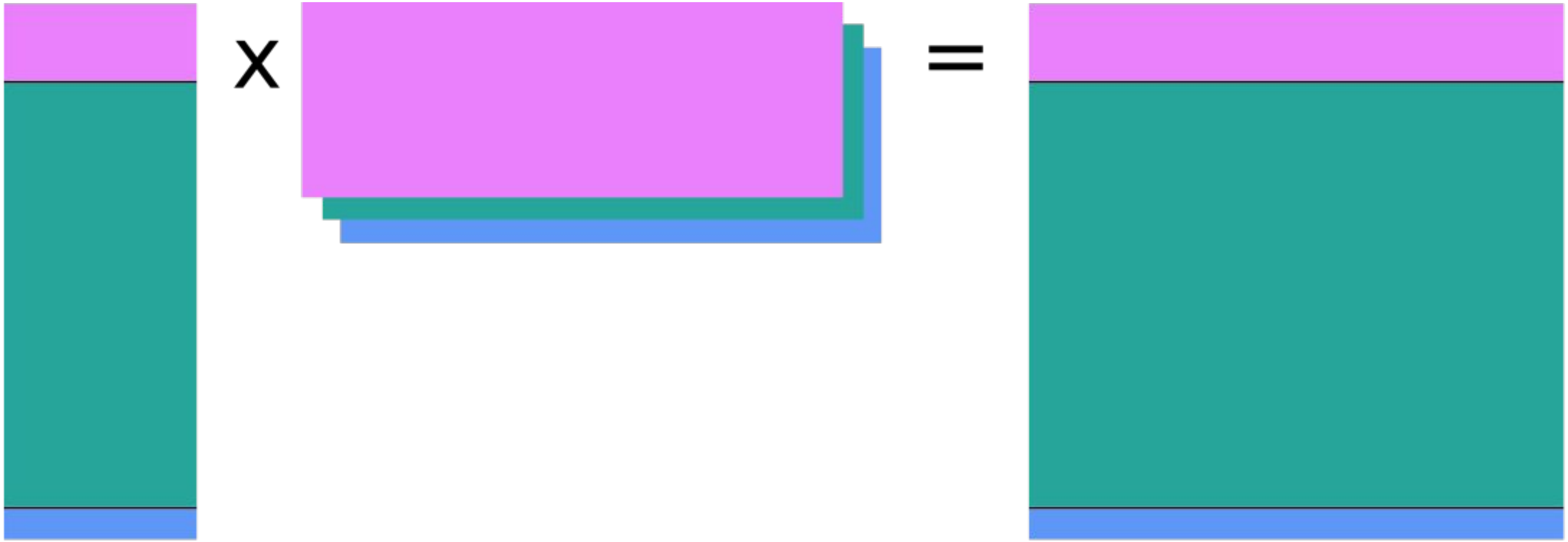
ragged dot



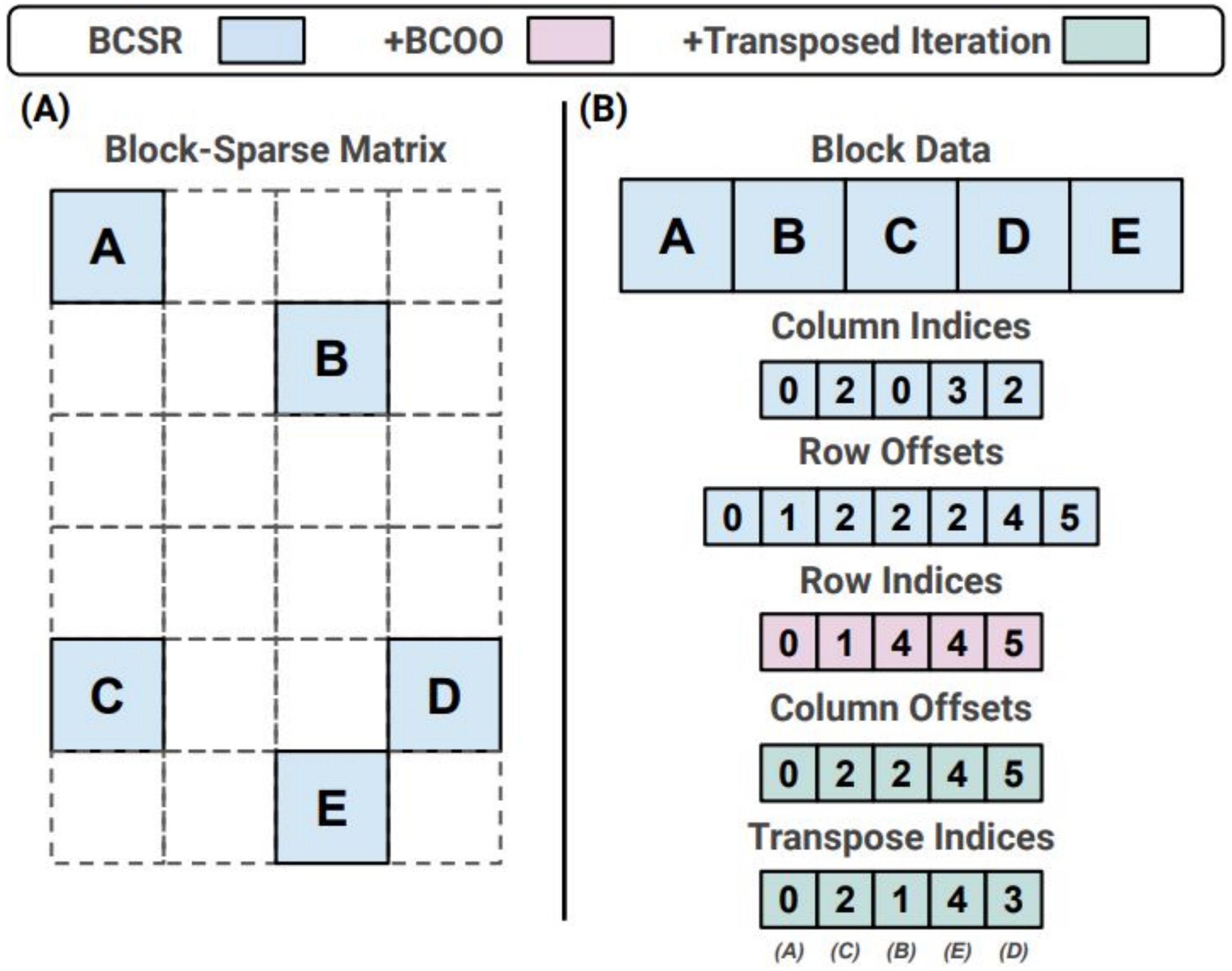
Robert Dyro
@rdyro

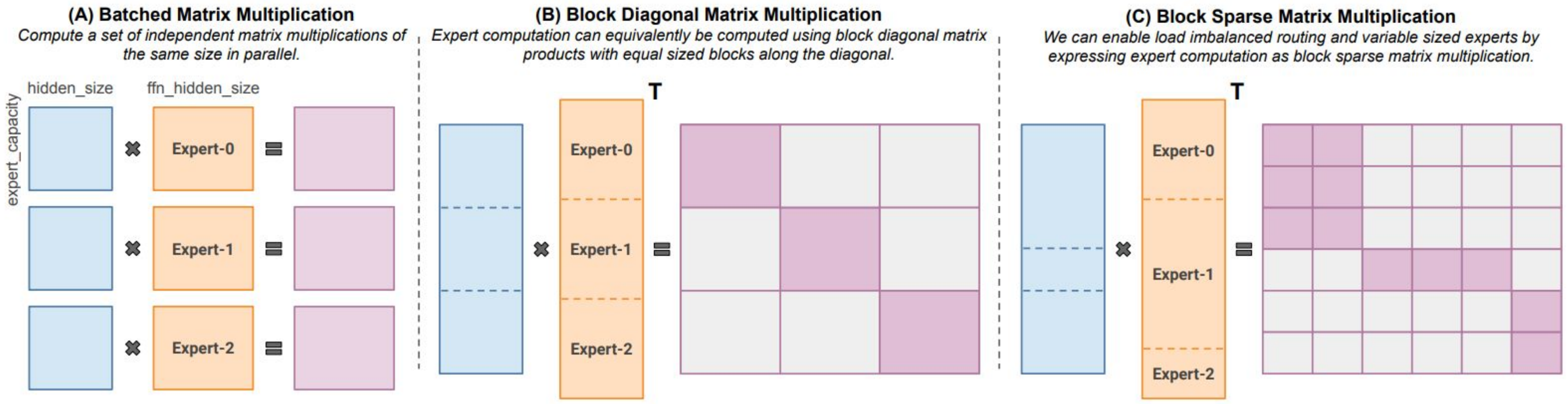
Intro

History

- *MegaBlocks: Efficient Sparse Training with Mixture-of-Experts,* Trevor Gale, Deepak Narayanan, Cliff Young, Matei Zaharia

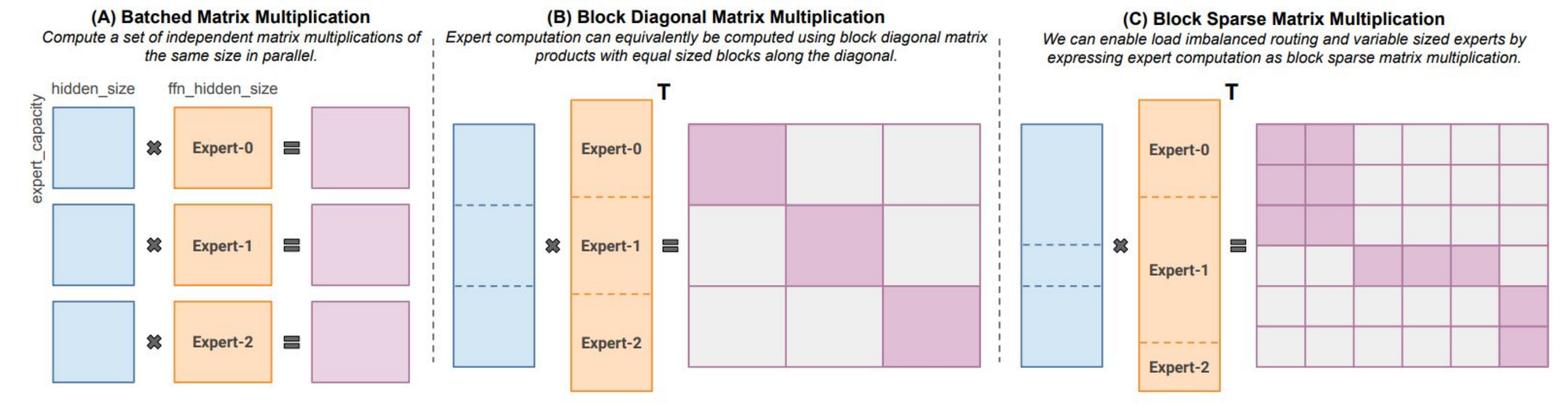
- traditional sparsity represented using CSC or CSR (on GPU)
- modern accelerators deal badly with sparsity
- **B**lock-**CSR** - **BCSR**





