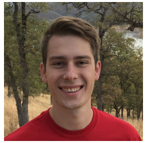
A Tour of **JAX**



Robert Dyro, Google
@rdyro

Outline

- JAX Key Ideas & How JAX works
- Getting Started with JAX & JAX Ecosystem
- JAX Advanced Techniques

Key Ideas

Familiar API
JAX provides a familiar NumPy-style API for ease of adoption by researchers and engineers

Transformations
JAX includes composable function transformations for compilation, batching, automatic differentiation, and parallelization

Run Anywhere
The same code executes on multiple backends, including CPU, GPU (NVIDIA & AMD), TPU & others!

The case for JAX

Fast Iteration

- Compiler-backed: fast execution without manual optimization
- Develop on CPU run on GPU/TPU

Research-ready & extensible

- Used for ML research at Google, Apple* and others and for scientific research (e.g., physics, astronomy)
- Manual control: custom kernels & ops

Small and Large Scale

- Low Latency Execution on a single CPU
- Easily scale to multi-device and multi-host computation

Google

* https://machinelearning.apple.com/research/introducing-apple-foundation-models

Key Ideas

1. Familiar API

2. Transformations

3. Run Anywhere

1. Familiar API

```python
import jax.numpy as jnp


def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)
```

```python
import numpy as jnp


def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)
```

2. **JAX** Transformations

CPU/GPU/TPU compilation

autodiff parallelization

```
import jax.numpy as jnp


def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)
```

Google

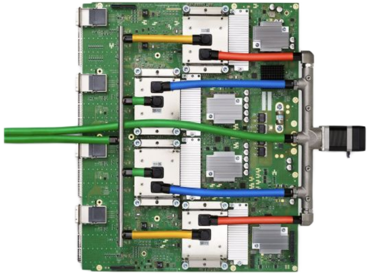
2. **JAX** Transformations

✓ CPU/GPU/TPU
compilation

autodiff
parallelization

```python
import jax.numpy as jnp


def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)
```


*



* Image: NVIDIA

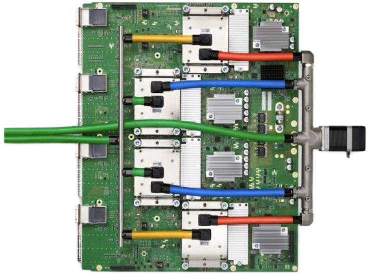
2. JAX Transformations

✅ CPU/GPU/TPU compilation

✅ autodiff parallelization

```python
import jax.numpy as jnp
from jax import grad

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)


gradient_fun = grad(loss)
```

*



* Image: NVIDIA

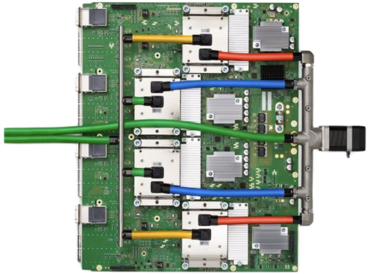
2. JAX Transformations

CPU/GPU/TPU compilation

autodiff parallelization

```
import jax.numpy as jnp
from jax import grad, vmap

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)


gradient_fun = grad(loss)
perexample_grads = vmap(grad(loss), in_axes=(None, 0))
```

*



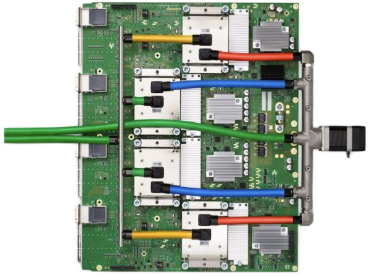
* Image: NVIDIA

2. **JAX** Transformations

✓ CPU/GPU/TPU ✓ autodiff
✓ compilation parallelization

```python
import jax.numpy as jnp
from jax import grad, vmap, jit

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)


gradient_fun = jit(grad(loss))
perexample_grads = jit(vmap(grad(loss), in_axes=(None, 0)))
```


*



Google

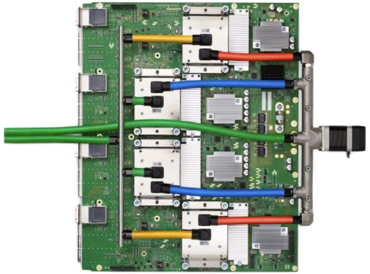
* Image: NVIDIA

2. JAX Transformations

✓ CPU/GPU/TPU ✓ autodiff
✓ compilation ✓ parallelization

```python
import jax.numpy as jnp
from jax import grad, vmap, jit

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)


gradient_fun = jit(grad(loss))
perexample_grads = jit(vmap(grad(loss), in_axes=(None, 0)),
                       in_shardings=..., out_shardings=...)
```

