```python
""" Core mathematics for Additive Quantization (AQ): initialization, reconstruction and beam search"""
import random
from typing import List, Optional, Tuple, Union

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.checkpoint import checkpoint
from tqdm.auto import trange

from src.kmeans import find_nearest_cluster, fit_faiss_kmeans, fit_kmeans, fit_kmeans_1d
from src.utils import ellipsis, maybe_script


class QuantizedLinear(nn.Module):
    def __init__(self, quantized_weight, bias: Optional[nn.Parameter]):
        super().__init__()
        self.out_features, self.in_features = quantized_weight.out_features, quantized_weight.in_features
        self.quantized_weight = quantized_weight
        self.bias = bias
        self.use_checkpoint = False

    def _forward(self, input: torch.Tensor):
        return F.linear(input, self.quantized_weight(), self.bias)

    def forward(self, input: torch.Tensor):
        if self.use_checkpoint and torch.is_grad_enabled():
            return checkpoint(
                self._forward, input, use_reentrant=False, preserve_rng_state=False, determinism_check="none"
            )
        return self._forward(input)


class QuantizedWeight(nn.Module):
    EPS = 1e-9

    def __init__(
        self,
        *,
        XTX: torch.Tensor,
        reference_weight: torch.Tensor,
        in_group_size: int,
        out_group_size: int,
        num_codebooks: int,
        nbits_per_codebook: int = 8,
        codebook_value_nbits: int = 16,
        codebook_value_num_groups: int = 1,
        scale_nbits: int = 0,
        scale_in_group_size: Optional[int] = None,
        scale_out_group_size: Optional[int] = None,
        straight_through_gradient: Optional[bool] = None,
        **init_kwargs,
    ):
        super().__init__()
        self.out_features, self.in_features = reference_weight.shape
        assert self.in_features % in_group_size == 0
        assert self.out_features % out_group_size == 0

        self.out_group_size, self.in_group_size = out_group_size, in_group_size
        self.num_codebooks = num_codebooks
        self.nbits_per_codebook = nbits_per_codebook
        self.codebook_size = codebook_size = 2**nbits_per_codebook
        self.codebook_value_nbits = codebook_value_nbits
        self.codebook_value_num_groups = codebook_value_num_groups
        self.codebook_value_clusters = None

        if scale_in_group_size is None:
            scale_in_group_size = in_group_size
        if scale_out_group_size is None:
            scale_out_group_size = out_group_size
        assert scale_out_group_size % out_group_size == 0
        assert scale_in_group_size % in_group_size == 0
```

```
        self.scale_in_group_size_factor = scale_in_group_size // in_group_size
        self.scale_out_group_size_factor = scale_out_group_size // out_group_size


        self.scales = self.scales_clusters = self.scales_indices = None
        if straight_through_gradient is None and scale_nbits > 0:
            straight_through_gradient = scale_nbits >= 6
        self.straight_through_gradient = straight_through_gradient
        self.scale_nbits = scale_nbits

        with torch.no_grad():
            weight_groupwise_for_scales = reference_weight.reshape(
                self.out_features // scale_out_group_size, scale_out_group_size,
                self.in_features // scale_in_group_size, scale_in_group_size
            ).swapaxes(1, 2)  # [num_out_groups, num_in_groups, out_group_size, in_group_size]

            if scale_nbits > 0:
                scales = weight_groupwise_for_scales.norm(dim=(2, 3), keepdim=True) + self.EPS
            else:
                scales = weight_groupwise_for_scales.flatten(1, -1).norm(dim=-1).view(-1, 1, 1, 1) + self.EPS
            # shape [num_out_groups, num_in_groups, 1, 1] if scale_nbits > 0 else [num_out_groups, num_in_groups, 1, 1]
            del weight_groupwise_for_scales


            weight_groupwise = reference_weight.reshape(
                self.out_features // out_group_size, out_group_size, self.in_features // in_group_size, in_group_size
            ).swapaxes(1, 2)  # [num_out_groups, num_in_groups, out_group_size, in_group_size]


            self.scales_are_lossless = scale_nbits == 0 or scale_nbits >= 16 or (2**scale_nbits >= scales.shape[1])
            if self.scales_are_lossless or self.straight_through_gradient:
                # ^-- this checks if scales can be preserved losslessly
                self.scales = nn.Parameter(scales, requires_grad=True)
            else:
                scales_clusters, scales_indices, _ = fit_kmeans_1d(scales.flatten(1, -1), k=2**scale_nbits)
                self.scales_clusters = nn.Parameter(scales_clusters, requires_grad=True)
