree(d_b_t);
3051 cudaFree(d_c);
3052 cudaFree(tma_a);
3053 cudaFree(tma_b);
3054 cublasDestroy(handle);
3055 }
3056 #endif
3057
3058 // =====
3059 // Bench: FlashAttention-2 Split-Q (template on kHeadDim for dispatch)
3060 // =====
3061 template <int kHeadDim, int kStagesK = 2, int kPadQ = 8, int kPadK = 8,
3062         int kPadV = 8, int kMmaAccF32 = 0>
3063 static void bench_fa_launch(int B, int H, int seqlen, int head_dim,
3064         half *h_o_ref, float *ref_o, half *d_q, half *d_k, half *d_v, half *d_o,
3065         float cudnn_tflops_f16) {
3066     static_assert(kHeadDim == 64 || kHeadDim == 128, "Only D=64 and D=128 are supported");
3067     constexpr bool kSwizzleQ = kPadQ == 0;
3068     constexpr bool kSwizzleK = kPadK == 0;
3069     constexpr bool kSwizzleV = kPadV == 0;
3070     constexpr int kMmaAtomM = 16;
3071     constexpr int kMmaAtomN = 8;
3072     constexpr int kMmaAtomK = 16;
3073     constexpr int kMmaTileSeqLenQ = 8;
3074     constexpr int kMmaTileSeqLenK = 1;
3075     constexpr int kMmaTileSeqLenP = 8;
3076     constexpr int kMmaTileHeadDimV = 1;
3077     constexpr int kValTileSeqLenQ = 1;
3078     constexpr int kValTileSeqLenK = 8;
3079     constexpr int kValTileSeqLenP = 1;
3080     constexpr int kValTileHeadDimV = kHeadDim / (8 * kMmaTileHeadDimV);
3081     constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ;
3082     constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK;
3083     constexpr const char *layout_name =
3084         kSwizzleQ && kSwizzleK && kSwizzleV ? "Swizzle" :
3085         kSwizzleQ && kSwizzleK ? "SwizzleQK" :
3086         kSwizzleQ && kSwizzleV ? "SwizzleQV" :
3087         kSwizzleK && kSwizzleV ? "SwizzleKV" :
3088         kSwizzleQ ? "SwizzleQ" :
3089         kSwizzleK ? "SwizzleK" :
3090         kSwizzleV ? "SwizzleV" : "Pad";
3091
3092     // Kernel requires seqlen >= Br (tile size); skip gracefully for short seqlen
3093     if (seqlen < Br) {
3094         char label[64];
3095         snprintf(label, sizeof(label), "FA2 (%s)",
3096                 layout_name);
3097         printf("| %-56s | %-9s | %-19s |\n", label,
3098                 "seqlen<Br", "None");
3099         return;
3100     }
3101     size_t smem_bytes =
3102         (Br * (kHeadDim + kPadQ) +
3103          kStagesK * Bc * (kHeadDim + kPadK) +
3104          Bc * (kHeadDim + kPadV)) * sizeof(half);
3105
3106     dim3 block(256);
3107     dim3 grid((seqlen + Br - 1) / Br, B * H);
3108

```

```

3109 cudaStream_t timing_stream;
3110 cudaStreamCreate(&timing_stream);
3111 cudaEvent_t start, stop;
3112 cudaEventCreate(&start);
3113 cudaEventCreate(&stop);
3114
3115 using FAKernel = void (*)(half *, half *, half *, half *, int, int);
3116 FAKernel fa_k = flash_attn_mma_stages_split_q<
3117     kHeadDim, kMmaAtomM, kMmaAtomN, kMmaAtomK, kMmaAccF32,
3118     kMmaTileSeqLenQ, kMmaTileSeqLenK,
3119     kMmaTileSeqLenP, kMmaTileHeadDimV, kValTileSeqLenQ, kValTileSeqLenK,
3120     kValTileSeqLenP, kValTileHeadDimV, kStagesK, kPadQ, kPadK, kPadV>;
3121 bool smem_ok = check_smem_feasible((const void *)fa_k, smem_bytes);
3122 if (smem_ok) {
3123     cudaFuncSetAttribute(fa_k, cudaFuncAttributeMaxDynamicSharedMemorySize, smem_bytes);
3124 }
3125
3126 // Warmup
3127 if (smem_ok) {
3128     for (int w = 0; w < g_warmup; w++)
3129         fa_k<<<grid, block, smem_bytes, timing_stream>>>(d_q, d_k, d_v, d_o, seqlen, H);
3130     check(cudaStreamSynchronize(timing_stream), "fa warmup sync");
3131 }
3132
3133 // Timed repeat
3134 cudaEventRecord(start, timing_stream);
3135 if (smem_ok) {
3136     for (int r = 0; r < g_repeat; r++)
3137         fa_k<<<grid, block, smem_bytes, timing_stream>>>(d_q, d_k, d_v, d_o, seqlen, H);
3138 }
3139 cudaEventRecord(stop, timing_stream);
3140 cudaEventSynchronize(stop);
3141
3142 float time_ms = 0;
3143 cudaEventElapsedTime(&time_ms, start, stop);
3144 time_ms /= g_repeat;
3145
3146 size_t sz = (size_t)B * H * seqlen * head_dim * sizeof(half);
3147 half *h_o = (half *)malloc(sz);
3148 check(cudaMemcpy(h_o, d_o, sz, cudaMemcpyDeviceToHost), "bench fa D2H");
3149
3150 int count = B * H * seqlen * head_dim;
3151 char label[64];
3152 snprintf(label, sizeof(label), "FA2 MMA Stages (Sk=%d, %s, %s)", kStagesK,
3153     layout_name, kMmaAccF32 ? "F32Acc" : "F16Acc");
3154 float max_err = 0.0f;
3155 bool checked = h_o_ref || ref_o;
3156 if (smem_ok && checked) {
3157     for (int i = 0; i < count; i++) {
3158         float ref_val = h_o_ref ? __half2float(h_o_ref[i]) : ref_o[i];
3159         float err = fabsf(__half2float(h_o[i]) - ref_val);
3160         if (err > max_err) max_err = err;
3161     }
3162 }
3163 if (smem_ok && checked) {
3164     float tflops = bench_fa_tflops(B, H, seqlen, head_dim, time_ms);
3165     bool is_fail = max_err >= 5e-1f;
3166     if (is_fail || should_print_fa_tflops(kMmaAccF32, tflops)) {
3167         char tflops_str[32];
3168         if (cudnn_tflops_f16 > 0)
3169             snprintf(tflops_str, sizeof(tflops_str), "%.1f/%.1f (%.2fx)",
3170                 tflops, cudnn_tflops_f16, tflops / cudnn_tflops_f16);
3171         else

```

```

3172     snprintf(tflops_str, sizeof(tflops_str), "%.1f", tflops);
3173     printf("| %-56s | %.3e | %-19s |\n", label, max_err,
3174           tflops_str);
3175 }
3176 } else if (smem_ok) {
3177     float tflops = bench_fa_tflops(B, H, seqlen, head_dim, time_ms);
3178     if (should_print_fa_tflops(kMmaAccF32, tflops)) {
3179         char tflops_str[32];
3180         snprintf(tflops_str, sizeof(tflops_str), "%.1f", tflops);
3181         printf("| %-56s | %-9s | %-19s |\n", label, "unchecked", tflops_str);
3182     }
3183 } else {
3184     if (g_debug)
3185         printf("| %-56s | %-9s | %-19s |\n", label, "SMEM too large", "None");
3186 }
3187
3188 cudaEventDestroy(start);
3189 cudaEventDestroy(stop);
3190 cudaStreamDestroy(timing_stream);
3191 free(h_o);
3192 }
3193
3194 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
3195 // Bench for flash_attn_tma_mma_ws_stages_split_q (SM120, D=64/128)
3196 template <int kHeadDim, int kStagesK, int kStagesV = 1, int kMmaAccF32 = 0>
3197 static void bench_fa_tma_mma_ws_launch(int B, int H, int seqlen, int head_dim,
3198                                       half *h_o_ref, float *ref_o,
3199                                       half *d_q, half *d_k, half *d_v,
3200                                       half *d_o, float cudnn_tflops_f16) {
3201     constexpr int kMmaAtomM = 16, kMmaAtomN = 8, kMmaAtomK = 16;
3202     constexpr int kMmaTileSeqLenQ = 8, kMmaTileSeqLenK = 1;
3203     constexpr int kMmaTileSeqLenP = 8, kMmaTileHeadDimV = 1;
3204     constexpr int kValTileSeqLenQ = 1, kValTileSeqLenK = 8;
3205     constexpr int kValTileSeqLenP = 1;
3206     constexpr int kValTileHeadDimV = kHeadDim / (8 * kMmaTileHeadDimV);
3207     constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ;
3208     constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK;
3209     constexpr int kNumThreads = 384;
3210
3211     if (seqlen < Br || seqlen % Br != 0 || seqlen % Bc != 0) {
3212         char label[64];
3213         snprintf(label, sizeof(label),
3214                 "FA2 TMA MMA WS (1 Consumer WG) (Sk=%d, Sv=%d, unaligned)", kStagesK, kStagesV);
3215         printf("| %-56s | %-9s | %-19s |\n", label, "SKIP", "None");
3216         return;
3217     }
3218
3219     // TMA descriptors: box innermost 固定 64 half (128B), 满足
3220     // CU_TENSOR_MAP_SWIZZLE_128B 对 box innermost ≤ 128B 的硬约束。
3221     // D=64: blocks_width=1, 单次 TMA 覆盖整行。
3222     // D=128: blocks_width=2, kernel producer 沿 head_dim 连续发 2 次 TMA,
3223     //         minor_coord = 0/64, 写入 chunk-major smem 布局 [2, Br, 64]。
3224     // Q/K/V gmem 是 [B, H, seqlen, head_dim] row-major, 作为 2D
3225     // [B*H*seqlen, head_dim] 矩阵描述。blocks_height = B*H*seqlen/tile_major。
3226     // kernel 用 (Nb_id*H+Nh_id)*N 偏移 major_coord。
3227     constexpr int kTmaBoxMinor = 64; // box innermost = 64 half = 128B
3228     constexpr int kTmaChunks = kHeadDim / kTmaBoxMinor;
3229     CUTensorMap *tma_q = allocate_and_create_tensor_map<Br, kTmaBoxMinor>(
3230         d_q, B * H * seqlen / Br, kTmaChunks);
3231     CUTensorMap *tma_k = allocate_and_create_tensor_map<Bc, kTmaBoxMinor>(
3232         d_k, B * H * seqlen / Bc, kTmaChunks);
3233     CUTensorMap *tma_v = allocate_and_create_tensor_map<Bc, kTmaBoxMinor>(
3234         d_v, B * H * seqlen / Bc, kTmaChunks);

```

