A Appendix

A.1 Extend Attention Kernel in Wave

```
def extend_attention_core(
    q,
    k,
    v,
    k_cache,
    v_cache,
    qo_indptr,
    kv_indptr,
    kv_indices,
    custom_mask,
    mask_offsets,
    c,
):
    c_reg = tkl.Register[H, D_KV, N_Q, tkl.f32](0.0)
    init_sum = tkl.Register[H, N_Q, tkl.f32](0.0)
    init_max = tkl.Register[H, N_Q, tkl.f32](-1e6)
    zero = tkl.Register[N_Q, N_KV, tkl.f32](0.0)
    neg_infinity = tkl.Register[N_Q, N_KV, tkl.f32](-1
     e6)
    layer_scale_reg = tkl.Register[H, N_Q, N_KV, tkl.
     f32](layer_scaling)
    if logit_cap > 0:
        logit_cap_reg = tkl.Register[H, N_Q, N_KV, tkl.
     f32](logit_cap)

    seq_extend_start_idx = tkw.read(qo_indptr,
     elements_per_thread=1)
    tkw.set_symbol(EXT_IDX, seq_extend_start_idx)
    seq_len_extend = (
        tkw.read(qo_indptr, elements_per_thread=1,
     source=(s + 1,), target=(s,))
        - seq_extend_start_idx
    )
    tkw.set_symbol(N_Q, seq_len_extend)
    seq_kv_start_idx = tkw.read(kv_indptr,
     elements_per_thread=1)
    tkw.set_symbol(KV_START_IDX, seq_kv_start_idx)
    seq_len_prefix = (
        tkw.read(kv_indptr, elements_per_thread=1,
     source=(s + 1,), target=(s,))
        - seq_kv_start_idx
    )
    tkw.set_symbol(N_KV, seq_len_prefix)
    if use_custom_mask:
        seq_len = seq_len_prefix + seq_len_extend
        tkw.set_symbol(SEQ_LEN, seq_len)
        seq_mask_start_idx = tkw.read(mask_offsets,
     elements_per_thread=1)
        tkw.set_symbol(MASK_START_IDX,
     seq_mask_start_idx)

    @tkw.iterate(N_KV, init_args=[init_max, init_sum,
     c_reg])
    def first_loop(
        partial_max: tkl.Register[H, N_Q, tkl.f32],
        partial_sum: tkl.Register[H, N_Q, tkl.f32],
        acc: tkl.Register[H, D_KV, N_Q, tkl.f32],
    ):
        q_reg = tkw.read(
            q,
            elements_per_thread=
     LOAD_ELEMS_PER_THREAD_QK,
            source=(n_q + EXT_IDX, h, d_q),
            target=(h, n_q, d_q),
        )
        block_indices_v = tkw.read(
            kv_indices,
            elements_per_thread=
     LOAD_ELEMS_PER_THREAD_PV,
            source=(n_kv + KV_START_IDX,),
            target=(n_kv,),
        )
        block_indices_k = tkw.read(
            kv_indices,
            elements_per_thread=1,
            source=(n_kv + KV_START_IDX,),
            target=(n_kv,),
        )
        k_reg = tkw.read(
            k_cache,
            elements_per_thread=
     LOAD_ELEMS_PER_THREAD_QK,
            source=(block_indices_k, h_kv // head_ratio,
     d_q),
            target=(h_kv, n_kv, d_q),
        )
        imm_reg = tkl.Register[H, N_KV, N_Q, tkl.f32
     ](0.0)
        inner_acc = tkw.mma(k_reg, q_reg, imm_reg,
     mfma_variant[0])
        x_j = tkw.permute(inner_acc, target_shape=[H,
     N_Q, N_KV])
        x_j = x_j * layer_scale_reg
        if logit_cap > 0:
            logit_cap_reg_inv = tkw.reciprocal(
     logit_cap_reg)
            x_j = logit_cap_reg * tkw.tanh_approx(x_j *
      logit_cap_reg_inv)
        n_kv_index = tkw.self_index(N_KV, tkl.i32)
        mask = tkw.apply_expr(n_kv_index, lambda x: x <
     N_KV)
        mask = tkw.broadcast(mask, target_shape=[N_Q,
     N_KV])
        mask = tkw.cast(mask, tkw.i1)
        if use_custom_mask:
            c_mask = tkw.read(
                custom_mask,
                elements_per_thread=
     STORE_ELEMS_PER_THREAD,
                source=(n_q * SEQ_LEN + MASK_START_IDX +