                self.scales_indices = nn.Parameter(scales_indices, requires_grad=False)

            weight_for_init = (weight_groupwise / self.get_scales()).swapaxes(1, 2).reshape_as(reference_weight)
            del weight_groupwise

        codes, codebooks = init_aq_kmeans(
            weight_for_init,
            num_codebooks=num_codebooks,
            out_group_size=out_group_size,
            in_group_size=in_group_size,
            codebook_size=self.codebook_size,
            **init_kwargs,
        )

        self.codebooks = nn.Parameter(
            codebooks, requires_grad=True
        )  # [num_codebooks, codebook_size, out_group_size, in_group_size]
        self.codes = nn.Parameter(codes, requires_grad=False)  #  [num_out_groups, num_in_groups, num_codebooks]

    def get_codebooks(self) -> torch.Tensor:
        """Get quantization codebooks or reconstruct them from second level quantization (see codebook_values_nbits)"""
        if self.codebook_value_nbits >= 16:
            return self.codebooks
        elif 0 < self.codebook_value_nbits < 16:
            with torch.no_grad():
                codebooks_dimshuffle = (
                    self.codebooks.reshape(
                        self.num_codebooks,
                        self.codebook_value_num_groups,
                        self.codebook_size // self.codebook_value_num_groups,
                        self.out_group_size,
```

```
                        self.in_group_size,
                    )
                    .permute(0, 1, 3, 4, 2)
                    .flatten(0, -2)
                )
                self.codebook_value_clusters, _unused, reconstructed_codebooks_dimshuffle = fit_kmeans_1d(
                    codebooks_dimshuffle,
                    k=2**self.codebook_value_nbits,
                    initial_clusters=self.codebook_value_clusters,
                )
                reconstructed_codebooks = (
                    reconstructed_codebooks_dimshuffle.view(
                        self.num_codebooks,
                        self.codebook_value_num_groups,
                        self.out_group_size,
                        self.in_group_size,
                        self.codebook_size // self.codebook_value_num_groups,
                    )
                    .permute(0, 1, 4, 2, 3)
                    .reshape_as(self.codebooks)
                )
            if torch.is_grad_enabled():
                reconstructed_codebooks = reconstructed_codebooks + (self.codebooks - self.codebooks.detach())
            return reconstructed_codebooks
        raise NotImplementedError(f"{self.codebook_value_nbits}-bit codebook values are not supported")

    def get_scales(self) -> torch.Tensor:
        """Get per-channel or per-group quantization scales or reconstruct those scales based on scales_nbits"""
        if self.scale_nbits == 0 or self.scales_are_lossless:
            scales = self.scales  # scales are not quantized or the quantization is lossless
        elif self.straight_through_gradient:
            with torch.no_grad():
                self.scales_clusters, _, dequantized_scales = fit_kmeans_1d(
                    self.scales.flatten(1, -1), k=2**self.scale_nbits, initial_clusters=self.scales_clusters
                )
                dequantized_scales = dequantized_scales.reshape_as(self.scales)
            if torch.is_grad_enabled() and self.scales.requires_grad:
                dequantized_scales = dequantized_scales + (self.scales - self.scales.detach())
            scales = dequantized_scales
        else:  # train scale codebook only
            scales = self.scales_clusters.gather(1, self.scales_indices)[:, :, None, None]

        if scales.numel() != scales.shape[0]: # group-wise scales   (i.e. not 1d scales)
            assert scales.ndim == 4
            assert scales.shape[2] == scales.shape[3] == 1
            # if replicate each scale several times
            scales = scales[:, None, :, None, :, :].tile(
                1, self.scale_out_group_size_factor, 1, self.scale_in_group_size_factor, 1, 1
            ).flatten(2, 3).flatten(0, 1)
        return scales

    def forward(self, selection: Union[slice, ellipsis, torch.Tensor] = ...):
        """
        Differentably reconstruct the weight (or parts thereof) from compressed components
        :param selection: By default, reconstruct the entire weight. If selection is specified, this method will instead
            reconstruct a portion of weight for the corresponding output dimensions (used for parallelism).
            The indices / slices must correspond to output channels (if out_group_size==1) or groups (if > 1).