```

3235 using FAKernel = void (*)(half *, half *, half *, half *, int, int,
3236                          const CUTensorMap *, const CUTensorMap *,
3237                          const CUTensorMap *);
3238
3239 FAKernel fa_k = flash_attn_tma_mma_ws_stages_split_q<
3240     kHeadDim, kMmaAtomM, kMmaAtomN, kMmaAtomK, kMmaAccF32, kMmaTileSeqLenQ,
3241     kMmaTileSeqLenK, kMmaTileSeqLenP, kMmaTileHeadDimV, kValTileSeqLenQ,
3242     kValTileSeqLenK, kValTileSeqLenP, kValTileHeadDimV, kStagesK, kStagesV,
3243     kNumThreads>;
3244
3245 size_t smem_bytes = (Br * kHeadDim + kStagesK * Bc * kHeadDim +
3246                     kStagesV * Bc * kHeadDim) * sizeof(half);
3247 bool smem_ok = check_smem_feasible((const void *)fa_k, smem_bytes);
3248 if (smem_ok) {
3249     cudaFuncSetAttribute(fa_k, cudaFuncAttributeMaxDynamicSharedMemorySize,
3250                         smem_bytes);
3251 }
3252
3253 dim3 block(kNumThreads);
3254 dim3 grid((seqlen + Br - 1) / Br, B * H);
3255
3256 cudaStream_t timing_stream;
3257 cudaStreamCreate(&timing_stream);
3258 cudaEvent_t start, stop;
3259 cudaEventCreate(&start);
3260 cudaEventCreate(&stop);
3261
3262 if (smem_ok) {
3263     for (int w = 0; w < g_warmup; ++w)
3264         fa_k<<<grid, block, smem_bytes, timing_stream>>>(d_q, d_k, d_v, d_o,
3265                                                         seqlen, H, tma_q, tma_k,
3266                                                         tma_v);
3267     check(cudaStreamSynchronize(timing_stream), "fa_tma_ws warmup sync");
3268 }
3269 cudaEventRecord(start, timing_stream);
3270 if (smem_ok) {
3271     for (int r = 0; r < g_repeat; ++r)
3272         fa_k<<<grid, block, smem_bytes, timing_stream>>>(d_q, d_k, d_v, d_o,
3273                                                         seqlen, H, tma_q, tma_k,
3274                                                         tma_v);
3275 }
3276 cudaEventRecord(stop, timing_stream);
3277 cudaEventSynchronize(stop);
3278
3279 float time_ms = 0;
3280 cudaEventElapsedTime(&time_ms, start, stop);
3281 time_ms /= g_repeat;
3282
3283 size_t sz = (size_t)B * H * seqlen * head_dim * sizeof(half);
3284 half *h_o = (half *)malloc(sz);
3285 check(cudaMemcpy(h_o, d_o, sz, cudaMemcpyDeviceToHost), "fa_tma_ws D2H");
3286 int count = B * H * seqlen * head_dim;
3287
3288 char label[64];
3289 snprintf(label, sizeof(label), "FA2 TMA MMA WS (1 Consumer WG) (Sk=%d, Sv=%d, %s)",
3290          kStagesK, kStagesV, kMmaAccF32 ? "F32Acc" : "F16Acc");
3291 float max_err = 0.0f;
3292 bool checked = h_o_ref || ref_o;
3293 if (smem_ok && checked) {
3294     for (int i = 0; i < count; ++i) {
3295         float ref_val = h_o_ref ? __half2float(h_o_ref[i]) : ref_o[i];
3296         float err = fabsf(__half2float(h_o[i]) - ref_val);
3297         if (err > max_err) max_err = err;

```

```

3298     }
3299 }
3300 if (smem_ok && checked) {
3301     float tflops = bench_fa_tflops(B, H, seqlen, head_dim, time_ms);
3302     bool is_fail = max_err >= 5e-1f;
3303     if (is_fail || should_print_fa_tflops(kMmaAccF32, tflops)) {
3304         char tflops_str[32];
3305         if (cudnn_tflops_f16 > 0)
3306             snprintf(tflops_str, sizeof(tflops_str), "%.1f/%.1f (%.2fx)",
3307                     tflops, cudnn_tflops_f16, tflops / cudnn_tflops_f16);
3308         else
3309             snprintf(tflops_str, sizeof(tflops_str), "%.1f", tflops);
3310         printf("| %-56s | %.3e | %-19s |\n", label, max_err,
3311             tflops_str);
3312     }
3313 } else if (smem_ok) {
3314     float tflops = bench_fa_tflops(B, H, seqlen, head_dim, time_ms);
3315     if (should_print_fa_tflops(kMmaAccF32, tflops)) {
3316         char tflops_str[32];
3317         snprintf(tflops_str, sizeof(tflops_str), "%.1f", tflops);
3318         printf("| %-56s | %-9s | %-19s |\n", label, "unchecked",
3319             tflops_str);
3320     }
3321 } else {
3322     if (g_debug)
3323         printf("| %-56s | %-9s | %-19s |\n", label, "SMEM too large",
3324             "None");
3325 }
3326
3327 cudaEventDestroy(start);
3328 cudaEventDestroy(stop);
3329 cudaStreamDestroy(timing_stream);
3330 free(h_o);
3331 cudaFree(tma_q);
3332 cudaFree(tma_k);
3333 cudaFree(tma_v);
3334 }
3335
3336 // D=64/128 dispatch wrapper for bench
3337 template <int kStagesK, int kStagesV = 1, int kMmaAccF32 = 0>
3338 static void bench_fa_tma_mma_ws_dispatch(int B, int H, int seqlen, int head_dim,
3339     half *h_o_ref, float *ref_o,
3340     half *d_q, half *d_k, half *d_v,
3341     half *d_o, float cudnn_tflops_f16) {
3342     if (head_dim == 64) {
3343         bench_fa_tma_mma_ws_launch<64, kStagesK, kStagesV, kMmaAccF32>(
3344             B, H, seqlen, head_dim, h_o_ref, ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3345     } else if (head_dim == 128) {
3346         bench_fa_tma_mma_ws_launch<128, kStagesK, kStagesV, kMmaAccF32>(
3347             B, H, seqlen, head_dim, h_o_ref, ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3348     } else {
3349         char label[64];
3350         snprintf(label, sizeof(label),
3351             "FA2 TMA MMA WS (1 Consumer WG) (Sk=%d, Sv=%d, D!=64/128)", kStagesK, kStagesV);
3352         printf("| %-56s | %-9s | %-19s |\n", label, "SKIP", "None");
3353     }
3354 }
3355
3356 // FA3-style dual-consumer bench (Br=64, per-WG independent K pipeline, Sv=1)
3357 template <int kHeadDim, int kStagesK, int kMmaAccF32 = 0>
3358 static void bench_fa_3_tma_ws_launch(int B, int H, int seqlen, int head_dim,
3359     half *h_o_ref, float *ref_o,
3360     half *d_q, half *d_k, half *d_v,

```

```

3361         half *d_o, float cudnn_tflops_f16) {
3362     constexpr int kMmaAtomM = 16, kMmaAtomN = 8, kMmaAtomK = 16;
3363     constexpr int kMmaTileSeqLenQ = 4, kMmaTileSeqLenK = 1;
3364     constexpr int kMmaTileSeqLenP = 4, kMmaTileHeadDimV = 1;
3365     constexpr int kValTileSeqLenQ = 1, kValTileSeqLenK = 8;
3366     constexpr int kValTileSeqLenP = 1;
3367     constexpr int kValTileHeadDimV = kHeadDim / (8 * kMmaTileHeadDimV);
3368     constexpr int kStagesV = 1; // per-WG V: always 1 buffer
3369     constexpr int kNumConsumerWGs = 2;
3370     constexpr int Br = kMmaAtomM * kMmaTileSeqLenQ * kValTileSeqLenQ; // 64
3371     constexpr int Bc = kMmaAtomN * kMmaTileSeqLenK * kValTileSeqLenK; // 64
3372     constexpr int kNumThreads = 384;
3373
3374     if (seqlen < Br || seqlen % Br != 0 || seqlen % Bc != 0) {
3375         char label[80];
3376         snprintf(label, sizeof(label),
3377             "FA3 TMA MMA WS (2 Consumer WG) (Sk=%d, Sv=1, unaligned)", kStagesK);
3378         printf("| %-56s | %-9s | %-19s |\n", label, "SKIP", "None");
3379         return;
3380     }
3381
3382     constexpr int kTmaBoxMinor = 64;
3383     constexpr int kTmaChunks = kHeadDim / kTmaBoxMinor;
3384     CUTensorMap *tma_q = allocate_and_create_tensor_map<Br, kTmaBoxMinor>(
3385         d_q, B * H * seqlen / Br, kTmaChunks);
3386     CUTensorMap *tma_k = allocate_and_create_tensor_map<Bc, kTmaBoxMinor>(
3387         d_k, B * H * seqlen / Bc, kTmaChunks);
3388     CUTensorMap *tma_v = allocate_and_create_tensor_map<Bc, kTmaBoxMinor>(
3389         d_v, B * H * seqlen / Bc, kTmaChunks);
3390
3391     using FAKernel = void (*)(half *, half *, half *, half *, int, int,
3392         const CUTensorMap *, const CUTensorMap *,
3393         const CUTensorMap *);
3394     FAKernel fa_k = flash_attn_3_tma_ws_stages_split_q<
3395         kHeadDim, kMmaAtomM, kMmaAtomN, kMmaAtomK, kMmaAccF32, kMmaTileSeqLenQ,
3396         kMmaTileSeqLenK, kMmaTileSeqLenP, kMmaTileHeadDimV, kValTileSeqLenQ,
3397         kValTileSeqLenK, kValTileSeqLenP, kValTileHeadDimV, kStagesK, kStagesV,
3398         kNumThreads>;
3399
3400     // smem = Q + K[2WGs * Sk * Bc*D] + V[2WGs * 1 * Bc*D]
3401     size_t smem_bytes = (Br * kHeadDim +
3402         kNumConsumerWGs * kStagesK * Bc * kHeadDim +
3403         kNumConsumerWGs * kStagesV * Bc * kHeadDim) * sizeof(half);
3404     bool smem_ok = check_smem_feasible((const void *)fa_k, smem_bytes);
3405     if (smem_ok) {
3406         cudaFuncSetAttribute(fa_k, cudaFuncAttributeMaxDynamicSharedMemorySize,
3407             smem_bytes);
3408     }
3409
3410     dim3 block(kNumThreads);
3411     dim3 grid((seqlen + Br - 1) / Br, B * H);
3412
3413     cudaStream_t timing_stream;
3414     cudaStreamCreate(&timing_stream);
3415     cudaEvent_t start, stop;
3416     cudaEventCreate(&start);
3417     cudaEventCreate(&stop);
3418
3419     if (smem_ok) {
3420         for (int w = 0; w < g_warmup; ++w)
3421             fa_k<<<grid, block, smem_bytes, timing_stream>>>(d_q, d_k, d_v, d_o,
3422                 seqlen, H, tma_q, tma_k,
3423                 tma_v);

```