History

Motivation



- some possible approaches:
  - pad blocks to the maximum size (memory inefficient)
  - loop over rows individually (accelerator unfriendly)

- decent approaches on GPU
  - use NVIDIA libraries, dispatch **each expert on CPU**
  - not bad with a good compiler and/or CUDA-graphs

Base Routine

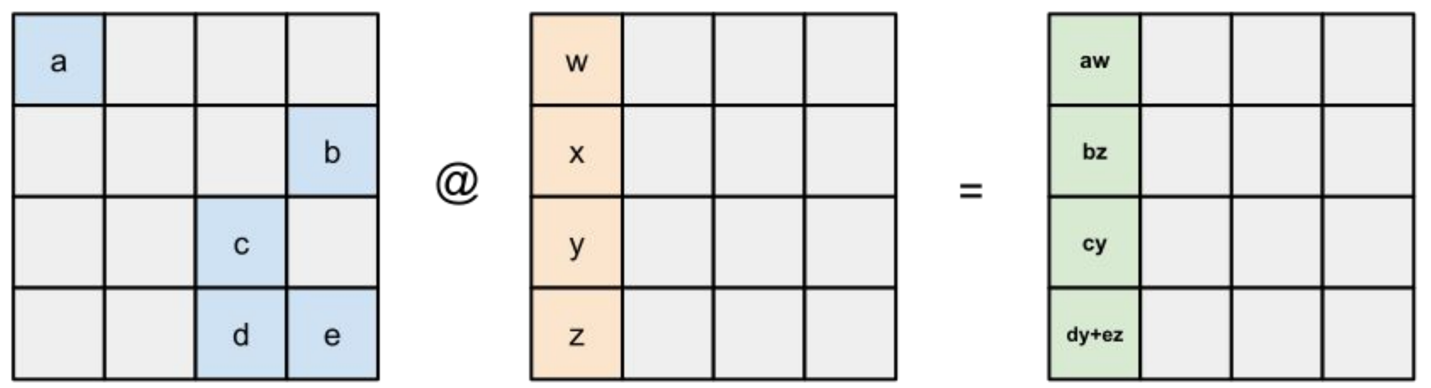
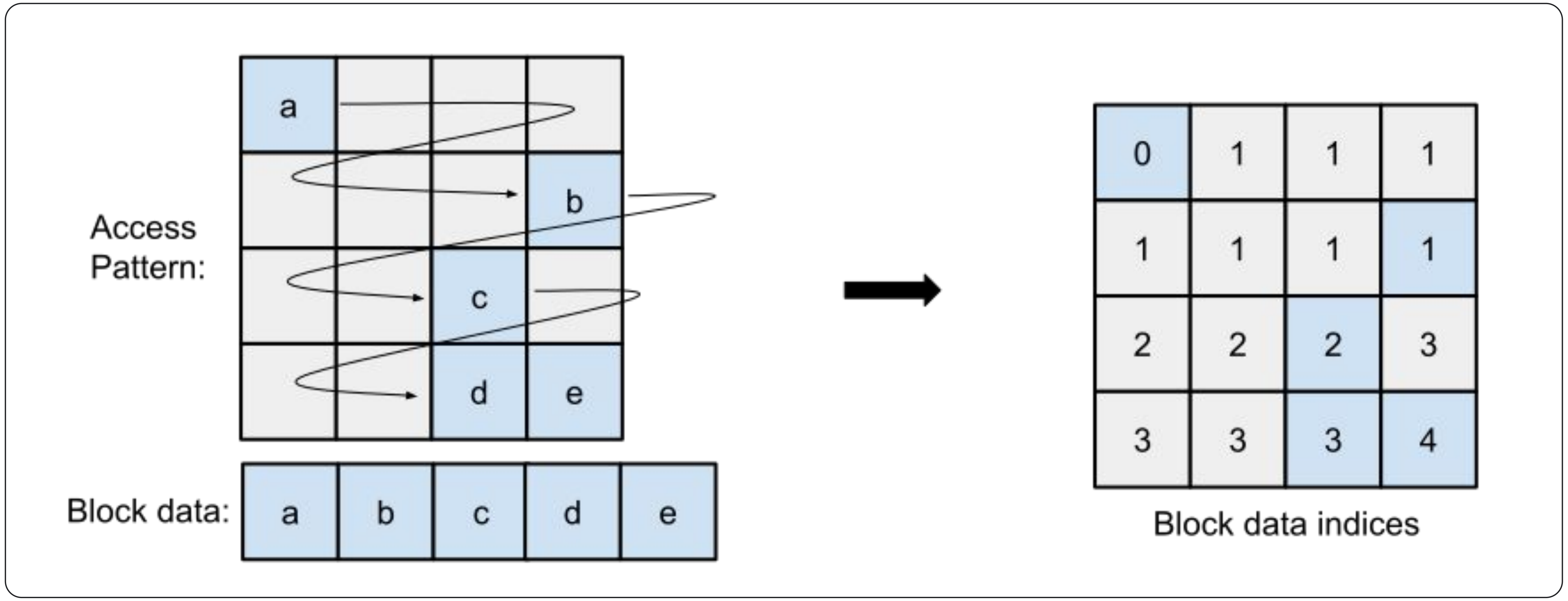


Fig: Block-sparse matrix mutliplication
docs.jax.dev/en/latest/pallas/tpu/sparse.html#example-sparse-dense-matrix-multiplication

Sparse computations in pallas
`pltpu.PrefetchScalarGridSpec`

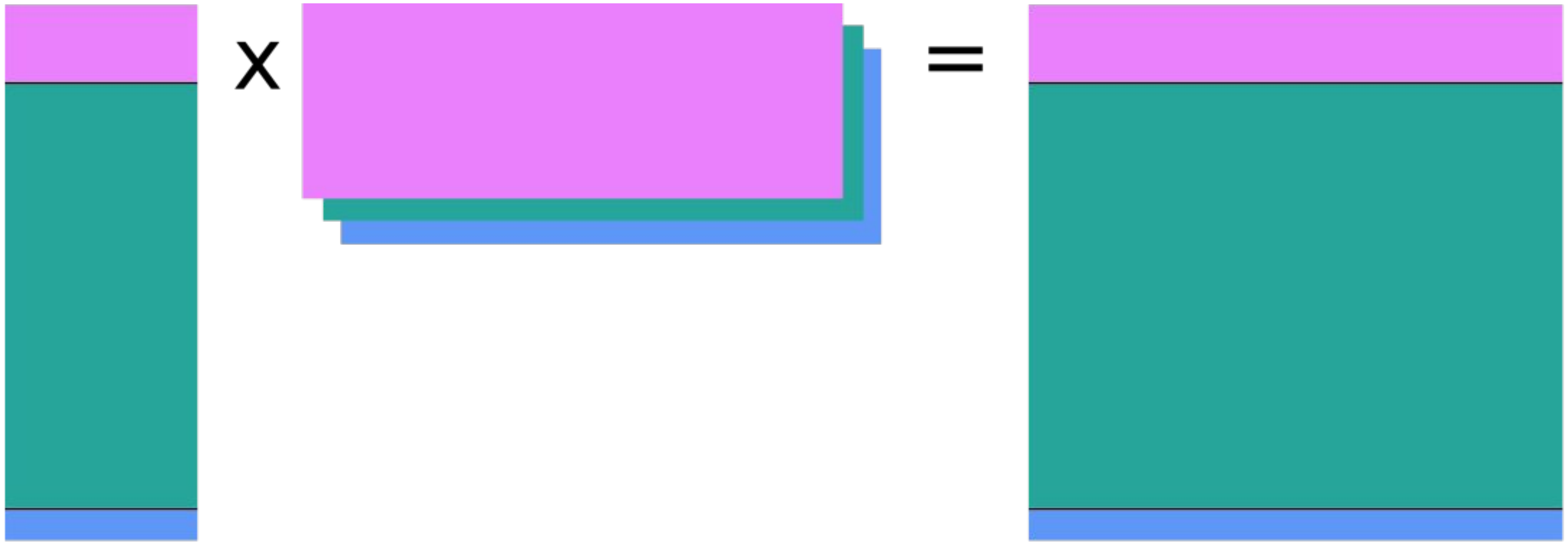


- "2, then 2" uses no BW
- read in the next block ASAP
- small penalty for visiting block at all
- can skip computation inside kernel

- pltpu.PrefetchScalarGridSpec is super useful
- generally pre-compute a metadata lookup table, then just iterate over entries
- example: `pl.BlockSpec(..., lambda i, j, scalar1_ref, scalar2_ref: (scalar1_ref[i], scalar2_ref[j]))`
- can re-implement the prefetch scalar grid using existing pallas APIs:
  https://github.com/openxla/tokamax/blob/main/tokamax/_src/mosaic_tpu.py#L126

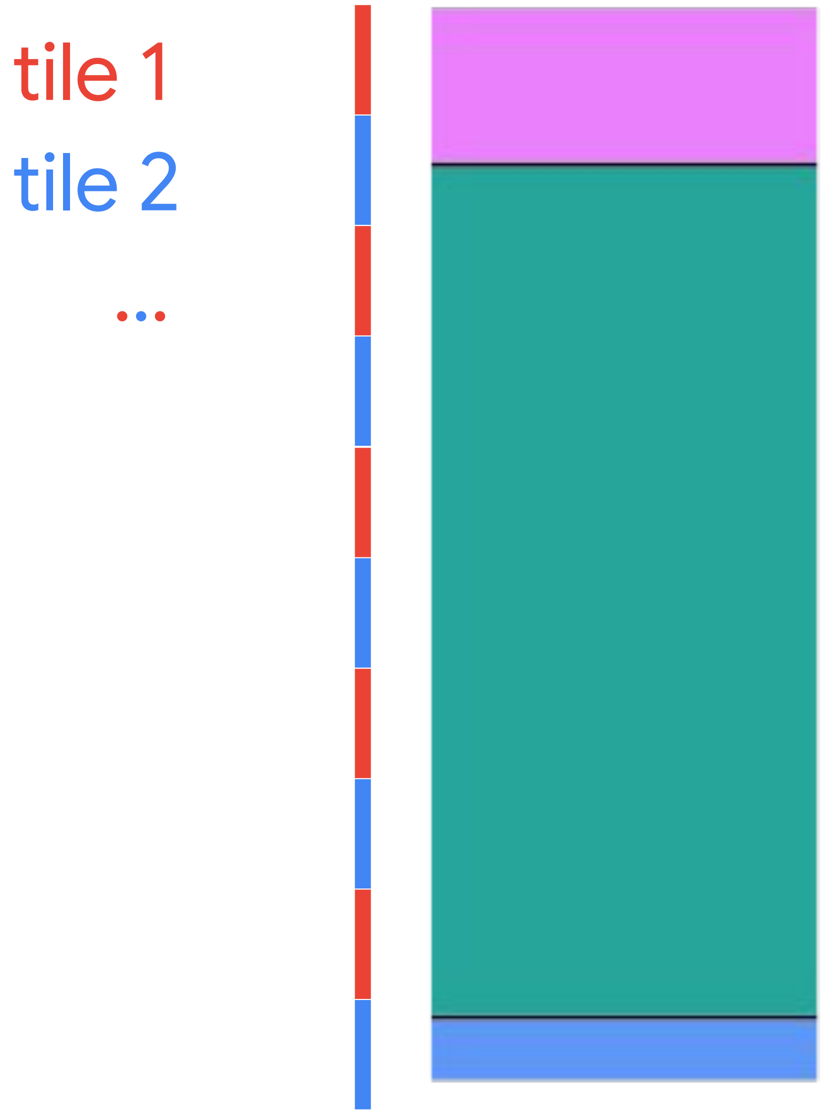
Modern “Megablox”