            Formally, the indices must be in range [ 0 , self.out_features // self.out_group_size )

        """
        weight = _dequantize_weight(self.codes[selection], self.get_codebooks(), self.get_scales()[selection])
        return weight

    @torch.no_grad()
    def beam_search_update_codes_(
```

```python
        self,
        XTX: torch.Tensor,
        reference_weight: torch.Tensor,
        *,
        selection: Union[slice, ellipsis, torch.LongTensor] = ...,
        **kwargs,
    ) -> torch:
        """
        Update self.codes in-place via beam search so as to minimize squared errors. Return the updated codes.
        :param XTX: pairwise products of input features matmul(X.transpose(), X), shape: [in_features, in_features]
        :note: if XTX is divided by dataset size, this function will return *mean* squared error
        :param reference_weight: original weight matrix that is being quantized, shape: [out_features, in_features]
        :note: if selection is specified, reference_weight must instead be [num_selected_out_features, in_features]
        :param selection:  By default, this function updates all codes, If selection specified, it will instead
            update only the codes for a portion of output dimensions (used for parallelism).
            The indices / slices must correspond to output channels (if out_group_size==1) or groups (if > 1).
            Formally, the indices must be in range [ 0 , self.out_features // self.out_group_size )
        :param beam_size: consider up to this many best encoding combinations (this param is passed through via kwargs)
        :param kwargs: any additional keyword arguments are forwarded to beam_search_optimal_codes function
        :returns: the updated codes
        """
        if self.scale_out_group_size_factor != 1:
            assert selection == ..., "todo implement selection with scale_out_group_size"
        self.codes[selection] = beam_search_optimal_codes(
            XTX=XTX,
            reference_weight=reference_weight,
            codebooks=self.get_codebooks(),
            prev_codes=self.codes[selection],
            scales=self.get_scales()[selection],
            **kwargs,
        )
        return self.codes[selection]

    def estimate_nbits_per_parameter(self) -> float:
        """Calculate the effective number of bits per original matrix parameters"""
        num_parameters = self.out_features * self.in_features
        group_size = self.out_group_size * self.in_group_size
        num_out_groups = self.out_features // self.out_group_size
        num_in_groups = self.in_features // self.in_group_size

        matrix_store = num_parameters // group_size * self.num_codebooks * self.nbits_per_codebook

        codebooks_store = self.num_codebooks * self.codebook_size * group_size * self.codebook_value_nbits
        if self.codebook_value_nbits < 16:
            codebooks_store += (
                2**self.codebook_value_nbits * self.num_codebooks * self.codebook_value_num_groups * group_size * 16
            )

        if self.scale_nbits >= 16 or 2**self.scale_nbits >= num_in_groups:  # group-wise scales in 16 bit
            scale_num_out_groups = num_out_groups // self.scale_out_group_size_factor
            scale_num_in_groups = num_in_groups // self.scale_in_group_size_factor
            scale_store = self.scale_nbits * scale_num_out_groups * scale_num_in_groups
        elif 0 < self.scale_nbits < 16:  # use scale quantization codebooks
            scale_store = self.scale_nbits * num_out_groups * num_in_groups
            scale_store += num_out_groups * 2**self.scale_nbits * 16
        elif self.scale_nbits == 0:  # no group-wise scales; use global 1d scales instead
            scale_store = num_out_groups * 16
        else:
            assert False

        return (matrix_store + codebooks_store + scale_store) / num_parameters
```

```python

    def extra_repr(self) -> str:
        return f"{self.out_features=}, {self.in_features=}, bits_per_parameter={self.estimate_nbits_per_parameter()}"


@torch.inference_mode()
def beam_search_optimal_codes(
    *,
    XTX: torch.Tensor,
    reference_weight: torch.Tensor,
    codebooks: torch.Tensor,
    prev_codes: torch.IntTensor,
    scales: Optional[torch.Tensor],
    beam_size: int,
    dim_rng: Optional[random.Random] = None,
    code_penalties: Optional[torch.Tensor] = None,
    verbose: bool,
):
    """
    :param XTX: pairwise products of input features matmul(X.transpose(), X), shape: [in_features, in_features]
    :note: if XTX is divided by dataset size, this function will return *mean* squared error
    :param reference_weight: original weight matrix that is being quantized, shape: [out_features, in_features]
    :param codebooks: look-up tables of codes, shape: [num_codebooks, codebook_size, out_group_siz, in_group_size]
    :param prev_codes: previous-best integer weight codes, shape: [num_out_groups, num_in_groups, num_codebooks]
    :param scales: weight will be multiplied by this factor, shape = [num_out_groups, num_in_groups or 1, 1, 1]
    :param dim_rng: a source of randomness to (optionally) shuffle the order in which the beam search runs
      None = update dimensions and codebooks in their natural order (0, 1, ..., n)
      random.Random(optional_seed) = shuffle dimensions at random, optionally using the specified seed

    :param beam_size: consider up to this many best encoding combinations
    :param code_penalties: a pytorch float tensor of shape [num_codebooks, codebook_size]
        Penalize the beam search objective by code_penalties[m][i] for every use of ith code from mth codebook
    :param verbose: if True, draw a progressbar and periodically print best loss
    :return: best quantization codes found, same shape as prev_codes

    :intuition: the beam search needs to produce weight codes that minimize MSE error
    - the codes are of shape [out_features / out_group_size, in_features / in_group_size, num_codebooks]

    Out of those three dimensions, out_features is "independent", i.e. changing code in
    one output feature does not increase the MSE error for another feature. Therefore,
    beam search for different output features can run in independently in parallel.