      n_kv,),
                target=(n_q, n_kv),
            )
            c_mask = tkw.cast(c_mask, tkw.i1)
            mask &= c_mask
        bias = tkw.select(mask, zero, neg_infinity)
        x_j = x_j + bias
        m_j = tkw.max(x_j, partial_max, dim=N_KV)
        e_delta_max = tkw.exp2(partial_max - m_j)
        e_delta = tkw.exp2(x_j - m_j)
        e_init = partial_sum * e_delta_max
        d_j = tkw.sum(e_delta, e_init, dim=N_KV)
        imm_f16 = tkw.cast(e_delta, wave_input_dtype)
        v_reg = tkw.read(
            v_cache,
            elements_per_thread=
     LOAD_ELEMS_PER_THREAD_PV,
            source=(block_indices_v, h_kv // head_ratio,
     d_kv),
            target=(h_kv, d_kv, n_kv),
        )
        new_acc = acc * e_delta_max
        acc = tkw.mma(v_reg, imm_f16, new_acc)
        return m_j, d_j, acc

    res_max, res_sum, res_mm = first_loop

    if is_causal:
        seq_len_extend = tkw.apply_expr(
            seq_len_extend, lambda x: sympy.Min(x, (
     WORKGROUP_0 + 1) * BLOCK_N_Q)
        )
    tkw.set_symbol(N_KV, seq_len_extend)
    if use_custom_mask:
        tkw.set_symbol(PREFIX_LEN, seq_len_prefix)

    @tkw.iterate(N_KV, init_args=[res_max, res_sum,
     res_mm])
    def second_loop(
        partial_max: tkl.Register[H, N_Q, tkl.f32],
        partial_sum: tkl.Register[H, N_Q, tkl.f32],
        acc: tkl.Register[H, D_KV, N_Q, tkl.f32],
    ):
        imm_reg = tkl.Register[H, N_KV, N_Q, tkl.f32
     ](0.0)
        q_reg = tkw.read(
            q,
            elements_per_thread=
     LOAD_ELEMS_PER_THREAD_QK,
            source=(n_q + EXT_IDX, h, d_q),
            target=(h, n_q, d_q),
        )
        k_reg = tkw.read(
            k,
            elements_per_thread=
     LOAD_ELEMS_PER_THREAD_QK,
            source=(n_kv + EXT_IDX, h_kv // head_ratio,
     d_q),
            target=(h_kv, n_kv, d_q),
        )
        inner_acc = tkw.mma(k_reg, q_reg, imm_reg,
     mfma_variant[0])
        x_j = tkw.permute(inner_acc, target_shape=[H,
     N_Q, N_KV])
        x_j = x_j * layer_scale_reg
        if logit_cap > 0:
            logit_cap_reg_inv = tkw.reciprocal(
     logit_cap_reg)
            x_j = logit_cap_reg * tkw.tanh_approx(x_j *
      logit_cap_reg_inv)
        n_kv_index = tkw.self_index(N_KV, tkl.i32)
```

```
        mask = tkw.apply_expr(n_kv_index, lambda x: x <
     N_KV)
        mask = tkw.broadcast(mask, target_shape=[N_Q,
    N_KV])
        if is_causal:
            n_q_index = tkw.self_index(N_Q, tkl.i32)
            n_q_index = tkw.broadcast(n_q_index,
    target_shape=[N_Q, N_KV])
            mask = (n_q_index >= n_kv_index) & mask
        mask = tkw.cast(mask, tkw.i1)
        if use_custom_mask:
            c_mask = tkw.read(
                custom_mask,
                elements_per_thread=
    STORE_ELEMS_PER_THREAD,
                source=(n_q * SEQ_LEN + MASK_START_IDX +
     n_kv + PREFIX_LEN,),
                target=(n_q, n_kv),
            )
            c_mask = tkw.cast(c_mask, tkw.i1)
            mask &= c_mask
        bias = tkw.select(mask, zero, neg_infinity)
        x_j = x_j + bias
        m_j = tkw.max(x_j, partial_max, dim=N_KV)
        e_delta_max = tkw.exp2(partial_max - m_j)
        e_delta = tkw.exp2(x_j - m_j)
        e_init = partial_sum * e_delta_max
        d_j = tkw.sum(e_delta, e_init, dim=N_KV)
        imm_f16 = tkw.cast(e_delta, wave_input_dtype)
