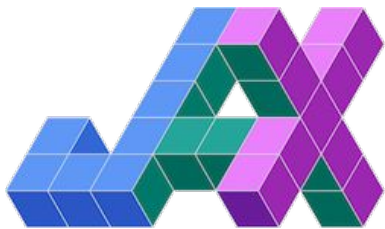


A Tour of JAX



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



Key Ideas

1. Familiar API

2. Transformations

3. Run Anywhere

1. Familiar API

```
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)
```

```
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)
```

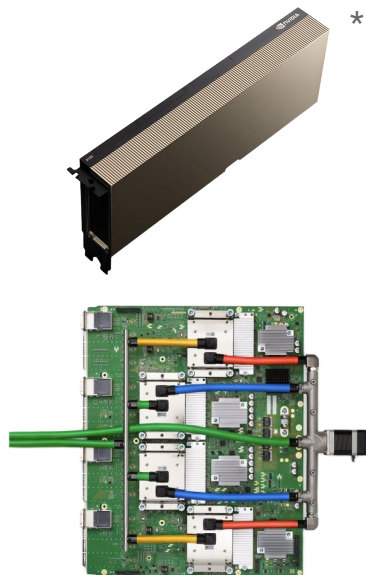
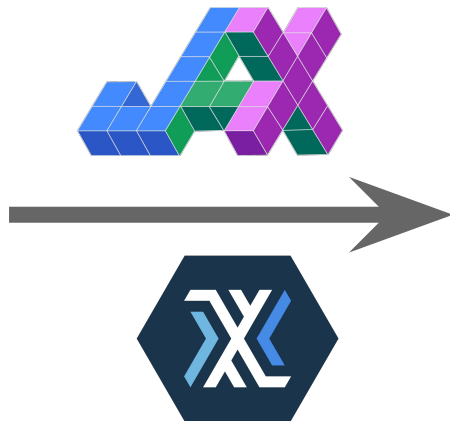
2. JAX Transformations

```
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)
```

✓ CPU/GPU/TPU
compilation

autodiff
parallelization



2. JAX Transformations

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

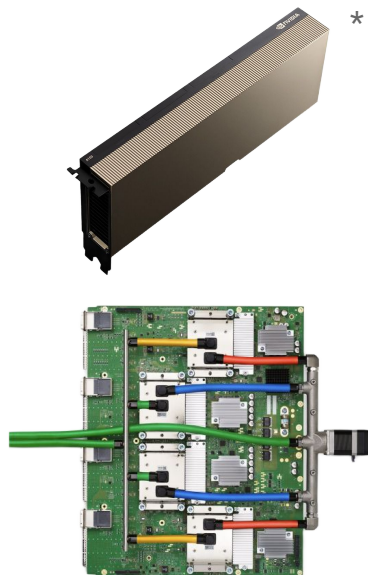
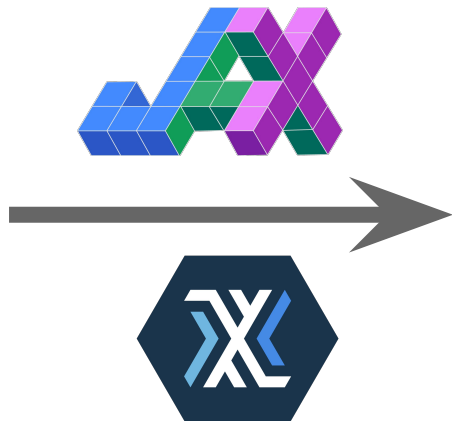
```
def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs

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