    Neither (in_features / in_group_size) nor (num_codebooks) dimension are independent:
    - changing the encoding for one feature can compensate the error from encoding another, OBC-style
    - for a single weight group, changing code in one codebook can affect the optimal choice in another codebook
    Therefore, beam search must go in a double loop over (in_features/in_group_size) and (num_codebooks) dimensions

    This leaves one choice: which dimension used for outer loop, and which one goes is in the inner loop?
    Due to the nature of beam search, interactions between dimensions of inner loop will be explored better.
    We chose to use (in_features/in_group_size) in the outer loop and (num_codebooks) for the inner loop.
    This is based on an intuition from GPTQ: you can get decent performance by quantizing each input unit ...
    ... greedily --- GPTQ does not change quantizations for previously quantized features and works fine.
    Therefore, we believe that we can also use a greedy approach to compensate error between input features.
    In turn, we believe that the codes used to encode the same weights (additively) are more inter-dependent.
    This should be treated as an educated guess with no proof and no ablation (as of the time
```

```
of writing).

    """
    num_out_groups, num_in_groups, num_codebooks = prev_codes.shape
    num_codebooks, codebook_size, out_group_size, in_group_size = codebooks.shape
    in_features = num_in_groups * in_group_size
    out_features = num_out_groups * out_group_size
    assert reference_weight.shape == (out_features, in_features)
    prev_weight = _dequantize_weight(prev_codes, codebooks, scales)

    # initialize all beam codes as previous codes - so they can be updated during beam search
    beam_codes = prev_codes.unsqueeze(0)
    # beam_codes shape: [current beam_size, num_out_groups, num_in_groups, num_codebooks], initial beam_size = 1
    beam_weights = prev_weight.unsqueeze(0)
    # beam_weights shape: [current beam_size, out_features, in_features], initial beam size = 1

    beam_losses = (
        _channelwise_squared_error(XTX, prev_weight, reference_weight)
        .reshape(1, num_out_groups, out_group_size)
        .sum(-1)
    )
    # beam_losses shape: [current beam_size, num_out_groups], initial beam_size = 1
    if code_penalties is not None:
        # Compute counts for each code in each codebook, initialize regularizer
        codebook_ids = torch.arange(num_codebooks, device=beam_losses.device).view(1, 1, 1, -1)
        per_channel_regularizers = code_penalties[codebook_ids, beam_codes].sum(dim=(2, 3))  # [beam_size, num_out_groups]
        del codebook_ids
        beam_losses += beam_losses + per_channel_regularizers

    if verbose:
        progressbar = trange(num_in_groups * num_codebooks)

    def _make_range(n: int) -> list:
        seq = list(range(n))
        if dim_rng is not None:
            dim_rng.shuffle(seq)
        return seq

    for input_group_index in _make_range(num_in_groups):
        for codebook_index in _make_range(num_codebooks):
            ### part 1: compute losses for every possible candidate for one given codebook and input group.
            # Currently, we compute errors for all output features in parallel in a vectorized fashion.
            best_losses, best_indices = _beam_search_squared_errors(
                XTX=XTX,
                reference_weight=reference_weight,
                codebooks=codebooks,
                scales=scales,
                beam_losses=beam_losses,
                beam_codes=beam_codes,
                beam_weights=beam_weights,
                input_group_index=input_group_index,
                codebook_index=codebook_index,
                k_best=beam_size,
                code_penalties=code_penalties,
            )  # [current beam_size, codebook_size, num_out_groups]