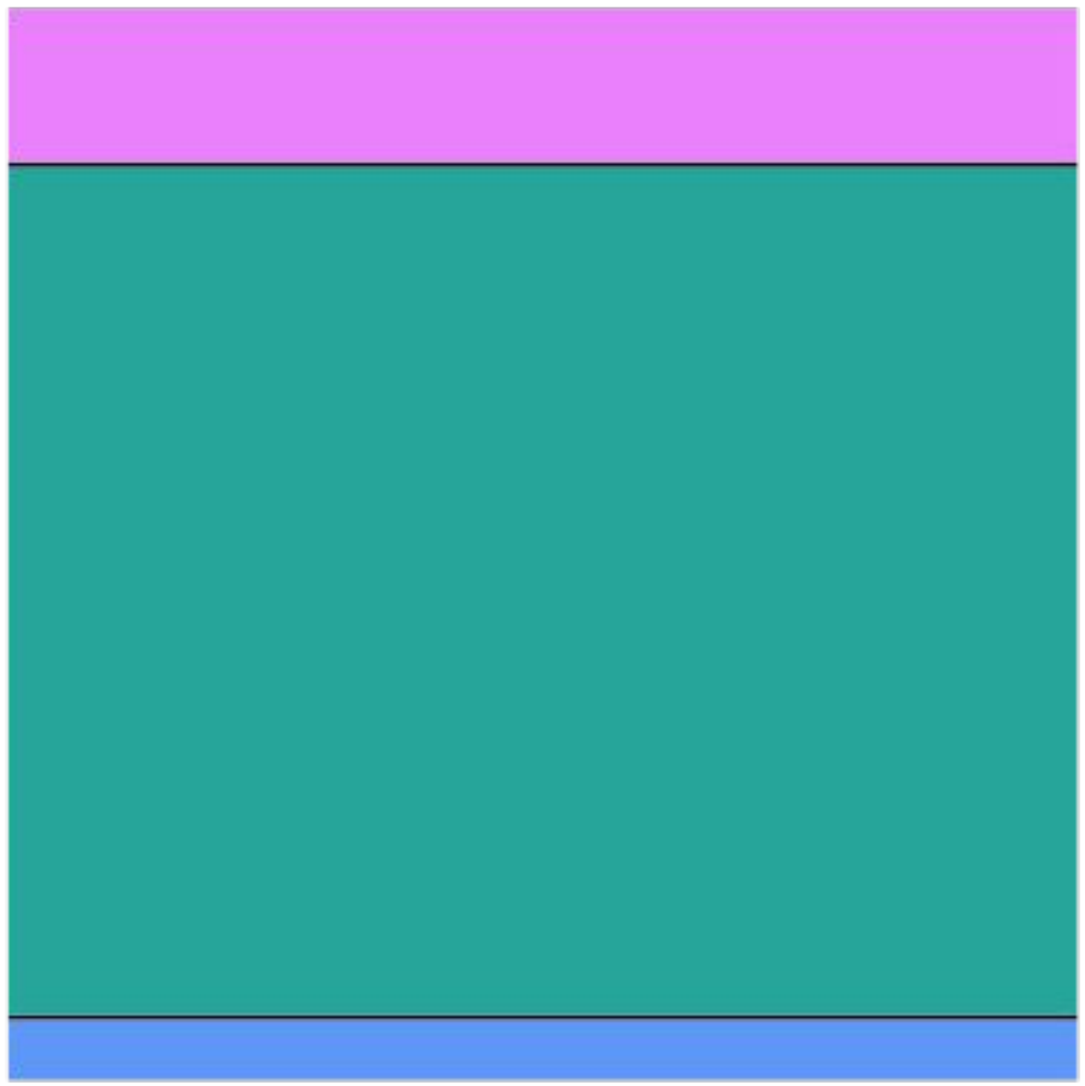
Megablox



- “ragged” representation usually means “pack stuff along one axis”
- the most dominant representation these days
- contiguous groups of rows belong to the (ordered) matrix in the stack
- LHS is 2D: `[m, k]`
- RHS is 3D: `[group, k, n]`
- OUT is 2D: `[m, n]`
- (theoretical) FLOPS are the same as for a simple matmul: `2·m·k·n`

Megablox
Accelerator Tiling & Tile Revisting





- need to tile evenly
  (very accelerator friendly)

- G - groups
- T - tile size
- N - tile number

- probability of having to "revisit" a tile: P(revisit) = 1 - 1 / T
- number of tiles revisited: G · P(revisit) = G · (1 - 1 / T)
- efficiency is tile number / actual tiles visited: N / (N + G · (1 - 1 / T))

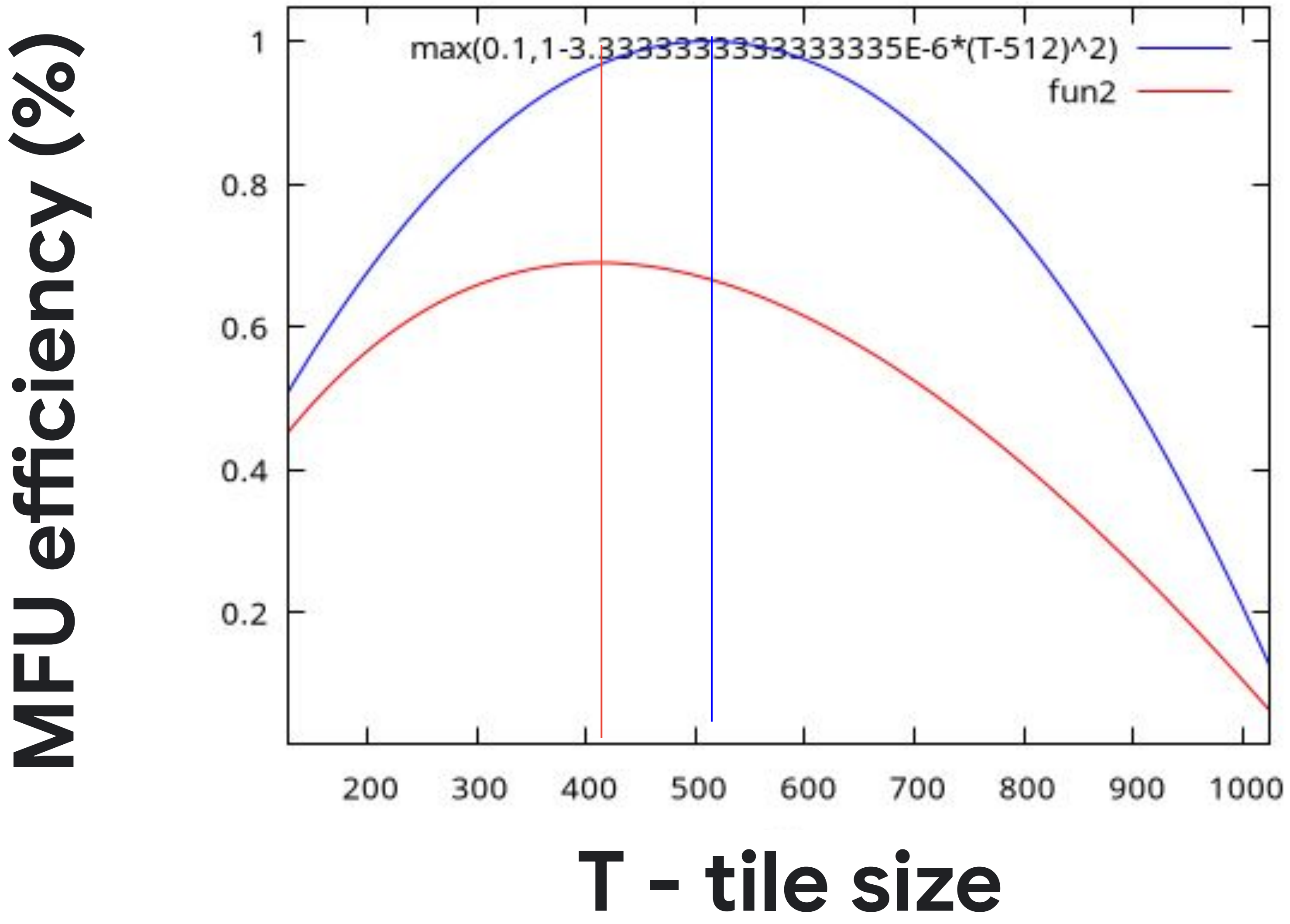
Megablox
Accelerator Efficiency

- G - groups
- T - tile size
- N - tile number

- probability of having to "revisit" a tile: P(revisit) = 1 - 1 / T
- number of tiles revisited: G · P(revisit) = G · (1 - 1 / T)
- efficiency is tile number / actual tiles visited: N / (N + G · (1 - 1 / T))

