Stream-K

Prateek Shukla

The first principles

For one output tile C_tile, GEMM computes:

C_tile = sum over K-tiles of (A_tile_k * B_tile_k)

If one CTA computes all K-tiles for that output tile, no inter-CTA reduction is needed.

If multiple CTAs each compute only a subset of K-tiles, each CTA produces a partial sum and those partial sums must be merged.

In this scheduler, that merge step is called fixup.

The unit of work

each work unit includes:

- tile coordinates (m_idx, n_idx, l_idx)

- k_idx: start k-tile within the output tile

- k_tile_count: number of k-tiles this work unit computed for this output tile

reduction need is decided by:

- full tile in k: k_tile_count == k_tiles_per_output_tile -> no reduction needed

- partial tile in k: k_tile_count != k_tiles_per_output_tile -> reduction needed

that is the key predicate behind requires_fixup(...).

Split

A single CTA's assigned range of k-tile iterations may land in the middle of an output tile. For example, with 3 output tiles of 90 k-tiles each and 4 CTA units:

```
Total k-iters: 270, divided among 4 units = ~67-68 each
Unit 0: iters [0,   67)  → tile 0: k-tiles [0,  67)     ← partial tile 0
Unit 1: iters [67, 135)  → tile 0: k-tiles [67, 90)     ← partial tile 0 (end)
                           tile 1: k-tiles [0,  45)     ← partial tile 1 (start)
Unit 2: iters [135, 203) → tile 1: k-tiles [45, 90)     ← partial tile 1 (end)
                           tile 2: k-tiles [0,  23)     ← partial tile 2 (start)
Unit 3: iters [203, 270) → tile 2: k-tiles [23, 90)     ← partial tile 2 (end)
```

A "split" is a CTA's contribution to a single output tile. Unit 1 above has two splits one for the tail of tile 0 and one for the head of tile 1. The code processes these one at a time (that's the k_tile_remaining loop in advance_to_next_work).

Tracking the triplet

For a single split (one CTA's work on one output tile):

`K_idx`: Where this split starts within the output tile's K dimension.

`K_idx = tile_iter_start - output_tile_iter_start`.

For Unit 0's work on tile 0, `K_idx` = 0. For Unit 1's work on tile 0, K_idx = 67.

`k_tile_count`: How many k-tiles this split processes. For Unit 0 on tile 0, that's 67. For Unit 1 on tile 0, that's 23 (= 90 - 67).

`is_final_split()`: `(K_idx + k_tile_count) == k_tiles_per_output_tile`. True when this split covers the end of the K dimension. Unit 1's split on tile 0 is a final split (67 + 23 = 90).

The three roles follow directly

Given that an output tile might have 2-4 CTAs each computing a portion of K:

`K_idx == 0` → you're the first split. You computed k-tiles [0, N). Nobody wrote to workspace before you. Just store.

`is_final_split() == true` and not separate reduction → you're the final split and epilogue owner. You computed k-tiles [X, 90). Wait for everyone before you, load their accumulated result, add yours, run the epilogue.

Everything else → you're a middle split. You computed k-tiles [A, B) where 0 < A < B < 90. You need to reduce your partials into what's already in workspace.

The lock: what it is physically

There's a contiguous array of ints in global memory, one per output tile (times num_barriers for multi-warp-group kernels).

The pointer to this array sits right after the reduction data buffer in the same allocation. Every lock starts at 0 when kernel launches.

The lock is a single integer that encodes how much K-dimension work has been completed and written to workspace for a given output tile. In normal (non separate reduction) mode, it counts cumulative k-tiles processed. It only ever increases.

The lock encodes progress in K-tile space. For deterministic mode, each split waits for the exact cumulative K-tile count that matches its starting position, enforcing a strict left-to-right reduction order. For non-deterministic mode, middle splits just need to know the workspace has been initialized (lock >= 1), and they race to atomically reduce into it.

Groups

Groups are an L2 cache locality optimization. They partition the stream-K units into G independent sub-groups, where each group collaborates only on its own subset of stream-K tiles. This is an optimization specific to stream-K as stream-K breaks the nice wave-based rasterization pattern because a single CTA might span tiles from different regions of the output grid, destroying that locality.

Without groups (G=1), all stream-K units share one big pool of K tiles across all stream-K output tiles. Unit 0 might work on tile 0 and tile 1, while unit 7 works on tile 5 and tile 6 completely different spatial positions. Their data in L2 cache doesn't overlap at all.

With groups, units 0 in each group will compute identical K extents of tiles that would be assigned in the same wave according to the rasterization order of the data-parallel formulation

Group hierarchy

Groups (up to 8, for L2 locality)

└── Each group contains multiple cluster-tiles

└── Each cluster contains multiple CTAs (thread blocks)

└── Each CTA processes K-tiles

The grouping is determined along the rasterization dimension. For example, if you rasterize along M and have problem_blocks_m / cluster_m = 4, you'd get 4 groups. Groups are interleaved in the output space. Final `output_tile_id` is computed as:

```
output_tile_id = (output_tile_id_in_group * num_groups) + group_idx
```

HyTiS

HyTiS solves wave quantization by making the partial wave use finer-grained tiles so more SMs stay busy. It's a purely spatial decomposition (M×N) with heterogeneous tile sizes, vs Stream-K's K-dimension decomposition with homogeneous tile sizes.

Instead of using stream-k, HyTiS uses two different tile sizes in one kernel launch:

- Large tiles (e.g., 128×256) for full waves ⟶ max throughput
- Small tiles (e.g., 64×64) for the partial wave ⟶ min latency

No reduction, no workspace, no barriers, no fixup

The tradeoff is that HyTiS can't help when the problem is small in M and N but large in K