https://www.gpumode.com/leaderboard/399

Basic Structure

The problem requires us to multiply fp8 2 matrices in column major layout, with group scalars for 128x128 blocks of B and 128x1 blocks of A. The MI300 FP8 MMA atom has rows packed in registers, so we have to transpose tiles.

The basic implementation follows the usual tiled matrix multiplication method: Each GPU block handles a `block_tile_m * block_tile_n` block matrix of C. The k dimension is divided into `block_tile_k` tiles. The block is then divided into `warp_tile_m`, `warp_tile_n`, and `warp_tile_k` sub-blocks, which correspond to the size of the MMA instruction. The MI300 supports 2 instructions for this operation: 32x32x16 and 16x16x32. The `block_tile_m * block_tile_n` matrix is divided into `warp_tile_m * warp_tile_n` tiles, and each warp in the block is assigned a rectangular block of tiles from the block's C tile. The block's A and B tile are kept in shared memory. In my implementation I always assume that `block_tile_[mnk] <= 128` to make the main loop simpler wrt loading group scalars.

The general loop is then as follows: * Load A and B group scalars * Load A block tile from global to registers * Load B block tile from global to registers * Transpose A block tile and store to shared memory. * Transpose B block tile and store to shared memory * Load A warp tiles from shared memory * Load B warp tiles from shared memory * FP8 MMA * Scaling + accumulate

Optimizations

A brief explanation of some optimizations that I've made. ### Memory Loading The MI300 has memory problems: The architecture cannot satisfy memory requests at the theoretical maximum rate. My personal theory is that there is a problem with memory requests between IODs, as memory performance gets better (per IOD) when partitioning the GPU in 4 or 8. This all means that memory loading needs to be done carefully to get good performance. There are some hand rules that I've exploited: - MI300 memory accesses should be as large as possible to avoid overhead of requesting memory. The maximum size that can be loaded in a single instruction is 128 bits / 16 bytes (`global_load_dwordx4`) - The MI300 *does not really care* about coalescing; we just need any individual memory request from a wave to hit **complete cache lines**. - The MI300's large cache distributed across the die adds overhead to memory loading. Any request that is not cached should bypass the cache. During GEMM, inputs have high cache hit rate in general, but outputs are only written once. - The MI300 supports the `nt` attribute on memory instructions which disables cache. Using this changes max memory loading performance from 4 TB/s to 4.4 TB/s. - In all cases of the challenge inputs, the matrix fits in L3 cache, so we don't really need to worry about efficient block tile access order initially.

Because of the 128-byte cache line size, we should aim for block tiles of `128 x X` to hit full cache lines. In the initial implementation for memory loading, I used divided threads such that the each contiguous group of `m / sizeof(load_type)` threads load a column of the block tile. `load_type` is the type used to perform memory loads, which directly corresponds to the amount of bytes transferred by a single thread in a single instruction. Usually, `load_type == __uint128_t`.