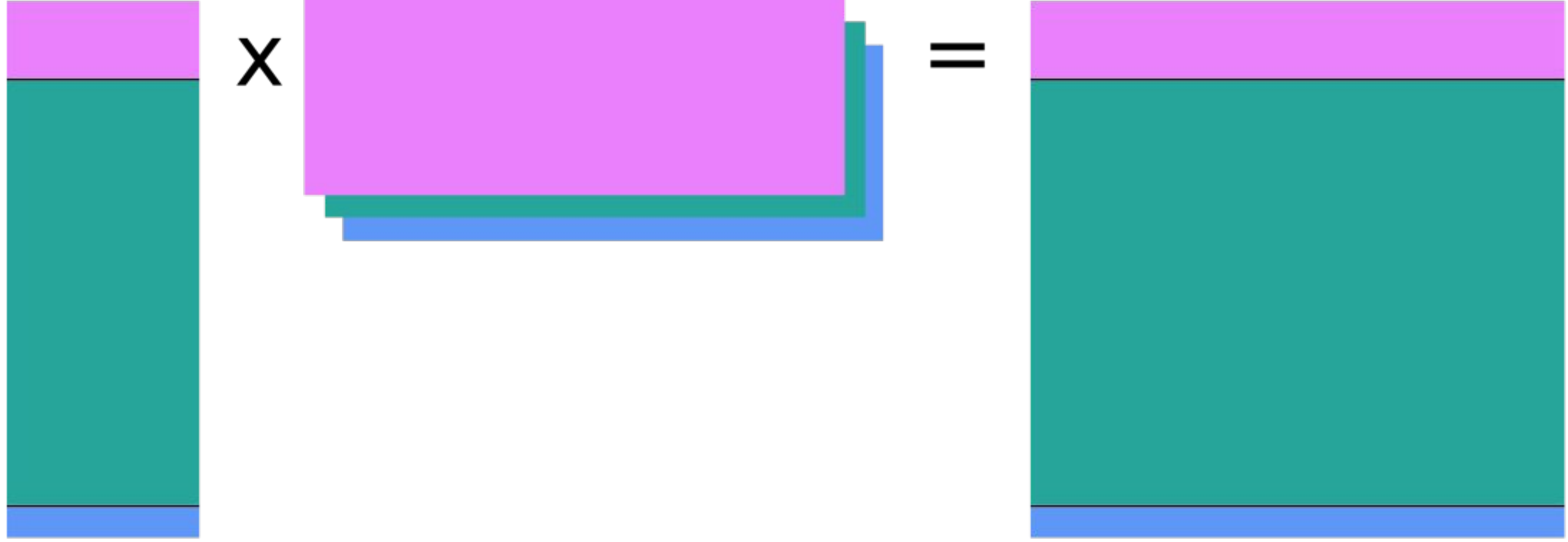
megablox efficiency model

- assume optimal matmul tile size model
- smaller tile sizes: more block revisiting
- overall lower MFU
- optimal megablox tile size skews smaller



Megablox
Derivatives

- LHS: `[m, k]`
- RHS: `[g, k, n]`
- OUT: `[m, n]`



d(LHS)
- needs to be [m, k]
- ragged_dot(
    d(OUT),
    RHS^T # transpose RHS
)
- FLOPS: `2·m·k·n` (still)
- reduction along the `n-axis`

d(RHS)
- needs to be [g, k, n]
- “transposed”_ragged_dot(
    LHS^T,
    d(OUT)
)
- FLOPS: `2·m·k·n` (still)
- lots of outer products
- reduction along: **masked** `m-axis`