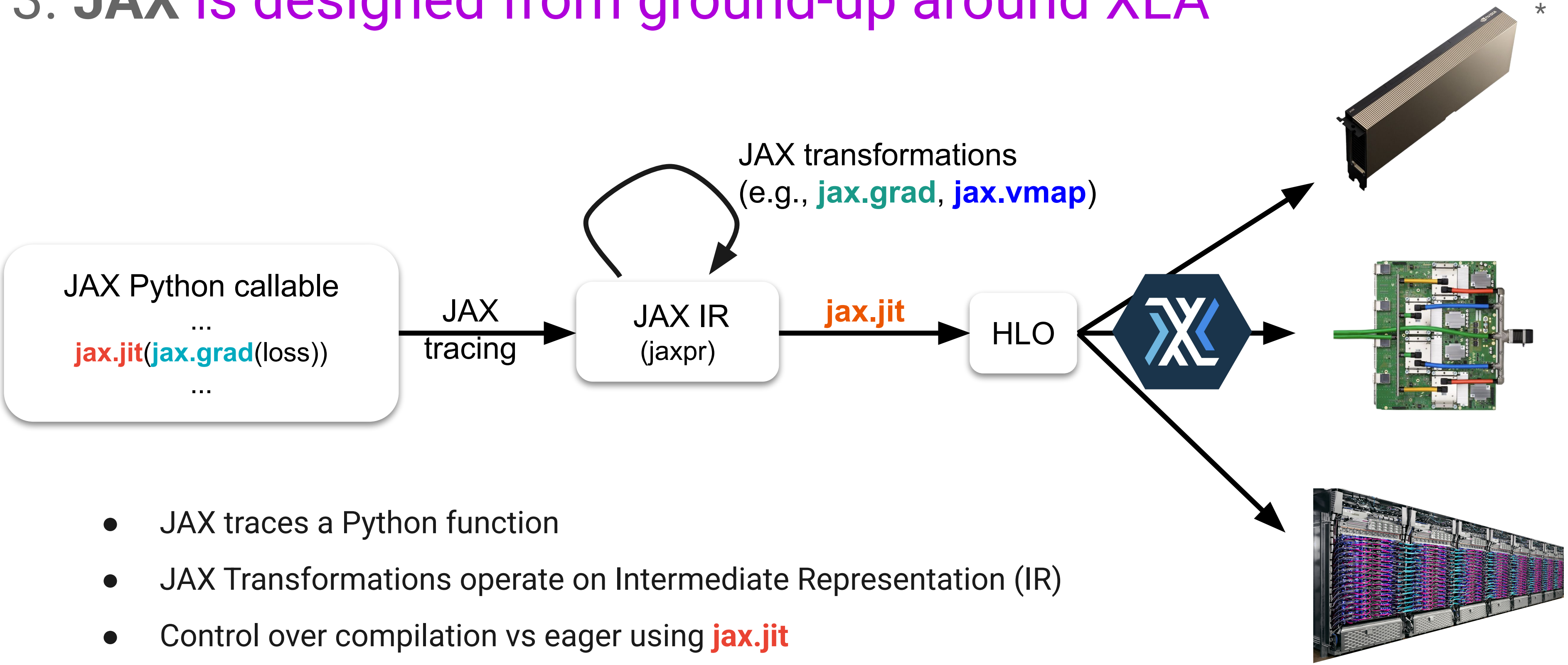

*



Google

* Image: NVIDIA

3. JAX is designed from ground-up around XLA



- JAX traces a Python function
- JAX Transformations operate on Intermediate Representation (IR)
- Control over compilation vs eager using jax.jit
- Same JAX code runs on CPU/GPU/TPU/pods

* Image: NVIDIA

How JAX Works: Python code → `jaxpr`

1. Use JAX primitives

```python
from jax.numpy import log
# from jax.lax import log

def f(x):
  return log(x) / log(2.)
```

2. Trace function

```python
x = jnp.ones([2])
jax.jit(f)(x)
```

```
ConcreteArray(
    val=[1., 1.],
    dtype=jnp.float32,
)
```

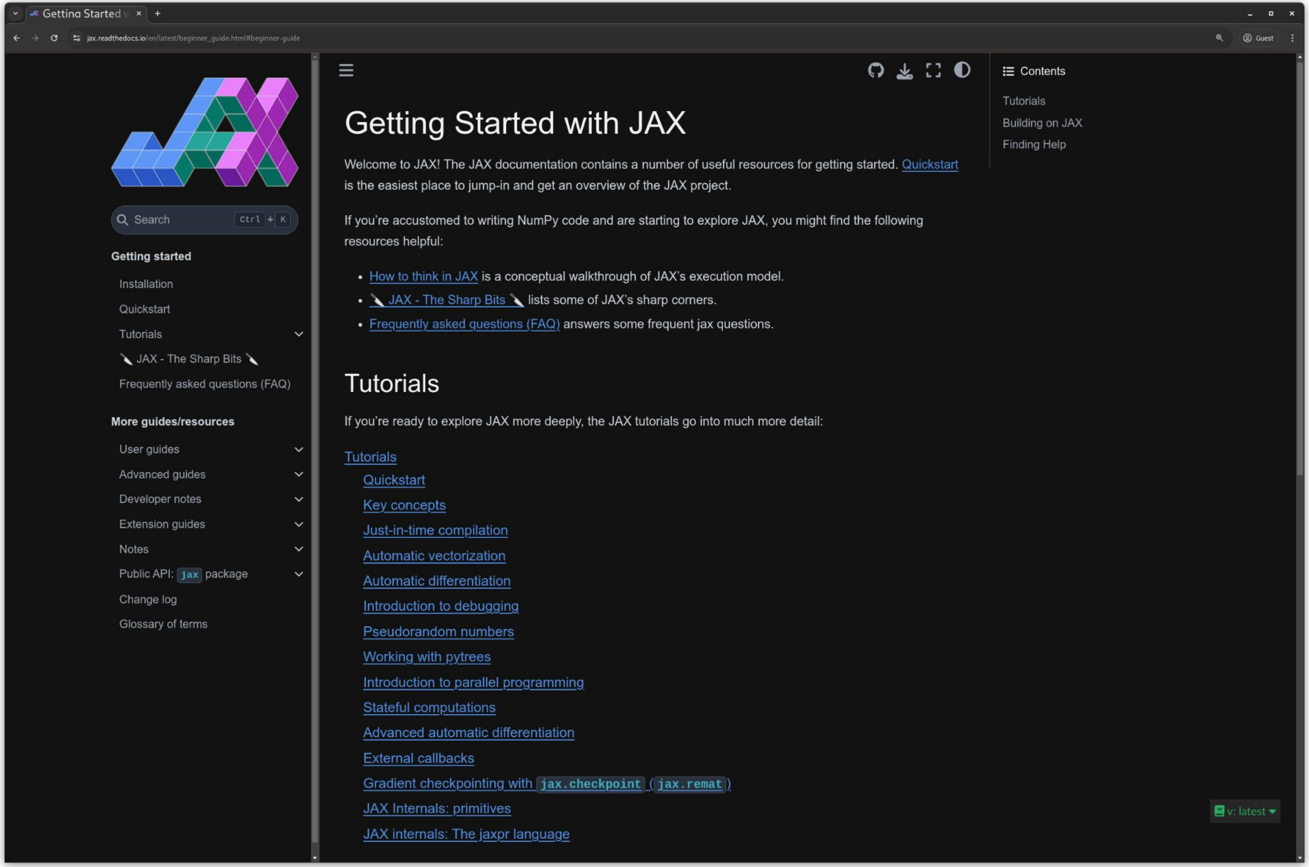
3. Record `jaxpr` `(JAX expression)`

```
{ lambda ; a:f32[]. let
   b:f32[] = log a
   c:f32[] = log 2.0
   d:f32[] = div b c
  in (d,) }
```

