JAX in Astro

Dan F-M & Skye W-M
2024-10-15

what is **JAX** anyways?*

*we know that you know what JAX is

JAX is a **platform** for **numerical computing**

JAX is 2* compilers in a trench coat

*or 3?

JAX is **modular** & has a thriving **ecosystem**

some random things about **JAX** that we **like**
and think you **might not know about** ...

```
debugging
```

```python
jax.debug.print("x = {}", x)

jax.config.update("jax_debug_nans", True)

with jax.log_compiles():
  ...

jax.jit(fun).lower( ... ).as_text()
jax.jit(fun).lower( ... ).compile().as_text()
```

2 **escape hatches**

```
jax.pure_callback

jax.experimental.pallas.pallas_call

jax.extend.ffi.ffi_call

jax.custom_jvp / jax.custom_vjp
```

3 parallelism

```python
import jax
import jax.numpy as jnp
import jax.sharding as shd

# Running on a TPU v5p 2x2. This assigns names to the two physical axes of the hardware.
mesh = jax.make_mesh(axis_shapes=(2, 2), axis_names=('x', 'y'))
def P(*mesh_axis_names):
  return shd.NamedSharding(mesh, shd.PartitionSpec(*mesh_axis_names))

# We shard W over the contracting dimension and x over batch and the hidden dim
W_sharding, x_sharding = P(None, 'y'), P('x', 'y')

# We create a matrix W and input activations x with this sharding.
W = jnp.zeros((8192, 2048), dtype=jnp.bfloat16, device=W_sharding)
x = jnp.zeros((8, 2048), dtype=jnp.bfloat16, device=x_sharding)

def matmul(W, x):
  return jnp.einsum('fd,bd->bf', W, x)

# We can explicitly compile the sharded matmul function here. This adds all the
# necessary comms (e.g. an AllReduce after the matmul).
jit_matmul = jax.jit(matmul, out_shardings=P('y', None)).lower(W, x).compile()

out = jit_matmul(W, x)
```

```
module @jit_matmul attributes {mhlo.num_partitions = 4 : i32, mhlo.num_replicas = 1 : i32} {
  func.func public @main(%arg0: tensor<8192x2048xbf16> {
      mhlo.layout_mode = "default",
      mhlo.sharding = "{devices=[1,2,2]<=[2,2]T(1,0) last_tile_dim_replicate}"},
    %arg1: tensor<8x2048xbf16> {
        mhlo.layout_mode = "default",
        mhlo.sharding = "{devices=[2,2]<=[4]}"})
    -> (tensor<8x8192xbf16> {
        jax.result_info = "",
        mhlo.layout_mode = "default",
        mhlo.sharding = "{devices=[2,1,2]<=[2,2]T(1,0) last_tile_dim_replicate}"}) {
    %0 = stablehlo.dot_general %arg1, %arg0, contracting_dims = [1] x [1],
             precision = [DEFAULT, DEFAULT] : (tensor<8x2048xbf16>, tensor<8192x2048xbf16>) -> tensor<8x8192xbf16>
    return %0 : tensor<8x8192xbf16>
  }
}
```

```
...

ENTRY %main.6_spmd (param.1.0: bf16[8192,1024], param.2: bf16[8,1024]) -> bf16[4,8192] {
  %param.1.0 = bf16[8192,1024]{1,0} parameter(0), sharding={devices=[1,2]<=[2]}, metadata={op_name="W"}
  %param.2 = bf16[8,1024]{1,0} parameter(1), sharding={devices=[1,2]<=[2]}, metadata={op_name="x"}
  %wrapped_convert = f32[8,1024]{1,0} fusion(bf16[8,1024]{1,0} %param.2), kind=kLoop, calls=%wrapped_convert_computation
  %wrapped_convert.1 = f32[8192,1024]{1,0} fusion(bf16[8192,1024]{1,0} %param.1.0), kind=kLoop,
calls=%wrapped_convert_computation.1
  %custom-call.1.0 = (f32[8,8192]{1,0}, s8[4194304]{0}) custom-call(f32[8,1024]{1,0} %wrapped_convert,
f32[8192,1024]{1,0} %wrapped_convert.1), custom_call_target="__cublas$gemm",
metadata={op_name="jit(matmul)/jit(main)/fd,bd->bf/dot_general" source_file="/home/danfm/demo/multi.py" source_line=18},
backend_config={"operation_queue_id":"0","wait_on_operation_queues":[],"gemm_backend_config":{"alpha_real":1,"alpha_imag"
:0,"beta":0,"dot_dimension_numbers":{"lhs_contracting_dimensions":["1"],"rhs_contracting_dimensions":["1"],"lhs_batch_dim
ensions":[],"rhs_batch_dimensions":[]},"precision_config":{"operand_precision":["DEFAULT","DEFAULT"],"algorithm":"ALG_UNS
ET"},"epilogue":"DEFAULT","damax_output":false,"selected_algorithm":"-1","lhs_stride":"8192","rhs_stride":"8388608","grad
_x":false,"grad_y":false},"force_earliest_schedule":false}
  %get-tuple-element.1 = f32[8,8192]{1,0} get-tuple-element((f32[8,8192]{1,0}, s8[4194304]{0}) %custom-call.1.0),
index=0, metadata={op_name="jit(matmul)/jit(main)/fd,bd->bf/dot_general" source_file="/home/danfm/demo/multi.py"
source_line=18}
  %wrapped_convert.2 = bf16[8,8192]{1,0} fusion(f32[8,8192]{1,0} %get-tuple-element.1), kind=kLoop,
calls=%wrapped_convert_computation.2
  %reduce-scatter-start = ((bf16[8,8192]{1,0}), bf16[4,8192]{1,0}) reduce-scatter-start(bf16[8,8192]{1,0}
%wrapped_convert.2), channel_id=2, replica_groups=[1,2]<=[2], use_global_device_ids=true, dimensions={0},
to_apply=%add.clone,
backend_config={"operation_queue_id":"0","wait_on_operation_queues":[],"collective_backend_config":{"is_sync":true,"no_pa
rallel_custom_call":false},"force_earliest_schedule":false}
  ROOT %reduce-scatter-done = bf16[4,8192]{1,0} reduce-scatter-done(((bf16[8,8192]{1,0}), bf16[4,8192]{1,0})
%reduce-scatter-start)
}
```