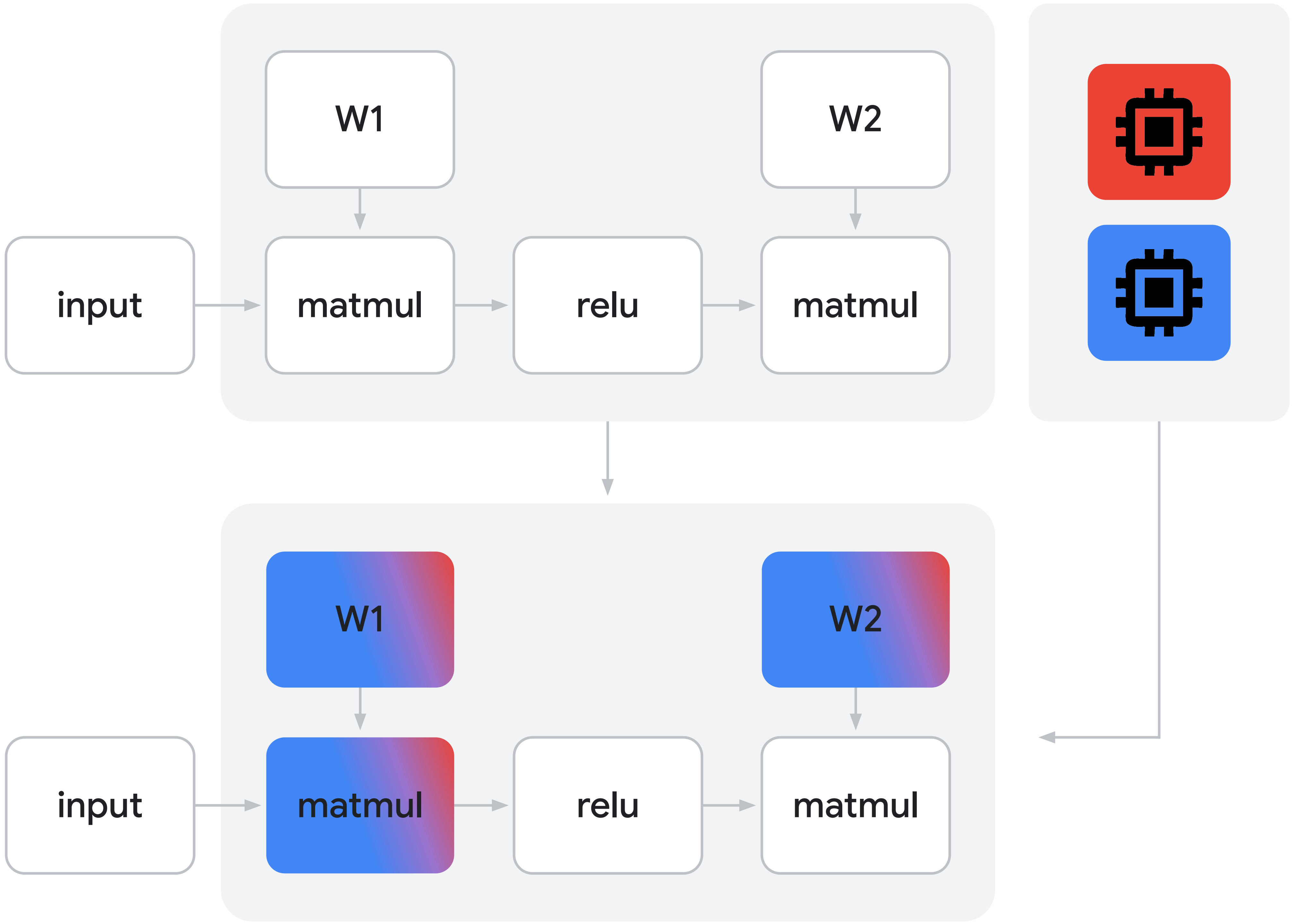
Megablox
Actual metadata computation

- don’t visit & skip empty blocks
  - this costs kernel dispatch latency
- compute tile and expert map flat
  - sometimes tile increase
  - sometimes expert
  - just a flat lookup table
- how to do this efficiently on TPU?
  - no loops, must use e.g., cumsum
  - computing metadata blocks TC

```python
def make_group_metadata(
    *,
    group_sizes: jnp.ndarray,
    m: int,
    tm: int,
    start_group: jnp.ndarray,
    num_nonzero_groups: int,
    visit_empty_groups: bool = True,
) -> GroupMetadata:
  """Create the metadata needed for grouped matmul computation.

  Args:
    group_sizes: A 1d, jnp.ndarray with shape [num_groups] and jnp.int32 dtype.
    m: The number of rows in lhs.
    tm: The m-dimension tile size being used.
    start_group: The group in group sizes to start computing from. This is
      particularly useful for when rhs num_groups is sharded.
    num_nonzero_groups: Number of groups in group sizes to compute on. Useful in
      combination with group_offset.
    visit_empty_groups: If True, do not squeeze tiles for empty groups out of
      the metadata. This is necessary for tgmm, where we at least need to zero
      the output for each group.

  Returns:
    tuple of:
      group_offsets: A 1d, jnp.ndarray with shape [num_groups+1] and jnp.int32
        dtype. group_offsets[i] indicates the row at which group [i] starts in
        the lhs matrix and group_offsets[i-1] = m.
      group_ids: A 1d, jnp.ndarray with shape [m_tiles + num_groups] and
        jnp.int32 dtype. group_ids[i] indicates which group grid index 'i' will
        work on.
      m_tile_ids: A 1d, jnp.ndarray with shape [m_tiles + num_groups] and
        jnp.int32. m_tile_ids[i] indicates which m-dimension tile grid index 'i'
        will work on.
    num_tiles: The number of m-dimension tiles to execute.
  """
  num_groups = group_sizes.shape[0]
  end_group = start_group + num_nonzero_groups - 1

  # Calculate the offset of each group, starting at zero. This metadata is
  # similar to row offsets in a CSR matrix. The following properties hold:
  #
  # group_offsets.shape = [num_groups + 1]
  # group_offsets[0] = 0
  # group_offsets[num_groups] = m
  #
  # The row at which group 'i' starts is group_offsets[i].
  group_ends = jnp.cumsum(group_sizes)
  group_offsets = jnp.concatenate([jnp.zeros(1, dtype=jnp.int32), group_ends])

  # Assign a group id to each grid index.
  #
  # If a group starts somewhere other than the start of a tile or ends somewhere
  # other than the end of a tile we need to compute that full tile. Calculate
  # the number of tiles for each group by rounding their end up to the nearest
  # 'tm' and their start down to the nearest 'tm'.

  # (1) Round the group_ends up to the nearest multiple of 'tm'.
  #
  # NOTE: This does not change group_offsets[num_groups], which is m
  # (because we enforce m is divisible by tm).
  rounded_group_ends = ((group_ends + tm - 1) // tm * tm).astype(jnp.int32)

  # (2) Round the group_starts down to the nearest multiple of 'tm'.
  group_starts = jnp.concatenate(
      [jnp.zeros(1, dtype=jnp.int32), group_ends[:-1]]
  )
  rounded_group_starts = group_starts // tm * tm

  # (3) Calculate the number of rows in each group.
  #
  # NOTE: Handle zero-sized groups as a special case. If the start for a
  # zero-sized group is not divisible by 'tm' its start will be rounded down and
  # its end will be rounded up such that its size will become 1 tile here.
  rounded_group_sizes = rounded_group_ends - rounded_group_starts
  rounded_group_sizes = jnp.where(group_sizes == 0, 0, rounded_group_sizes)

  # (4) Convert the group sizes from units of rows to unit of 'tm' sized tiles.
  #
  # An m-dimension tile is 'owned' by group 'i' if the first row of the tile
  # belongs to group 'i'. In addition to owned tiles, each group can have 0 or 1
  # initial partial tiles if it's first row does not occur in the first row of a
  # tile. The '0-th' group never has a partial tile because it always starts at
  # the 0-th row.
  #
  # If no group has a partial tile, the total number of tiles is equal to
  # 'm // tm'. If every group has a partial except the 0-th group, the total
  # number of tiles is equal to 'm // tm + num_groups - 1'. Thus we know that
  #
  # tiles_m <= group_tiles.sum() <= tiles_m + num_groups - 1
  #
  # Where tiles_m = m // tm.
  #
  # NOTE: All group sizes are divisible by 'tm' because of the rounding in steps
  # (1) and (2) so this division is exact.
  group_tiles = rounded_group_sizes // tm

  if visit_empty_groups:
    # Insert one tile for empty groups.
    group_tiles = jnp.where(group_sizes == 0, 1, group_tiles)

  # Create the group ids for each grid index based on the tile counts for each
  # group.
  #
  # NOTE: This repeat(...) will pad group_ids with the final group id if
  # group_tiles.sum() < tiles_m + num_groups - 1. The kernel grid will be sized
  # such that we only execute the necessary number of tiles.
  tiles_m = _calculate_num_tiles(m, tm)
  group_ids = jnp.repeat(
      jnp.arange(num_groups, dtype=jnp.int32),
      group_tiles,
      total_repeat_length=tiles_m + num_groups - 1,
  )

  # Assign an m-dimension tile id to each grid index.
  #
  # NOTE: Output tiles can only be re-visited consecutively. The following
  # procedure guarantees that m-dimension tile indices respect this.

  # (1) Calculate how many times each m-dimension tile will be visited.
  #
  # Each tile is guaranteed to be visited once by the group that owns the tile.
  # The remaining possible visits occur when a group starts inside of a tile at
  # a position other than the first row. We can calculate which m-dimension tile
  # each group starts in by floor-dividing its offset with `tm` and then count
  # tile visits with a histogram.
  #
  # To avoid double counting tile visits from the group that owns the tile,
  # filter these out by assigning their tile id to `tile_m` (one beyond the max)
  # such that they're ignored by the subsequent histogram. Also filter out any
  # group which is empty.
  #
  # TODO(tgale): Invert the 'partial_tile_mask' predicates to be more clear.
  partial_tile_mask = jnp.logical_or(
      (group_offsets[:-1] % tm) == 0, group_sizes == 0
  )

  # Explicitly enable tiles for zero sized groups, if specified. This covers
  # zero sized groups that start on a tile-aligned row and those that do not.
  if visit_empty_groups:
    partial_tile_mask = jnp.where(group_sizes == 0, 0, partial_tile_mask)

  partial_tile_ids = jnp.where(
      partial_tile_mask, tiles_m, group_offsets[:-1] // tm
  )

  tile_visits = (
      jnp.histogram(partial_tile_ids, bins=tiles_m, range=(0, tiles_m - 1))[0]
      + 1
  )

  # Create the m-dimension tile ids for each grid index based on the visit
  # counts for each tile.
  m_tile_ids = jnp.repeat(
      jnp.arange(tiles_m, dtype=jnp.int32),
      tile_visits.astype(jnp.int32),
      total_repeat_length=tiles_m + num_groups - 1,
  )

  # Account for sharding.
  #
  # Find the start of the groups owned by our shard and shift the group_ids and
  # m_tile_ids s.t. the metadata for our tiles are at the front of the arrays.
  #
  # TODO(tgale): Move this offset into the kernel to avoid these rolls.
  first_tile_in_shard = (group_ids < start_group).sum()
  group_ids = jnp.roll(group_ids, shift=-first_tile_in_shard, axis=0)
  m_tile_ids = jnp.roll(m_tile_ids, shift=-first_tile_in_shard, axis=0)

  # Calculate the number of tiles we need to compute for our shard.
  #
  # Remove tile visits that belong to a group not in our shard.
  iota = jnp.arange(num_groups, dtype=jnp.int32)
  active_group_mask = jnp.logical_and(iota <= end_group, iota >= start_group)
  group_tiles = jnp.where(active_group_mask, group_tiles, 0)
  num_tiles = group_tiles.sum()
  return (group_offsets, group_ids, m_tile_ids), num_tiles
```

The MoE Layer in JAX

Sharded MLP (or MoE)

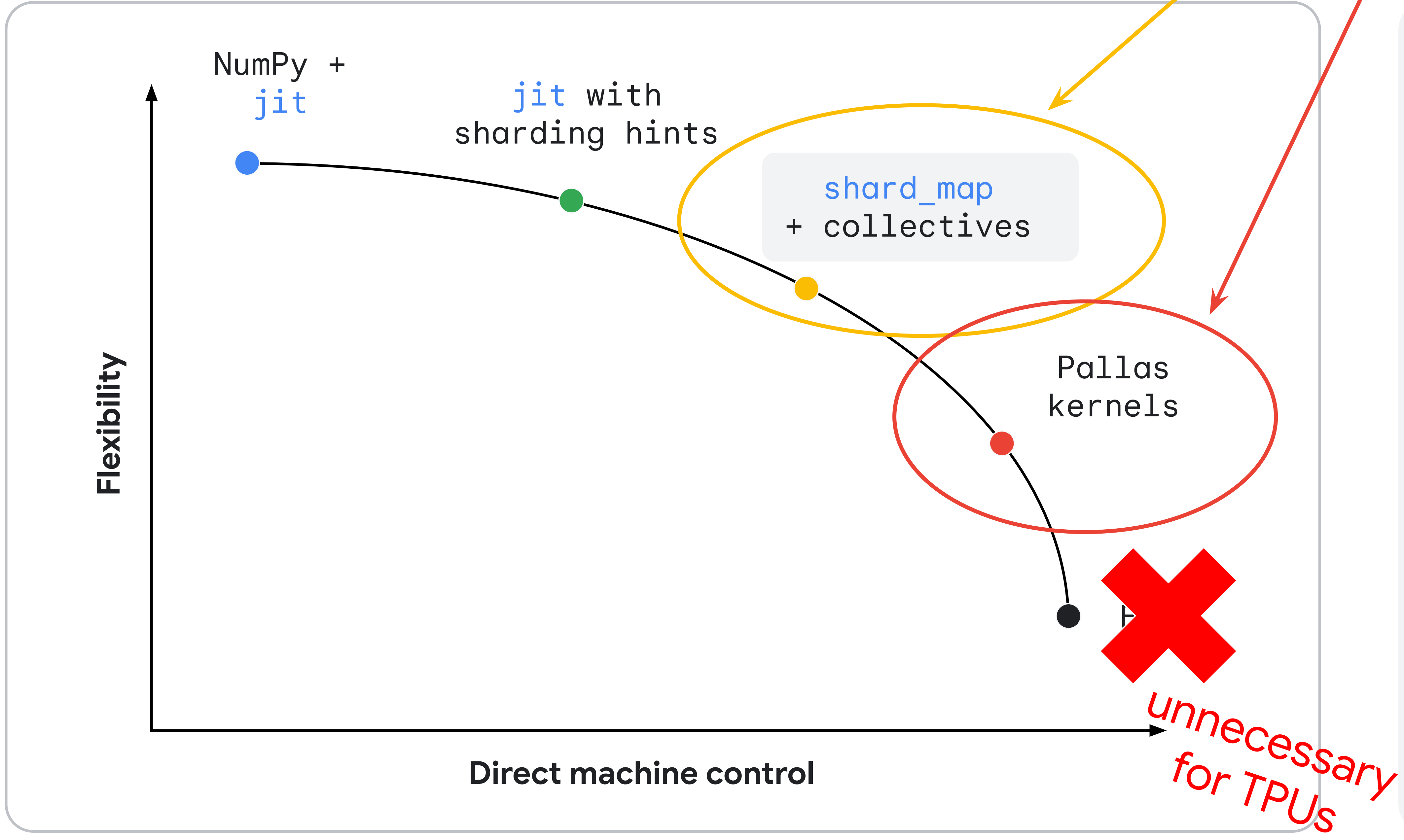


JAX: The escape hatch hierarchy

Flexibility vs. control

for MoE

for Megablox



Math

- Compiler, take the wheel!
- Here's a hint
- I'll handle comms
- Kernel languages
- FFI

Physics

MoE *Prep*

helper lambdas

token routing
(just computing the index map)

in specs and out specs for shard_map

constants for the MoE closure

```python
def moe_block(x: jax.Array, layer: MoELayer, cfg: Config):
    assert x.ndim == 3
    l2p = lambda *axes: logical_to_physical(axes, cfg.rules)
    _psc = lambda z, spec: reshard(z, P(*spec))
    psc = lambda z, spec: _qpsc(z, spec) if is_type(z, QuantArray) else _psc(z, spec)

    # we're decoding or device count does not divide total token count
    replicated_routing = x.shape[-2] == 1 or (x.shape[-2] * x.shape[-3]) % jax.device_count() != 0
    topk_weights, topk_idx = _route_tokens_to_moe_experts(x, layer.w_router, replicated_routing, cfg)
    tensor_axname, expert_axname = l2p("moe_e_tp")[0], l2p("moe_e_ep")[0]

    x_spec = l2p("batch", "sequence", None)
    topk_weights_spec, topk_idx_spec = l2p("batch", "sequence", None), l2p("batch", "sequence", None)
    out_spec = l2p("batch", "sequence", None)

    we_gate_spec = l2p("moe_e_ep", None, "moe_e_tp")
    we_up_spec = l2p("moe_e_ep", None, "moe_e_tp")
    we_down_spec = l2p("moe_e_ep", "moe_e_tp", None)
    we_gate = psc(layer.we_gate, we_gate_spec)
    we_up = psc(layer.we_up, we_up_spec)
    we_down = psc(layer.we_down, we_down_spec)

    in_specs = (x_spec, we_gate_spec, we_up_spec, we_down_spec, topk_weights_spec, topk_idx_spec)

    is_embedding_sharded = l2p("act_embed")[0] is not None
    if is_embedding_sharded:  # activations are sharded
        out_spec = P(*(out_spec[:-1] + (tensor_axname,)))  # override last axis name

    expert_count = cfg.mesh.axis_sizes[cfg.mesh.axis_names.index(expert_axname)] if expert_axname is not None else 1
    tensor_count = cfg.mesh.axis_sizes[cfg.mesh.axis_names.index(tensor_axname)] if tensor_axname is not None else 1
    assert cfg.moe_num_experts % expert_count == 0
    expert_size = cfg.moe_num_experts // expert_count
```