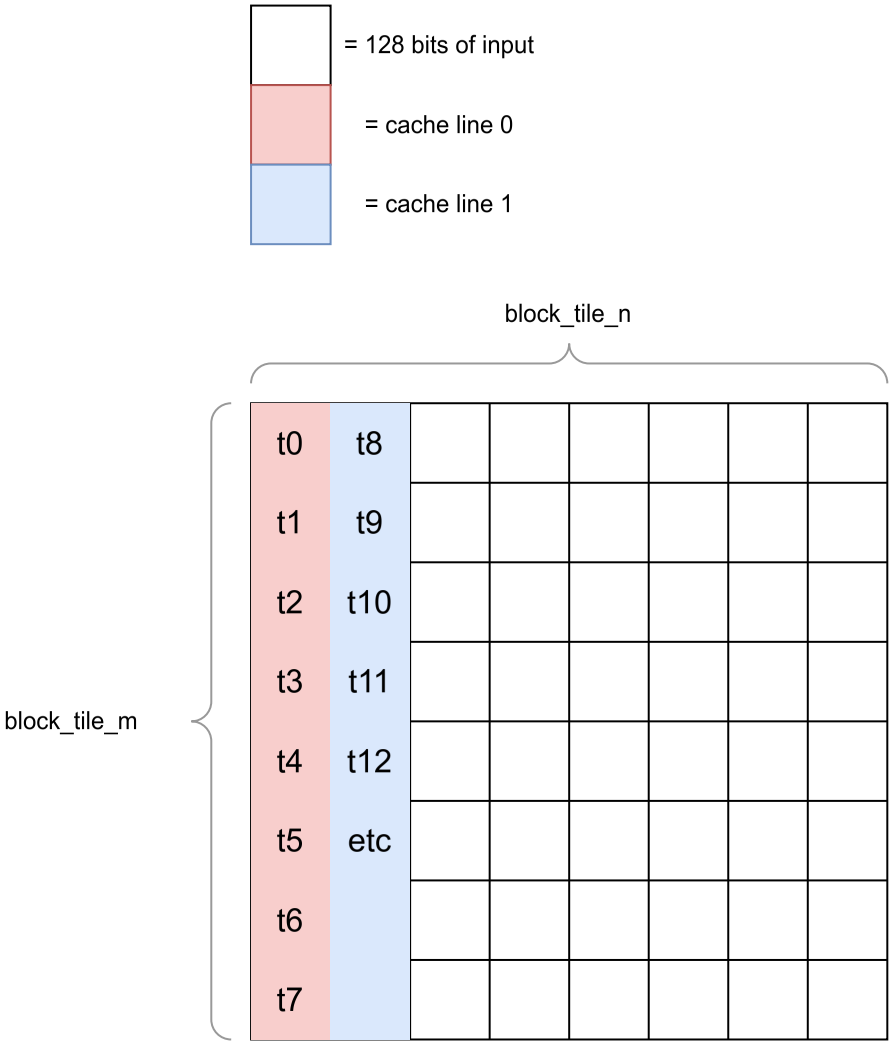

Figure 1: Memory Loading Pattern

Transposing input tiles in registers using warp shuffle

My initial implementation simply reinterpreted loaded `__uint128_t` groups into bytes and wrote each bytes into shared memory. This is inefficient, however, as the combination of bit shuffles, masks, and `ds_write_b8` is relatively slow. Instead 4x4 bytes of the input are transposed using cross-lane shuffle operations and `__builtin_amdgcn_perm`, AMD's version of `__byteperm`. This happens in 2 steps: First, 2x2 groups are transposed by performing a shuffle and a byteperm. Then, groups of 2x2 are transposed using another shuffle and byteperm. After this, we can use regular `ds_write_b32` to write to shared memory, which is much faster. ### Transposing input tiles in registers using dpp quad shuffles AMD architectures have multiple types of shuffle instructions. The fastest is known as a `DPP` shuffle. The quad-shuffle variant of this allows arbitrary permutation between groups of 4 threads of the input, but using the memory loading method outlined above, this the 4x4 groups are `m / sizeof(load_type)` lanes apart. By using our knowledge that coalescing doesn't matter as long as the instruction hits complete cache lines, we can re-organize our memory loading pattern to load 4x4 tiles into neighboring lanes.

Using buffers for memory loading

MI300 does not have a TMA or similar like Hopper, but it is possible to use MUBUF instructions (like `buffer_load_dwordx4`) to perform bounds checking. These support 1-dimensional bounds checking only; 2D bounds checking is performed by manually checking the bounds and adding an offset to the loaded pointer. This is slightly cheaper than performing branch-based bounds checking, optimizes better in general, and produces more readable assembly. LLVM knows of these instructions, but the builtins are only exposed in ROCm 6.4. In ROCm 6.3, we have to manually invoke the LLVM IR instruction. In my solution, the `buffer` class is used to abstract memory loading in this way.

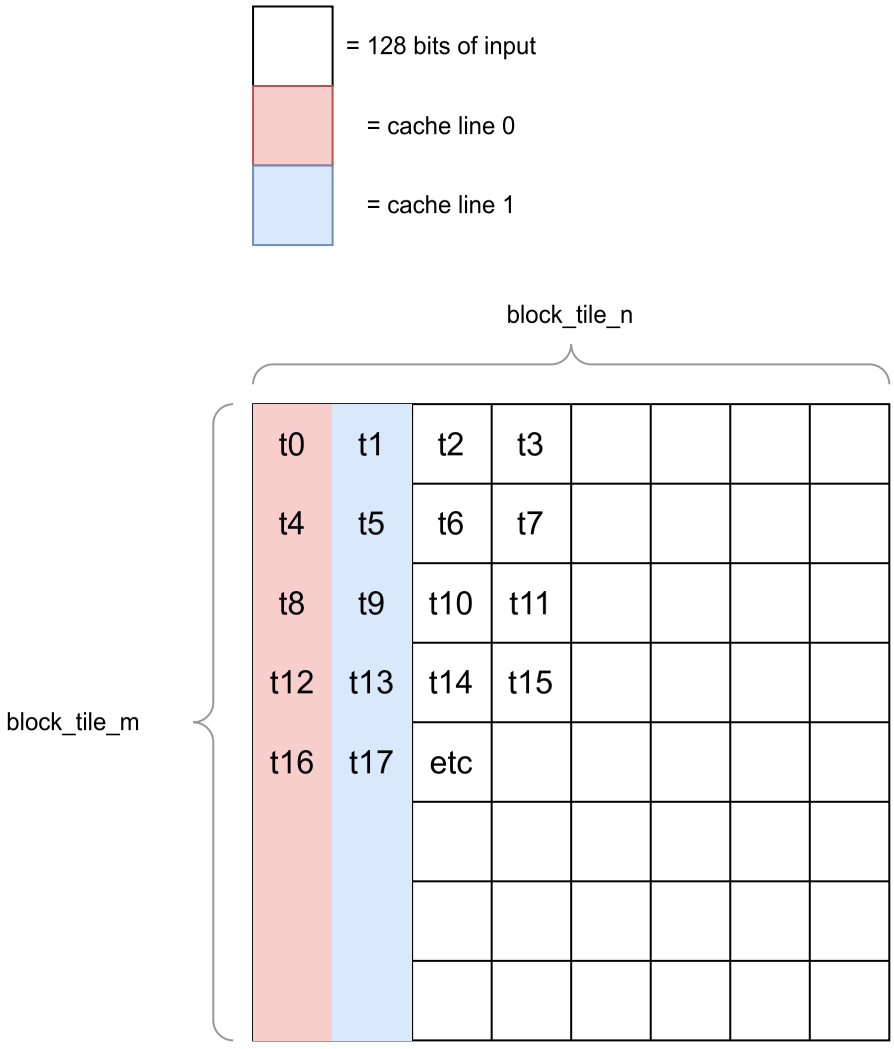

Figure 2: Memory Loading Pattern