

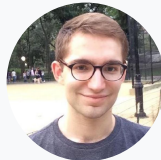
# High Performance LLMs From First Principles (2024)

# Pallas

A kernel language for GPUs and TPUs



Sharad Vikram



Adam Paszke

& many others from JAX and XLA!

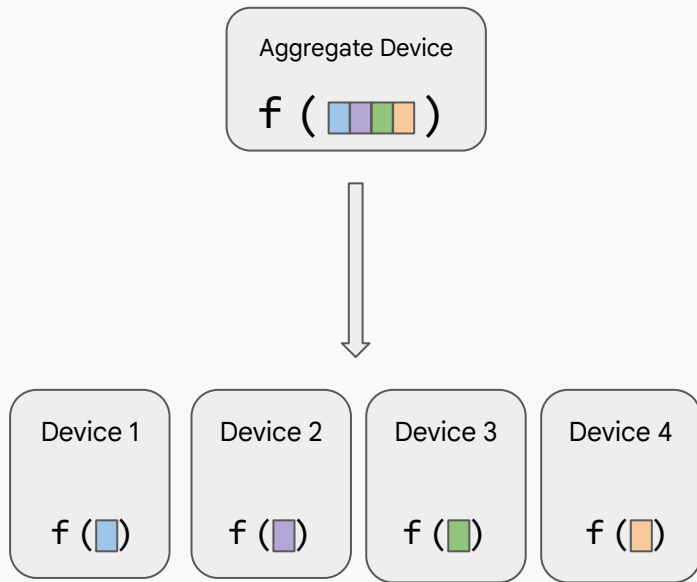
Motivation: Control

# JAX and Control

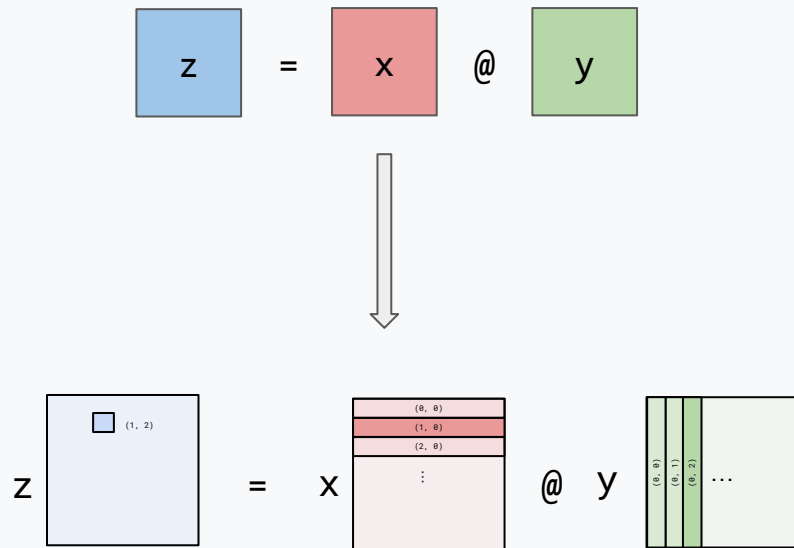
- JAX can be considered a **declarative** library for numerical computing
  - We stage out user programs and a compiler (XLA) determines how they are executed
  - Being compiler-friendly has been a huge success story
- On the other hand, power users/use-cases often want to:
  - Use advanced algorithms that compiler doesn't yet support (e.g. flash attention)
  - Use the hardware in different ways than are exposed by the compiler
- Goals:
  - Add features that provide **control** - give controls when users want it (manual OR automatic)
  - Expose them in an intuitive, accessible way to not alienate users

# New APIs in JAX for Control

`shard_map`  
Per-device parallelism



`pallas`  
Custom kernels for accelerators



# Pallas Goals

## Some challenges in writing kernels

- Often not expressible in Python
- Need to carefully consider memory access patterns
- Can be hardware/platform specific (not portable)
- Boilerplate in integrating with existing frameworks

## Our Approach

- Everything in Python
- Expose control over memory access
- Shared API across platforms
- Integrates with JAX + transformations