```

3424     check(cudaStreamSynchronize(timing_stream), "fa3_tma_ws warmup sync");
3425 }
3426 cudaEventRecord(start, timing_stream);
3427 if (smem_ok) {
3428     for (int r = 0; r < g_repeat; ++r)
3429         fa_k<<<grid, block, smem_bytes, timing_stream>>>(d_q, d_k, d_v, d_o,
3430                                                         seqlen, H, tma_q, tma_k,
3431                                                         tma_v);
3432 }
3433 cudaEventRecord(stop, timing_stream);
3434 cudaEventSynchronize(stop);
3435
3436 float time_ms = 0;
3437 cudaEventElapsedTime(&time_ms, start, stop);
3438 time_ms /= g_repeat;
3439
3440 size_t sz = (size_t)B * H * seqlen * head_dim * sizeof(half);
3441 half *h_o = (half *)malloc(sz);
3442 check(cudaMemcpy(h_o, d_o, sz, cudaMemcpyDeviceToHost), "fa3_tma_ws D2H");
3443 int count = B * H * seqlen * head_dim;
3444
3445 char label[80];
3446 snprintf(label, sizeof(label), "FA3 TMA MMA WS (2 Consumer WG) (Sk=%d, Sv=1, %s)",
3447          kStagesK, kMmaAccF32 ? "F32Acc" : "F16Acc");
3448 float max_err = 0.0f;
3449 bool checked = h_o_ref || ref_o;
3450 if (smem_ok && checked) {
3451     for (int i = 0; i < count; ++i) {
3452         float ref_val = h_o_ref ? __half2float(h_o_ref[i]) : ref_o[i];
3453         float err = fabsf(__half2float(h_o[i]) - ref_val);
3454         if (err > max_err) max_err = err;
3455     }
3456 }
3457 if (smem_ok && checked) {
3458     float tflops = bench_fa_tflops(B, H, seqlen, head_dim, time_ms);
3459     bool is_fail = max_err >= 5e-1f;
3460     if (is_fail || should_print_fa_tflops(kMmaAccF32, tflops)) {
3461         char tflops_str[32];
3462         if (cudnn_tflops_f16 > 0)
3463             snprintf(tflops_str, sizeof(tflops_str), "%.1f/%.1f (%.2fx)", tflops,
3464                    cudnn_tflops_f16, tflops / cudnn_tflops_f16);
3465         else
3466             snprintf(tflops_str, sizeof(tflops_str), "%.1f", tflops);
3467         printf("| %-56s | %.3e | %-19s |\n", label, max_err,
3468            tflops_str);
3469     }
3470 } else if (smem_ok) {
3471     float tflops = bench_fa_tflops(B, H, seqlen, head_dim, time_ms);
3472     if (should_print_fa_tflops(kMmaAccF32, tflops)) {
3473         char tflops_str[32];
3474         snprintf(tflops_str, sizeof(tflops_str), "%.1f", tflops);
3475         printf("| %-56s | %-9s | %-19s |\n", label, "unchecked",
3476            tflops_str);
3477     }
3478 } else {
3479     if (g_debug)
3480         printf("| %-56s | %-9s | %-19s |\n", label, "SMEM too large",
3481            "None");
3482 }
3483
3484 cudaEventDestroy(start);
3485 cudaEventDestroy(stop);
3486 cudaStreamDestroy(timing_stream);

```

```

3487     free(h_o);
3488     cudaFree(tma_q);
3489     cudaFree(tma_k);
3490     cudaFree(tma_v);
3491 }
3492
3493 // D=64/128 dispatch wrapper for FA3-style bench
3494 template <int kMmaAccF32 = 0>
3495 static void bench_fa_3_tma_ws_dispatch(int B, int H, int seqlen, int head_dim,
3496                                       half *h_o_ref, float *ref_o,
3497                                       half *d_q, half *d_k, half *d_v,
3498                                       half *d_o, float cudnn_tflops_f16) {
3499     // Try stages up to 4; bench_fa_3_tma_ws_launch internally checks
3500     // cudaDevAttrMaxSharedMemoryPerBlockOptin and skips configs that
3501     // exceed the actual device limit (e.g. Hopper 224KB vs Blackwell 101KB).
3502     if (head_dim == 64) {
3503         bench_fa_3_tma_ws_launch<64, 1, kMmaAccF32>(B, H, seqlen, head_dim, h_o_ref,
3504                                                     ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3505         bench_fa_3_tma_ws_launch<64, 2, kMmaAccF32>(B, H, seqlen, head_dim, h_o_ref,
3506                                                     ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3507         bench_fa_3_tma_ws_launch<64, 3, kMmaAccF32>(B, H, seqlen, head_dim, h_o_ref,
3508                                                     ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3509         bench_fa_3_tma_ws_launch<64, 4, kMmaAccF32>(B, H, seqlen, head_dim, h_o_ref,
3510                                                     ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3511     } else if (head_dim == 128) {
3512         bench_fa_3_tma_ws_launch<128, 1, kMmaAccF32>(B, H, seqlen, head_dim, h_o_ref,
3513                                                     ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3514         bench_fa_3_tma_ws_launch<128, 2, kMmaAccF32>(B, H, seqlen, head_dim, h_o_ref,
3515                                                     ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3516         bench_fa_3_tma_ws_launch<128, 3, kMmaAccF32>(B, H, seqlen, head_dim, h_o_ref,
3517                                                     ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3518         bench_fa_3_tma_ws_launch<128, 4, kMmaAccF32>(B, H, seqlen, head_dim, h_o_ref,
3519                                                     ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f16);
3520     } else {
3521         char label[64];
3522         snprintf(label, sizeof(label), "FA3 TMA MMA WS (2 Consumer WG) (D!=64/128)");
3523         printf("| %-56s | %-9s | %-19s |\n", label, "SKIP", "None");
3524     }
3525 }
3526
3527 #if defined(NOTES_V2_ENABLE_CUTE)
3528 template <int kHeadDim, int kStagesK = 1>
3529 static void bench_fa_3_tma_mma_ws_cute_launch(
3530     int B, int H, int seqlen, half *h_o_ref, float *ref_o,
3531     half *d_q, half *d_k, half *d_v, half *d_o,
3532     float cudnn_tflops_f32) {
3533     using namespace cute;
3534     using Traits = fa_cute::FlashAttn3CuTeTraits<kHeadDim>;
3535     using SmemLayout = typename Traits::SmemLayoutQKV;
3536     constexpr int kNumConsumers = 2;
3537     if (seqlen < 64 || seqlen % 64 != 0) {
3538         char label[80];
3539         snprintf(label, sizeof(label),
3540             "FA3 CuTe TMA MMA WS (2 Consumer WG) (Sk=%d, Sv=1, unaligned)", kStagesK);
3541         printf("| %-56s | %-9s | %-19s |\n", label, "SKIP", "None");
3542         return;
3543     }
3544
3545     int rows = B * H * seqlen;
3546     auto make_tma = [=](half *pointer) {
3547         auto tensor = make_tensor(
3548             make_gmem_ptr(reinterpret_cast<cutlass::half_t *>(pointer)),
3549             make_shape(rows, Int<kHeadDim>{}),

```

```

3550     make_stride(Int<kHeadDim>{}, _1{}));
3551     return make_tma_copy(
3552         SM90_TMA_LOAD{}, tensor, SmemLayout{},
3553         Shape<_64, Int<kHeadDim>>{}, _1{});
3554 };
3555 auto tma_q = make_tma(d_q);
3556 auto tma_k = make_tma(d_k);
3557 auto tma_v = make_tma(d_v);
3558 auto kernel = flash_attn_3_tma_mma_ws_split_q_cute<
3559     kHeadDim, decltype(tma_q), decltype(tma_k), decltype(tma_v),
3560     kStagesK>;
3561 auto acc_o = partition_fragment_C(
3562     typename Traits::TiledMma{}, Shape<_64, Int<kHeadDim>>{});
3563 constexpr int kTiles = 1 + kNumConsumers * kStagesK + kNumConsumers;
3564 constexpr int kTilesBytes = kTiles * cosize(SmemLayout{}) * sizeof(cutlass::half_t);
3565 int merge_bytes = 128 * size(acc_o) * sizeof(float) + 128 * sizeof(float4);
3566 int smem_bytes = max(kTilesBytes, merge_bytes);
3567 bool smem_ok = check_smem_feasible((const void *)kernel, smem_bytes);
3568 if (!smem_ok) {
3569     char label[80];
3570     snprintf(label, sizeof(label),
3571         "FA3 CuTe TMA MMA WS (2 Consumer WG) (Sk=%d, Sv=1, SMEM)", kStagesK);
3572     if (g_debug)
3573         printf("| %-56s | %-9s | %-19s |\n", label, "SMEM too large", "None");
3574     return;
3575 }
3576 check(cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
3577     smem_bytes),
3578     "bench cute fa3 set smem");
3579
3580 dim3 grid(seqlen / 64, B * H);
3581 cudaStream_t stream;
3582 cudaEvent_t start;
3583 cudaEvent_t stop;
3584 check(cudaStreamCreate(&stream), "bench cute fa3 stream");
3585 check(cudaEventCreate(&start), "bench cute fa3 event start");
3586 check(cudaEventCreate(&stop), "bench cute fa3 event stop");
3587 for (int warmup = 0; warmup < g_warmup; ++warmup) {
3588     kernel<<<grid, 384, smem_bytes, stream>>>(
3589         tma_q, tma_k, tma_v,
3590         reinterpret_cast<cutlass::half_t *>(d_o), rows, seqlen);
3591 }
3592 check(cudaStreamSynchronize(stream), "bench cute fa3 warmup sync");
3593 check(cudaEventRecord(start, stream), "bench cute fa3 record start");
3594 for (int repeat = 0; repeat < g_repeat; ++repeat) {
3595     kernel<<<grid, 384, smem_bytes, stream>>>(
3596         tma_q, tma_k, tma_v,
3597         reinterpret_cast<cutlass::half_t *>(d_o), rows, seqlen);
3598 }
3599 check(cudaEventRecord(stop, stream), "bench cute fa3 record stop");
3600 check(cudaEventSynchronize(stop), "bench cute fa3 timing sync");
3601 float time_ms = 0.0f;
3602 check(cudaEventElapsedTime(&time_ms, start, stop), "bench cute fa3 elapsed");
3603 time_ms /= g_repeat;
3604
3605 size_t count = (size_t)rows * kHeadDim;
3606 half *h_o = (half *)malloc(count * sizeof(half));
3607 check(cudaMemcpy(h_o, d_o, count * sizeof(half), cudaMemcpyDeviceToHost),
3608     "bench cute fa3 D2H");
3609 float max_err = 0.0f;
3610 bool checked = h_o_ref || ref_o;
3611 if (checked) {
3612     for (size_t idx = 0; idx < count; ++idx) {

```