            # part 2: select beam_size new best codes and re-arrange beam to account for the fact that ...
            # ... sometimes two or more top candidates originate from the same source in previous beam
            beam_codes, beam_weights, beam_losses = _beam_search_select_best(
                beam_codes=beam_codes,
                beam_weights=beam_weights,
                codebooks=codebooks,
                scales=scales,
                input_group_index=input_group_index,
                codebook_index=codebook_index,
                best_losses=best_losses,
                best_indices=best_indices,
```

```
                beam_size=beam_size,
            )

            if verbose:
                progressbar.update()
                if (input_group_index * num_codebooks + codebook_index) % verbose != 0:
                    continue  # if update is an integer, compute metrics every (this many) beam search steps
                best_loss = beam_losses.min(0).values.sum().item() / out_features
                info = f"in_group {input_group_index} / {num_in_groups} "
                info += f"| codebook {codebook_index} / {num_codebooks} "
                if code_penalties is None:
                    info += f"| loss {best_loss:.10f}"
                else:  # un-regularize to restore MSE loss, report sparsity rate
                    codebook_ids = torch.arange(num_codebooks, device=beam_losses.device).view(1, 1, 1, -1)
                    best_cand_regularizer = code_penalties[codebook_ids, beam_codes[0]].sum() / num_out_groups
                    del codebook_ids
                    best_loss = best_loss - best_cand_regularizer   # report loss without the regularizer part
                    info += f"| loss {best_loss:.5f} | reg {best_cand_regularizer:.5f} |"
                del best_loss
                progressbar.desc = info
    return beam_codes[0]


@maybe_script
def _dequantize_weight(
    codes: torch.Tensor, codebooks: torch.Tensor, scales: Optional[torch.Tensor] = None
) -> torch.Tensor:
    """
    Decode float weights from quantization codes. Differentiable.
    :param codes: tensor of integer quantization codes, shape [*dims, num_out_groups, num_in_groups, num_codebooks]
    :param codebooks: tensor of vectors for each quantization code, [num_codebooks, codebook_size, out_group_size, in_group_size]
    :param scales: weight will be multiplied by this factor, must be broadcastble with [*dims, out_groups, num_in_groups, out_group_size, in_group_size]
    :return: reconstructed weight tensor of shape [*dims, num_in_groups*group_size]
    """
    num_out_groups, num_in_groups, num_codebooks = codes.shape[-3:]
    num_codebooks, codebook_size, out_group_size, in_group_size = codebooks.shape
    out_features = num_out_groups * out_group_size
    in_features = num_in_groups * in_group_size
    codebook_offsets = torch.arange(
        0, num_codebooks * codebook_size, codebook_size, device=codes.device
    )  # shape: [num_codebooks]
    reconstructed_weight_flat = F.embedding_bag(
        codes.flatten(0, -2) + codebook_offsets, codebooks.flatten(0, 1).flatten(-2, -1), mode="sum"
    )  # [prod(dims) * num_out_groups * num_in_groups, out_group_size * in_group_size]

    reconstructed_weight_groupwise = reconstructed_weight_flat.view(
        list(codes.shape[:-3]) + [num_out_groups, num_in_groups, out_group_size, in_group_size]
    )
    if scales is not None:
        reconstructed_weight_groupwise = reconstructed_weight_groupwise.mul(scales)
    return reconstructed_weight_groupwise.swapaxes(-3, -2).reshape(list(codes.shape[:-3]) + [out_features, in_features])


@maybe_script
def _beam_search_squared_errors(
    XTX: torch.Tensor,
    reference_weight: torch.Tensor,
    codebooks: torch.Tensor,
    scales: Optional[torch.Tensor],
    beam_losses: torch.Tensor,
    beam_codes: torch.Tensor,
    beam_weights: torch.Tensor,
    input_group_index: int,
    codebook_index: int,
    k_best: int,
```

```
    code_penalties: Optional[torch.Tensor] = None,
) -> tuple[torch.Tensor, torch.Tensor]:
    """
    Compute MSE or sum-of-squared-error losses for all possible ways to replace quantization codes for one input group
     and one codebook. Works in parallel for all output-dimension groups.

    :param XTX: pairwise products of input features matmul(X.transpose(), X), shape: [in_features, in_features]
    :note: if both XTX *and* beam_loses are divided by dataset size, this function will return mean squared error
    :param reference_weight: original weight matrix that is being quantized, shape: [out_features, in_features]
    :param codebooks: look-up tables of codes, shape: [num_codebooks, codebook_size, out_group_size, in_group_size]
    :param scales: weight will be multiplied by this factor, [num_out_groups, num_in_groups, 1, 1]

    :param beam_losses: sum-of-squared-error for each hypothesis in beam and for each output channel;
        shape: [beam_size, num_out_groups]
    :param beam_codes: a tensor with best weight codes, shape: [beam_size, num_out_groups, num_in_groups, num_codebooks]
    :param beam_weights: a tensor with de-quantized beam_codes, shape: [beam_size, out_features, in_features]
    :param input_group_index: an index of one group of in_features that is being re-encoded
    :param codebook_index: an index of one codebook for that group of features that is being re-encoded
    :return: tuple(Tensor, Tensor) of 3d tensor of shape = [beam_size, k_best, num_out_groups].