## Related: Triton

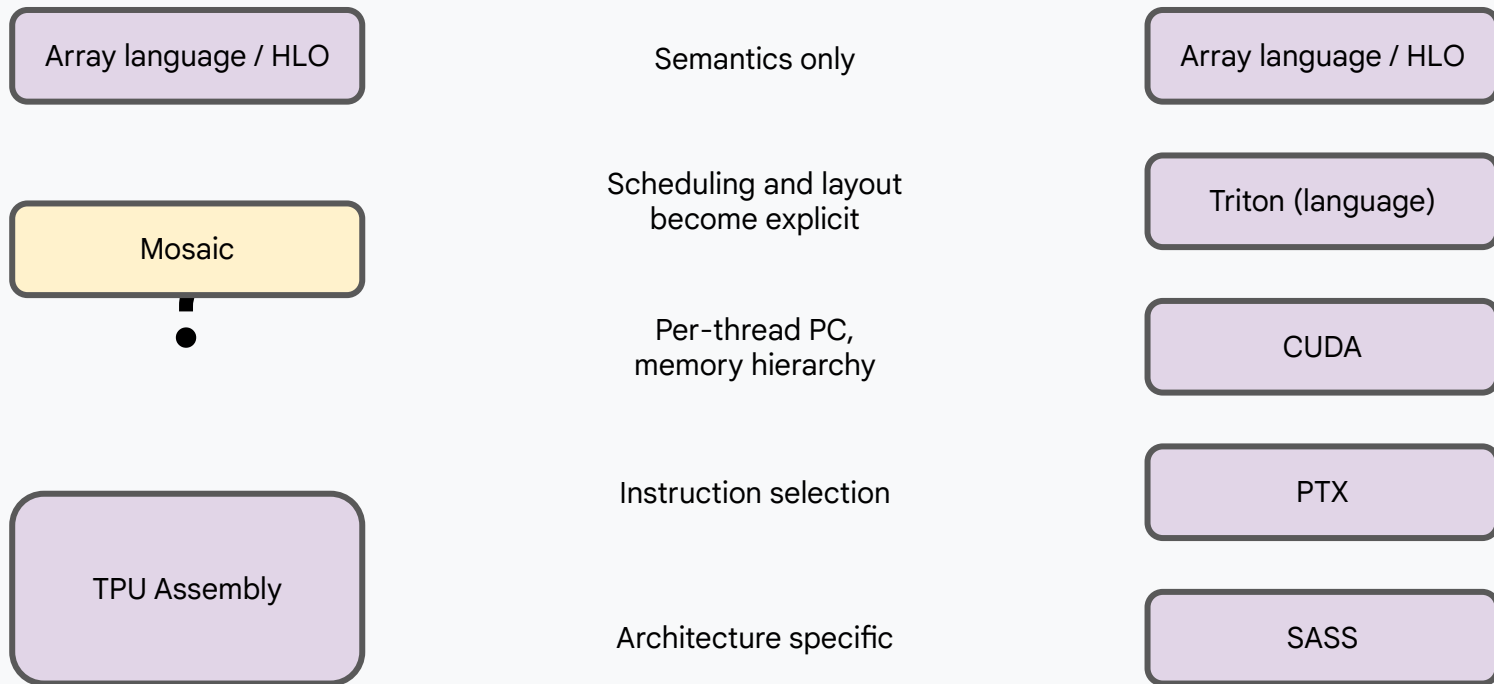
- Open source, primary development at OpenAI
- **Significantly** more productive than CUDA
- Adoption throughout industry (OpenAI, Meta, NVidia, AMD, etc.)
- Default codegen for TorchInductor (now also used by XLA)
- Still **not quite as productive** as NumPy/JAX
  - Unfamiliar Python DSL
  - Inconvenient embedding techniques
  - Missing out on automatic transforms



What's the story on  
TPUs?



# GPU & TPU language abstraction ladder



# Mosaic in one slide

- MLIR based syntax and implementation
  - Right now based on a mix of standard dialects + TPU dialect
- Similar abstraction level as Triton
- Embeddable into larger XLA:TPU computations as a CustomCall HLO
- Lowers to TPU assembly (skips HLO)
- Takes logically-shaped vector/matrix operations and tiles them over hardware registers and units (e.g. we tile over (8,128)-size vector registers).

```
func.func @main(%i: i32, %j: i32, %k: i32,
               %lhs: memref<256x256xf32>, %rhs: memref<256x256xf32>,
               %out: memref<256x256xf32>) attributes {
  iteration_bounds = array<i64: 4, 4, 4>,
  window_params = [
    {transform_indices = @transform_lhs, window_bounds = array<i64: 256, 256>},
    {transform_indices = @transform_rhs, window_bounds = array<i64: 256, 256>},
    {transform_indices = @transform_out, window_bounds = array<i64: 256, 256>}
  ] {
    %cst = arith.constant dense<0.000000e+00> : vector<256x256xf32>
    %cst_0 = arith.constant dense<0> : vector<256x256xi32>
    %c128 = arith.constant 128 : index
    %c0 = arith.constant 0 : index
    %0 = vector.broadcast %k : i32 to vector<256x256xi32>
    %1 = arith.cmpi eq, %0, %cst_0 : vector<256x256xi32>
    %2 = vector.load %rhs[%c0, %c0] : memref<256x256xf32>, vector<256x256xf32>
    %3 = vector.load %lhs[%c0, %c0] : memref<256x256xf32>, vector<256x256xf32>
    %4 = vector.load %out[%c0, %c0] : memref<256x256xf32>, vector<256x256xf32>
    %5 = arith.select %1, %cst, %4 : vector<256x256xi1>, vector<256x256xf32>
    %6 = vector.contract {...} %3, %2, %5
          : vector<256x256xf32>, vector<256x256xf32> into vector<256x256xf32>
    vector.store %6, %out[%c0, %c0] : memref<256x256xf32>, vector<256x256xf32>
    return
  }
}

func.func @transform_lhs(%i: i32, %j: i32, %k: i32) -> (i32, i32) {
  return %i, %k : i32, i32
}

func.func @transform_rhs(%i: i32, %j: i32, %k: i32) -> (i32, i32) {
  return %k, %j : i32, i32
}

func.func @transform_out(%i: i32, %j: i32, %k: i32) -> (i32, i32) {
  return %i, %j : i32, i32
}
```