```

3613     float reference = h_o_ref ? __half2float(h_o_ref[idx]) : ref_o[idx];
3614     max_err = max(max_err, fabsf(__half2float(h_o[idx]) - reference));
3615 }
3616 }
3617 float tflops = bench_fa_tflops(B, H, seqlen, kHeadDim, time_ms);
3618 char label[80];
3619 snprintf(label, sizeof(label),
3620     "FA3 CuTe TMA MMA WS (2 Consumer WG) (Sk=%d, Sv=1, F32Acc)", kStagesK);
3621 bool is_fail = checked && max_err >= 5e-1f;
3622 if (is_fail || should_print_fa_tflops(1, tflops)) {
3623     char performance[32];
3624     if (cudnn_tflops_f32 > 0.0f) {
3625         snprintf(performance, sizeof(performance), "%.1f/%.1f (%.2fx)",
3626             tflops, cudnn_tflops_f32, tflops / cudnn_tflops_f32);
3627     } else {
3628         snprintf(performance, sizeof(performance), "%.1f", tflops);
3629     }
3630     printf("| %-56s | %.3e | %-19s |\n", label, max_err, performance);
3631 }
3632
3633 free(h_o);
3634 cudaEventDestroy(start);
3635 cudaEventDestroy(stop);
3636 cudaStreamDestroy(stream);
3637 }
3638
3639 static void bench_fa_3_tma_mma_ws_cute_dispatch(
3640     int B, int H, int seqlen, int head_dim,
3641     half *h_o_ref, float *ref_o,
3642     half *d_q, half *d_k, half *d_v, half *d_o,
3643     float cudnn_tflops_f32) {
3644     if (head_dim == 64) {
3645         bench_fa_3_tma_mma_ws_cute_launch<64, 1>(B, H, seqlen, h_o_ref, ref_o,
3646             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3647         bench_fa_3_tma_mma_ws_cute_launch<64, 2>(B, H, seqlen, h_o_ref, ref_o,
3648             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3649         bench_fa_3_tma_mma_ws_cute_launch<64, 3>(B, H, seqlen, h_o_ref, ref_o,
3650             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3651         bench_fa_3_tma_mma_ws_cute_launch<64, 4>(B, H, seqlen, h_o_ref, ref_o,
3652             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3653     } else if (head_dim == 128) {
3654         bench_fa_3_tma_mma_ws_cute_launch<128, 1>(B, H, seqlen, h_o_ref, ref_o,
3655             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3656         bench_fa_3_tma_mma_ws_cute_launch<128, 2>(B, H, seqlen, h_o_ref, ref_o,
3657             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3658     }
3659 }
3660
3661 #if defined(NOTES_V2_ENABLE_CUTE) && defined(NOTES_V2_ENABLE_TMA_MMA_WS)
3662 // FA2-style CuTe cp.async bench: single consumer, Br=128, no TMA/WS.
3663 // V 单 buffer, V 在 QK 之前发起 cp.async, 通过 QK+softmax 隐藏 V 延迟.
3664 template <int kHeadDim, int kStagesK>
3665 static void bench_fa_2_mma_stages_cute_launch(
3666     int B, int H, int seqlen, half *h_o_ref, float *ref_o,
3667     half *d_q, half *d_k, half *d_v, half *d_o,
3668     float cudnn_tflops_f32) {
3669     using namespace cute;
3670     using Traits = fa_cute::FlashAttn2CuTeTraits<kHeadDim>;
3671     using SmemLayoutQ = typename Traits::SmemLayoutQ;
3672     using SmemLayoutKV = typename Traits::SmemLayoutKV;
3673     constexpr int kBr = 128;
3674     if (seqlen < kBr || seqlen % kBr != 0 || seqlen % 64 != 0) {
3675         char label[96];

```

```

3676     snprintf(label, sizeof(label),
3677              "FA2 CuTe MMA Stages (Sk=%d, unaligned)", kStagesK);
3678     printf("| %-56s | %-9s | %-19s |\n", label, "SKIP", "None");
3679     return;
3680 }
3681
3682 int rows = B * H * seqLen;
3683 auto kernel = flash_attn_mma_stages_split_q_cute<kHeadDim, kStagesK>;
3684 int smem_bytes = (size(SmemLayoutQ{}) +
3685                  kStagesK * size(SmemLayoutKV{}) +
3686                  1 * size(SmemLayoutKV{})) *
3687                  sizeof(cutlass::half_t);
3688 bool smem_ok = check_smem_feasible((const void *)kernel, smem_bytes);
3689 if (!smem_ok) {
3690     char label[96];
3691     snprintf(label, sizeof(label),
3692              "FA2 CuTe MMA Stages (Sk=%d, SMEM)", kStagesK);
3693     if (g_debug)
3694         printf("| %-56s | %-9s | %-19s |\n", label, "SMEM too large", "None");
3695     return;
3696 }
3697 check(cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
3698                             smem_bytes),
3699        "bench cute fa2 cpasync set smem");
3700
3701 dim3 grid(seqLen / kBr, B * H);
3702 cudaStream_t stream;
3703 cudaEvent_t start;
3704 cudaEvent_t stop;
3705 check(cudaStreamCreate(&stream), "bench cute fa2 cpasync stream");
3706 check(cudaEventCreate(&start), "bench cute fa2 cpasync event start");
3707 check(cudaEventCreate(&stop), "bench cute fa2 cpasync event stop");
3708 for (int warmup = 0; warmup < g_warmup; ++warmup) {
3709     kernel<<<grid, 256, smem_bytes, stream>>>{
3710         reinterpret_cast<cutlass::half_t *>(d_q),
3711         reinterpret_cast<cutlass::half_t *>(d_k),
3712         reinterpret_cast<cutlass::half_t *>(d_v),
3713         reinterpret_cast<cutlass::half_t *>(d_o), rows, seqLen);
3714     }
3715     check(cudaStreamSynchronize(stream), "bench cute fa2 cpasync warmup sync");
3716     check(cudaEventRecord(start, stream), "bench cute fa2 cpasync record start");
3717     for (int repeat = 0; repeat < g_repeat; ++repeat) {
3718         kernel<<<grid, 256, smem_bytes, stream>>>{
3719             reinterpret_cast<cutlass::half_t *>(d_q),
3720             reinterpret_cast<cutlass::half_t *>(d_k),
3721             reinterpret_cast<cutlass::half_t *>(d_v),
3722             reinterpret_cast<cutlass::half_t *>(d_o), rows, seqLen);
3723         }
3724         check(cudaEventRecord(stop, stream), "bench cute fa2 cpasync record stop");
3725         check(cudaEventSynchronize(stop), "bench cute fa2 cpasync timing sync");
3726         float time_ms = 0.0f;
3727         check(cudaEventElapsedTime(&time_ms, start, stop),
3728                "bench cute fa2 cpasync elapsed");
3729         time_ms /= g_repeat;
3730
3731         size_t count = (size_t)rows * kHeadDim;
3732         half *h_o = (half *)malloc(count * sizeof(half));
3733         check(cudaMemcpy(h_o, d_o, count * sizeof(half), cudaMemcpyDeviceToHost),
3734                "bench cute fa2 cpasync D2H");
3735         float max_err = 0.0f;
3736         bool checked = h_o_ref || ref_o;
3737         if (checked) {
3738             for (size_t idx = 0; idx < count; ++idx) {

```

```

3739     float reference = h_o_ref ? __half2float(h_o_ref[idx]) : ref_o[idx];
3740     max_err = max(max_err, fabsf(__half2float(h_o[idx]) - reference));
3741 }
3742 }
3743 float tflops = bench_fa_tfllops(B, H, seqlen, kHeadDim, time_ms);
3744 char label[96];
3745 snprintf(label, sizeof(label),
3746     "FA2 CuTe MMA Stages (Sk=%d, F32Acc)", kStagesK);
3747 bool is_fail = checked && max_err >= 5e-1f;
3748 if (is_fail || should_print_fa_tfllops(1, tflops)) {
3749     char performance[32];
3750     if (cudnn_tfllops_f32 > 0.0f) {
3751         snprintf(performance, sizeof(performance), "%.1f/%.1f (%.2fx)",
3752             tflops, cudnn_tfllops_f32, tflops / cudnn_tfllops_f32);
3753     } else {
3754         snprintf(performance, sizeof(performance), "%.1f", tflops);
3755     }
3756     printf("| %-56s | %.3e | %-19s |\n", label, max_err, performance);
3757 }
3758
3759 free(h_o);
3760 cudaEventDestroy(start);
3761 cudaEventDestroy(stop);
3762 cudaStreamDestroy(stream);
3763 }
3764
3765 static void bench_fa_2_mma_stages_cute_dispatch(
3766     int B, int H, int seqlen, int head_dim,
3767     half *h_o_ref, float *ref_o,
3768     half *d_q, half *d_k, half *d_v, half *d_o,
3769     float cudnn_tfllops_f32) {
3770     if (head_dim == 64) {
3771         bench_fa_2_mma_stages_cute_launch<64, 1>(B, H, seqlen, h_o_ref, ref_o,
3772             d_q, d_k, d_v, d_o, cudnn_tfllops_f32);
3773         bench_fa_2_mma_stages_cute_launch<64, 2>(B, H, seqlen, h_o_ref, ref_o,
3774             d_q, d_k, d_v, d_o, cudnn_tfllops_f32);
3775         bench_fa_2_mma_stages_cute_launch<64, 3>(B, H, seqlen, h_o_ref, ref_o,
3776             d_q, d_k, d_v, d_o, cudnn_tfllops_f32);
3777         bench_fa_2_mma_stages_cute_launch<64, 4>(B, H, seqlen, h_o_ref, ref_o,
3778             d_q, d_k, d_v, d_o, cudnn_tfllops_f32);
3779     } else if (head_dim == 128) {
3780         bench_fa_2_mma_stages_cute_launch<128, 1>(B, H, seqlen, h_o_ref, ref_o,
3781             d_q, d_k, d_v, d_o, cudnn_tfllops_f32);
3782         bench_fa_2_mma_stages_cute_launch<128, 2>(B, H, seqlen, h_o_ref, ref_o,
3783             d_q, d_k, d_v, d_o, cudnn_tfllops_f32);
3784     }
3785 }
3786 #endif
3787
3788 // FA2-style CuTe bench: single consumer, Br=128, no merge.
3789 template <int kHeadDim, int kStagesK, int kStagesV = 1>
3790 static void bench_fa_2_tma_mma_ws_cute_launch(
3791     int B, int H, int seqlen, half *h_o_ref, float *ref_o,
3792     half *d_q, half *d_k, half *d_v, half *d_o,
3793     float cudnn_tfllops_f32) {
3794     using namespace cute;
3795     using Traits = fa_cute::FlashAttn2CuTeTraits<kHeadDim>;
3796     using SmemLayoutQ = typename Traits::SmemLayoutQ;
3797     using SmemLayoutKV = typename Traits::SmemLayoutKV;
3798     constexpr int kBr = 128;
3799     if (seqlen < kBr || seqlen % kBr != 0 || seqlen % 64 != 0) {
3800         char label[80];
3801         snprintf(label, sizeof(label),

```