Outline

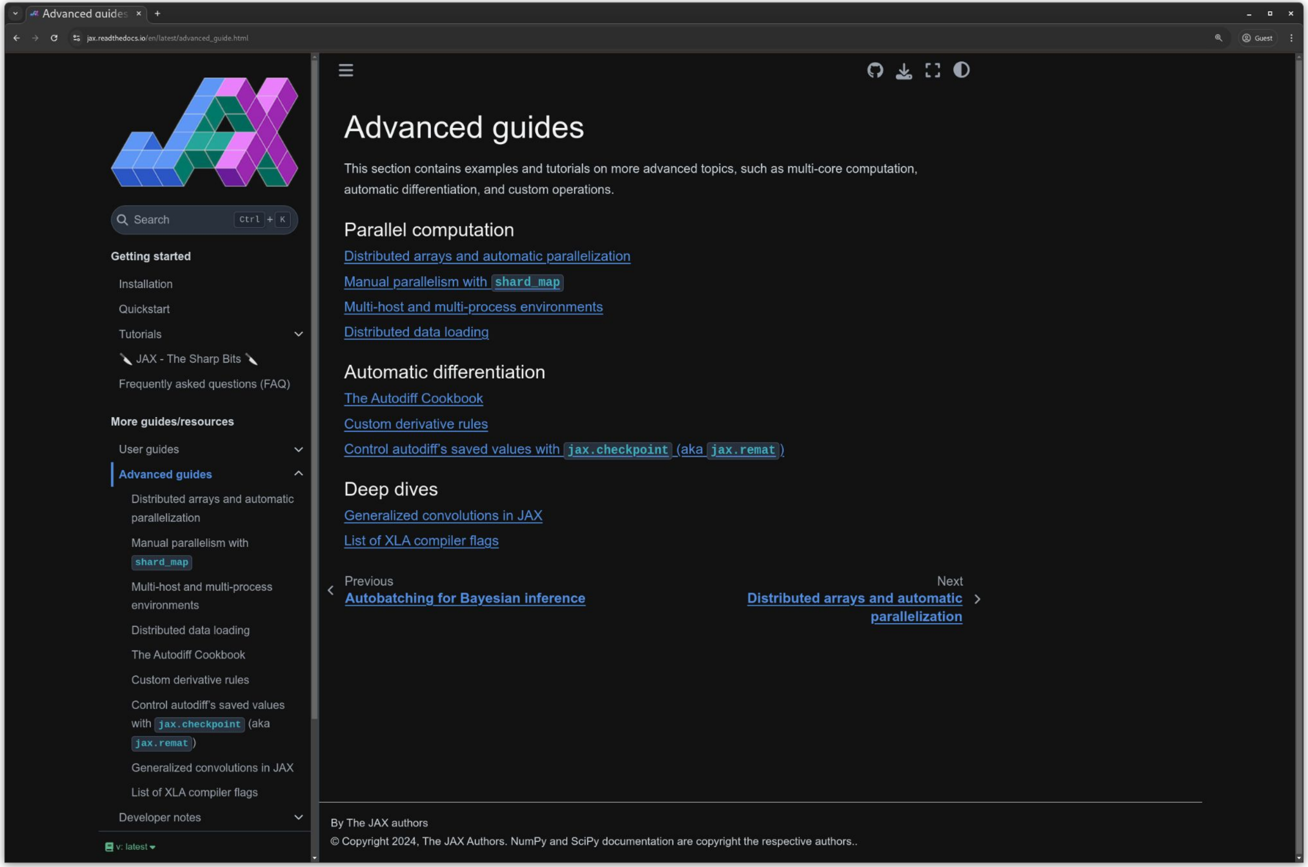
- JAX Key Ideas & How JAX works
- Getting Started with JAX & JAX Ecosystem
- JAX Advanced Techniques