# Pallas on TPU: “Hello world” (a.k.a. add)

*Shaped references*

```
def add_kernel(x_ref, y_ref, z_ref):  
    z_ref[...] = x_ref[...] + y_ref[...]
```

Load + compute using JAX NumPy! + store

```
x, y = ... # JAX arrays
```

```
assert x.shape == y.shape
```

```
jax_add_kernel = pl.pallas_call(  
    add_kernel,
```

```
    out_shape=jax.ShapeDtypeStruct(x.shape, x.dtype),  
)
```

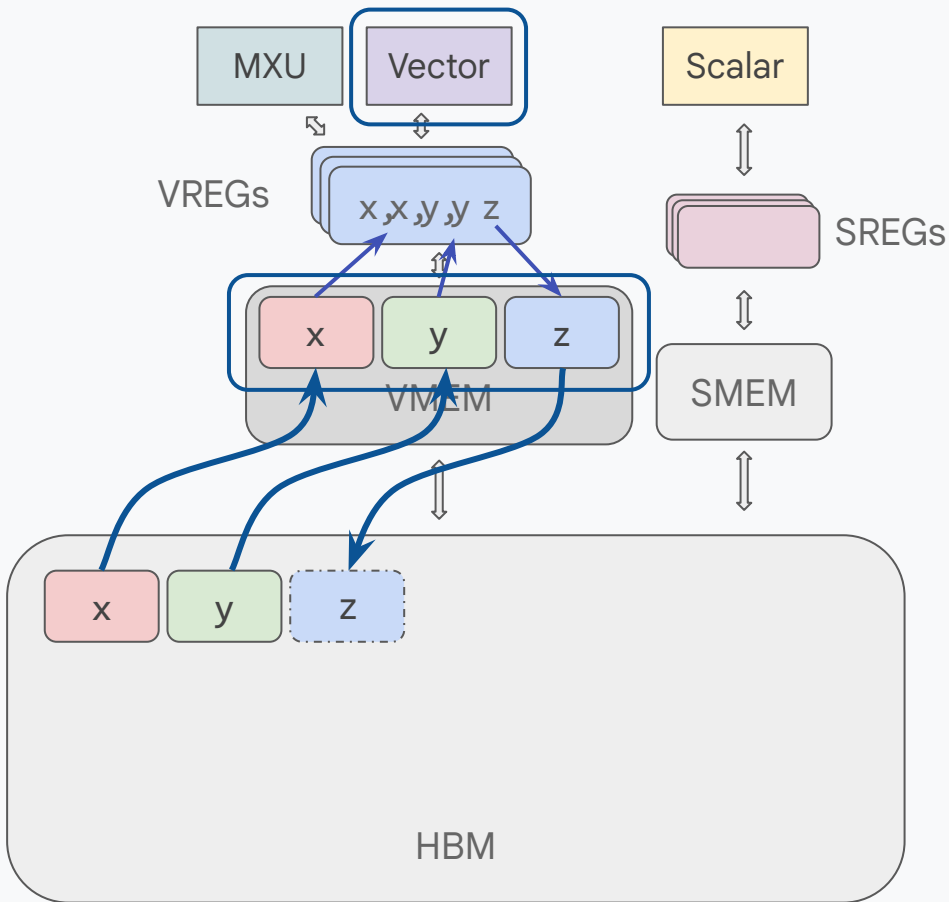
```
z = jax_add_kernel(x, y)
```

*pallas\_call lifts a kernel to a JAX function*

# Breakdown

```
def add_kernel(x_ref, y_ref, z_ref):  
    z_ref[...] = x_ref[...] + y_ref[...]  
x, y = ... # JAX arrays  
assert x.shape == y.shape  
jax_add_kernel = pl.pallas_call(  
    add_kernel,  
    out_shape=jax.ShapeDtypeStruct(x.shape, x.dtype),  
)  
z = jax_add_kernel(x, y)
```