```

3802         "FA2 CuTe TMA MMA WS (1 Consumer WG) (Sk=%d, Sv=%d, unaligned)",
3803         kStagesK, kStagesV);
3804     printf("| %-56s | %-9s | %-19s |\n", label, "SKIP", "None");
3805     return;
3806 }
3807
3808 int rows = B * H * seqLen;
3809 // Q tile: [128, D], K/V tile: [64, D]
3810 auto make_tma_q = [=]() {
3811     auto tensor = make_tensor(
3812         make_gmem_ptr(reinterpret_cast<cutlass::half_t*>(d_q)),
3813         make_shape(rows, Int<kHeadDim>{}),
3814         make_stride(Int<kHeadDim>{}, _1{}));
3815     return make_tma_copy(
3816         SM90_TMA_LOAD{}, tensor, SmemLayoutQ{},
3817         Shape<_128, Int<kHeadDim>>{}, _1{});
3818 };
3819 auto make_tma_kv = [=](half *pointer) {
3820     auto tensor = make_tensor(
3821         make_gmem_ptr(reinterpret_cast<cutlass::half_t*>(pointer)),
3822         make_shape(rows, Int<kHeadDim>{}),
3823         make_stride(Int<kHeadDim>{}, _1{}));
3824     return make_tma_copy(
3825         SM90_TMA_LOAD{}, tensor, SmemLayoutKV{},
3826         Shape<_64, Int<kHeadDim>>{}, _1{});
3827 };
3828 auto tma_q = make_tma_q();
3829 auto tma_k = make_tma_kv(d_k);
3830 auto tma_v = make_tma_kv(d_v);
3831 auto kernel = flash_attn_tma_mma_ws_split_q_cute<
3832     kHeadDim, decltype(tma_q), decltype(tma_k), decltype(tma_v),
3833     kStagesK, kStagesV>;
3834 // smem = Q[128*D] + K[Sk*64*D] + V[Sv*64*D] (no merge scratch)
3835 int smem_bytes = (cosize(SmemLayoutQ{}) +
3836     kStagesK * cosize(SmemLayoutKV{}) +
3837     kStagesV * cosize(SmemLayoutKV{})) *
3838     sizeof(cutlass::half_t);
3839 bool smem_ok = check_smem_feasible((const void *)kernel, smem_bytes);
3840 if (!smem_ok) {
3841     char label[80];
3842     snprintf(label, sizeof(label),
3843         "FA2 CuTe TMA MMA WS (1 Consumer WG) (Sk=%d, Sv=%d, SMEM)",
3844         kStagesK, kStagesV);
3845     if (g_debug)
3846         printf("| %-56s | %-9s | %-19s |\n", label, "SMEM too large", "None");
3847     return;
3848 }
3849 check(cudaFuncSetAttribute(kernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
3850     smem_bytes),
3851     "bench cute fa2 set smem");
3852
3853 dim3 grid(seqLen / kBr, B * H);
3854 cudaStream_t stream;
3855 cudaEvent_t start;
3856 cudaEvent_t stop;
3857 check(cudaStreamCreate(&stream), "bench cute fa2 stream");
3858 check(cudaEventCreate(&start), "bench cute fa2 event start");
3859 check(cudaEventCreate(&stop), "bench cute fa2 event stop");
3860 for (int warmup = 0; warmup < g_warmup; ++warmup) {
3861     kernel<<<grid, 384, smem_bytes, stream>>>(
3862         tma_q, tma_k, tma_v,
3863         reinterpret_cast<cutlass::half_t*>(d_o), rows, seqLen);
3864 }

```

```

3865 check(cudaStreamSynchronize(stream), "bench cute fa2 warmup sync");
3866 check(cudaEventRecord(start, stream), "bench cute fa2 record start");
3867 for (int repeat = 0; repeat < g_repeat; ++repeat) {
3868     kernel<<<grid, 384, smem_bytes, stream>>>(
3869         tma_q, tma_k, tma_v,
3870         reinterpret_cast<cutlass::half_t *>(d_o), rows, seqlen);
3871 }
3872 check(cudaEventRecord(stop, stream), "bench cute fa2 record stop");
3873 check(cudaEventSynchronize(stop), "bench cute fa2 timing sync");
3874 float time_ms = 0.0f;
3875 check(cudaEventElapsedTime(&time_ms, start, stop), "bench cute fa2 elapsed");
3876 time_ms /= g_repeat;
3877
3878 size_t count = (size_t)rows * kHeadDim;
3879 half *h_o = (half *)malloc(count * sizeof(half));
3880 check(cudaMemcpy(h_o, d_o, count * sizeof(half), cudaMemcpyDeviceToHost),
3881     "bench cute fa2 D2H");
3882 float max_err = 0.0f;
3883 bool checked = h_o_ref || ref_o;
3884 if (checked) {
3885     for (size_t idx = 0; idx < count; ++idx) {
3886         float reference = h_o_ref ? __half2float(h_o_ref[idx]) : ref_o[idx];
3887         max_err = max(max_err, fabsf(__half2float(h_o[idx]) - reference));
3888     }
3889 }
3890 float tflops = bench_fa_tflops(B, H, seqlen, kHeadDim, time_ms);
3891 char label[80];
3892 snprintf(label, sizeof(label),
3893     "FA2 CuTe TMA MMA WS (1 Consumer WG) (Sk=%d, Sv=%d, F32Acc)",
3894     kStagesK, kStagesV);
3895 bool is_fail = checked && max_err >= 5e-1f;
3896 if (is_fail || should_print_fa_tflops(1, tflops)) {
3897     char performance[32];
3898     if (cudnn_tflops_f32 > 0.0f) {
3899         snprintf(performance, sizeof(performance), "%.1f/%.1f (%.2fx)",
3900             tflops, cudnn_tflops_f32, tflops / cudnn_tflops_f32);
3901     } else {
3902         snprintf(performance, sizeof(performance), "%.1f", tflops);
3903     }
3904     printf("| %-56s | %.3e | %-19s |\n", label, max_err, performance);
3905 }
3906
3907 free(h_o);
3908 cudaEventDestroy(start);
3909 cudaEventDestroy(stop);
3910 cudaStreamDestroy(stream);
3911 }
3912
3913 static void bench_fa_2_tma_mma_ws_cute_dispatch(
3914     int B, int H, int seqlen, int head_dim,
3915     half *h_o_ref, float *ref_o,
3916     half *d_q, half *d_k, half *d_v, half *d_o,
3917     float cudnn_tflops_f32) {
3918     if (head_dim == 64) {
3919         bench_fa_2_tma_mma_ws_cute_launch<64, 1, 1>(B, H, seqlen, h_o_ref, ref_o,
3920             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3921         bench_fa_2_tma_mma_ws_cute_launch<64, 2, 1>(B, H, seqlen, h_o_ref, ref_o,
3922             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3923         bench_fa_2_tma_mma_ws_cute_launch<64, 3, 1>(B, H, seqlen, h_o_ref, ref_o,
3924             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3925         bench_fa_2_tma_mma_ws_cute_launch<64, 4, 1>(B, H, seqlen, h_o_ref, ref_o,
3926             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3927         bench_fa_2_tma_mma_ws_cute_launch<64, 2, 2>(B, H, seqlen, h_o_ref, ref_o,

```

```

3928     d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3929 } else if (head_dim == 128) {
3930     bench_fa_2_tma_mma_ws_cute_launch<128, 1, 1>(B, H, seqlen, h_o_ref, ref_o,
3931         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3932     bench_fa_2_tma_mma_ws_cute_launch<128, 2, 1>(B, H, seqlen, h_o_ref, ref_o,
3933         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3934     bench_fa_2_tma_mma_ws_cute_launch<128, 3, 1>(B, H, seqlen, h_o_ref, ref_o,
3935         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
3936 }
3937 }
3938 #endif
3939 #endif /* NOTES_V2_ENABLE_TMA_MMA_WS */
3940
3941 #if defined(NOTES_V2_ENABLE_CUDNN)
3942 static float bench_cudnn_sdpa_tflops(half *d_q, half *d_k, half *d_v,
3943     half *d_o_ref, int B, int H, int seqlen,
3944     int head_dim, fe::DataType_t compute_type) {
3945     cudnnHandle_t handle;
3946     cudnnCreate(&handle);
3947     auto graph = std::make_shared<fe::graph::Graph>();
3948     graph->set_io_data_type(fe::DataType_t::HALF)
3949         .set_intermediate_data_type(fe::DataType_t::FLOAT)
3950         .set_compute_data_type(compute_type);
3951
3952     auto Q = graph->tensor(fe::graph::Tensor_attributes()
3953         .set_uid(1).set_dim({B, H, seqlen, head_dim})
3954         .set_stride({H * seqlen * head_dim, seqlen * head_dim, head_dim, 1}));
3955     auto K = graph->tensor(fe::graph::Tensor_attributes()
3956         .set_uid(2).set_dim({B, H, seqlen, head_dim})
3957         .set_stride({H * seqlen * head_dim, seqlen * head_dim, head_dim, 1}));
3958     auto V = graph->tensor(fe::graph::Tensor_attributes()
3959         .set_uid(3).set_dim({B, H, seqlen, head_dim})
3960         .set_stride({H * seqlen * head_dim, seqlen * head_dim, head_dim, 1}));
3961
3962     auto [O_sdpa, Stats] = graph->sdpa(Q, K, V,
3963         fe::graph::SDPA_attributes()
3964             .set_name("sdpa_ref")
3965             .set_attn_scale(1.0f / sqrtf((float)head_dim)));
3966
3967     O_sdpa->set_output(true).set_uid(4)
3968         .set_dim({B, H, seqlen, head_dim})
3969         .set_stride({H * seqlen * head_dim, seqlen * head_dim, head_dim, 1});
3970
3971     auto build_status = graph->build(handle, {fe::HeurMode_t::A, fe::HeurMode_t::FALLBACK});
3972     float tflops = -1.0f;
3973     if (build_status.is_good()) {
3974         std::unordered_map<fe::graph::Tensor_attributes::uid_t, void*> vp = {
3975             {1, d_q}, {2, d_k}, {3, d_v}, {4, d_o_ref}};
3976         int64_t ws_size = 0;
3977         if (graph->get_workspace_size(ws_size).is_good()) {
3978             int8_t *d_ws = nullptr;
3979             if (ws_size > 0) check(cudaMalloc(&d_ws, ws_size), "bench cudnn ws");
3980             if (graph->execute(handle, vp, d_ws).is_good()) {
3981                 check(cudaDeviceSynchronize(), "bench cudnn sync");
3982                 for (int w = 1; w < g_warmup; ++w)
3983                     (void)graph->execute(handle, vp, d_ws);
3984                 cudaDeviceSynchronize();
3985                 cudaEvent_t ev_s, ev_e;
3986                 cudaEventCreate(&ev_s); cudaEventCreate(&ev_e);
3987                 cudaEventRecord(ev_s);
3988                 for (int r = 0; r < g_repeat; ++r)
3989                     (void)graph->execute(handle, vp, d_ws);
3990                 cudaEventRecord(ev_e);

```