Getting Started



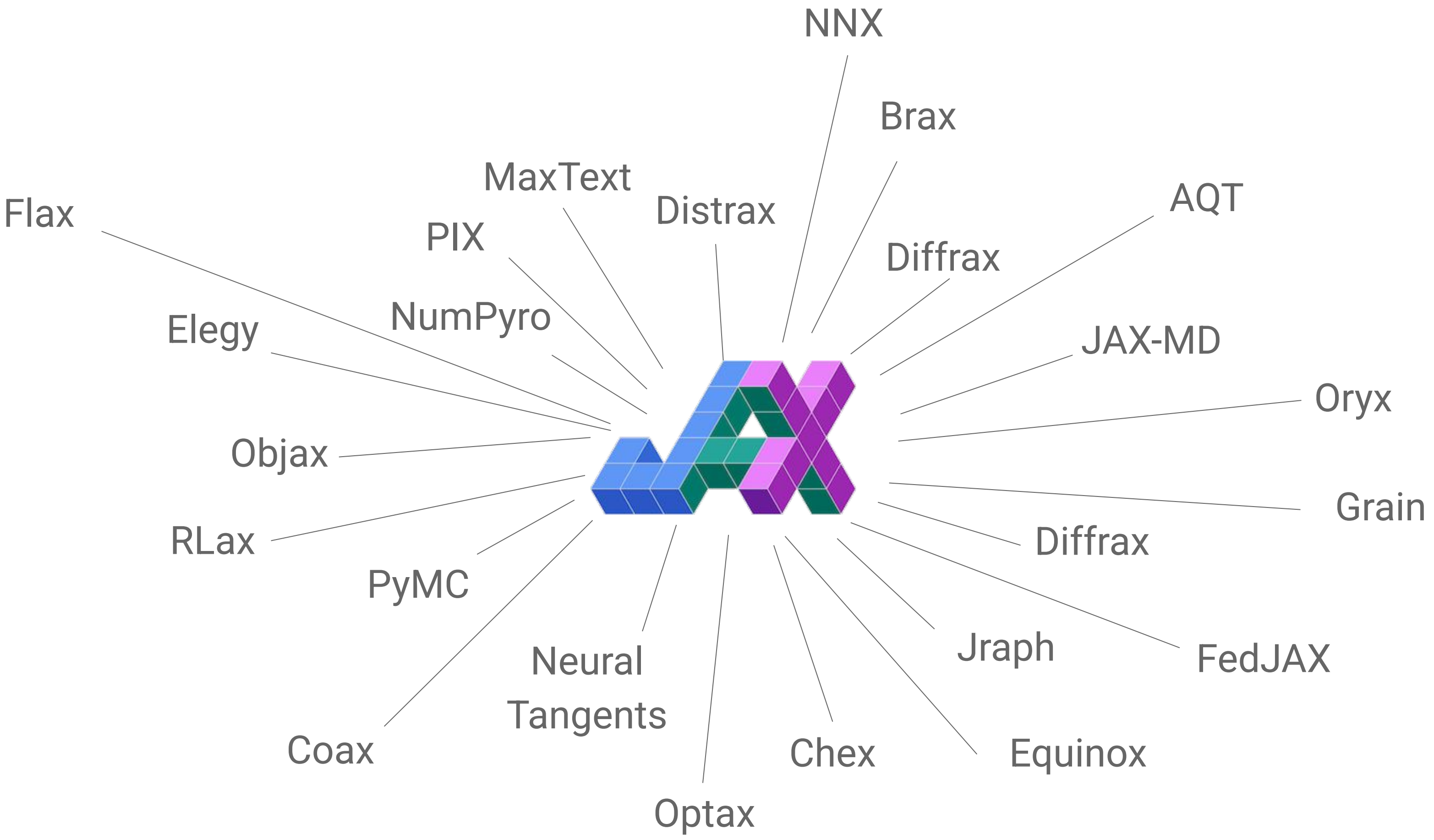
Google

Getting Started



Google

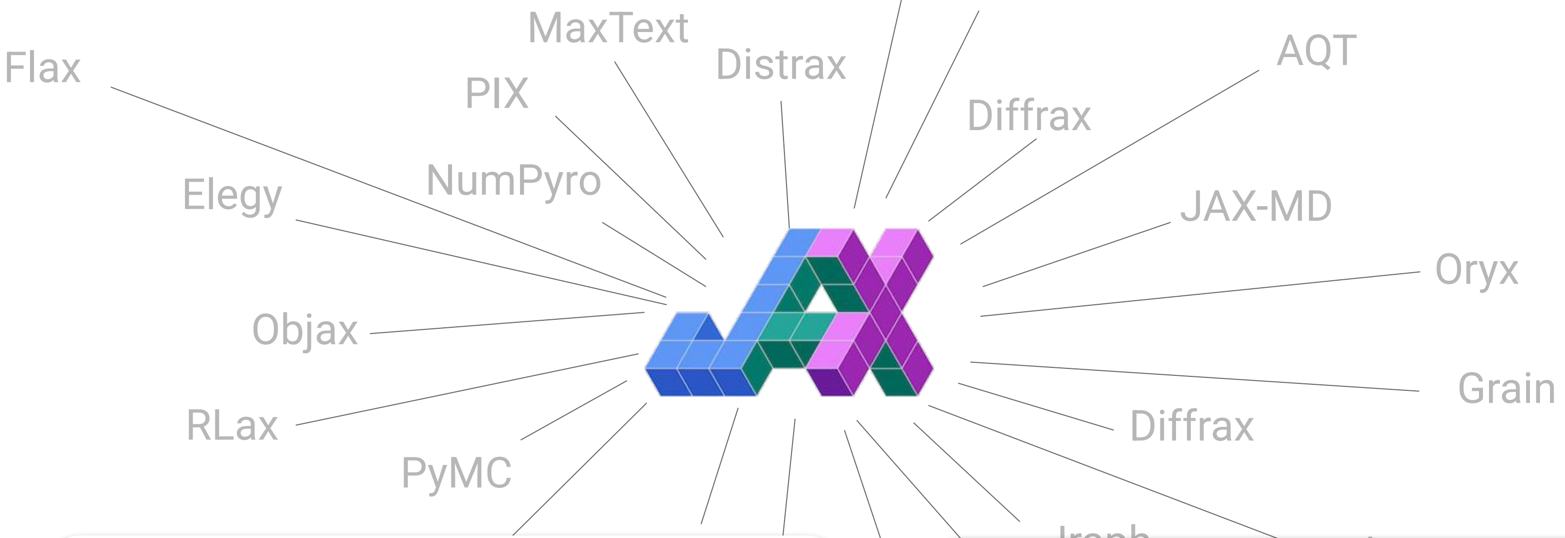
JAX Ecosystem



JAX Ecosystem

NNX

Object Oriented Neural Nets

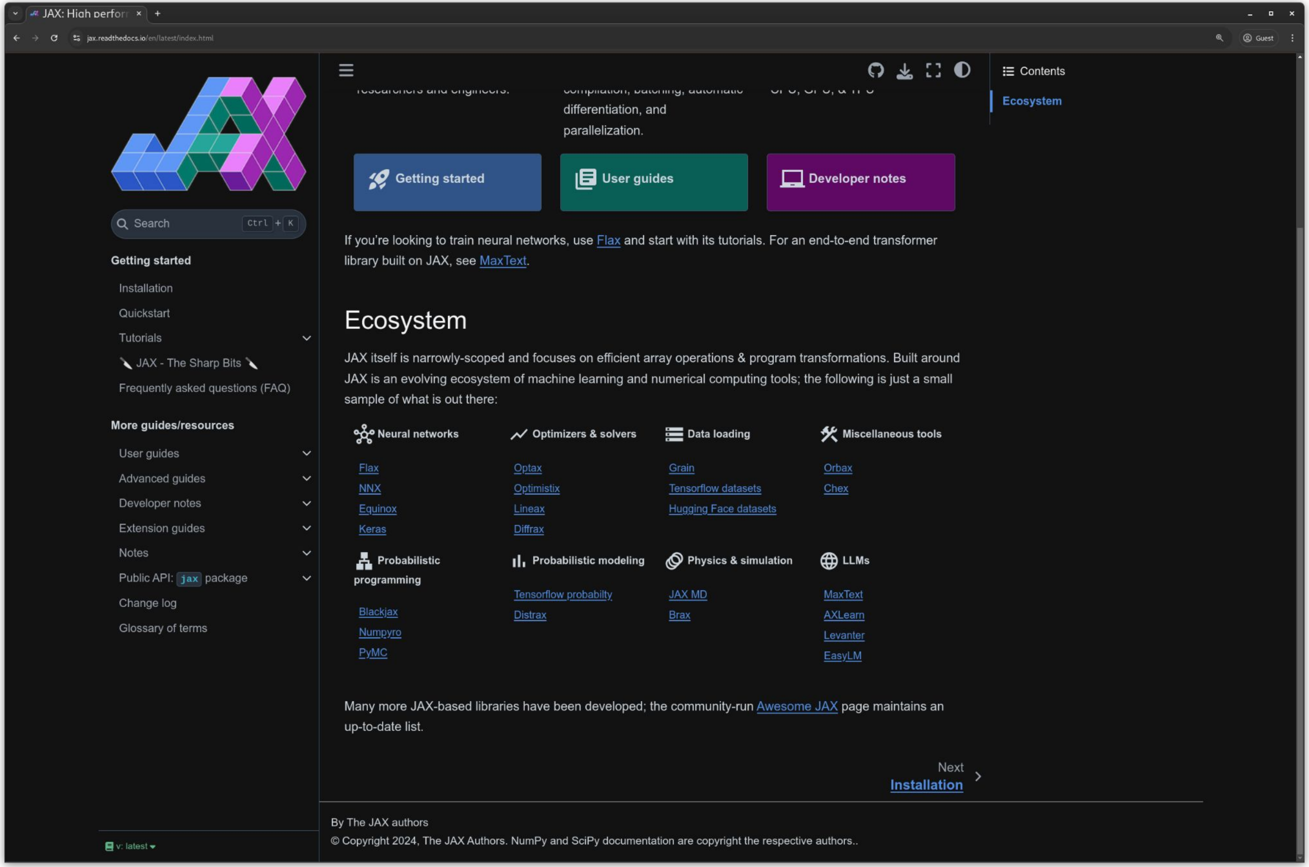


Optax

Fast Optimizers in JAX

Community List:
https://github.com/n2cholas/awesome-jax

Up-to-date list on jax.dev



Outline

- JAX Key Ideas & How JAX works
- Getting Started with JAX & JAX Ecosystem
- JAX Advanced Techniques