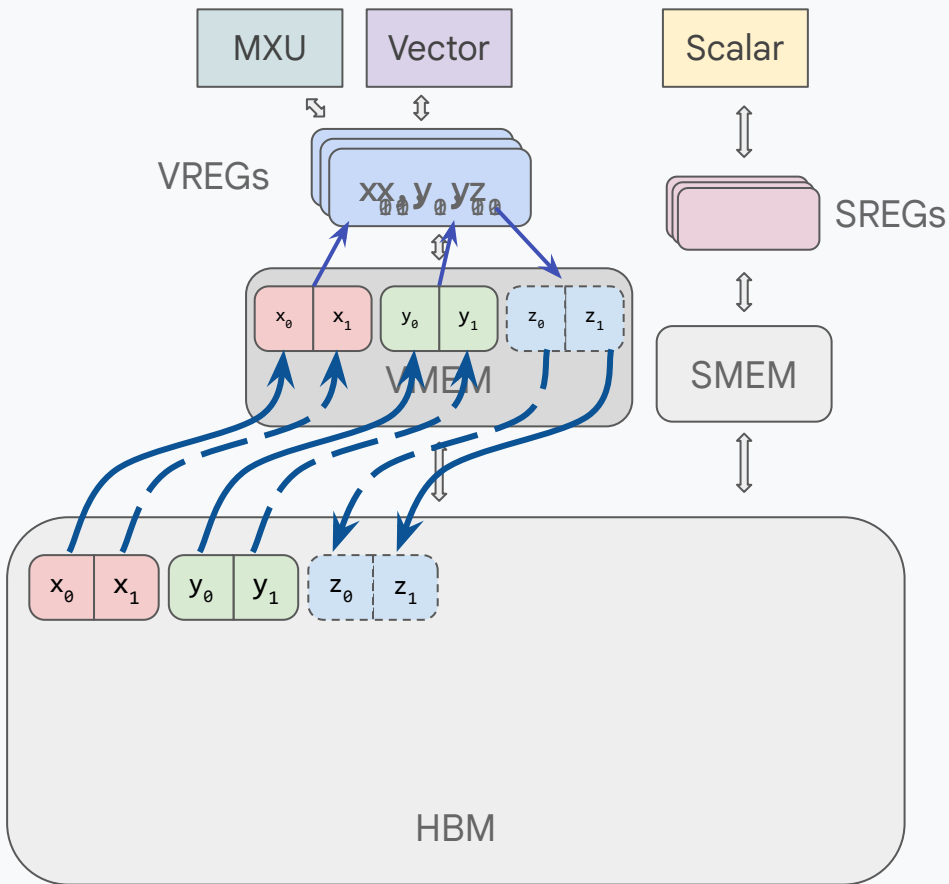
- Allocate output buffer for z
- Allocate VMEM scratch space
- Copy x, y from HBM into VMEM
- Load x, y from VMEM into VREGs
- Add x and y in MXU core
- Store z in VMEM
- Copy z from VMEM to HBM



# Pipelining

Idea: overlap loading/storing with compute  
by tiling inputs/outputs

- Allocate output buffer for  $z$
- Allocate VMEM scratch space
- Copy  $x_0, y_0$  from HBM into VMEM
- Start copying  $x_1, y_1$  from HBM into VMEM
- Load  $x_0, y_0$  into VREGs
- Add  $x_0$  and  $y_0$  using vector core
- Store  $z_0$  in VMEM
- Start copying  $z_0$  from VMEM into HBM
- Wait until  $x_1, y_1$  are done copying into VMEM
- Load  $x_1, y_1$  into VREGs
- Add  $x_1$  and  $y_1$  into using vector core
- Store  $z_1$  into VMEM
- Wait until  $z_0$  is done copying into HBM
- Copy  $z_1$  from VMEM into HBM



# How do expose a pipelining API to users?

Key idea: we can automatically pipeline user code if we can identify which copies we need to do ahead of time.

We specify a BlockSpec for each input/output.

- `block_shape`: what shapes to carve up the array
- `index_map`: which block a particular instance of the kernel read from/writes to

# BlockSpec example: matrix multiplication

```
def matmul(x, y, *, bm, bn, bk):
    m, n, k = x.shape[0], y.shape[1], x.shape[1]

    def matmul_kernel(x_ref, y_ref, o_ref):
        # x_ref, y_ref, o_refs now refer to blocks of the inputs/outputs
        o_ref[:, :] = jnp.dot(x_ref[:, :], y_ref[:, :])

    z = pl.pallas_call(
        matmul_kernel,
        out_shape=jax.ShapeDtypeStruct((m, n), jnp.float32),
        in_specs=[
            pl.BlockSpec(lambda i, _: (i, 0), (bm, x.shape[1])),
            pl.BlockSpec(lambda _, j: (0, j), (y.shape[0], bn)),
        ],
        out_specs=pl.BlockSpec(lambda i, j: (i, j), (bm, bn)),
        grid=(jt.cdiv(m, bm), jt.cdiv(n, bn)))(x, y)

    return z
```

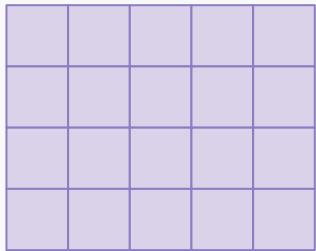
# BlockSpec example: matrix multiplication

```
def matmul(x, y, *, bm, bn, bk):  
    m, n, k = x.shape[0], y.shape[1], x.shape[1]  
  
    def matmul_kernel(x_ref, y_ref, o_ref):  
        # x_ref, y_ref, o_refs now refer to blocks of the inputs/outputs  
        o_ref[:, :] = jnp.dot(x_ref[:, :], y_ref[:, :])  
  
    z = pl.pallas_call(  
        matmul_kernel,  
        out_shape=jax.ShapeDtypeStruct((m, n), jnp.float32),  
        in_specs=[  
            pl.BlockSpec(lambda i, _: (i, 0), (bm, x.shape[1])),  
            pl.BlockSpec(lambda _, j: (0, j), (y.shape[0], bn)),  
        ],  
        out_specs=pl.BlockSpec(lambda i, j: (i, j), (bm, bn)),  
        grid=(jt.cdiv(m, bm), jt.cdiv(n, bn)))(x, y)  
  
    return z
```