```

3991     cudaEventSynchronize(ev_e);
3992     float time_ms = 0;
3993     cudaEventElapsedTime(&time_ms, ev_s, ev_e);
3994     time_ms /= g_repeat;
3995     tflops = bench_fa_tflops(B, H, seqlen, head_dim, time_ms);
3996     cudaEventDestroy(ev_s); cudaEventDestroy(ev_e);
3997 }
3998 if (d_ws) cudaFree(d_ws);
3999 }
4000 }
4001 cudnnDestroy(handle);
4002 return tflops;
4003 }
4004 #endif
4005
4006 static void bench_flash_attn(int B, int H, int N, int D) {
4007     int seqlen = N, head_dim = D;
4008
4009     if (head_dim != 64 && head_dim != 128) {
4010         printf("| %-56s | %-9s | %-19s |\n", "FlashAttention-2", "unsupported D", "None");
4011         return;
4012     }
4013
4014     size_t sz = (size_t)B * H * seqlen * head_dim * sizeof(half);
4015
4016     srand(42);
4017     half *h_q = (half *)malloc(sz);
4018     half *h_k = (half *)malloc(sz);
4019     half *h_v = (half *)malloc(sz);
4020     for (int i = 0; i < B * H * seqlen * head_dim; i++) {
4021         h_q[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
4022         h_k[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
4023         h_v[i] = __float2half(((float)rand() / RAND_MAX) * 2.0f - 1.0f);
4024     }
4025
4026     // Device allocations — shared by kernel and reference
4027     half *d_q, *d_k, *d_v, *d_o;
4028     check(cudaMalloc(&d_q, sz), "bench fa alloc Q");
4029     check(cudaMalloc(&d_k, sz), "bench fa alloc K");
4030     check(cudaMalloc(&d_v, sz), "bench fa alloc V");
4031     check(cudaMalloc(&d_o, sz), "bench fa alloc O");
4032     check(cudaMemcpy(d_q, h_q, sz, cudaMemcpyHostToDevice), "bench fa H2D Q");
4033     check(cudaMemcpy(d_k, h_k, sz, cudaMemcpyHostToDevice), "bench fa H2D K");
4034     check(cudaMemcpy(d_v, h_v, sz, cudaMemcpyHostToDevice), "bench fa H2D V");
4035
4036     // Reference output: either cuDNN (half) or CPU (float)
4037     half *h_o_ref = nullptr;
4038     float *ref_o = nullptr;
4039     float cudnn_tflops_f16 = -1.0f, cudnn_tflops_f32 = -1.0f;
4040     int ref_count = B * H * seqlen * head_dim;
4041
4042     #if defined(NOTES_V2_ENABLE_CUDNN)
4043     if (!g_fa_skip_check) {
4044         half *d_o_ref;
4045         check(cudaMalloc(&d_o_ref, sz), "bench fa alloc O_ref");
4046         cudnn_tflops_f16 = bench_cudnn_sdpa_tflops(d_q, d_k, d_v, d_o_ref,
4047                                                     B, H, seqlen, head_dim,
4048                                                     fe::DataType_t::HALF);
4049         bool cudnn_ok = (cudnn_tflops_f16 > 0);
4050         if (cudnn_ok) {
4051             cudnn_tflops_f32 = bench_cudnn_sdpa_tflops(d_q, d_k, d_v, d_o_ref,
4052                                                         B, H, seqlen, head_dim,
4053                                                         fe::DataType_t::FLOAT);

```

```

4054     h_o_ref = (half *)malloc(sz);
4055     check(cudaMemcpy(h_o_ref, d_o_ref, sz, cudaMemcpyDeviceToHost), "bench fa D2H ref");
4056 } else {
4057     fprintf(stderr, "cudnn SDPA unsupported on this SM, using CPU ref\n");
4058 }
4059 cudaFree(d_o_ref);
4060 }
4061 #endif
4062
4063 // CPU reference fallback
4064 if (!g_fa_skip_check && !h_o_ref) {
4065     float *ref_q = (float *)malloc(sz * 4 / sizeof(half));
4066     float *ref_k = (float *)malloc(sz * 4 / sizeof(half));
4067     float *ref_v = (float *)malloc(sz * 4 / sizeof(half));
4068     ref_o = (float *)malloc(sz * 4 / sizeof(half));
4069     for (int i = 0; i < ref_count; i++) {
4070         ref_q[i] = __half2float(h_q[i]);
4071         ref_k[i] = __half2float(h_k[i]);
4072         ref_v[i] = __half2float(h_v[i]);
4073     }
4074
4075     float scale = 1.0f / sqrtf((float)head_dim);
4076     for (int bi = 0; bi < B * H; bi++) {
4077         for (int qi = 0; qi < seqlen; qi++) {
4078             float smax = -INFINITY;
4079             float *S = (float *)malloc((size_t)seqlen * sizeof(float));
4080             for (int kj = 0; kj < seqlen; kj++) {
4081                 float s = 0.0f;
4082                 for (int d = 0; d < head_dim; d++)
4083                     s += ref_q[bi * seqlen * head_dim + qi * head_dim + d] *
4084                         ref_k[bi * seqlen * head_dim + kj * head_dim + d];
4085                 S[kj] = s * scale;
4086                 if (S[kj] > smax) smax = S[kj];
4087             }
4088             double sum_exp = 0.0;
4089             for (int kj = 0; kj < seqlen; kj++)
4090                 sum_exp += (double)expf(S[kj] - smax);
4091             float inv_sum = 1.0f / (float)sum_exp;
4092             for (int d = 0; d < head_dim; d++) {
4093                 double o_acc = 0.0;
4094                 for (int kj = 0; kj < seqlen; kj++)
4095                     o_acc += (double)(expf(S[kj] - smax) * inv_sum) *
4096                         ref_v[bi * seqlen * head_dim + kj * head_dim + d];
4097                 ref_o[bi * seqlen * head_dim + qi * head_dim + d] = (float)o_acc;
4098             }
4099             free(S);
4100         }
4101     }
4102     free(ref_q);
4103     free(ref_k);
4104     free(ref_v);
4105 }
4106
4107 if (g_bench_fa3_cute_only) {
4108 #if defined(NOTES_V2_ENABLE_CUTE) && defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4109     bench_fa3_tma_mma_ws_cute_dispatch(
4110         B, H, seqlen, head_dim, h_o_ref, ref_o,
4111         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4112 #endif
4113     free(h_q);
4114     free(h_k);
4115     free(h_v);
4116     free(h_o_ref);

```

```

4117 free(ref_o);
4118 cudaFree(d_q);
4119 cudaFree(d_k);
4120 cudaFree(d_v);
4121 cudaFree(d_o);
4122 return;
4123 }
4124
4125 if (head_dim == 64) {
4126     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::Pad) {
4127         cudaDeviceSynchronize();
4128         bench_fa_launch<64, 1, 8, 8, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4129             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4130         cudaDeviceSynchronize();
4131         bench_fa_launch<64, 2, 8, 8, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4132             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4133         cudaDeviceSynchronize();
4134         bench_fa_launch<64, 1, 8, 8, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4135             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4136         cudaDeviceSynchronize();
4137         bench_fa_launch<64, 2, 8, 8, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4138             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4139     }
4140     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleQ) {
4141         cudaDeviceSynchronize();
4142         bench_fa_launch<64, 2, 0, 8, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4143             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4144         cudaDeviceSynchronize();
4145         bench_fa_launch<64, 2, 0, 8, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4146             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4147     }
4148     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleK) {
4149         cudaDeviceSynchronize();
4150         bench_fa_launch<64, 2, 8, 0, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4151             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4152         cudaDeviceSynchronize();
4153         bench_fa_launch<64, 2, 8, 0, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4154             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4155     }
4156     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleV) {
4157         cudaDeviceSynchronize();
4158         bench_fa_launch<64, 2, 8, 8, 0, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4159             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4160         cudaDeviceSynchronize();
4161         bench_fa_launch<64, 2, 8, 8, 0, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4162             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4163     }
4164     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleQK) {
4165         cudaDeviceSynchronize();
4166         bench_fa_launch<64, 2, 0, 0, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4167             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4168         cudaDeviceSynchronize();
4169         bench_fa_launch<64, 2, 0, 0, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4170             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4171     }
4172     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleQV) {
4173         cudaDeviceSynchronize();
4174         bench_fa_launch<64, 2, 0, 8, 0, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4175             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4176         cudaDeviceSynchronize();
4177         bench_fa_launch<64, 2, 0, 8, 0, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4178             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4179     }

```

```

4180 if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleKV) {
4181     cudaDeviceSynchronize();
4182     bench_fa_launch<64, 2, 8, 0, 0, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4183                                         d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4184     cudaDeviceSynchronize();
4185     bench_fa_launch<64, 2, 8, 0, 0, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4186                                         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4187 }
4188 if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::Swizzle) {
4189     cudaDeviceSynchronize();
4190     bench_fa_launch<64, 2, 0, 0, 0, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4191                                         d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4192     cudaDeviceSynchronize();
4193     bench_fa_launch<64, 2, 0, 0, 0, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4194                                         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4195 }
4196 #if defined(NOTES_V2_ENABLE_CUTE)
4197 {
4198     cudaDeviceSynchronize();
4199     bench_fa_2_mma_stages_cute_dispatch(
4200         B, H, seqlen, head_dim, h_o_ref, ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4201 }
4202 #endif
4203 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4204 {
4205     cudaDeviceSynchronize();
4206     bench_fa_tma_mma_ws_dispatch<1, 1, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4207                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4208     cudaDeviceSynchronize();
4209     bench_fa_tma_mma_ws_dispatch<2, 1, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4210                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4211     cudaDeviceSynchronize();
4212     bench_fa_tma_mma_ws_dispatch<3, 1, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4213                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4214     cudaDeviceSynchronize();
4215     bench_fa_tma_mma_ws_dispatch<4, 1, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4216                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4217     cudaDeviceSynchronize();
4218     bench_fa_tma_mma_ws_dispatch<2, 2, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4219                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4220     // F32Acc variants
4221     cudaDeviceSynchronize();
4222     bench_fa_tma_mma_ws_dispatch<1, 1, 1>(B, H, seqlen, head_dim, h_o_ref,
4223                                             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4224     cudaDeviceSynchronize();
4225     bench_fa_tma_mma_ws_dispatch<2, 1, 1>(B, H, seqlen, head_dim, h_o_ref,
4226                                             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4227     cudaDeviceSynchronize();
4228     bench_fa_tma_mma_ws_dispatch<3, 1, 1>(B, H, seqlen, head_dim, h_o_ref,
4229                                             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4230     cudaDeviceSynchronize();
4231     bench_fa_tma_mma_ws_dispatch<4, 1, 1>(B, H, seqlen, head_dim, h_o_ref,
4232                                             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4233     cudaDeviceSynchronize();
4234     bench_fa_tma_mma_ws_dispatch<2, 2, 1>(B, H, seqlen, head_dim, h_o_ref,
4235                                             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4236     // FA3-style dual-consumer
4237     cudaDeviceSynchronize();
4238     bench_fa_3_tma_ws_dispatch<0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4239                                     d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4240     cudaDeviceSynchronize();
4241     bench_fa_3_tma_ws_dispatch<1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4242                                     d_q, d_k, d_v, d_o, cudnn_tflops_f32);

```