JAX Advanced Techniques

Sharding & Shard Map
Control over data, computation and communication

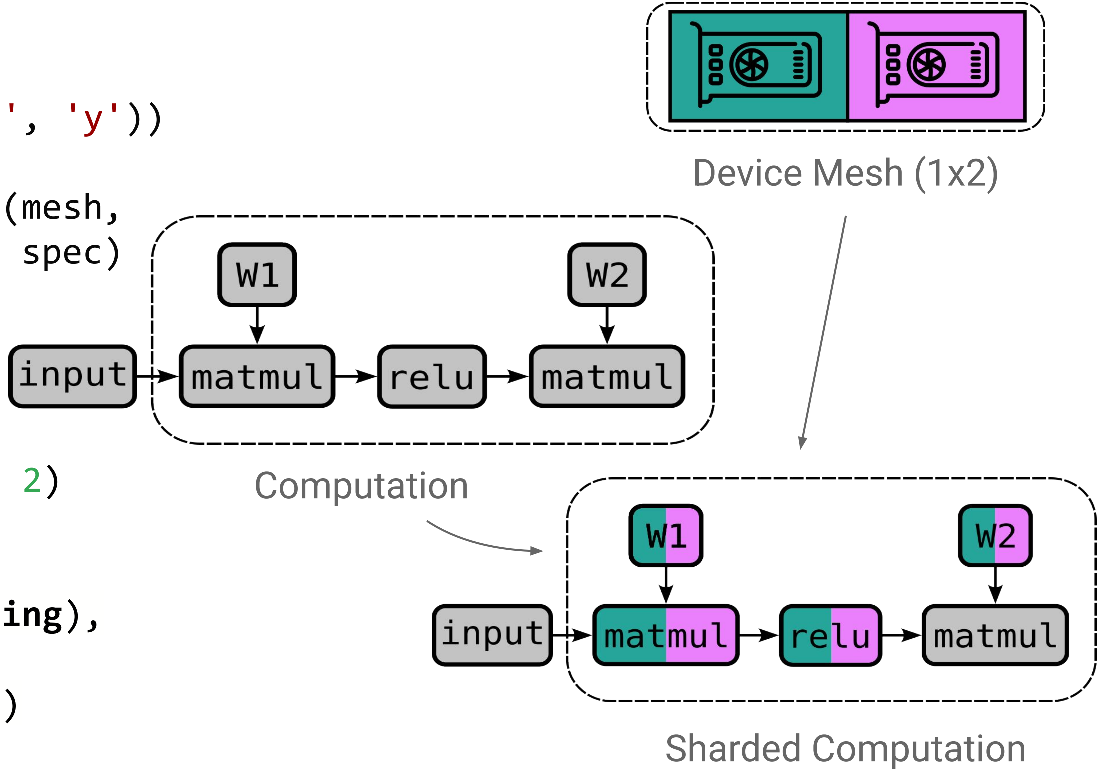
Multi-device and Multi-host Computation via Sharding

```
import jax.numpy as jnp
from jax import grad, vmap, jit

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)

params_sharded = jax.tree.map(
    lambda x: jax.device_put(x, sharding),
    params)
jit(grad(loss))(params_sharded, batch)
# OR
jit(grad(loss), in_shardings=...)(params,
                                  batch)
```

```
from jax.sharding import use_mesh
from jax.sharding import NamedSharding
from jax.sharding import PartitionSpec as P

mesh = jax.make_mesh((2, 4), ('x', 'y'))
spec = P('x', 'y')
with use_mesh(mesh):
     sharding = spec
# OR
sharding = NamedSharding(mesh, spec)
```

Automatic Parallel Computation with Array Sharding

```
from jax.sharding import Mesh
from jax.sharding import PartitionSpec as P

mesh = Mesh(jnp.array(
    jax.devices()).reshape(2, 4), ('x', 'y'))
spec = P('x', 'y')
sharding = jax.sharding.NamedSharding(mesh,
                                       spec)


def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)

params_sharded = jax.tree.map(
    lambda x: jax.device_put(x, sharding),
    params)
jit(grad(loss))(params_sharded, batch)
```

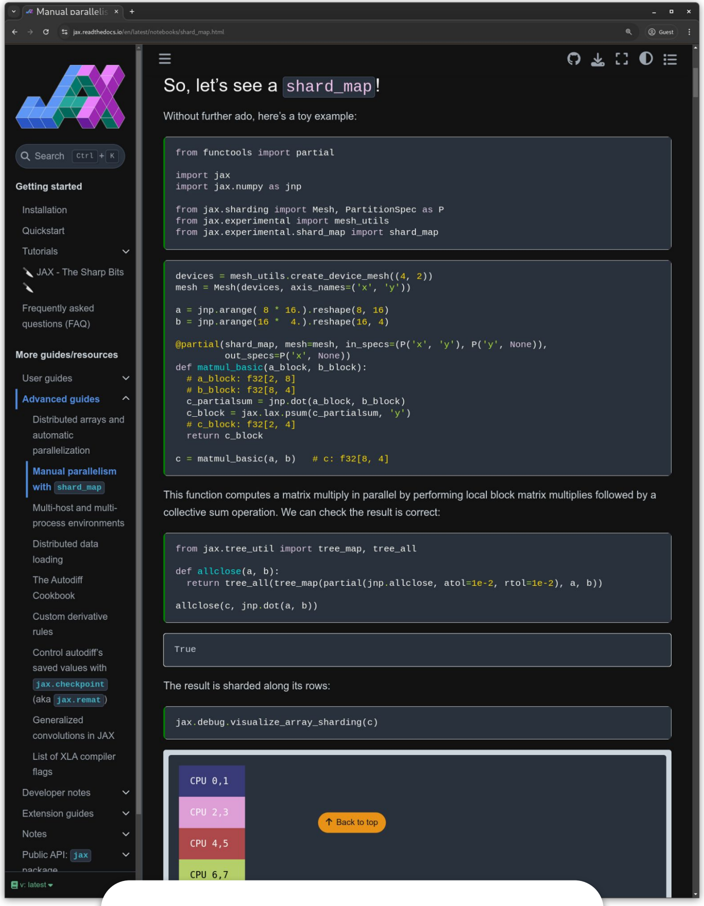


Google

Via GSPMD: arXiv [2105.04663]