        v_reg = tkw.read(
            v,
            elements_per_thread=
    LOAD_ELEMS_PER_THREAD_PV,
            source=(n_kv + EXT_IDX, h_kv // head_ratio,
     d_kv),
            target=(h_kv, d_kv, n_kv),
        )
        new_acc = acc * e_delta_max
        acc = tkw.mma(v_reg, imm_f16, new_acc)
        return m_j, d_j, acc

    # repeat represents the results of the loop
    res_max, res_sum, res_mm = second_loop
    reciprocal_sum = tkw.reciprocal(res_sum)
    res = res_mm * reciprocal_sum
    if wave_output_dtype != tkl.f32:
        res = tkw.cast(res, wave_output_dtype)
    tkw.write(
        res,
        c,
        source=(h, d_kv, n_q),
        target=(n_q + EXT_IDX, h, d_kv),
        elements_per_thread=STORE_ELEMS_PER_THREAD,
    )
```

A.2 Persistent GEMM Kernel in Wave

```
def get_persistent_gemm_kernel(
    shape: tuple[int, int, int],
    mfma_variant: MMAType,
    threads_per_wave: int = 64,
    block_shape: Optional[tuple[int, int, int]] = None,
    waves_per_block: Optional[tuple[int, int, int]] =
     None,
    num_ctas: Optional[int] = None,
):
    """
    Creates a persistent GEMM kernel that uses a single
      workgroup dimension
    with linearized CTA dims. Each CTA iterates over
     multiple output tiles.

    """
    if not block_shape:
        block_shape = (128, 256, 64)

    if not waves_per_block:
        waves_per_block = (4, 1)

    m, n, k = shape
    block_m, block_n, block_k = block_shape

    m_tiles = (m + block_m - 1) // block_m
    n_tiles = (n + block_n - 1) // block_n
    total_tiles = m_tiles * n_tiles

    if num_ctas is None:
        num_ctas = 304  # Or number of compute units
     depending on the device

    # Symbols
    M = tkl.sym.M
    N = tkl.sym.N
    K = tkl.sym.K
    BLOCK_M = tkl.sym.BLOCK_M
    BLOCK_N = tkl.sym.BLOCK_N
    BLOCK_K = tkl.sym.BLOCK_K
    ADDRESS_SPACE = tkl.sym.ADDRESS_SPACE
    TOTAL_TILES = tkl.sym.TOTAL_TILES
    NUM_CTAS = tkl.sym.NUM_CTAS
    TILE_IDX = tkl.sym.TILE_IDX
    CTA_M_OFFSET = tkl.sym.CTA_M_OFFSET
    CTA_N_OFFSET = tkl.sym.CTA_N_OFFSET
    N_TILES = tkl.sym.N_TILES

    # Index mappings for manual offset control
    i = tkw.IndexMapping.iterator(0)
    j = tkw.IndexMapping.iterator(1)

    a_read_mapping = tkw.IndexMapping(
        num_iterators=2,
        inputs={M: i + CTA_M_OFFSET, K: j},
        outputs={M: i, K: j},
    )

    b_read_mapping = tkw.IndexMapping(
        num_iterators=2,
        inputs={N: i + CTA_N_OFFSET, K: j},
        outputs={N: i, K: j},
    )

    c_write_mapping = tkw.IndexMapping(
        num_iterators=2,
        inputs={M: i, N: j},
        outputs={M: i + CTA_M_OFFSET, N: j +
     CTA_N_OFFSET},
    )

    constraints = [
        tkw.GridConstraint(NUM_CTAS),
        tkw.WorkgroupConstraint(M, BLOCK_M, 0),
        tkw.WorkgroupConstraint(N, BLOCK_N, 1),
        tkw.TilingConstraint(K, BLOCK_K),
        tkw.TilingConstraint(TILE_IDX),
        tkw.WaveConstraint(M, BLOCK_M / waves_per_block
     [0]),
        tkw.WaveConstraint(N, BLOCK_N / waves_per_block
     [1]),
        tkw.HardwareConstraint(
            threads_per_wave=threads_per_wave,
            mma_type=mfma_variant,
            vector_shapes={TILE_IDX: 0},
        ),
    ]

    @tkw.wave(constraints)
    def persistent_gemm(
        a: tkl.Memory[M, K, ADDRESS_SPACE, tkl.f16],
        b: tkl.Memory[N, K, ADDRESS_SPACE, tkl.f16],
        c: tkl.Memory[M, N, GLOBAL_ADDRESS_SPACE, tkl.