```

4243 #if defined(NOTES_V2_ENABLE_CUTE)
4244     cudaDeviceSynchronize();
4245     bench_fa_2_tma_mma_ws_cute_dispatch(B, H, seqlen, head_dim, h_o_ref, ref_o,
4246                                         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4247     cudaDeviceSynchronize();
4248     bench_fa_3_tma_mma_ws_cute_dispatch(B, H, seqlen, head_dim, h_o_ref, ref_o,
4249                                         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4250 #endif
4251 }
4252 #endif
4253 } else {
4254     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::Pad) {
4255         cudaDeviceSynchronize();
4256         bench_fa_launch<128, 1, 8, 8, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4257                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4258         cudaDeviceSynchronize();
4259         bench_fa_launch<128, 2, 8, 8, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4260                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4261         cudaDeviceSynchronize();
4262         bench_fa_launch<128, 1, 8, 8, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4263                                             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4264         cudaDeviceSynchronize();
4265         bench_fa_launch<128, 2, 8, 8, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4266                                             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4267     }
4268     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleQ) {
4269         cudaDeviceSynchronize();
4270         bench_fa_launch<128, 2, 0, 8, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4271                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4272         cudaDeviceSynchronize();
4273         bench_fa_launch<128, 2, 0, 8, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4274                                             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4275     }
4276     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleK) {
4277         cudaDeviceSynchronize();
4278         bench_fa_launch<128, 2, 8, 0, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4279                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4280         cudaDeviceSynchronize();
4281         bench_fa_launch<128, 2, 8, 0, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4282                                             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4283     }
4284     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleV) {
4285         cudaDeviceSynchronize();
4286         bench_fa_launch<128, 2, 8, 8, 0, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4287                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4288         cudaDeviceSynchronize();
4289         bench_fa_launch<128, 2, 8, 8, 0, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4290                                             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4291     }
4292     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleQK) {
4293         cudaDeviceSynchronize();
4294         bench_fa_launch<128, 2, 0, 0, 8, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4295                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4296         cudaDeviceSynchronize();
4297         bench_fa_launch<128, 2, 0, 0, 8, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4298                                             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4299     }
4300     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleQV) {
4301         cudaDeviceSynchronize();
4302         bench_fa_launch<128, 2, 0, 8, 0, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4303                                             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4304         cudaDeviceSynchronize();
4305         bench_fa_launch<128, 2, 0, 8, 0, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,

```

```

4306         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4307     }
4308     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::SwizzleKV) {
4309         cudaDeviceSynchronize();
4310         bench_fa_launch<128, 2, 8, 0, 0, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4311             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4312         cudaDeviceSynchronize();
4313         bench_fa_launch<128, 2, 8, 0, 0, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4314             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4315     }
4316     if (g_fa_layout == FALayout::All || g_fa_layout == FALayout::Swizzle) {
4317         cudaDeviceSynchronize();
4318         bench_fa_launch<128, 2, 0, 0, 0, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4319             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4320         cudaDeviceSynchronize();
4321         bench_fa_launch<128, 2, 0, 0, 0, 1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4322             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4323     }
4324     #if defined(NOTES_V2_ENABLE_CUTE)
4325     {
4326         cudaDeviceSynchronize();
4327         bench_fa_2_mma_stages_cute_dispatch(
4328             B, H, seqlen, head_dim, h_o_ref, ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4329     }
4330     #endif
4331     #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4332     {
4333         cudaDeviceSynchronize();
4334         bench_fa_tma_mma_ws_dispatch<1, 1, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4335             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4336         cudaDeviceSynchronize();
4337         bench_fa_tma_mma_ws_dispatch<2, 1, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4338             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4339         cudaDeviceSynchronize();
4340         bench_fa_tma_mma_ws_dispatch<3, 1, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4341             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4342         cudaDeviceSynchronize();
4343         bench_fa_tma_mma_ws_dispatch<2, 2, 0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4344             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4345         // F32Acc variants
4346         cudaDeviceSynchronize();
4347         bench_fa_tma_mma_ws_dispatch<1, 1, 1>(B, H, seqlen, head_dim, h_o_ref,
4348             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4349         cudaDeviceSynchronize();
4350         bench_fa_tma_mma_ws_dispatch<2, 1, 1>(B, H, seqlen, head_dim, h_o_ref,
4351             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4352         cudaDeviceSynchronize();
4353         bench_fa_tma_mma_ws_dispatch<3, 1, 1>(B, H, seqlen, head_dim, h_o_ref,
4354             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4355         cudaDeviceSynchronize();
4356         bench_fa_tma_mma_ws_dispatch<2, 2, 1>(B, H, seqlen, head_dim, h_o_ref,
4357             ref_o, d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4358         // FA3-style dual-consumer
4359         cudaDeviceSynchronize();
4360         bench_fa_3_tma_ws_dispatch<0>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4361             d_q, d_k, d_v, d_o, cudnn_tflops_f16);
4362         cudaDeviceSynchronize();
4363         bench_fa_3_tma_ws_dispatch<1>(B, H, seqlen, head_dim, h_o_ref, ref_o,
4364             d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4365     }
4366     #if defined(NOTES_V2_ENABLE_CUTE)
4367     cudaDeviceSynchronize();
4368     bench_fa_2_tma_mma_ws_cute_dispatch(B, H, seqlen, head_dim, h_o_ref, ref_o,
4369         d_q, d_k, d_v, d_o, cudnn_tflops_f32);

```

```

4369     cudaDeviceSynchronize();
4370     bench_fa_3_tma_mma_ws_cute_dispatch(B, H, seqlen, head_dim, h_o_ref, ref_o,
4371         d_q, d_k, d_v, d_o, cudnn_tflops_f32);
4372     #endif
4373 }
4374 #endif
4375 }
4376 cudaDeviceSynchronize();
4377
4378 free(h_q);
4379 free(h_k);
4380 free(h_v);
4381 free(h_o_ref);
4382 free(ref_o);
4383 cudaFree(d_q);
4384 cudaFree(d_k);
4385 cudaFree(d_v);
4386 cudaFree(d_o);
4387 }
4388
4389 // Host 端 v1/v2 等价性测试: 遍历 kColStride/i/j, 断言 swizzle_v1_impl == swizzle_v2_impl.
4390 // 用于在启用 NOTES_V2_ENABLE_SWIZZLE_V2 前后验证 v2 与 v1 bit-exact 等价.
4391 // kColStride=8 在 v2 内回退 v1, 故 8 也应恒等.
4392 template <int kColStride>
4393 static void swizzle_equiv_check_one(long &total, long &fail) {
4394     for (int i = 0; i < 256; ++i) {
4395         for (int j = 0; j < kColStride; ++j) {
4396             int v1 = swizzle_v1_impl<kColStride>(i, j);
4397             int v2 = swizzle_v2_impl<kColStride>(i, j);
4398             ++total;
4399             if (v1 != v2) {
4400                 ++fail;
4401                 if (fail <= 10)
4402                     printf(" MISMATCH cs=%d i=%d j=%d: v1=%d v2=%d\n", kColStride, i, j, v1, v2);
4403             }
4404         }
4405     }
4406 }
4407
4408 static void test_swizzle_equiv() {
4409     long total = 0, fail = 0;
4410     swizzle_equiv_check_one<8>(total, fail);
4411     swizzle_equiv_check_one<16>(total, fail);
4412     swizzle_equiv_check_one<32>(total, fail);
4413     swizzle_equiv_check_one<64>(total, fail);
4414     printf("| %-56s | %-9s |\n",
4415         "Swizzle v1/v2 equiv (host)", fail == 0 ? "ALL PASS" : "FAIL");
4416     printf(" total=%ld fail=%ld\n", total, fail);
4417 }
4418
4419 int main(int argc, char *argv[]) {
4420     #if defined(NOTES_V2_ENABLE_WGMMMA) || defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4421         cuInit(0);
4422     #endif
4423
4424     for (int i = 1; i < argc; i++) {
4425         if (strcmp(argv[i], "--bench-hgemm") == 0) {
4426             g_bench_hgemm = true;
4427         } else if (strcmp(argv[i], "--bench") == 0) {
4428             g_bench_hgemm = true;
4429             g_bench_fa = true;
4430         } else if (strcmp(argv[i], "--bench-fa") == 0) {
4431             g_bench_fa = true;

```

```

4432 } else if (strcmp(argv[i], "--bench-fa3-cute") == 0) {
4433     g_bench_fa = true;
4434     g_bench_fa3_cute_only = true;
4435 } else if (strcmp(argv[i], "--bench-hgemm-all") == 0) {
4436     g_bench_hgemm_all = true;
4437 } else if (strcmp(argv[i], "--bench-fa-all") == 0) {
4438     g_bench_fa = true;
4439     g_fa_layout = FALayout::All;
4440 } else if (strcmp(argv[i], "--bench-all") == 0) {
4441     g_bench_all = true;
4442     g_fa_layout = FALayout::All;
4443 } else if (strcmp(argv[i], "--mnk") == 0 && i + 1 < argc) {
4444     sscanf(argv[++i], "%d,%d,%d", &g_bench_M, &g_bench_N, &g_bench_K);
4445 } else if (strcmp(argv[i], "--bhnd") == 0 && i + 1 < argc) {
4446     sscanf(argv[++i], "%d,%d,%d,%d", &g_bench_B, &g_bench_H, &g_bench_Nfa, &g_bench_D);
4447 } else if (strcmp(argv[i], "--fa-layout") == 0 && i + 1 < argc) {
4448     const char *layout = argv[++i];
4449     if (strcmp(layout, "all") == 0)
4450         g_fa_layout = FALayout::All;
4451     else if (strcmp(layout, "pad") == 0)
4452         g_fa_layout = FALayout::Pad;
4453     else if (strcmp(layout, "swizzle-q") == 0)
4454         g_fa_layout = FALayout::SwizzleQ;
4455     else if (strcmp(layout, "swizzle-k") == 0)
4456         g_fa_layout = FALayout::SwizzleK;
4457     else if (strcmp(layout, "swizzle-v") == 0)
4458         g_fa_layout = FALayout::SwizzleV;
4459     else if (strcmp(layout, "swizzle-qk") == 0)
4460         g_fa_layout = FALayout::SwizzleQK;
4461     else if (strcmp(layout, "swizzle-qv") == 0)
4462         g_fa_layout = FALayout::SwizzleQV;
4463     else if (strcmp(layout, "swizzle-kv") == 0)
4464         g_fa_layout = FALayout::SwizzleKV;
4465     else if (strcmp(layout, "swizzle") == 0)
4466         g_fa_layout = FALayout::Swizzle;
4467     else {
4468         fprintf(stderr, "Unsupported FA layout: %s\n", layout);
4469         return EXIT_FAILURE;
4470     }
4471 } else if (strcmp(argv[i], "--fa-skip-check") == 0) {
4472     g_fa_skip_check = true;
4473 } else if (strcmp(argv[i], "--swizzle-eq-check") == 0) {
4474     g_swizzle_eq_check = true;
4475 } else if (strcmp(argv[i], "--debug") == 0) {
4476     g_debug = true;
4477 } else if (strcmp(argv[i], "--verbose") == 0) {
4478     g_verbose = true;
4479 } else if (strcmp(argv[i], "--warmup") == 0 && i + 1 < argc) {
4480     g_warmup = atoi(argv[++i]);
4481 } else if (strcmp(argv[i], "--repeat") == 0 && i + 1 < argc) {
4482     g_repeat = atoi(argv[++i]);
4483 }
4484 }
4485
4486 if (g_swizzle_eq_check) {
4487     printf("=== notes-v2.cu swizzle v1/v2 equivalence check ===\n");
4488     printf("| %-56s | %-9s |\n", "Kernel", "Max Err");
4489     printf("|-----|-----|\n");
4490     test_swizzle_equiv();
4491     printf("=== Done ===\n");
4492     return 0;
4493 }
4494

```