# BlockSpec example: matrix multiplication

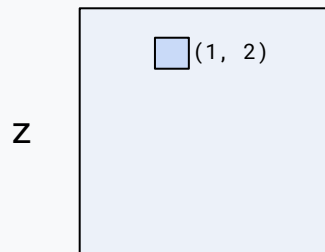
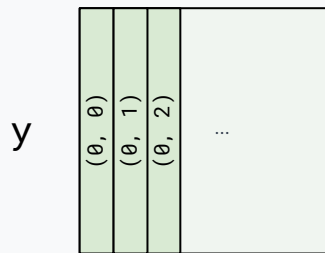
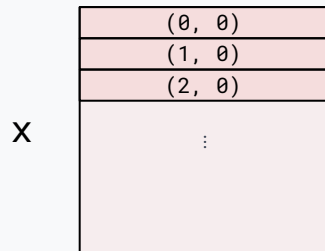
```
def matmul(x, y, *, bm, bn, bk):  
    m, n, k = x.shape[0], y.shape[1], x.shape[1]  
  
    def matmul_kernel(x_ref, y_ref, o_ref):  
        # x_ref, y_ref, o_refs now refer to blocks of the inputs/outputs  
        o_ref[:, :] = jnp.dot(x_ref[:, :], y_ref[:, :])  
  
    z = pl.pallas_call(  
        matmul_kernel,  
        out_shape=jax.ShapeDtypeStruct((m, n), jnp.float32),  
        in_specs=[  
            pl.BlockSpec(lambda i, _: (i, 0), (bm, x.shape[1])),  
            pl.BlockSpec(lambda _, j: (0, j), (y.shape[0], bn)),  
        ],  
        out_specs=pl.BlockSpec(lambda i, j: (i, j), (bm, bn)),  
        grid=(jt.cdiv(m, bm), jt.cdiv(n, bn)))(x, y)  
    return z
```



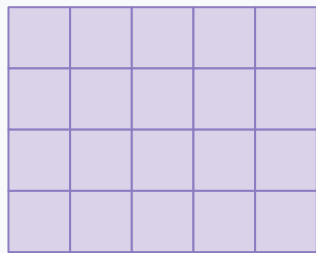
program instances

The **grid** argument determines the shape of the grid of programs.

# BlockSpec example: matrix multiplication



data

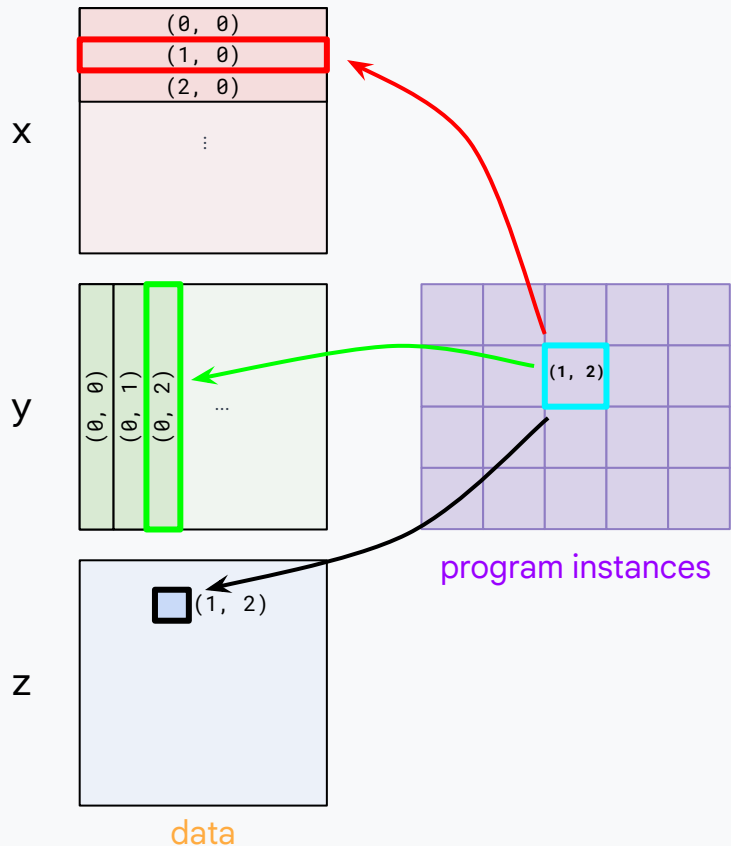


program instances

```
def matmul(x, y, *, bm, bn, bk):  
    m, n, k = x.shape[0], y.shape[1], x.shape[1]  
  
    def matmul_kernel(x_ref, y_ref, o_ref):  
        # x_ref, y_ref, o_refs now refer to blocks of the inputs/outputs  
        o_ref[:, :] = jnp.dot(x_ref[:, :], y_ref[:, :])  
  
    z = pl.pallas_call(  
        matmul_kernel,  
        out_shape=jax.ShapeDtypeStruct((m, n), jnp.float32),  
        in_specs=[  
            pl.BlockSpec(lambda i, _: (i, 0), (bm, x.shape[1])),  
            pl.BlockSpec(lambda _, j: (0, j), (y.shape[0], bn)),  
        ],  
        out_specs=pl.BlockSpec(lambda i, j: (i, j), (bm, bn)),  
        grid=(jt.cdiv(m, bm), jt.cdiv(n, bn)))(x, y)  
  
    return z
```

Each **block\_shape** argument to a **BlockSpec** determines how corresponding inputs are carved into blocks.

