Mosaic GPU

A DSL for Fast Hopper Kernels
in Python

Adam Paszke (& many others from JAX and Google!)

Motivation

- Pre-packaged libraries are never a complete solution
- Compilers take lots of work to adapt and develop!
- Personal opinion: C++ metaprogramming isn't great for kernels!
- We need a better playground for:
  - rapid development, and
  - careful manual implementations.
- Also: a new Pallas backend for Hopper and onward

Current design

- Integrated into JAX
- MLIR-based tracing DSL
- Small, hackable Python core: only ~3.5k LOC
- 1 warpgroup (≈1 SM) = 1 Mosaic thread
- Lots of warpgroup-level helpers (demo coming up!)
- But, low-level programming always accessible (though rarely needed)
    - Can literally stick thread-level code or inline PTX into a kernel

Single-block kernel

```
def kernel(ctx, l_gmem, r_gmem, o_gmem, _):
  l = mgpu.FragmentedArray.load_strided(l_gmem)
  r = mgpu.FragmentedArray.load_strided(r_gmem)
  (l + r).store_untiled(o_gmem)
```

Inputs/outputs become references as kernel args

```
io_shape = mgpu.ShapeDtypeStruct((128, 128), np.float16)

f = mgpu.as_gpu_kernel(
    kernel, (1, 1, 1), (128, 1, 1), (io_shape, io_shape), io_shape, ()
)
```

grid block inputs outputs

Multi-block kernels

```
def kernel(ctx, l_gmem, r_gmem, o_gmem, _):

  start_offset = arith.muli(gpu.block_id(gpu.Dimension.x), arith.constant(index, 64))
  slc = mgpu.DynamicSlice(start_offset, length=64)

  l = mgpu.FragmentedArray.load_strided(memref_slice(l_gmem, slc))
  r = mgpu.FragmentedArray.load_strided(memref_slice(r_gmem, slc))
  (l + r).store_untiled(memref_slice(o_gmem, slc))
```

Multidimensional references:
no pointer arithmetic required

```
io_shape = mgpu.ShapeDtypeStruct((128, 128), np.float16)

f = mgpu.as_gpu_kernel(
    kernel, (2, 1, 1), (128, 1, 1), (io_shape, io_shape), io_shape, ()

)
```

grid has 2 blocks now!

Async copy (TMA)

```
def kernel(ctx, l_gmem, r_gmem, o_gmem, allocs):
  (l_smem, r_smem, o_smem), barriers = allocs
  start_offset = arith.muli(gpu.block_id(gpu.Dimension.x), arith.constant(index, 64))
  slc = mgpu.DynamicSlice(start_offset, length=64)

  ctx.async_copy(src_ref=l_gmem, dst_ref=l_smem, barrier=barriers[0], gmem_slice=slc)
  ctx.async_copy(src_ref=r_gmem, dst_ref=r_smem, barrier=barriers[1], gmem_slice=slc)
  barriers[0].wait()
  barriers[1].wait()
  l = mgpu.FragmentedArray.load_strided(l_smem)
  r = mgpu.FragmentedArray.load_strided(r_smem)
  (l + r).store_untiled(o_smem)
  ctx.async_copy(src_ref=o_smem, dst_ref=o_gmem, gmem_slice=slc)
  ctx.await_async_copy(0)


io_shape = mgpu.ShapeDtypeStruct((128, 128), np.float16)
slc_shape = mgpu.ShapeDtypeStruct((64, 128), np.float16)
f = mgpu.as_gpu_kernel(
    kernel, (2, 1, 1), (128, 1, 1), (io_shape, io_shape), io_shape,
    ([slc_shape] * 3, mgpu.TMABarrier(num_barriers=2)),
)
```