```

4495 #if defined(NOTES_V2_ENABLE_CUTE) && defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4496     if (argc >= 2 && strcmp(argv[1], "--fa3-cute") == 0) {
4497         printf("=== CuTe FA3 TMA MMA WS correctness ===\n");
4498         printf("| %-56s | %-9s |\n", "Kernel", "Max Err");
4499         printf("|-----|-----|\n");
4500         test_flash_attn_3_tma_mma_ws_split_q_cute<64>();
4501         test_flash_attn_3_tma_mma_ws_split_q_cute<128>();
4502         return 0;
4503     }
4504     if (argc >= 2 && strcmp(argv[1], "--fa3-cute-tma-smoke") == 0) {
4505         printf("=== CuTe TMA copy smoke ===\n");
4506         printf("| %-56s | %-9s |\n", "Kernel", "Max Err");
4507         printf("|-----|-----|\n");
4508         test_flash_attn_3_cute_tma_copy_smoke<64>();
4509         test_flash_attn_3_cute_tma_copy_smoke<128>();
4510         return 0;
4511     }
4512     if (argc >= 2 && strcmp(argv[1], "--fa2-cute-cpasync") == 0) {
4513         printf("=== CuTe FA2 MMA Stages (cp.async) correctness ===\n");
4514         printf("| %-56s | %-9s |\n", "Kernel", "Max Err");
4515         printf("|-----|-----|\n");
4516         test_flash_attn_mma_stages_split_q_cute<64>();
4517         test_flash_attn_mma_stages_split_q_cute<128>();
4518         return 0;
4519     }
4520     if (argc >= 2 && strcmp(argv[1], "--fa2-cute") == 0) {
4521         printf("=== CuTe FA2 TMA MMA WS correctness ===\n");
4522         printf("| %-56s | %-9s |\n", "Kernel", "Max Err");
4523         printf("|-----|-----|\n");
4524         test_flash_attn_tma_mma_ws_split_q_cute<64>();
4525         test_flash_attn_tma_mma_ws_split_q_cute<128>();
4526         return 0;
4527     }
4528 #endif
4529
4530 if (g_bench_hgemm || g_bench_fa || g_bench_all) {
4531     printf("=== notes-v2.cu bench mode ===\n");
4532     printf("HGEMM: M=%d N=%d K=%d   FA: B=%d H=%d N=%d D=%d\n",
4533           g_bench_M, g_bench_N, g_bench_K,
4534           g_bench_B, g_bench_H, g_bench_Nfa, g_bench_D);
4535     printf("| %-56s | %-9s | %-19s |\n", "Kernel", "Max Err", "TFLOPS/cu{BLAS,DNN}");
4536     printf(
4537         "|-----|-----|-----|\n"
4538     );
4539
4540     if (g_bench_hgemm || g_bench_hgemm_all || g_bench_all) {
4541 #if defined(NOTES_V2_ENABLE_CUTE)
4542         bench_hgemm_cute(g_bench_M, g_bench_N, g_bench_K);
4543 #endif
4544         if (g_bench_hgemm_all || g_bench_all) {
4545             bench_hgemm_mma(g_bench_M, g_bench_N, g_bench_K);
4546             bench_hgemm_swizzle(g_bench_M, g_bench_N, g_bench_K);
4547 #if defined(NOTES_V2_ENABLE_WGMMA)
4548             bench_hgemm_wgmma(g_bench_M, g_bench_N, g_bench_K);
4549 #endif
4550 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4551             bench_hgemm_tma_mma_ws(g_bench_M, g_bench_N, g_bench_K);
4552 #endif
4553         }
4554     }
4555     if (g_bench_fa || g_bench_all)
4556         bench_flash_attn(g_bench_B, g_bench_H, g_bench_Nfa, g_bench_D);
4557

```

```

4558     printf("=== Bench done ===\n");
4559     return 0;
4560 }
4561
4562 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4563 if (argc >= 2 && strcmp(argv[1], "--tma-mma-ws") == 0) {
4564     int M = 128, N = 128, K = 64;
4565     if (argc > 4) {
4566         M = atoi(argv[2]);
4567         N = atoi(argv[3]);
4568         K = atoi(argv[4]);
4569     }
4570     printf("=== SM120 TMA MMA WS validation ===\n");
4571     printf("| %-56s | %-9s |\n", "Kernel", "Max Err");
4572     printf("|-----|-----|\n");
4573     test_hgemm_tma_mma_ws(M, N, K);
4574     return 0;
4575 }
4576 #endif
4577 int M = 1024, N = 1024, K = 1024;
4578 if (argc > 3) { M = atoi(argv[1]); N = atoi(argv[2]); K = atoi(argv[3]); }
4579
4580 printf("=== notes-v2.cu verification harness ===\n");
4581 printf("| %-56s | %-9s |\n", "Kernel", "Max Err");
4582 printf("|-----|-----|\n");
4583
4584 test_block_reduce(N);
4585 test_dot(N);
4586 test_relu(1024);
4587 test_elementwise(1024);
4588 test_histogram(1024);
4589 test_merge_attn_states(512, 16, 128);
4590 test_softmax(256);
4591 test_rms_norm(8, 128);
4592 test_layer_norm(8, 128);
4593 test_rope(8, 128);
4594 test_mat_transpose(256, 256);
4595 test_mat_transpose_padded(256, 256);
4596 test_sgemv(256, 128);
4597 test_sgemm(M, N, K);
4598 test_hgemm_mma(M, N, K);
4599 test_hgemm_swizzle(M, N, K);
4600 #if (defined(NOTES_V2_ENABLE_CUTE))
4601     test_hgemm_cute(M, N, K);
4602 #endif
4603 #if (defined(NOTES_V2_ENABLE_WGMMA))
4604     test_hgemm_wgmma(M, N, K);
4605 #endif
4606 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4607     test_hgemm_tma_mma_ws(M, N, K);
4608 #endif
4609     test_flash_attn(1024, 64);
4610 #if defined(NOTES_V2_ENABLE_TMA_MMA_WS)
4611     test_flash_attn_tma_mma_ws(1024, 64);
4612     test_flash_attn_tma_mma_ws(1024, 128);
4613     test_flash_attn_3_tma_ws(1024, 64);
4614     test_flash_attn_3_tma_ws(1024, 128);
4615 #endif
4616
4617 printf("=== All tests done ===\n");
4618 return 0;
4619 }
4620

```

```

4621 // =====
4622 // Quick build & run reference
4623 // =====
4624 // # sm_86 + CuTe (Ampere, RTX 30/40 系列, 无 WGMMMA) :
4625 // nvcc -std=c++20 -O2 -arch=sm_86 -DNOTES_V2_ENABLE_CUTE \
4626 // -I ../../third-party/cutlass/include \
4627 // -lcublas -lcuda notes-v2.cu -o notes_v2_cute_sm86.bin
4628 //
4629 // # sm_89 单独编译 + 运行:
4630 // nvcc -std=c++20 -O2 -arch=sm_89 -lcublas -lcuda notes-v2.cu -o notes_v2_sm89.bin
4631 //
4632 // # sm_89 + CuTe (需要 CUTLASS include 路径, 编译 CUDA_HOME 指向的 CUTLASS 或
4633 // 项目内置 third-party/cutlass) :
4634 // nvcc -std=c++20 -O2 -arch=sm_89 -DNOTES_V2_ENABLE_CUTE \
4635 // -I ../../third-party/cutlass/include \
4636 // -lcublas -lcuda notes-v2.cu -o notes_v2_cute_sm89.bin
4637 //
4638 // # sm_90a 单独编译 + 运行 (需要 Hopper GPU, H100/H200 均可) :
4639 // nvcc -std=c++20 -O2 -gencode arch=compute_90a,code=sm_90a \
4640 // -DNOTES_V2_ENABLE_WGMMMA -lcublas -lcuda notes-v2.cu -o notes_v2_sm90.bin
4641 //
4642 // # sm_90a + CuTe + WGMMMA:
4643 // nvcc -std=c++20 -O2 -gencode arch=compute_90a,code=sm_90a \
4644 // -DNOTES_V2_ENABLE_CUTE -DNOTES_V2_ENABLE_WGMMMA \
4645 // -I ../../third-party/cutlass/include \
4646 // -lcublas -lcuda notes-v2.cu -o notes_v2_cute_sm90.bin
4647 //
4648 // # sm_120a TMA + warp-specialized mma.sync (CUDA Toolkit >= 13.2;
4649 // RTX PRO 5000 / RTX 5090):
4650 // nvcc -std=c++20 -O2 -arch=sm_120a -DNOTES_V2_ENABLE_TMA_MMA_WS \
4651 // -lcublas -lcuda notes-v2.cu -o notes_v2_tma_mma_ws_sm120.bin
4652 // ./notes_v2_tma_mma_ws_sm120.bin --tma-mma-ws 1024 1024 1024
4653 //
4654 // # sm_120a + CUTE + TMA_MMA_WS + cuDNN SDPA (CUDA Toolkit >= 13.2;
4655 // requires cudnn-frontend submodule: git submodule update --init):
4656 // nvcc -std=c++20 -O2 -arch=sm_120a \
4657 // -DNOTES_V2_ENABLE_CUTE -DNOTES_V2_ENABLE_TMA_MMA_WS -DNOTES_V2_ENABLE_CUDNN \
4658 // -I ../../third-party/cutlass/include -I ../../third-party/cudnn-frontend/include \
4659 // -L/usr/local/cuda-13.2/targets/x86_64-linux/lib/stubs \
4660 // -lcublas -lcudnn -lnvrtc -lcuda notes-v2.cu -o notes_v2_cute_ws_sm120a.bin
4661 // # Default FA bench (kPadQ=kPadK=kPadV=8 only):
4662 // ./notes_v2_cute_ws_sm120a.bin --bench --bench-fa --bhnd 1,48,4096,64
4663 // # All FA layout variants:
4664 // ./notes_v2_cute_ws_sm120a.bin --bench --bench-fa-all --bhnd 1,48,4096,64
4665 // # FA TMA MMA WS bench (SM120, D=64 only, kStagesK=2/3):
4666 // ./notes_v2_cute_ws_sm120a.bin --bench-fa-tma-ws --bhnd 1,32,4096,64

```