# BlockSpec example: matrix multiplication



```
def matmul(x, y, *, bm, bn, bk):  
    m, n, k = x.shape[0], y.shape[1], x.shape[1]  
  
    def matmul_kernel(x_ref, y_ref, o_ref):  
        # x_ref, y_ref, o_refs now refer to blocks of the inputs/outputs  
        o_ref[:, :] = jnp.dot(x_ref[:, :], y_ref[:, :])  
  
    z = pl.pallas_call(  
        matmul_kernel,  
        out_shape=jax.ShapeDtypeStruct((m, n), jnp.float32),  
        in_specs=[  
            pl.BlockSpec(lambda i, _: (i, 0), (bm, x.shape[1])),  
            pl.BlockSpec(lambda _, j: (0, j), (y.shape[0], bn)),  
        ],  
        out_specs=pl.BlockSpec(lambda i, j: (i, j), (bm, bn)),  
        grid=(jt.cdiv(m, bm), jt.cdiv(n, bn)))(x, y)  
  
    return z
```

Each **input\_map** argument to a BlockSpec maps program grid coordinates to a block of the corresponding input or output array.

## Aside: you might have seen this

- SPMD partitioning
- `linalg.generic`
- ...

# Pallas on TPU: Matmul

Kernel is written assuming inputs and outputs are in VMEM

```
def matmul_kernel(x_ref, y_ref, z_ref, acc_ref):  
    @pl.when(pl.program_id(2) == 0)  
    def _():  
        acc_ref[...] = jnp.zeros_like(acc_ref)  
  
    acc_ref[...] += jnp.dot(x_ref[...], y_ref[...])  
  
    @pl.when(pl.program_id(2) == pl.num_programs(2) - 1)  
    def _():  
        z_ref[...] = acc_ref[...].astype(z_ref.dtype)
```

Run kernel in a loop (specified by grid),  
where HBM <-> VMEM transfers are  
pipelined

```
@functools.partial(jax.jit, static_argnames=['bm', 'bk', 'bn'])  
def matmul(  
    x: jax.Array,  
    y: jax.Array,  
    *,  
    bm: int = 128,  
    bk: int = 128,  
    bn: int = 128,  
):  
    m, k = x.shape  
    _, n = y.shape  
    return pl.pallas_call(  
        matmul_kernel,  
        grid_spec=pltpu.PrefetchScalarGridSpec(  
            num_scalar_prefetch=0,  
            in_specs=[  
                pl.BlockSpec(lambda i, j, k: (i, k), (bm, bk)),  
                pl.BlockSpec(lambda i, j, k: (k, j), (bk, bn)),  
            ],  
            out_specs=pl.BlockSpec(lambda i, j, k: (i, j), (bm, bn)),  
            scratch_shapes=[pltpu.VMEM((bm, bn), jnp.float32)],  
            grid=(m // bm, n // bn, k // bk),  
        ),  
        out_shape=jax.ShapeDtypeStruct((m, n), x.dtype),  
    )(x, y)
```

BlockSpecs chunk up the inputs and  
outputs according to the loop iteration

# Pallas Benefits

- `jax.numpy` support
- Support for JAX-style transforms
  - `vmap`
  - `jvp`
- Tracing-based embedding (easy metaprogramming!)
- Emulation via HLO
  - Develop/debug on CPU
  - All standard JAX debugging tools work



# Dynamic Computation

# Scalar Prefetch

pallas\_call kicks off a HBM <-> VMEM pipeline.

We can determine the memory access pattern of this pipeline by prefetching runtime values into SMEM, a.k.a. “scalar prefetch”.

num\_scalar\_prefetch specifies how many leading arguments to the kernel to prefetch into SMEM