MoE *Compute*

sort indices

gather tokens
(potentially increase their count)

group sizes via bincount

up and gate projection
(first tensor-parallel stage)

down projection
(second tensor-parallel stage)

weight the tokens

all-gather tokens
and reduce across experts

```python
    @partial(shard_map, mesh=cfg.mesh, in_specs=in_specs, out_specs=out_spec, check_rep=False)
    def _expert_fn(x, we_gate, we_up, we_down, topk_weights, topk_idx):
        (b, s, d), e = x.shape, cfg.moe_experts_per_tok
        expert_idx = jax.lax.axis_index(expert_axname) if expert_axname is not None else 0
        tensor_idx = jax.lax.axis_index(tensor_axname) if tensor_axname is not None else 0
        del tensor_idx
        topk_idx_ = topk_idx.reshape(-1)
        valid_group_mask_ = (topk_idx_ >= expert_size * expert_idx) & (topk_idx_ < expert_size * (expert_idx + 1))
        expert_mapped_topk_idx_ = jnp.where(valid_group_mask_, topk_idx_ - expert_idx * expert_size, 2**30)

        sort_idx_ = jnp.argsort(expert_mapped_topk_idx_, axis=-1)  # [b * s * e]
        isort_idx_ = jnp.argsort(sort_idx_)

        topk_idx_sort_ = topk_idx_[sort_idx_]  # [b * s * e]
        expert_mapped_topk_idx_sort_ = expert_mapped_topk_idx_[sort_idx_]
        valid_group_mask_sort_ = expert_mapped_topk_idx_sort_ < 2**30
        expert_mapped_topk_idx_sort_ = jnp.where(expert_mapped_topk_idx_sort_ < 2**30, expert_mapped_topk_idx_sort_, 0)

        # equivalent to:
        # ```
        # x_repeat_ = jnp.repeat(x.reshape((-1, x.shape[-1])), e, axis=0)
        # x_repeat_sort_ = jnp.take_along_axis(x_repeat_, sort_idx_[:, None], axis=-2)  # [b * s, d]
        # ```
        x_repeat_sort_ = jnp.take_along_axis(
            x.reshape((-1, x.shape[-1])),
            sort_idx_[:, None] // e,
            axis=-2,  # index trick to avoid jnp.repeat
        )  # [b * s * e, d]

        group_sizes = jnp.bincount(topk_idx_sort_, length=cfg.moe_num_experts)
        group_sizes_shard = jax.lax.dynamic_slice_in_dim(group_sizes, expert_idx * expert_size, expert_size, 0)

        with jax.named_scope("we_gate"):
            ff_gate = _moe_gmm(x_repeat_sort_, we_gate, group_sizes_shard, expert_mapped_topk_idx_sort_, cfg)
            ff_gate = jax.nn.silu(ff_gate)
            ff_gate = jnp.where(valid_group_mask_sort_[..., None], ff_gate, 0)
        with jax.named_scope("we_up"):
            ff_up = _moe_gmm(x_repeat_sort_, we_up, group_sizes_shard, expert_mapped_topk_idx_sort_, cfg)
        ff_gate_up = jnp.where(valid_group_mask_sort_[..., None], ff_gate * ff_up, 0)
        with jax.named_scope("we_down"):
            ff_out = _moe_gmm(ff_gate_up, we_down, group_sizes_shard, expert_mapped_topk_idx_sort_, cfg)
            ff_out = jnp.where(valid_group_mask_sort_[..., None], ff_out, 0)  # expensive

        ff_out = ff_out * topk_weights.reshape(-1)[sort_idx_][:, None]

        with jax.named_scope("unpermute"):
            ff_out = jnp.take_along_axis(ff_out, isort_idx_[..., None], axis=-2)
        with jax.named_scope("expert_summing"):
            ff_out_expert = jnp.sum(ff_out.reshape((b * s, e, d)), -2)
            ff_out_expert = ff_out_expert.astype(cfg.dtype)

        with jax.named_scope("experts_collective"):
            # collectives
            psum_axes = tensor_axname if expert_axname is None else (expert_axname, tensor_axname)
            ff_out_expert = jax.lax.psum(ff_out_expert, psum_axes)
            ff_out_expert = ff_out_expert.reshape((b, s, ff_out_expert.shape[-1]))
            return ff_out_expert

    with jax.named_scope("moe_routed_expert"):
        x_ = psc(x, x_spec)
        ff_out_expert = _expert_fn(x_, we_gate, we_up, we_down, topk_weights, topk_idx)[..., : x.shape[-1]]
    return psc(ff_out_expert, l2p("batch", "sequence", "act_embed"))
```

Quantization

Quantizing Matmuls

most hardware computes matmuls 2x as fast in lower precision (int8 / fp8)

full-channel quantization:

- compute-bound: ([m, k], [m, 1]) @ ([k, n], [1, n]) = ([m, k] @ [k, n]) * [m, 1] * [1, n]
- HBM BW-bound:
  - option 1 (scale out): [m, k]@ ([k, n], [1, n]) = ([m, k] @ [k, n]) * [1, n]
  - option 1 (scale in): [m, k]@ ([k, n], [k, 1]) = ([m, k] * [k, 1]) @ [k, n]

what about subchannel quantization?

- similar, but each tile size along reduction dimension receives separate scale
- ([tile_m, tile_k] @ [tile_k, tile_n]) * [tile_m, 1] * [1, tile_n]

Quantizing Ragged Dot

LHS: `[m, k]`
RHS: `[g, k, n]`
OUT: `[m, n]`

OUT: `ragged_dot(LHS, RHS)`
`d(LHS): ragged_dot(d(OUT), RHS`^{T}`)`
`d(RHS): transposed_ragged_dot(LHS`^{T}`, d(OUT))`

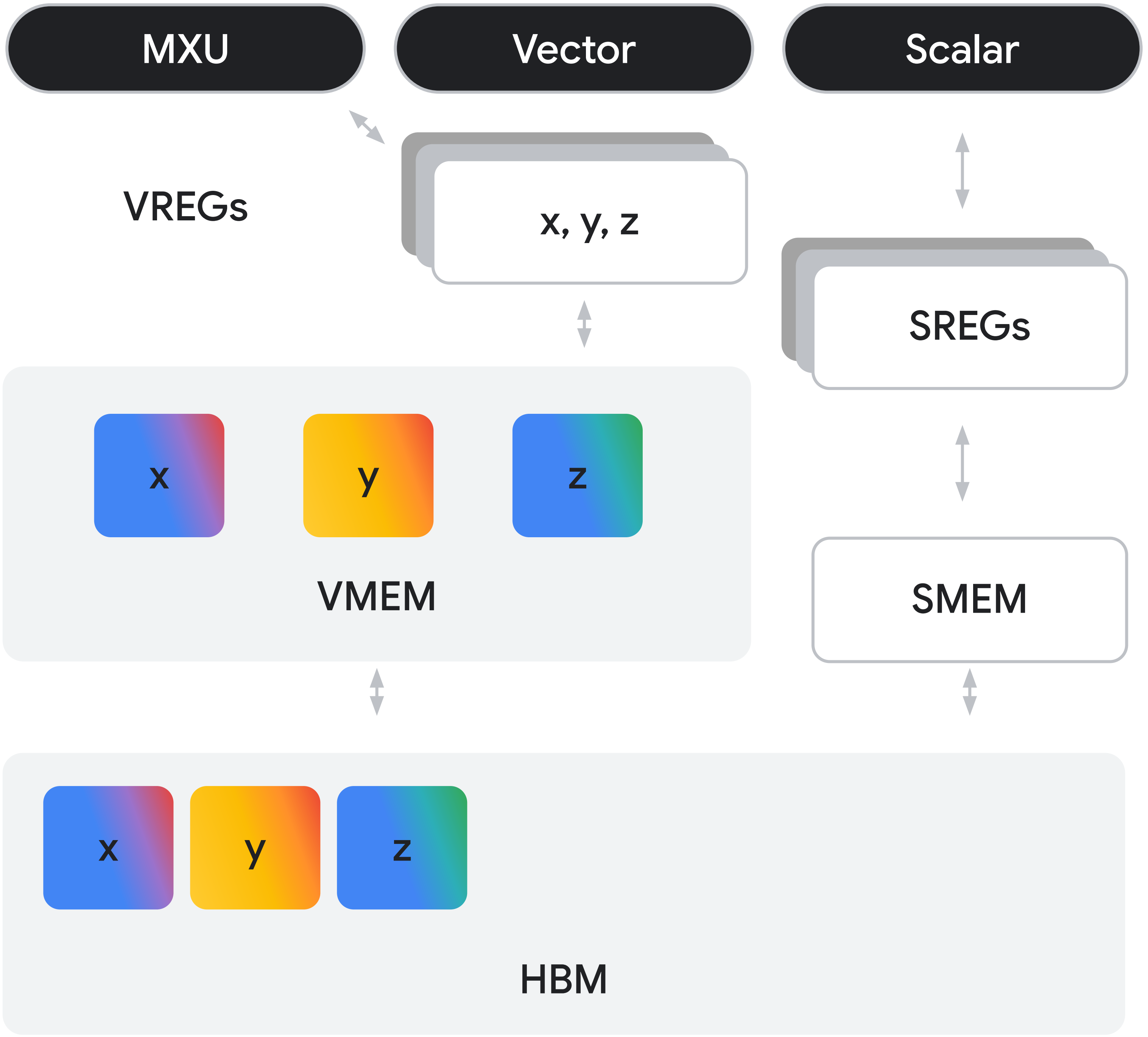
- quantizing very similar to matmul quantization strategies
- full-channel quantization:
  - ([m, k], [m, 1]) @ ([k, n], [1, n]) = ([m, k] @ [k, n]) * [m, 1] * [1, n]

- usually want single scale (or just a few scales, subchannel) along **reduction axis**
- reduction axis **changes** (matmuls have the same problem)