SMEM arguments

SMEM requirements

TMA transforms

128x128 -> 2x2x64x64

```
def kernel(ctx, l_gmem, r_gmem, o_gmem, allocs):
  (l_smem, r_smem, o_smem), barriers = allocs
  start_offset = arith.muli(gpu.block_id(gpu.Dimension.x), arith.constant(index, 64))
  slc = mgpu.DynamicSlice(start_offset, length=64)
  transform = dict(swizzle=128, gmem_transform=mgpu.TileTransform((64, 64)))
  ctx.async_copy(src_ref=l_gmem, dst_ref=l_smem, barrier=barriers[0], gmem_slice=slc, **transform)
  ctx.async_copy(src_ref=r_gmem, dst_ref=r_smem, barrier=barriers[1], gmem_slice=slc, **transform)
  barriers[0].wait()
  barriers[1].wait()
  l = mgpu.FragmentedArray.load_strided(l_smem)
  r = mgpu.FragmentedArray.load_strided(r_smem)
  (l + r).store_untiled(o_smem)
  ctx.async_copy(src_ref=o_smem, dst_ref=o_gmem, gmem_slice=slc, **transform)
  ctx.await_async_copy(0)


io_shape = mgpu.ShapeDtypeStruct((128, 128), np.float16)
slc_shape = mgpu.ShapeDtypeStruct(mgpu.tile_shape((64, 128), (64, 64)), np.float16)
f = mgpu.as_gpu_kernel(
    kernel, (2, 1, 1), (128, 1, 1), (io_shape, io_shape), io_shape,
    ([slc_shape] * 3, mgpu.TMABarrier(num_barriers=2)),
)
```

WGMMA

```
def kernel(ctx, l_gmem, r_gmem, o_gmem, allocs):
  (l_smem, r_smem, o_smem), barriers = allocs
  start_offset = arith.muli(gpu.block_id(gpu.Dimension.x), arith.constant(index, 64))
  slc = mgpu.DynamicSlice(start_offset, length=64)
  transform = dict(swizzle=128, gmem_transform=mgpu.TileTransform((64, 64)))
  ctx.async_copy(src_ref=l_gmem, dst_ref=l_smem, barrier=barriers[0], gmem_slice=slc, **transform)
  ctx.async_copy(src_ref=r_gmem, dst_ref=r_smem, barrier=barriers[1], **transform)
  barriers[0].wait()
  barriers[1].wait()
  acc = mgpu.WGMMAAccumulator.zero(64, 128, ir.F16Type.get())
  acc = mgpu.wgmma(acc, l_smem, r_smem)
  nvvm.wgmma_commit_group_sync_aligned()
  nvvm.wgmma_wait_group_sync_aligned(0)
  acc.value.store_tiled(o_smem, swizzle=128)
  ctx.async_copy(src_ref=o_smem, dst_ref=o_gmem, gmem_slice=slc, **transform)
  ctx.await_async_copy(0)

…
f = mgpu.as_gpu_kernel(
    kernel, (2, 1, 1), (128, 1, 1), (l_shape, r_shape), o_shape,
    ([sl_shape, sro_shape, sro_shape], mgpu.TMABarrier(num_barriers=2)),
)
```

Block clusters + TMA multicast

```
def kernel(ctx, l_gmem, r_gmem, o_gmem, allocs):
  (l_smem, r_smem, o_smem), barriers = allocs
  start_offset = arith.muli(gpu.block_id(gpu.Dimension.x), arith.constant(index, 64))
  slc = mgpu.DynamicSlice(start_offset, length=64)
  transform = dict(swizzle=128, gmem_transform=mgpu.TileTransform((64, 64)))
  ctx.async_copy(src_ref=l_gmem, dst_ref=l_smem, barrier=barriers[0], gmem_slice=slc, **transform)
  ctx.async_copy(src_ref=r_gmem, dst_ref=r_smem, barrier=barriers[1], collective=gpu.Dimension.x, **transform)
  barriers[0].wait()
  barriers[1].wait()
  acc = mgpu.WGMMAAccumulator.zero(64, 128, ir.F16Type.get())
  acc = mgpu.wgmma(acc, l_smem, r_smem)
  nvvm.wgmma_commit_group_sync_aligned()
  nvvm.wgmma_wait_group_sync_aligned(0)
  acc.value.store_tiled(o_smem, swizzle=128)
  ctx.async_copy(src_ref=o_smem, dst_ref=o_gmem, gmem_slice=slc, **transform)
  ctx.await_async_copy(0)

…
f = mgpu.as_gpu_kernel(
    kernel, (2, 1, 1), (128, 1, 1), (l_shape, r_shape), o_shape,
    ([sl_shape, sro_shape, sro_shape], mgpu.TMABarrier(num_barriers=2)),
    cluster=(2, 1, 1),
)
```