```
gradient_fun = grad(loss)
```

✓ CPU/GPU/TPU
compilation

✓ autodiff
parallelization



2. JAX Transformations

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

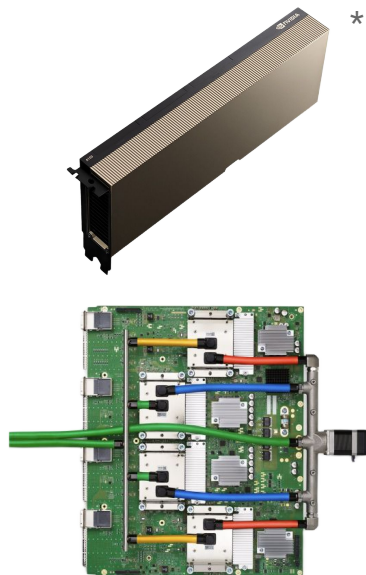
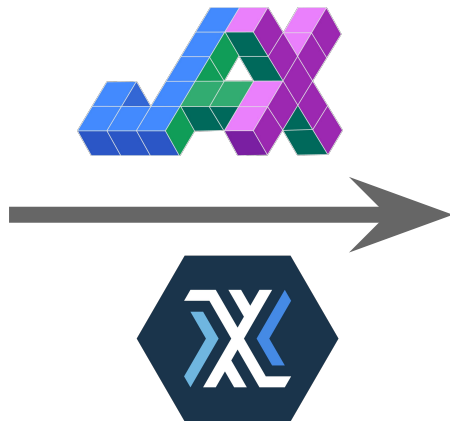
```
def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs
```

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

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

✓ CPU/GPU/TPU
compilation

✓ autodiff
parallelization



2. JAX Transformations

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

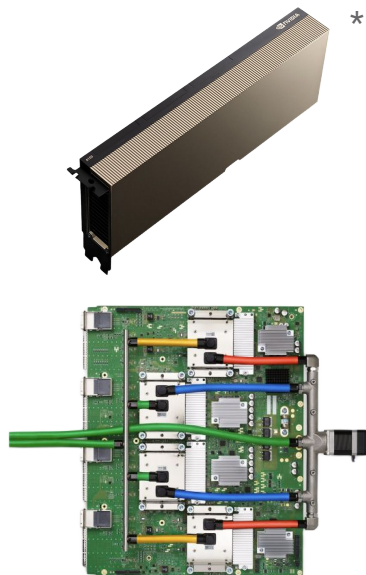
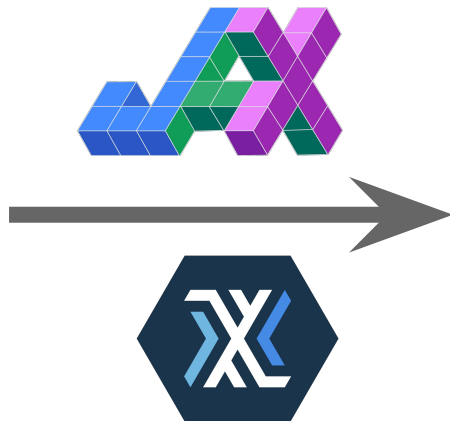
```
def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs
```