     f32],
    ):
        condition = TILE_IDX < TOTAL_TILES
        init_tile_id = tkw.scalar(WORKGROUP_0, tkl.i32)

        @tkw.iterate(TILE_IDX, start=init_tile_id,
     condition=condition, init_args=[])
        def persistent_loop():
            tile_id = tkw.scalar(TILE_IDX, tkl.i32)
            m_offset = (tile_id // tkw.scalar(N_TILES,
     tkl.i32)) * tkw.scalar(
                BLOCK_M, tkl.i32
            )
            n_offset = (tile_id % tkw.scalar(N_TILES,
     tkl.i32)) * tkw.scalar(
                BLOCK_N, tkl.i32
            )
            tkw.set_symbol(CTA_M_OFFSET, m_offset)
            tkw.set_symbol(CTA_N_OFFSET, n_offset)

            c_reg = tkl.Register[M, N, tkl.f32](0.0)

            @tkw.iterate(axis=K, init_args=[c_reg])
            def k_loop(acc: tkl.Register[M, N, tkl.f32])
      -> tkl.Register[M, N, tkl.f32]:
                a_reg = tkw.read(a, mapping=
    a_read_mapping)
                b_reg = tkw.read(b, mapping=
    b_read_mapping)
                acc = tkw.mma(a_reg, b_reg, acc)
                return acc

            tkw.write(k_loop, c, mapping=
    c_write_mapping)
```

```
        num_ctas_scalar = tkw.scalar(NUM_CTAS, tkl.
 i32)
        next_idx = tile_id + num_ctas_scalar
        tkw.set_symbol(TILE_IDX, next_idx)

    hyperparams = {
        ADDRESS_SPACE: SHARED_ADDRESS_SPACE,
        BLOCK_M: block_m,
        BLOCK_N: block_n,
        BLOCK_K: block_k,
        M: m,
        N: n,
        K: k,
        TOTAL_TILES: total_tiles,
        N_TILES: n_tiles,
        NUM_CTAS: num_ctas,
    }

    return persistent_gemm, hyperparams
```

A.3 Mixture-of-Experts (MoE) Kernel in Wave

```
def get_fused_moe_gemm(
    m: int,
    n: int,
    k: int,
    e: int,
    block_shape: int,
    total_elems: int,
    num_experts: int,
    mfma_variant: MMAType,
    datatype: DataType,
):
    M = tkl.sym.M
    N = tkl.sym.N
    K = tkl.sym.K
    E = sym.E
    TOTAL_ELEMS = sym.TOTAL_ELEMS
    NUM_BLOCKS = sym.NUM_BLOCKS
    BLOCK_SHAPE = sym.BLOCK_SHAPE
    PAD_VALUE = sym.PAD_VALUE

    # Define workgroup tile sizes
    BLOCK_M = sym.BLOCK_M
    BLOCK_N = sym.BLOCK_N
    BLOCK_K = sym.BLOCK_K

    # Define the address space for our memory buffers
    ADDRESS_SPACE_A = sym.ADDRESS_SPACE_A
    ADDRESS_SPACE_B = sym.ADDRESS_SPACE_B
    ADDRESS_SPACE_C = sym.ADDRESS_SPACE_C

    IDX = sym.IDX
    SCATTER_IDX = sym.SCATTER_IDX
    # Define constraints for the kernel
    constraints = [
        tkw.WorkgroupConstraint(M, BLOCK_M, 0),
        tkw.WorkgroupConstraint(N, BLOCK_N, 1),
        tkw.WorkgroupConstraint(TOTAL_ELEMS,
 BLOCK_SHAPE, 2),
        tkw.TilingConstraint(K, BLOCK_K),
        tkw.WaveConstraint(M, BLOCK_M / 2),
        tkw.WaveConstraint(N, BLOCK_N / 2),
        tkw.WaveConstraint(TOTAL_ELEMS, BLOCK_SHAPE),
        tkw.HardwareConstraint(
            threads_per_wave=64,
            mma_type=tkw.MMAType.F32_16x16x16_F16,
            vector_shapes={
                E: E,
                TOTAL_ELEMS: TOTAL_ELEMS,
                BLOCK_SHAPE: BLOCK_SHAPE,