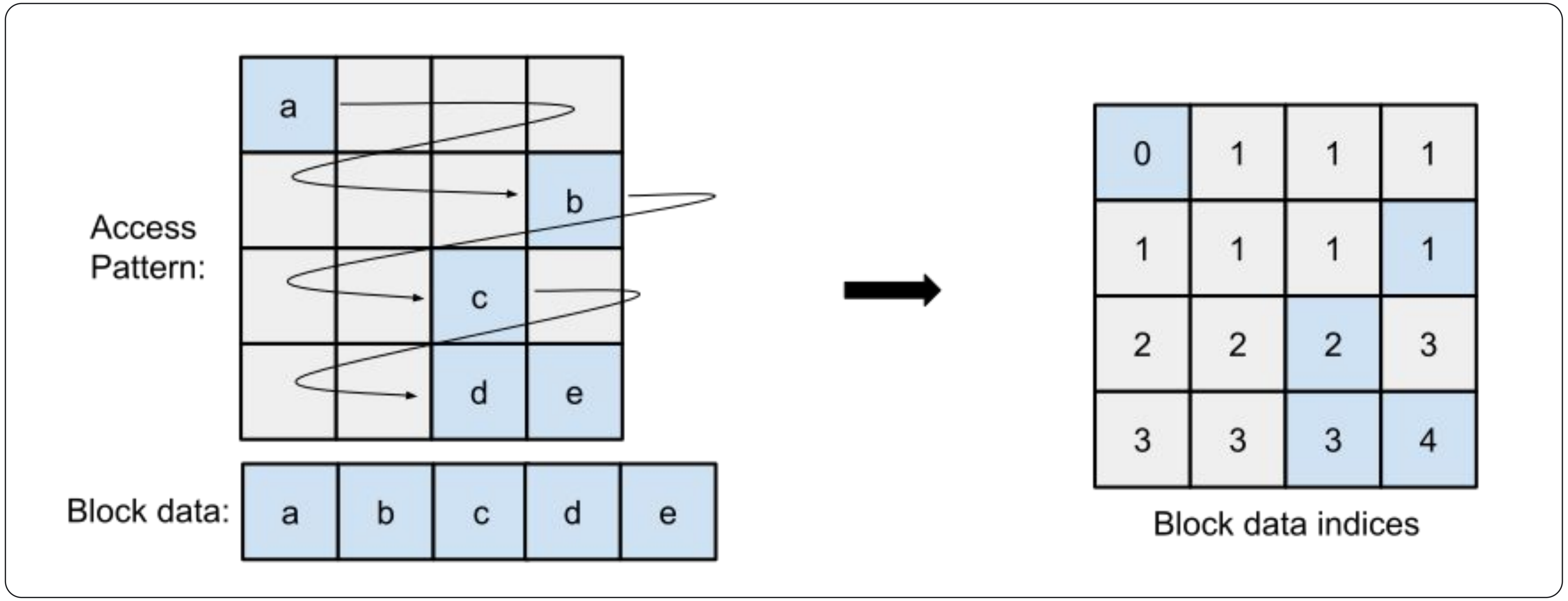
OUT: k-axis d(LHS): n-axis d(RHS): m-axis

Matmuls & **Megablox** in Pallas

TPU Memory pipeline



Sparse computations in pallas
`pltpu.PrefetchScalarGridSpec`



- “2, then 2” uses no BW
- read in the next block ASAP
- small penalty for visiting block at all
- can skip computation inside kernel

- pltpu.PrefetchScalarGridSpec is super useful
- generally pre-compute a metadata lookup table, then just iterate over entries
- example: `pl.BlockSpec(..., lambda i, j, scalar1_ref, scalar2_ref: (scalar1_ref[i], scalar2_ref[j]))`
- can re-implement the prefetch scalar grid using existing pallas APIs: https://github.com/openxla/tokamax/blob/main/tokamax/_src/mosaic_tpu.py#L126

Writing a Matmul on TPUs

- a super simple kernel
  - `pl.dot` to get correct “output” dtype
  - we’re issuing a hardware matmul
- 3D grid, no need for anything special
- **megablox** kernel is essentially this
  - apart from metadata of course
  - some row masking for tile overlap

```
def matmul_kernel(x_ref, y_ref, out_ref):
    out_ref[...] = pl.dot(x_ref[...], y_ref[...]).astype(out_ref.dtype)

def matmul(A, B, block_m, block_n, block_k):
    in_specs = [
        pl.BlockSpec((block_m, block_k), lambda i, j, k: (i, k)),
        pl.BlockSpec((block_k, block_n), lambda i, j, k: (k, j))
    ]
    out_specs = pl.BlockSpec((block_m, block_n), lambda i, j, k: (i, j))
    return pl.pallas_call(
        matmul_kernel,
        out_shape=jax.ShapeDtypeStruct((m, n), "bfloat16"),
        grid=(pl.cdiv(m, block_m), pl.cdiv(n, block_n), pl.cdiv(k, block_k)),
        in_specs=in_specs, out_specs=out_specs,
    )(A, B)
C = jax.jit(partial(matmul, block_m=2048, block_n=1024, block_k=1024))(A, B)
```

matmul: gist.github.com/rdyro/dd149ef5650f185bd96ec0666a23de9e
megablox: [github.com/openxla/tokamax ... pallas_mosaic_tpu_kernel.py](github.com/openxla/tokamax)

Tuning is (essentially) necessary
tune-jax

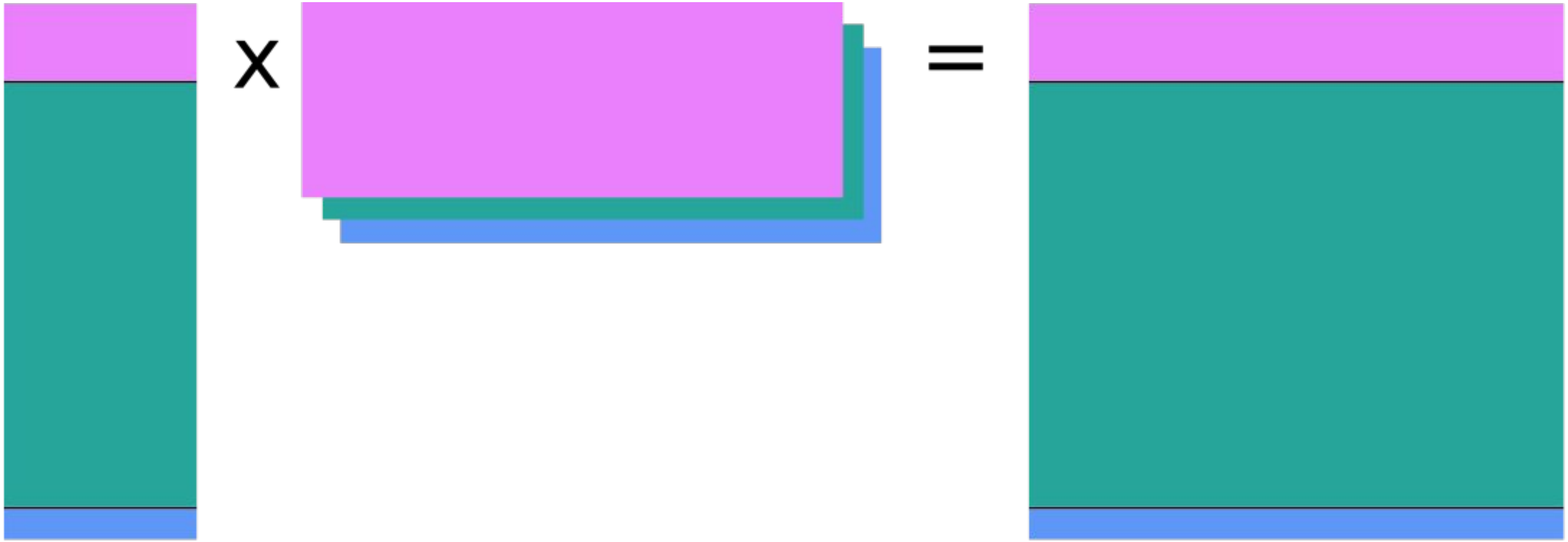
```python
import tune_jax
tune_jax.logger.setLevel("INFO") # print some info for sanity
tiles = [512, 1024, 2048, 4096] # any multiple of 128 will do
hyperparams = dict(
    block_m=tiles,
    block_k=tiles,
    block_n=tiles,
)
fn = tune_jax.tune(matmul, hyperparams=hyperparams)
_ = fn(A, B) # run to tune
print(tune_jax.tabulate(fn)) # print results nicely
```

```
Compiling...: 100%|██████████| 64/64 [00:39<00:00,  1.62it/s]
Compiling...: 100%|██████████| 39/39 [00:02<00:00, 16.27it/s]
Profiling tpu:   0%|          | 0/5 [00:00<?, ?it/s]
Saving optimization profile to `/tmp/tuning_profile_2025-10-19_02:15:04_87s164fh`

Profiling tpu: 100%|██████████| 5/5 [00:03<00:00,  1.66it/s]

 id    block_m    block_k    block_n    t_mean (s)    t_std (s)
----  ---------  ---------  ---------  ------------  -----------
  34       2048        512       2048    1.3220e-03   4.6820e-07
  49       4096        512       1024    1.3353e-03   2.1772e-07
...
   4        512       1024        512    3.4113e-03   8.1699e-05
   0        512        512        512    4.3446e-03   4.1993e-06
```

- worst result is 3.3x slower
- gap can get much wider for more complicated kernels
- `tune_jax` - available on PyPI
  - compiles in parallel (multi-core CPU speedup)
  - timing via automatic xprof parsing

future work & ragged dot on GPU

Future work



- collective fusions into the ragged dot kernel
- efficient quantization support
  - reusing quantized matrices in the backwards pass
  - in-kernel dynamic quantization