Prefetched values are provided as SMEM  
Refs to BlockSpec functions and can be  
used to select blocks

```
def body(_, x_ref, o_ref):
    o_ref[...] = x_ref[...]

s = jnp.array([4, 3, 2, 5, 3, 5, 2, 7], jnp.int32)
x = jnp.arange(8 * 8 * 128, dtype=jnp.int32).reshape((8 * 8, 128))

def _x_transform(i, s_ref):
    return s_ref[i], 0

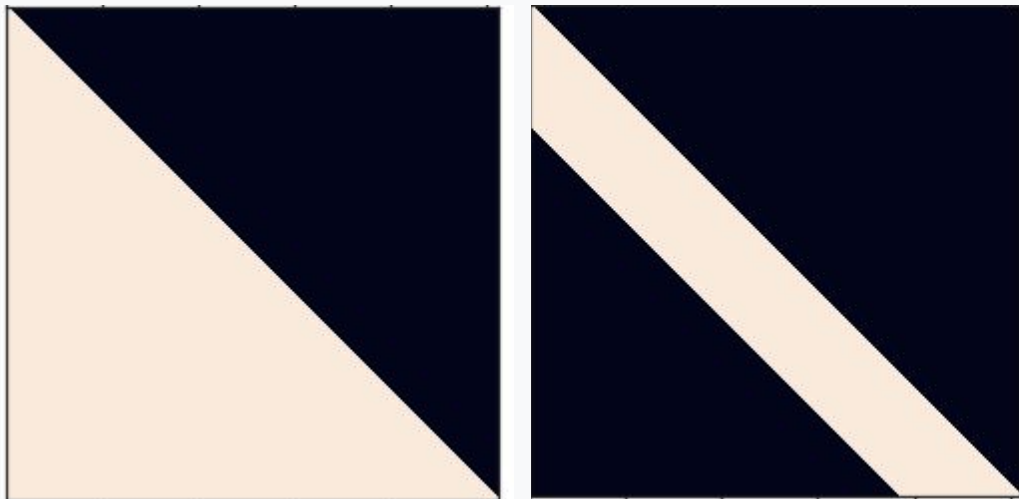
out = pl.pallas_call(
    body,
    out_shape=jax.ShapeDtypeStruct(x.shape, jnp.int32),
    grid_spec=pltpu.PrefetchScalarGridSpec(
        num_scalar_prefetch=1,
        in_specs=[
            pl.BlockSpec(_x_transform, (x.shape[0] // 8, x.shape[1])),
        ],
        out_specs=pl.BlockSpec(lambda i, _: (i, 0),
                                (x.shape[0] // 8, x.shape[1])),
        grid=8,
    ),
)(s, x)
```



# Case study: Splash attention

Splash (“sparse flash”) attention is a kernel that performs memory-efficient flash attention when provided a mask.

Preprocess mask into scalar prefetch metadata

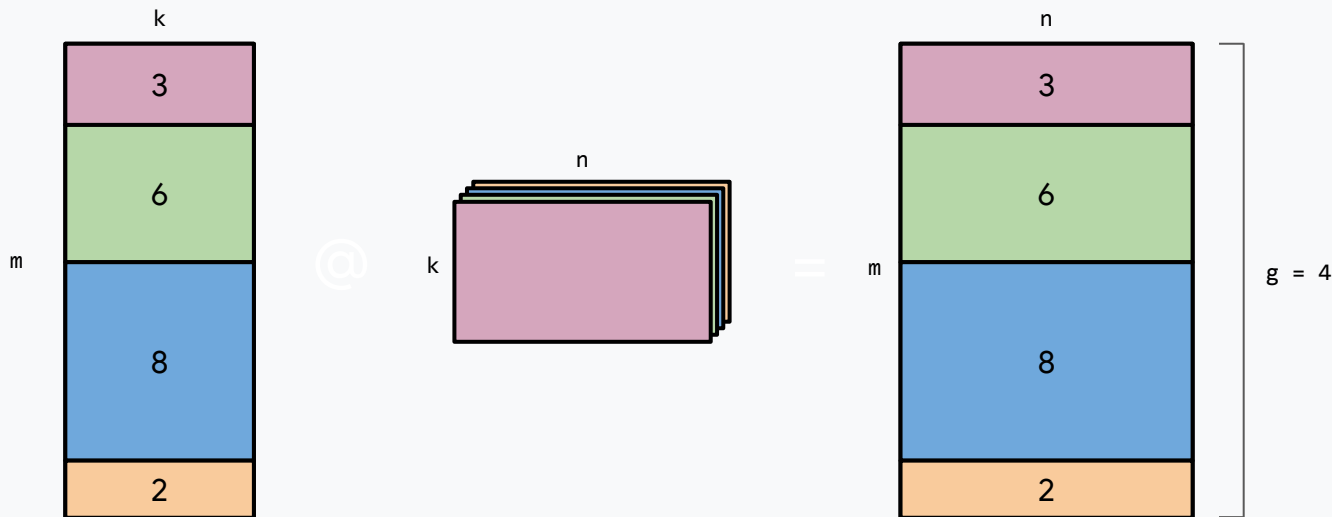


block\_mask  
data\_next  
mask\_next  
partial\_mask

# Case study: Grouped matmul

We have a matmul where

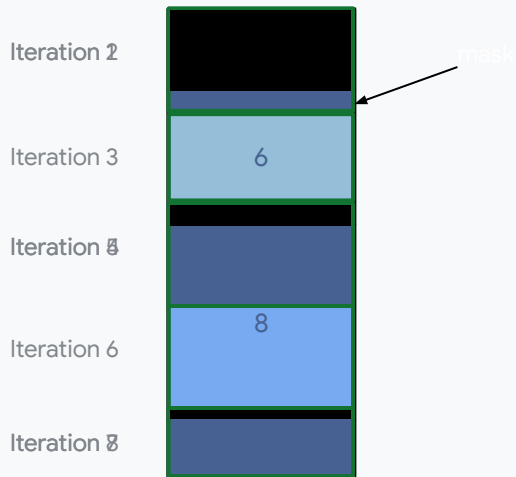
- $[m, k]$  LHS
- $[g, k, n]$  RHS
- $[g]$  group\_sizes, which determines how many elements on the  $m$  dimension are assigned to each group.