                M: 16,
                N: 16,
                K: 16,
                NUM_BLOCKS: NUM_BLOCKS,
            },
        ),
    ]

    i = tkw.IndexMapping.iterator(0)
    j = tkw.IndexMapping.iterator(1)
    e = tkw.IndexMapping.iterator(2)
    d0 = tkw.IndexMapping.dynamic_val(0)

    b_read_map = tkw.IndexMapping(
        num_iterators=2,
        inputs={E: IDX, N: i, K: j},
        outputs={N: i, K: j},
    )

    a_read_map = tkw.IndexMapping(
        num_iterators=2,
        inputs={M: d0, K: j},
        outputs={M: i, K: j},
        dynamic_val_mappings={M: i},
    )

    a_back_write_map = tkw.IndexMapping(
        num_iterators=2,
        inputs={M: i, K: j},
        outputs={NUM_BLOCKS: WORKGROUP_2, M: d0, K: j},
        dynamic_val_mappings={M: i},
    )

    a_back_read_map = tkw.IndexMapping(
        num_iterators=2,
        inputs={NUM_BLOCKS: WORKGROUP_2, M: i, K: j},
        outputs={M: i, K: j},
    )

    c_back_write_map = tkw.IndexMapping(
        num_iterators=2,
        inputs={M: i, N: j},
        outputs={NUM_BLOCKS: WORKGROUP_2, M: i, N: j},
    )

    c_back_read_map = tkw.IndexMapping(
        num_iterators=2,
        inputs={NUM_BLOCKS: WORKGROUP_2, M: d0, N: j},
        outputs={M: i, N: j},
        dynamic_val_mappings={M: i},
    )

    c_write_map = tkw.IndexMapping(
        num_iterators=2,
        inputs={M: i, N: j},
        outputs={M: d0, N: j},
        dynamic_val_mappings={M: i},
    )

    dyn_reorder_a_read_map = tkw.IndexMapping(
        num_iterators=1,
        inputs={TOTAL_ELEMS: d0},
        outputs={TOTAL_ELEMS: i},
        dynamic_val_mappings={TOTAL_ELEMS: i},
    )

    expert_id_read_map = tkw.IndexMapping(
        num_iterators=1,
        inputs={NUM_BLOCKS: d0},
        outputs={NUM_BLOCKS: i},
        dynamic_val_mappings={NUM_BLOCKS: i},
    )

    @tkw.wave(constraints)
    def fused_moe_gemm(
        a: Memory[M, K, ADDRESS_SPACE_A, f16],  # Input
  matrix A
        b: Memory[E, N, K, ADDRESS_SPACE_B, f16],  #
 Input matrix B
        reorder_a: Memory[TOTAL_ELEMS, ADDRESS_SPACE_A,
  i32],  # Input matrix A
        expert_ids: Memory[NUM_BLOCKS, ADDRESS_SPACE_A,
  i32],  # Input matrix A
        a_back: Memory[NUM_BLOCKS, M, K,
 ADDRESS_SPACE_A, f16],  # Output matrix A
        c: Memory[M, N, ADDRESS_SPACE_C, f32],  #
 Output matrix C
        c_back: Memory[NUM_BLOCKS, M, N,
 ADDRESS_SPACE_C, f32],  # Output matrix C
    ):
        # Initialize the accumulator register with
 zeros
        zero_reg = Register[M, K, f16](0.0)
        zero_reg_mn = Register[M, N, f32](0.0)
        tkw.write(zero_reg, a_back)
        tkw.write(zero_reg_mn, c_back)
        mock_reg = tkw.read(reorder_a)

        wid = tkw.scalar(WORKGROUP_2, i32)
        expert_id = tkw.read(
            expert_ids, mapping=expert_id_read_map,
 mapping_dynamic_vals=(wid,)
        )
        tkw.set_symbol(IDX, expert_id)
        condition = THREAD_0 < BLOCK_SHAPE

        @tkw.conditional(condition)
        def gather_op():
            tid = tkw.Register[TOTAL_ELEMS, i32](