Manual Parallelization - Shard Map

```python
import jax.numpy as jnp
from jax.experimental.shard_map import shard_map

devices = mesh_utils.create_device_mesh((4, 2))
mesh = Mesh(devices, axis_names=('x', 'y'))

a = jnp.arange( 8 * 16).reshape(8, 16)
b = jnp.arange(16 *  4).reshape(16, 4)

@partial(shard_map, mesh=mesh,
    in_specs=(P('x', 'y'), P('y', None)),
    out_specs=P('x', None))
def matmul_basic(a_block, b_block):
  # a_block: f32[2, 8]
  # b_block: f32[8, 4]
  z_partialsum = jnp.dot(a_block, b_block)  # compute
  z_block = jax.lax.psum(z_partialsum, 'y') # comms
  # c_block: f32[2, 4]
  return z_block
```



https://jax.readthedocs.io/en/latest/notebooks/shard_map.html

JAX Advanced Techniques

Sharding & Shard Map
Control over data, computation and communication

Pallas
Custom high-performance kernels on TPU and GPU

Pallas How? Hello World

references, not arrays

```python
def add_kernel(x_ref, y_ref, out_ref):
  x = x_ref[:,:]
  y = y_ref[:,:]
  out = x + y
  out_ref[:,:] = out
```

load value from ref – NumPy indexing

jax.numpy on arrays – as usual

write result – no return!

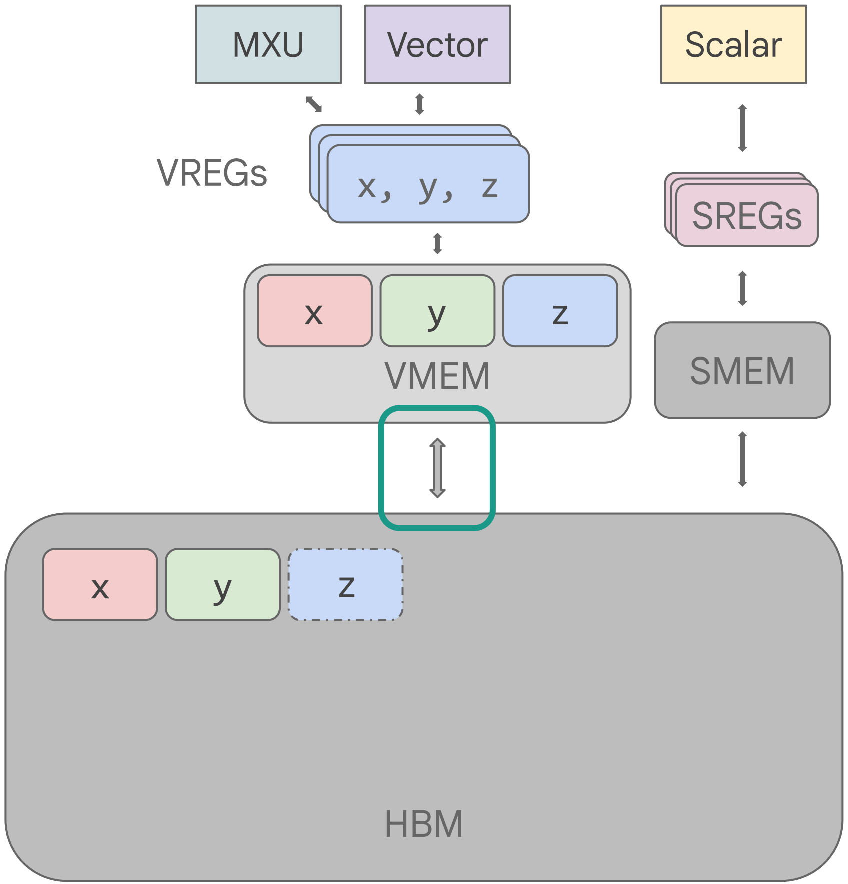
```python
x, y = ...  # JAX arrays
assert x.shape == y.shape and x.ndim == 2
jax_add_kernel = pl.pallas_call(
    add_kernel,
    out_shape=jax.ShapeDtypeStruct(x.shape, x.dtype),
)
z = jax_add_kernel(x, y)
```