        First one is float tensor of losses of k_best lowest square errors for each beam and out_group
        Second one is int64 tensor of indices of k_best lowest square errors for each beam and out_group

    :note: The code computes MSE using the square-of-difference expansion
     ||X@W.T - sum_i X@(Bi@Ci).T||^2 = ||X@W.T||^2 - 2 <X@W.T, sum_i X@(Bi@Ci).T> + ||sum_i X@Bi@Ci||^2
    where X[nsamples,in_features] is calibration data, W[out_features, in_features] is the reference weight,
       C[num_codebooks, codebook_size, in_features] are learned codebooks (Ci has shape [codebook_size, out_features])
       B[num_codebooks, out_features, codebook_size] are one-hot encoded indices (quantization codes)
    The formula above uses a single group per output "neuron" and a single group.
    The algorithm below generalizes the formula for multiple groups and codebooks.

    Furthermore, the algorithm does not compute the entire formula. Instead, it begins from some baseline loss
    and computes the change in loss from changing a single code to every possible altearnative code.
    When computing the changed loss, the algorithm only computes the few affected parts of the loss formula above.
    """
    num_codebooks, codebook_size, out_group_size, in_group_size = codebooks.shape
    beam_size, num_out_groups, num_in_groups, num_codebooks = beam_codes.shape
    out_features = num_out_groups * out_group_size

    input_group_slice = slice(input_group_index * in_group_size, (input_group_index + 1) * in_group_size)

    prev_codes_part = beam_codes[:, :, input_group_index, codebook_index]  # [beam_size, num_out_groups]

    if scales is not None:
        scales_part = scales[:, input_group_index % scales.shape[1], :, :]  # [num_out_groups, 1, 1]
    else:
        scales_part = torch.empty(0, device=XTX.device)
    prev_part_dequantized = F.embedding(prev_codes_part, codebooks[codebook_index].flatten(-2, -1)).view(
        beam_size, out_features, in_group_size
    )  # previous codes de-quantized

    prev_weight_part = prev_part_dequantized
```

```
    if scales is not None:
        prev_weight_part = (
            prev_weight_part.view(beam_size, num_out_groups, out_group_size, in_group_size)
            .mul(scales_part)
            .view(beam_size, out_features, in_group_size)
        )

    cand_weights = codebooks[codebook_index]  # [codebook_size, out_group_size, in_group_size], all replacement codes

    delta_weight_without_part = reference_weight - beam_weights
    delta_weight_without_part[:, :, input_group_slice] += prev_weight_part

    # dWTXTX is equivalent to < X @ (W - \sum BiCi except current codebook), X @ SOMETHING >
    dWTXTXg = delta_weight_without_part @ XTX[..., input_group_slice]  # [beam_size, out_features, in_group_size]
    # below: use torch.matmul to compute broadcasted batch matrix multiplication; see matmul docs

    XnewBkC_norms_sq = torch.bmm(
        (cand_weights.flatten(0, 1) @ XTX[input_group_slice, input_group_slice]).view(
            codebook_size, 1, out_group_size * in_group_size
        ),
        cand_weights.view(codebook_size, out_group_size * in_group_size, 1),
    ).reshape(
        codebook_size, 1
    )  # [codebook_size, num_out_groups]
    if scales is not None:
        XnewBkC_norms_sq = XnewBkC_norms_sq.mul(scales_part.square().reshape(1, num_out_groups))

    best_losses = torch.empty(
        (beam_size, k_best, num_out_groups), dtype=XTX.dtype, device=XTX.device
    )  # shape: [beam_size, k_best, num_out_groups]
    best_indices = torch.empty(
        (beam_size, k_best, num_out_groups),
        dtype=torch.int64,
        device=XTX.device,
    )
    for beam_id in range(beam_size):
        dot_products = (
            torch.einsum(
                "mg,og->mo",
                cand_weights.reshape(codebook_size, out_group_size * in_group_size),
                dWTXTXg[beam_id].view(num_out_groups, out_group_size * in_group_size),
            )
            .sub_(
                torch.einsum(
                    "og,og->o",
                    prev_part_dequantized[beam_id].reshape(num_out_groups, out_group_size * in_group_size),
                    dWTXTXg[beam_id].view(num_out_groups, out_group_size * in_group_size),
                ).view(1, num_out_groups)