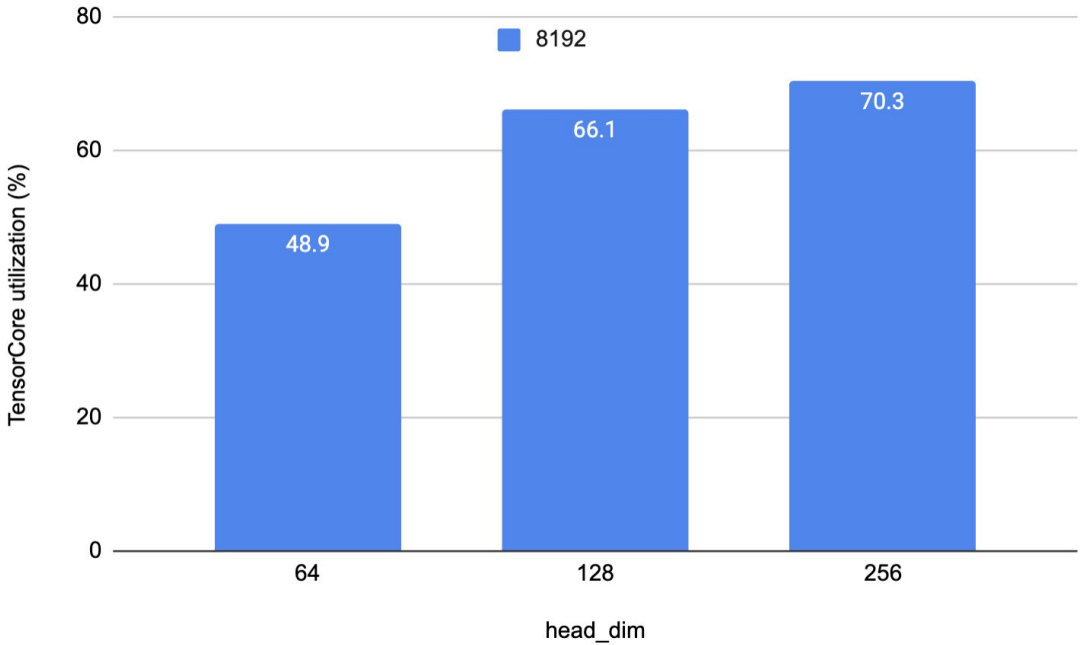
We've used most Hopper features!

```
def kernel(ctx, l_gmem, r_gmem, o_gmem, allocs):
  (l_smem, r_smem, o_smem), barriers = allocs
  start_offset = arith.muli(gpu.block_id(gpu.Dimension.x), arith.constant(index, 64))
  slc = mgpu.DynamicSlice(start_offset, length=64)
  transform = dict(swizzle=128, gmem_transform=mgpu.TileTransform((64, 64)))
  ctx.async_copy(src_ref=l_gmem, dst_ref=l_smem, barrier=barriers[0], gmem_slice=slc, **transform)
  ctx.async_copy(src_ref=r_gmem, dst_ref=r_smem, barrier=barriers[1], collective=gpu.Dimension.x, **transform)
  barriers[0].wait()
  barriers[1].wait()
  acc = mgpu.WGMMAAccumulator.zero(64, 128, ir.F16Type.get())
  acc = mgpu.wgmma(acc, l_smem, r_smem)
  nvvm.wgmma_commit_group_sync_aligned()
  nvvm.wgmma_wait_group_sync_aligned(0)
  acc.value.store_tiled(o_smem, swizzle=128)
  ctx.async_copy(src_ref=o_smem, dst_ref=o_gmem, gmem_slice=slc, **transform)
  ctx.await_async_copy(0)

…
f = mgpu.as_gpu_kernel(
    kernel, (2, 1, 1), (128, 1, 1), (l_shape, r_shape), o_shape,
    ([sl_shape, sro_shape, sro_shape], mgpu.TMABarrier(num_barriers=2)),
    cluster=(2, 1, 1),
)
```