`pallas_call` lifts the kernel to a JAX function

apply the kernel function to values

Pallas Why? Memory Pipeline

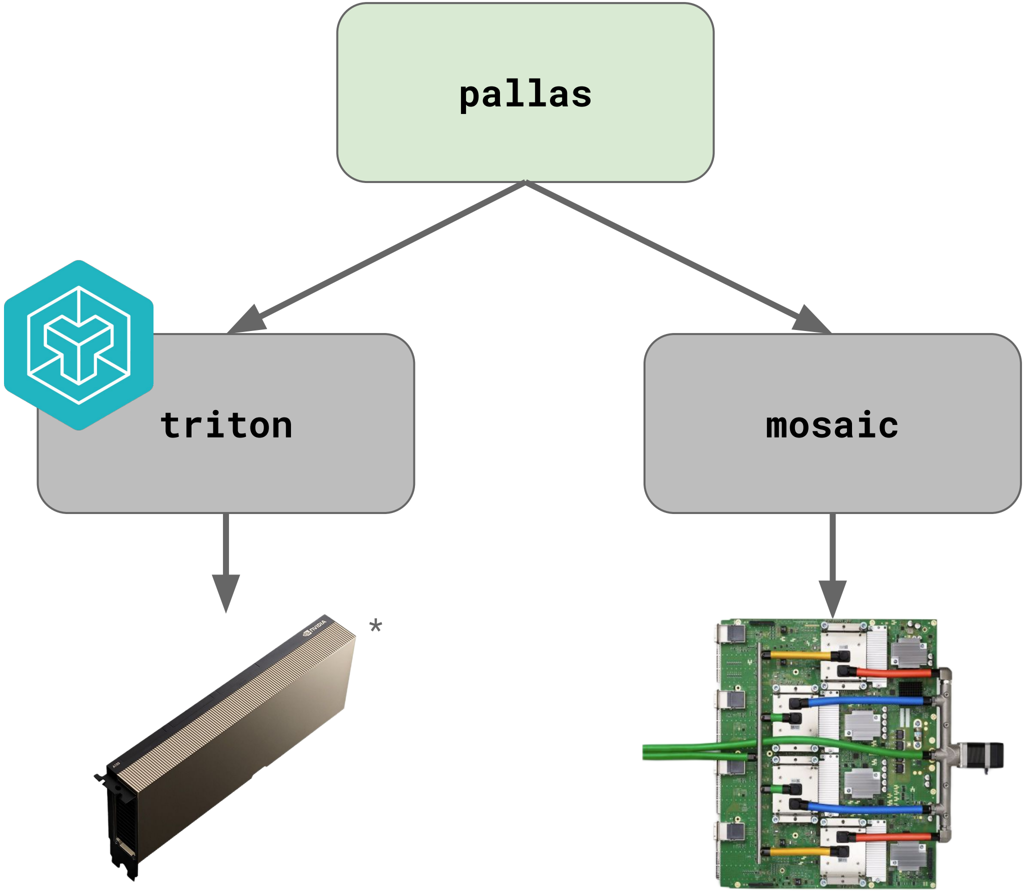
```python
def add_kernel(x_ref, y_ref, out_ref):
  x = x_ref[:,:]
  y = y_ref[:,:]
  out = x + y
  out_ref[:,:] = out


jax_add_kernel = pl.pallas_call(
  add_kernel,
  out_shape=jax.ShapeDtypeStruct(x.shape, x.dtype),
  in_specs=[pl.BlockSpec(lambda i, j: (i, j), (m, n)),
            pl.BlockSpec(lambda i, j: (i, j), (m, n))],
  out_specs=pl.BlockSpec(lambda i, j: (i, j), (m, n)),
  grid=(M, N),
)
```



Pipeline

HBM ↔ VMEM

Pallas targets TPU and GPU



Google

* Image: NVIDIA

JAX Advanced Techniques

Sharding & Shard Map
Control over data, computation and communication

Pallas
Custom high-performance kernels on TPU and GPU

Foreign Function Interface
Allow JAX to call external functions written in other languages

External Callbacks: The Simple Case

```python
import jax

# for debugging
jax.debug.callback(python_fn, arg1, arg2)

# for no-side-effect callbacks
out_shape = jax.ShapeDtypeStruct(shape, dtype)
jax.pure_callback(python_fn, out_shape, arg1, arg2)
# memory transferred to host device (CPU)
```

Google

Calling External Functions in JAX

```
float ComputeRmsNorm(float eps, int64_t size,
                     const float *x, float *y)
{
    ...
}

XLA_FFI_DEFINE_HANDLER_SYMBOL(...
```

Step 1: External Implementation → link to Python

```
from jax.extend.ffi import register_ffi_target
import cpp_module

register_ffi_target('target', cpp_module.cpp_fn)
```

Step 2: Register FFI function in XLA (JAX compiler)

```
from jax.extend.ffi import ffi_call

out_type = jax.ShapeDtypeStruct(...)

ffi_call('target', out_type, x,
         parameter=jnp.float32(0.5))
```

Step 3: Call from JAX eager and **under JIT**

`jax.extend.ffi`

Python bindings

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

Google

Foreign Function Interface (FFI) via C++

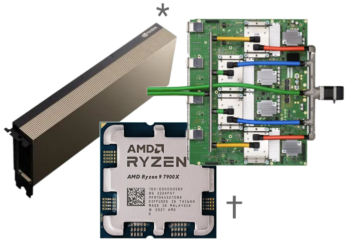
```
#include "xla/ffi/api/ffi.h"  // no dependencies!

Error ffi_call(
               float parameter,
               AnyBuffer input1,
               Result<Buffer<F32>> output1,
               ...) {
  auto shape = input.dimensions();
  auto dtype = input.element_type();
}
```

Google

A Single Change for CUDA

```
#include "xla/ffi/api/ffi.h"  // no dependencies!

Error ffi_call(CUStream stream,
               float parameter,
               AnyBuffer input1,
               Result<Buffer<F32>> output1,
               ...) {
  auto shape = input.dimensions();
  auto dtype = input.element_type();
}
```

Summary



Fast Iteration

High-level, compiler based

Performant & scalable to very large computations
Run on any: CPU/GPU/TPU
Fast and scalable

Cutting edge research
Machine learning, optimization, physics simulations, biology, ...

Contact info:

rdyro@google.com

@rdyro

https://github.com/jax-ml/jax

* Image: NVIDIA
† Image: AMD

How JAX Works: Tracing

Math

$$f(x) = \frac{\log(x)}{\log(2)}$$

implement

Python code

```
def f(x):
  return log(x) / log(2.)
```

trace

Jaxpr

```
{ lambda ; a:f32[]. let
    b:f32[] = log a
    c:f32[] = log p2.0
    d:f32[] = div b c
  in (d,) }
```

- very powerful
  (can do anything)
- difficult to analyze

- restricted
  (describes computations)
- easy to analyze

Google