            )
            .view(codebook_size, num_out_groups)
        )
        if scales is not None:
            dot_products = dot_products.mul_(scales_part.reshape(1, num_out_groups))

        XoldBkC_norms_sq = torch.bmm(
            (prev_weight_part[beam_id] @ XTX[input_group_slice, input_group_slice]).view(
                num_out_groups, 1, out_group_size * in_group_size
            ),
            prev_weight_part[beam_id].view(num_out_groups, out_group_size * in_group_size, 1),
        ).reshape(1, num_out_groups)

        # finally, combine them to get MSE
        candidate_squared_errors = (
            beam_losses[beam_id, None, :] - 2 * dot_products + XnewBkC_norms_sq - XoldBkC_norms_sq
        )  # shape: [codebook_size, num_out_groups]

        if code_penalties is not None:
            prev_code_penalties = code_penalties[codebook_index][prev_codes_part[beam_id]]  # [codebook_size]
```

```
            candidate_squared_errors[:, :] -= prev_code_penalties[None, :]  # refund penalty for the replaced code
            candidate_squared_errors[:, :] += code_penalties[codebook_index, :, None]  # add penalty for new code

        best_beam_squared_errors, best_beam_indices = torch.topk(
            candidate_squared_errors, k_best, dim=0, largest=False, sorted=False
        )
        best_losses[beam_id] = best_beam_squared_errors
        best_indices[beam_id] = best_beam_indices

    return best_losses, best_indices


@maybe_script
def _beam_search_select_best(
    beam_codes: torch.Tensor,
    beam_weights: torch.Tensor,
    codebooks: torch.Tensor,
    scales: Optional[torch.Tensor],
    input_group_index: int,
    codebook_index: int,
    best_losses: torch.Tensor,
    best_indices: torch.Tensor,
    beam_size: int,
) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    """
    Select top-:beam_size: and reorder beam accordingly, return new beam
    :param beam_codes: a tensor with best weight codes, shape: [beam_size, num_out_groups, num_in_groups, num_codebooks]
    :param beam_weights: a tensor with de-quantized beam_codes, shape: [beam_size, out_features, in_features]
    :param codebooks: a tensor with look-up tables of codes, shape: [num_codebooks, codebook_size, out_group_size, in_group_size]
    :param scales: weight will be multiplied by this factor, [num_out_groups, num_in_groups, 1, 1]

    :param input_group_index: an index of one group of in_features that is being re-encoded
    :param codebook_index: an index of one codebook for that group of features that is being re-encoded
    :param best_losses: a 3d tensor of losses of k_best lowest square errors for each beam and out group,
        shape = [beam_size, k_best, num_out_groups]
    :param best_indices: a 3d tensor of indices of k_best lowest square errors for each beam and out group,
        shape = [beam_size, k_best, num_out_groups]
    :param beam_size: how many top hypotheses should be selected

    :returns: new (beam_codes, beam_weights, beam_losses)
    """
    dtype = best_losses.dtype
    device = best_losses.device
    _prev_beam_size, k_best, num_out_groups = best_losses.shape
    _prev_beam_size, out_features, in_features = beam_weights.shape
    _prev_beam_size, num_out_groups, num_in_groups, num_codebooks = beam_codes.shape
    flat_best = best_losses.flatten(0, 1).topk(dim=0, k=beam_size, largest=False)
    best_hypo_source_ids = flat_best.indices // k_best
    arange_out_groups = torch.arange(num_out_groups, device=device)
    best_hypo_codes = best_indices.flatten(0, 1)[flat_best.indices, arange_out_groups].reshape(
        beam_size, num_out_groups
    )
    # ^-- shape: [beam_size, num_out_groups]

    # reorder beam codes and weights
    new_beam_codes = torch.full(
        size=(len(best_hypo_codes), num_out_groups, num_in_groups, num_codebooks),
        fill_value=-1,
        dtype=beam_codes.dtype,
        device=device,
    )  # [beam_size, num_out_groups, num_in_groups, num_codebooks]
    new_beam_weights = torch.empty(len(best_hypo_codes), out_features, in_features, dtype=dtype, device=device)

    for beam_index in range(len(best_hypo_codes)):
```

```
        new_beam_codes[beam_index, :, ...] = beam_codes[best_hypo_source_ids[beam_index, :], arange_out_groups, ...]