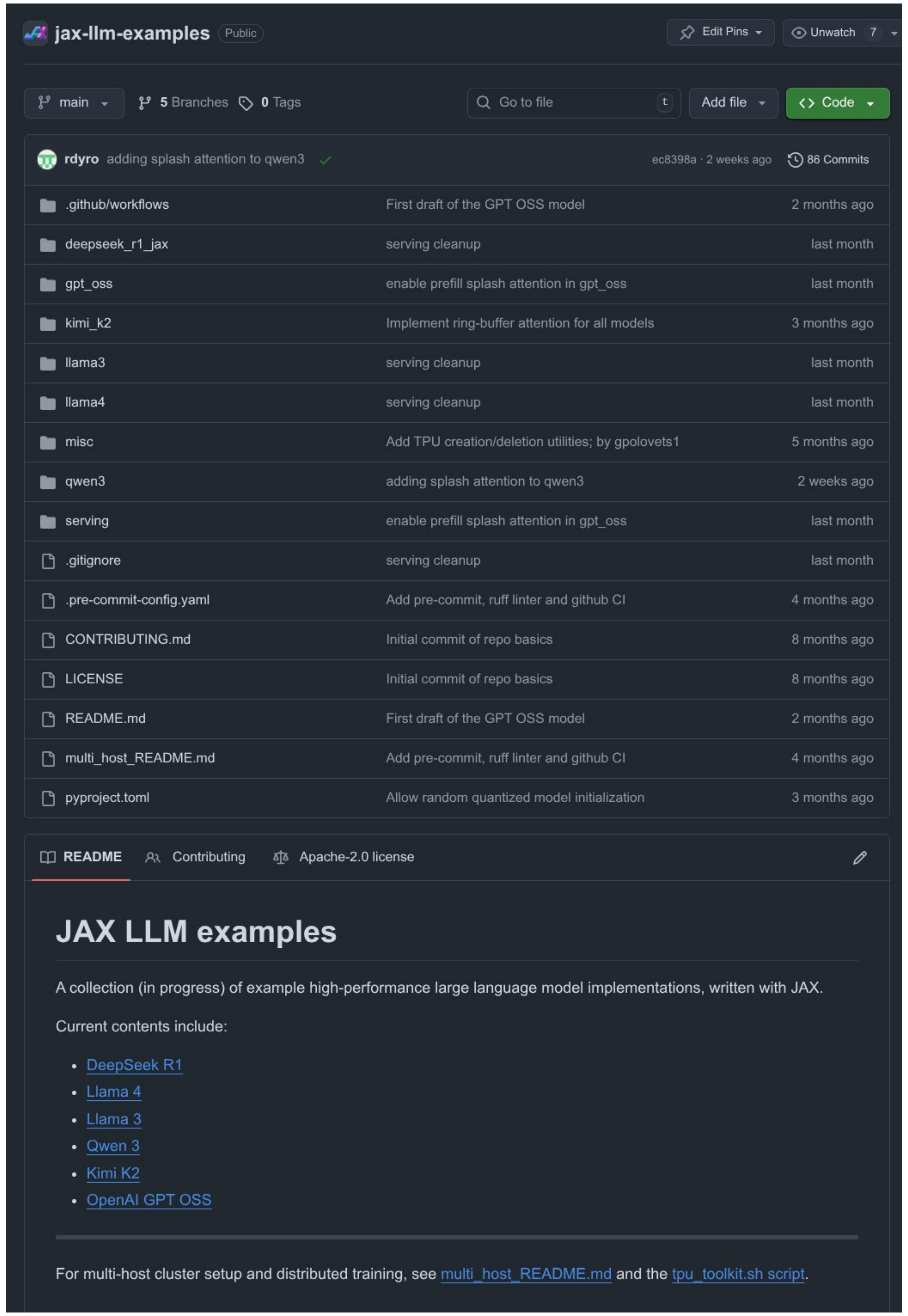
ragged dot on GPU

- originally in CUTLASS/CUBLAS
- kernel languages these days mostly
  - (many) triton implementations
  - e.g., (for JAX) https://github.com/rdyro/gpu_ragged_dot
    - not particularly efficient (no wgmma)

- Mosaic GPU implementation
  - matmul itself more complicated (166 lines)
  - more dynamic control over tiles
  - github.com/jax-ml/jax ... blackwell_ragged_dot_mgpu.py

Extras

jax-llm-examples
end-to-end inference

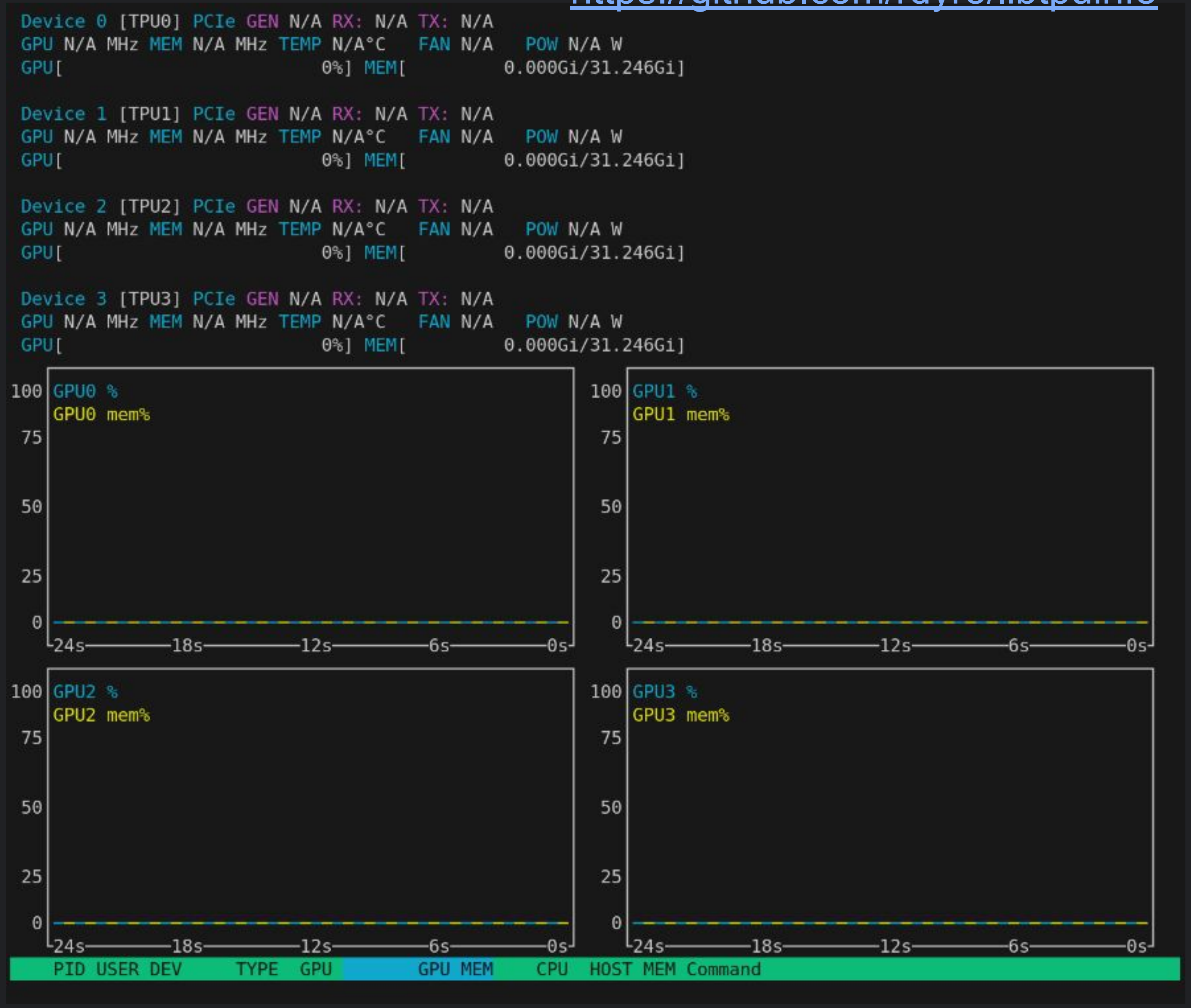



```python
@partial(jax.shard_map, mesh=cfg.mesh, in_specs=in_specs, out_specs=out_spec, check_vma=False)
def _expert_fn(x, we_gate_up, we_gate_up_bias, we_down, we_down_bias, topk_weights, topk_idx):
    (b, s, d), e = x.shape, cfg.moe_experts_per_tok
    expert_idx = jax.lax.axis_index(expert_axname) if expert_axname is not None else 0
    tensor_idx = jax.lax.axis_index(tensor_axname) if tensor_axname is not None else 0
    topk_idx_ = topk_idx.reshape(-1)
    valid_group_mask_ = (topk_idx_ >= expert_size * expert_idx) & (topk_idx_ < expert_size * (expert_idx + 1))
    expert_mapped_topk_idx_ = jnp.where(valid_group_mask_, topk_idx_ - expert_idx * expert_size, 2**30)

    sort_idx_ = jnp.argsort(expert_mapped_topk_idx_, axis=-1)  # [b * s * e]
    isort_idx_ = jnp.argsort(sort_idx_)

    if cfg.ep_strategy == "prefill":
        truncate_size = round(2 * sort_idx_.size / expert_count)
        sort_idx_, isort_idx_ = sort_idx_[:truncate_size], isort_idx_[:truncate_size]

    topk_idx_sort_ = topk_idx_[sort_idx_]  # [b * s * e]
    expert_mapped_topk_idx_sort_ = expert_mapped_topk_idx_[sort_idx_]
    valid_group_mask_sort_ = expert_mapped_topk_idx_sort_ < 2**30
    expert_mapped_topk_idx_sort_ = jnp.where(expert_mapped_topk_idx_sort_ < 2**30, expert_mapped_topk_idx_sort_, 0)

    # equivalent to:
    # ```
    # x_repeat_ = jnp.repeat(x.reshape((-1, x.shape[-1])), e, axis=0)
    # x_repeat_sort_ = jnp.take_along_axis(x_repeat_, sort_idx_[:, None], axis=-2)  # [b * s, d]
    # ```
    x_repeat_sort_ = jnp.take_along_axis(x.reshape((-1, x.shape[-1])), sort_idx_[:, None] // e, axis=-2)
    # [b * s * e, d] # "// e" is an index trick to avoid jnp.repeat

    group_sizes = jnp.bincount(topk_idx_sort_, length=cfg.moe_num_experts)
    group_sizes_shard = jax.lax.dynamic_slice_in_dim(group_sizes, expert_idx * expert_size, expert_size, 0)

    with jax.named_scope("we_gate"):
        ff_gate_up = _moe_gmm(x_repeat_sort_, we_gate_up, group_sizes_shard, expert_mapped_topk_idx_sort_, cfg)
        ff_gate_up = ff_gate_up + we_gate_up_bias[expert_mapped_topk_idx_sort_, :]
        ff_gate = jnp.clip(ff_gate_up[..., ::2], max=cfg.moe_gate_up_limit)
        ff_up = jnp.clip(ff_gate_up[..., 1::2], min=-cfg.moe_gate_up_limit, max=cfg.moe_gate_up_limit)
        ff_gate_up = (ff_up + 1) * (ff_gate * jax.nn.sigmoid(ff_gate * cfg.moe_gate_up_alpha))
        ff_gate_up = jnp.where(valid_group_mask_sort_[..., None], ff_gate_up, 0)
    with jax.named_scope("we_down"):
        ff_out = _moe_gmm(ff_gate_up, we_down, group_sizes_shard, expert_mapped_topk_idx_sort_, cfg)
        ff_out = ff_out + (tensor_idx == 0) * we_down_bias[expert_mapped_topk_idx_sort_, :]
        ff_out = jnp.where(valid_group_mask_sort_[..., None], ff_out, 0)  # expensive

    if cfg.ep_strategy == "prefill":
        rs_shape = math.ceil((ff_out.shape[-1] // tensor_count) / 256) * 256 * tensor_count
        pad_size = rs_shape - ff_out.shape[-1]
        ff_out = jnp.pad(ff_out, ((0, 0), (0, pad_size)))
        ff_out = jax.lax.psum_scatter(ff_out, axis_name=tensor_axname, scatter_dimension=1, tiled=True)
```

Extras

nvtop: TPU support

https://github.com/syllo/nvtop
https://github.com/rdyro/libtpuinfo




```
wget https://github.com/rdyro/libtpuinfo/releases/download/v0.0.1/libtpuinfo-linux-x86_64.so
sudo mv libtpuinfo-linux-x86_64.so /lib/libtpuinfo.so
git clone https://github.com/Syllo/nvtop.git
cd nvtop && mkdir build && cd build && cmake -DTPU_SUPPORT=ON .. && make
sudo cp ./src/nvtop /usr/local/bin
```