multi-host parallelism

1. jax.distributed.initialize(…)
   a. no args needed for SLURM, Open MPI, Cloud TPU

2. run your Python code on every host!

https://jax.readthedocs.io/en/latest/multi_process.html

4 ecosystem

https://jax.readthedocs.io/en/latest/index.html

Ecosystem

JAX itself is narrowly-scoped and focuses on efficient array operations & program transformations. Built around JAX is an evolving ecosystem of machine learning and numerical computing tools; the following is just a small sample of what is out there:

Neural networks

- Flax
- NNX
- Equinox
- Keras

Optimizers & solvers

- Optax
- Optimistix
- Lineax
- Diffrax

Data loading

- Grain
- Tensorflow datasets
- Hugging Face datasets

Miscellaneous tools

- Orbax
- Chex

Probabilistic programming

- Blackjax
- Numpyro
- PyMC

Probabilistic modeling

- Tensorflow probabilty
- Distrax

Physics & simulation

- JAX MD
- Brax

LLMs

- MaxText
- AXLearn
- Levanter
- EasyLM

nnx

next-gen flax
https://flax.readthedocs.io

```python
from flax import nnx
import optax


class Model(nnx.Module):
  def __init__(self, din, dmid, dout, rngs: nnx.Rngs):
    self.linear = nnx.Linear(din, dmid, rngs=rngs)
    self.bn = nnx.BatchNorm(dmid, rngs=rngs)
    self.dropout = nnx.Dropout(0.2, rngs=rngs)
    self.linear_out = nnx.Linear(dmid, dout, rngs=rngs)

  def __call__(self, x):
    x = nnx.relu(self.dropout(self.bn(self.linear(x))))
    return self.linear_out(x)

model = Model(2, 64, 3, rngs=nnx.Rngs(0))  # eager initialization
optimizer = nnx.Optimizer(model, optax.adam(1e-3))  # reference sharing

@nnx.jit  # automatic state management for JAX transforms
def train_step(model, optimizer, x, y):
  def loss_fn(model):
    y_pred = model(x)  # call methods directly
    return ((y_pred - y) ** 2).mean()

  loss, grads = nnx.value_and_grad(loss_fn)(model)
  optimizer.update(grads)  # in-place updates

  return loss
```

jax-ai-stack

jax-ai-stack.github.com

- JAX: the core JAX package, which includes array operations and program transformations like jit, vmap, grad, etc.
- flax: build neural networks with JAX
- ml_dtypes: NumPy dtype extensions for machine learning.
- optax: gradient processing and optimization in JAX.
- orbax: checkpointing and persistence utilities for JAX.

jax longevity

- Google depends on it
  - Gemini, other research, TPUs
- modularity
- XLA contributions from hardware vendors
  - Nvidia, AMD, Amazon
- jax team obsessed with jax

discussion

What are some ways that JAX has improved your research/workflow/general happiness?

JAX pain points; let's get specific, not just "docs are bad"

How could JAX be better for you?