TMA
WGMMA
Block clusters
TMA multicast

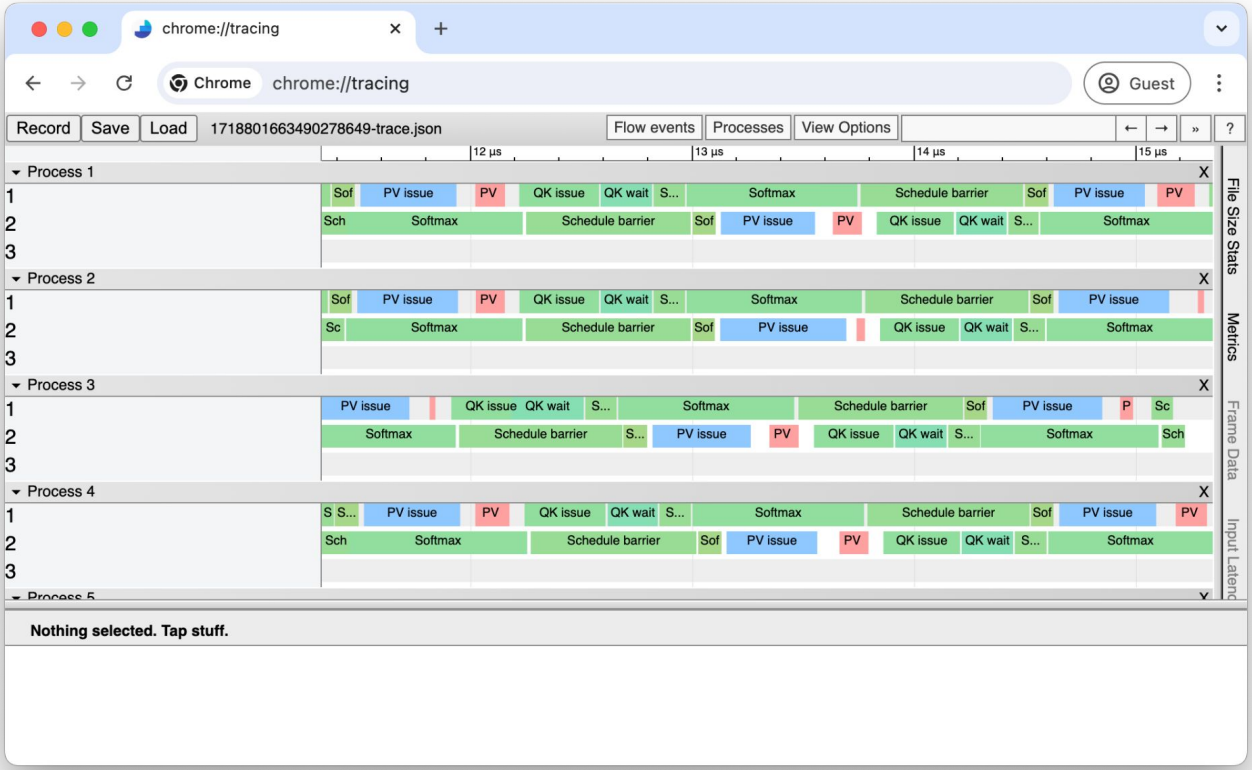
Case study: FlashAttention 3



batch size = 2, dim = 2048, sequence length = 8192, forward only

https://github.com/google/jax/blob/main/jax/experimental/mosaic/gpu/examples/flash_attention.py#L146-L350

Warpgroup-level profiling



How do I get it?

Comes by default with JAX GPU (`jax.experimental.mosaic.gpu`):

```
pip install -U jax[cuda12]
pip install -U jax[cuda12] --pre -f https://storage.googleapis.com/jax-releases/jax_nightly_releases.html
```

Can I use it with PyTorch?

Absolutely!

`mgpu.as_gpu_kernel`

`mgpu.as_torch_gpu_kernel`

Takeaways

- Hopper kernels can be reasonably concise
- You can quickly prototype them in Python now
- … but they remain *somewhat low-level*
- Next step: Pallas integration
  - `a * 4` is a lot nicer than `arith.muli(a, arith.constant(a.type, 4))`
  - Nicer abstractions for common patterns (synchronization, pipelining, …)

⚠️ Experimental software ⚠️

Thank you!

apaszke@google.com