Problem 1: FP8 Groupwise GEMM

The edge between traditional "quantization" and low-precision operators lies at the ***float8*** datatype. In this problem, you will be implementing and optimizing a **W8A8** grouped generalized matrix multiplication (GEMM) kernel with $(128, 128)$ block-wise scaling factors. In this problem, **you are not implementing the standard BLAS GEMM, so pay close attention to the instructions below**. We describe the problem below to avoid any ambiguity regarding the problem.

- The input matrices are in NT (non-transposed) format.
- You will only need to compute $\mathbf{A} @ \mathbf{B}$ with proper scaling factors, and you will store to an output matrix that is already allocated on-device.
    - The matrices will be in FP8, the scale factors in FP32, and the accumulator / final output in BF16.
- All input tensors **will start in on-device memory** (HBM / DRAM).
- Constants / shapes are guaranteed to divide 128.

Formally, given:

- A set of constants M, N, K.
- a random FP8 (**e4m3**) input tensor $\mathbf{A} \in \mathbb{FP}_8{}^{M \times K}$ stored in column-major order.
- a random FP8 (**e4m3**) input tensor $\mathbf{B} \in \mathbb{FP}_8{}^{N \times K}$ stored in column-major order.
- a zero-initialized BF16 (**e8m7**) accumulator tensor $\mathbf{C} \in \mathbb{BF}_{16}{}^{M \times N}$

The scaling factors are applied as unique scalars to every 1×128-block chunk in the rows of $\mathbf{A}$ and every 128×128-block chunk in $\mathbf{B}$. So the

- LHS scaling factor tensor $\alpha \in \mathbb{FP}_{32}{}^{M \times (K/128)}$ stored in column-major order.

- RHS scaling factor tensor $\beta \in \mathbb{FP}_{32}{}^{(N/128)\times(K/128)}$ stored in column-major order.

All of these tensors **will start in on-device memory** (HBM / DRAM), and you will compute

$$\mathbf{C} = \text{bfloat16}(\alpha_{\text{broadcasted}} \cdot \mathbf{A} \times \mathbf{B}^T \cdot \beta_{\text{broadcasted}})$$

The explanation above might be a little confusing (the block-wise scaling is really nice when doing tiled matrix multiplication), so below is an illustration of how the LHS and RHS scalars broadcast and multiply (elementwise) with the input matrices:

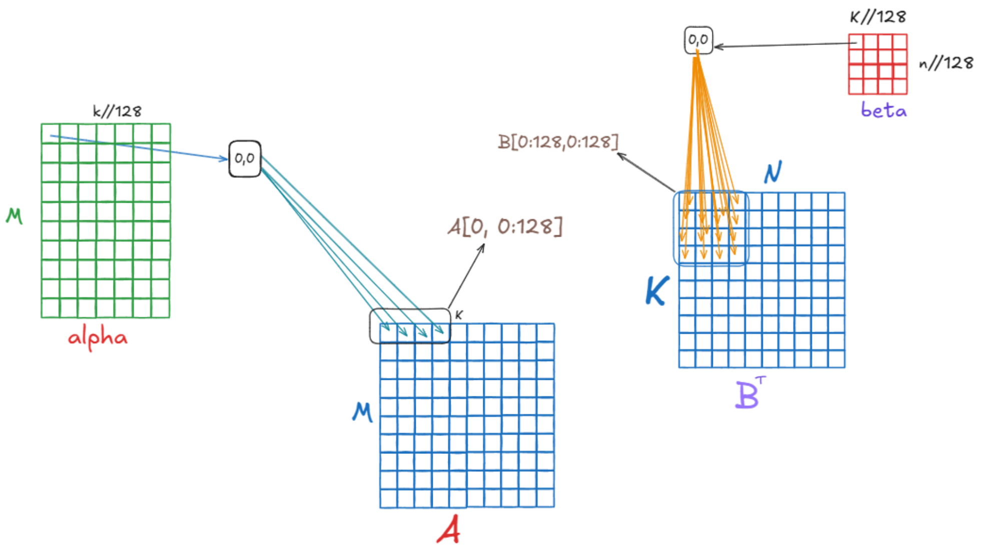

Figure 1 (by the amazing Shekhar from AMD!). We show how the LHS (alpha) and RHS (beta) scaling factors are multiplied elementwise into $\mathbf{A}$ and $\mathbf{B}$ respectively. Each element in alpha maps to a 1×128 chunk in $\mathbf{A}$, while each element in beta maps to a 128×128 chunk in $\mathbf{B}$!

Code Version. If you like reading code better, you are essentially writing a ROCm kernel for:

```
# Defaults to ROCm if AMD GPU available.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
BLOCK_K = 128
BLOCK_N = 128

# Example of FP8 (e4m3) matrices
A = torch.randn(M, K, dtype=torch.float8_e4m3fnuz, device=device)
B = torch.randn(N, K, dtype=torch.float8_e4m3fnuz, device=device)
lhs = torch.randn(M, K // BLOCK_K, dtype=torch.float32, device=device)
rhs = torch.randn(N // BLOCK_N, K // BLOCK_K,
                  dtype=torch.float32, device=device)
C = torch.randn(M, N, dtype=torch.bfloat16, device=device)

# You will be implementing this kernel.
# The inputs will be wrapped in a tuple argument called "data".
def kernel(A: torch.Tensor,    # [M, K]
           B: torch.Tensor,    # [N, K]
           C: torch.Tensor,    # [M, N]
           lhs: torch.Tensor,  # [M, K // 128]
           rhs: torch.Tensor): # [N // 128, K // 128]
  # Constants
  m = a.shape[0]
  n = b.shape[0]
  k = a.shape[1]
  scale_n = rhs.shape[0]
  scale_k = rhs.shape[1]

  # Apply scaling to input 'a'
  # Shape: [m, scale_k, BLOCK_K]
  lhs = lhs.unsqueeze(-1).repeat(1, 1, BLOCK_K)
  lhs = lhs.reshape(m, scale_k * BLOCK_K)
  lhs = lhs[:, :k] # If k doesn't cleanly divide 128
```

```
    # Apply scaling to input 'b'
    rhs = (
        rhs.view(-1, 1)
        .repeat(1, BLOCK_N * BLOCK_K)
        .view(scale_n, scale_k, BLOCK_N, BLOCK_K)
        # Reorder dimensions: [scale_n, blk_n, scale_k, blk_k]
        .permute(0, 2, 1, 3)
        .reshape(scale_n * BLOCK_N, scale_k * BLOCK_K)
    )
    rhs = rhs[:n, :k]

    # Compute matmul
    C = (torch.matmul(
                lhs * A.to(torch.float32),
                (B.T).to(torch.float32) * rhs
              ).to(torch.bfloat16)

      return C
```

Note. In the example above, the tensors are row-major order, but all tensors including the scale factors (see the reference code for more detail) will be provided in the NT format, so column major order (e.g. M x K, N x K column-major).

Problem Constraints and Scoring:

The ranking criteria is the **geometric mean** of the benchmark results. For the grand prize, your kernel will be evaluated against the speed of light analysis and the solution closest to the speed of light will be awarded the grand price.

The speed of light numbers provided by AMD are:

M	N	K	time[µs]
1024	1536	7168	8.6331019
1024	4608	7168	25.8936898
6144	1536	7168	51.7775517

6144	4608	7168	155.2989590
1024	7168	256	3.1671426
6144	7168	256	17.2712935

Problem shapes:

- $M, K, N \in (1024, 1536, 7168)$
- $M, K, N \in (1024, 4608, 7168)$
- $M, K, N \in (6144, 1536, 7168)$
- $M, K, N \in (6144, 4608, 7168)$
- $M, K, N \in (1024, 7168, 256)$
- $M, K, N \in (6144, 7168, 256)$