# Case study: Grouped matmul

Key Idea: Treat it like a matmul\*

\*Except tile up the m dimension and revisit tiles located on group boundaries + mask the computation appropriately



```
# Create the metadata we need for computation.
group_ids, m_tile_ids, tiles_m = make_group_metadata(
    group_sizes=group_sizes,
    m=m,
    tm=tm,
)
# group_ids[i]: which group are we processing on the i-th
# iteration
# m_tiles_id[i]: which tile of m to load on the i-th iteration
# tiles_m: how many iterations

# group_ids and m_tiles_id are passed in via scalar prefetch

def lhs_transform_indices(n_i, grid_id, k_i, group_metadata):
    group_ids, m_tile_ids = group_metadata
    return m_tile_ids[grid_id], k_i

def rhs_transform_indices(n_i, grid_id, k_i, group_metadata, group_offset):
    # rhs is (num_groups, k, n). Load the [tk, tn] matrix based on the group
    # id for this m-tile.
    group_ids, m_tile_ids = group_metadata
    return group_ids[grid_id], k_i, n_i
```

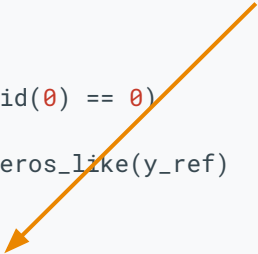
# Dynamic Grid

Grids have been static up to this point. With the dynamic grid feature, grids can now be determined at runtime.

```
def kernel(y_ref):
    @pl.when(pl.program_id(0) == 0)
    def _init():
        y_ref[...] = jnp.zeros_like(y_ref)
        y_ref[...] += 1

@jax.jit
def dynamic_kernel(steps):
    return self.pallas_call(
        kernel,
        grid=(steps * 2,),
        out_specs=pl.BlockSpec(lambda i: (0, 0), shape),
        out_shape=jax.ShapeDtypeStruct(shape, jnp.float32),
    )()
```

steps is an input to this JITed computation and is therefore a value only known at runtime



grid can have dynamic values in it!



# Asynchronous Memory Copying

# APIs for async copying (DMAs)

```
def kernel(x_hbm_ref, y_ref):  
    def body(sem):  
        dma1 = pltpu.async_copy(  
            x_hbm_ref.at[0], y_ref.at[p1.ds(0, 8)], sem  
        )  
        dma2 = pltpu.async_copy(  
            x_hbm_ref.at[1], y_ref.at[p1.ds(8, 8)], sem  
        )  
        # do something here  
        dma1.wait()  
        dma2.wait()  
    pltpu.run_scoped(body, pltpu.SemaphoreType.DMA)
```

Kick off asynchronous  
memory copy between two  
Refs

Block until it's done

Stack allocate semaphore for  
synchronization

# Extensible pipeline emitter

```
def matmul_pipeline(x_ref, y_ref, z_ref):  
    @pl.when(pl.program_id(2) == 0)  
    def _():  
        z_ref[...] = jnp.zeros(z_ref.shape, jnp.float32)  
  
        z_ref[...] += x_ref[...] @ y_ref[...]
```

Kernel is passed  
in HBM Refs (not  
VMEM)

```
def matmul_hbm_kernel(x_hbm_ref, y_hbm_ref, z_hbm_ref):  
    pltpu.emit_pipeline(  
        matmul_pipeline,  
        grid=(4, 4, 4),  
        in_specs=[  
            pl.BlockSpec(lambda i, j, k: (i, k), (128, 128)),  
            pl.BlockSpec(lambda i, j, k: (k, j), (128, 128)),  
        ],  
        out_specs=pl.BlockSpec(lambda i, j, k: (i, j), (128, 128)),  
    )(x_hbm_ref, y_hbm_ref, z_hbm_ref)
```

Emit  
HBM-VMEM  
pipeline  
inside of  
outer kernel

# APIs for remote async copying (rDMAs)

```
def kernel(x_ref, y_ref):  
    def body(ready_sem, send_sem, recv_sem):
```

```
        my_id = lax.axis_index('x')  
        num_devices = lax.psum(1, 'x')  
        neighbor = lax.rem(my_id + 1, num_devices)
```

```
        pltpu.semaphore_signal(ready_sem, device_id=neighbor)  
        pltpu.semaphore_wait(ready_sem)
```

```
        send_done, recv_done = pltpu.async_remote_copy(  
            x_ref, y_ref, send_sem, recv_sem, device_id=neighbor  
        )  
        # do something here  
        send_done.wait()  
        recv_done.wait()
```

```
    pltpu.run_scoped(body,  
        pltpu.SemaphoreType.REGULAR,  
        pltpu.SemaphoreType.DMA,  
        pltpu.SemaphoreType.DMA)
```

Use JAX functions to  
compute information about  
our position in the mesh

Synchronize  
with neighbor

Kick off and  
wait for rDMA  
to finish



# Case study: collective matmuls

By combining rDMAs with the custom pipeline emitter, we can now write our own collective matmuls.



For latency-bound collective matmuls, we can even fuse the outer and inner pipelines together to avoid bubbles (by using callbacks in the custom pipeline emitter API).

Thank you!