        new_beam_codes[beam_index, :, input_group_index, codebook_index] = best_hypo_codes[beam_index, :]
        new_beam_weights[beam_index, :, :] = _dequantize_weight(new_beam_codes[beam_index, ...], codebooks, scales)

    # Note: the code above can be further accelerated by 1) vectorzing loop and ...
    # ... 2) updating new_beam_weights only for the chosen input group
    return new_beam_codes, new_beam_weights, flat_best.values


@maybe_script
def _channelwise_squared_error(XTX: torch.Tensor, weight: torch.Tensor, reference_weight: torch.Tensor):
    """
    Compute per-channel squared error between X @ weight_or_weights and X @ reference_weight
    :param XTX: pairwise products of input features matmul(X.transpose(), X), shape: [in_features, in_features]
    :note: if XTX is divided by dataset size, this function will return *mean* squared error
    :param weight: predicted/reconstructed weights of shape [*dims, out_features, in_features]
    :param reference_weight: reference weight of shape [out_features, in_features]
    :return: per-channel squared errors of shape [*dims, out_features]
    """
    XW_norm_square = torch.matmul(weight[..., :, None, :], (weight @ XTX)[..., :, :, None]).flatten(-3)
    XWreference_norm_square = torch.bmm(reference_weight[:, None, :], (reference_weight @ XTX)[:, :, None]).flatten(-3)
    dot_product = torch.matmul((reference_weight @ XTX)[:, None, :], weight[..., :, :, None]).flatten(-3)
    return XW_norm_square - 2 * dot_product + XWreference_norm_square


@torch.no_grad()
def init_aq_kmeans(
    reference_weight: torch.Tensor,
    *,
    num_codebooks: int,
    out_group_size: int,
    in_group_size: int,
    codebook_size: int,
    verbose: bool = False,
    use_faiss: bool = False,
    max_points_per_centroid: Optional[int] = None,
    max_iter: int = 1000,
    devices: Optional[List[torch.device]] = None,
    **kwargs,
):
    """
    Create initial codes and codebooks using residual K-means clustering of weights
    :params reference_weight, num_codebooks, out_group_size, in_group_size, nbits, verbose: same as in QuantizedWeight
    :params use_faiss  whether to use faiss implementation of kmeans or pure torch
    :params max_point_per_centorid maximum data point per cluster
    :param kwargs: any additional params are forwarded to fit_kmeans
    """
    out_features, in_features = reference_weight.shape
    num_out_groups = out_features // out_group_size
    num_in_groups = in_features // in_group_size
    weight_residue = (
        reference_weight.reshape(num_out_groups, out_group_size, num_in_groups, in_group_size)
        .clone()
        .swapaxes(-3, -2)
        .reshape(num_out_groups * num_in_groups, out_group_size * in_group_size)
    )
    codebooks = []
    codes = []

    if max_points_per_centroid is not None:
        print("Clustering:", max_points_per_centroid * codebook_size, "points from", weight_residue.shape[0])

    for _ in trange(num_codebooks, desc="initializing with kmeans") if verbose else range(num_codebooks):
        if use_faiss:
```

```
            codebook_i, codes_i, reconstructed_weight_i = fit_faiss_kmeans(
                weight_residue,
                k=codebook_size,
                max_iter=max_iter,
                gpu=(weight_residue.device.type == "cuda"),
                max_points_per_centroid=max_points_per_centroid,
            )
        else:
            chosen_ids = None
            if max_points_per_centroid is not None:
                chosen_ids = torch.randperm(weight_residue.shape[0], device=weight_residue.device)[
                    : max_points_per_centroid * codebook_size
                ]
            codebook_i, _, _ = fit_kmeans(
                weight_residue if chosen_ids is None else weight_residue[chosen_ids, :],
                k=codebook_size,
                max_iter=max_iter,
                devices=devices,
                **kwargs,
            )
            codes_i, reconstructed_weight_i = find_nearest_cluster(weight_residue, codebook_i, devices=devices)

        codes_i = codes_i.reshape(num_out_groups, num_in_groups, 1)
        codebook_i = codebook_i.reshape(1, codebook_size, out_group_size, in_group_size)
        weight_residue -= reconstructed_weight_i
        codes.append(codes_i)
        codebooks.append(codebook_i)
        del reconstructed_weight_i
    codebooks = torch.cat(codebooks, dim=0)
    codes = torch.cat(codes, dim=-1)
    return codes, codebooks
```