 THREAD_0)
            wid = tkw.Register[TOTAL_ELEMS, i32](
 WORKGROUP_2)
            tid_offset = tkw.Register[TOTAL_ELEMS, i32
 ](BLOCK_SHAPE) * wid + tid
            reordered_idx = tkw.read(
```

```
            reorder_a,
            mapping=dyn_reorder_a_read_map,
            mapping_dynamic_vals=(tid_offset,),
        )

        tkw.set_symbol(SCATTER_IDX, reordered_idx)
        is_not_padding = SCATTER_IDX < PAD_VALUE

        @tkw.conditional(is_not_padding)
        def then():
            @tkw.iterate(K, init_args=[])
            def copy_row():
                a_row_data = tkw.read(
                    a,
                    mapping=a_read_map,
                    mapping_dynamic_vals=(
reordered_idx,),
                    elements_per_thread=16,
                )

                tkw.write(
                    a_row_data,
                    a_back,
                    mapping=a_back_write_map,
                    mapping_dynamic_vals=(tid,),
                    elements_per_thread=16,
                )

    tkw.workgroup_barrier()
    c_reg = Register[M, N, f32](0.0)

    # Iterate over the K dimension to compute the
dot product
    @tkw.iterate(K, init_args=[c_reg])
    def gemm_compute(acc: Register[M, N, f32]) ->
Register[M, N, f32]:
        # Load elements from A and B
        a_reg = tkw.read(a_back, mapping=
a_back_read_map)
        b_reg = tkw.read(b, mapping=b_read_map)

        # Compute matrix multiplication and
accumulate
        acc = tkw.mma(a_reg, b_reg, acc)
        return acc

    tkw.write(gemm_compute, c_back, mapping=
c_back_write_map)

    @tkw.conditional(condition)
    def scatter_op():
        tid = tkw.Register[TOTAL_ELEMS, i32](
THREAD_0)
        wid = tkw.Register[TOTAL_ELEMS, i32](
WORKGROUP_2)
        tid_offset = tkw.Register[TOTAL_ELEMS, i32
](BLOCK_SHAPE) * wid + tid
        reordered_idx = tkw.read(
            reorder_a,
            mapping=dyn_reorder_a_read_map,
            mapping_dynamic_vals=(tid_offset,),
        )

        tkw.set_symbol(SCATTER_IDX, reordered_idx)
        is_not_padding = SCATTER_IDX < PAD_VALUE

        @tkw.conditional(is_not_padding)
        def then():
            c_row_data = tkw.read(
                c_back,
                mapping=c_back_read_map,
                mapping_dynamic_vals=(tid,),
                elements_per_thread=16,
            )
            tkw.write(
                c_row_data,
                c,
                mapping=c_write_map,
                mapping_dynamic_vals=(reordered_idx
,),
                elements_per_thread=16,
            )

# Set hyperparameters for compilation
hyperparams = {
    ADDRESS_SPACE_A: GLOBAL_ADDRESS_SPACE,
    ADDRESS_SPACE_B: GLOBAL_ADDRESS_SPACE,
    ADDRESS_SPACE_C: GLOBAL_ADDRESS_SPACE,
    BLOCK_M: 64,
    BLOCK_N: 64,
    BLOCK_K: 32,
    M: m,
    N: n,
    K: k,
    E: num_experts,
    BLOCK_SHAPE: block_shape,
    TOTAL_ELEMS: total_elems,
    NUM_BLOCKS: (total_elems + block_shape - 1) //
 block_shape,
    PAD_VALUE: m,
}

return fused_moe_gemm, hyperparams
```