```
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)))
```

✓ CPU/GPU/TPU
✓ compilation

✓ autodiff
parallelization



2. JAX Transformations

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

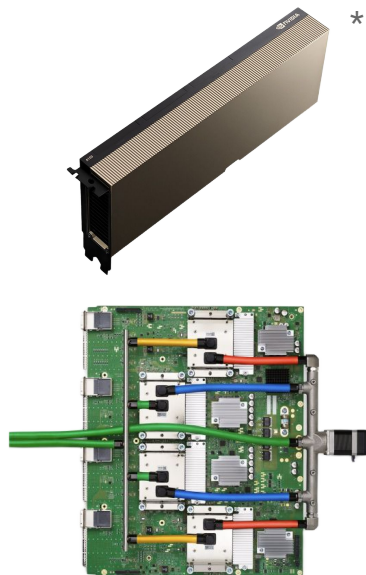
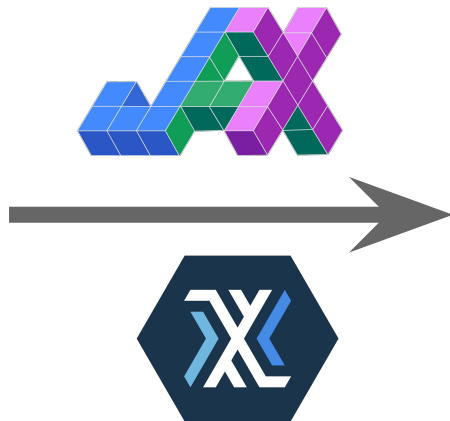
```
def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs
```

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

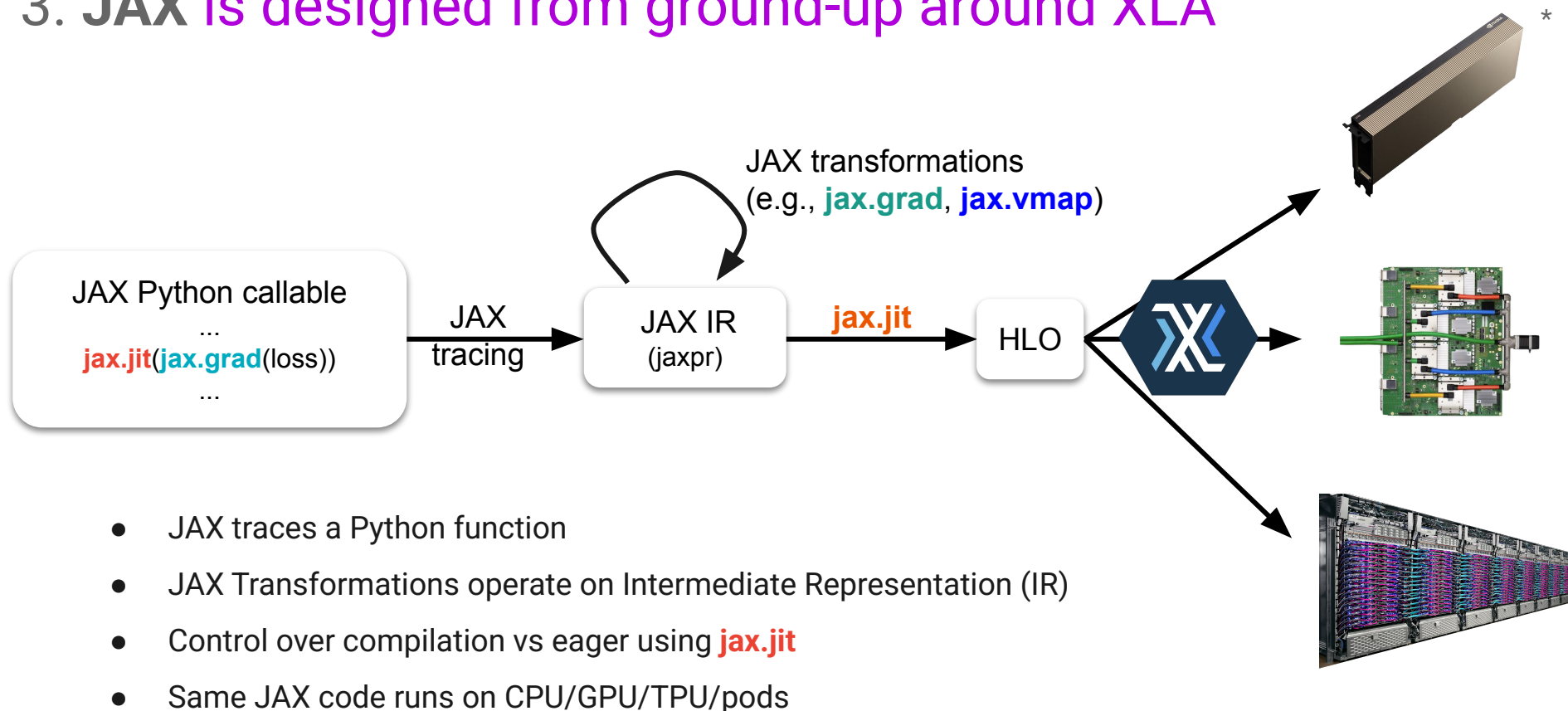
```
gradient_fun = jit(grad(loss))
perexample_grads = jit(vmap(grad(loss), in_axes=(None, 0)),
                        in_shardings=..., out_shardings=...)
```

✓ CPU/GPU/TPU
✓ compilation

✓ autodiff
✓ parallelization




3. JAX is designed from ground-up around XLA



How JAX Works: Python code → jaxpr

1. Use JAX primitives

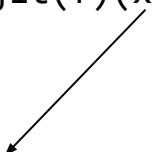
```
from jax.numpy import log
# from jax.lax import log
```



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

2. Trace function

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



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

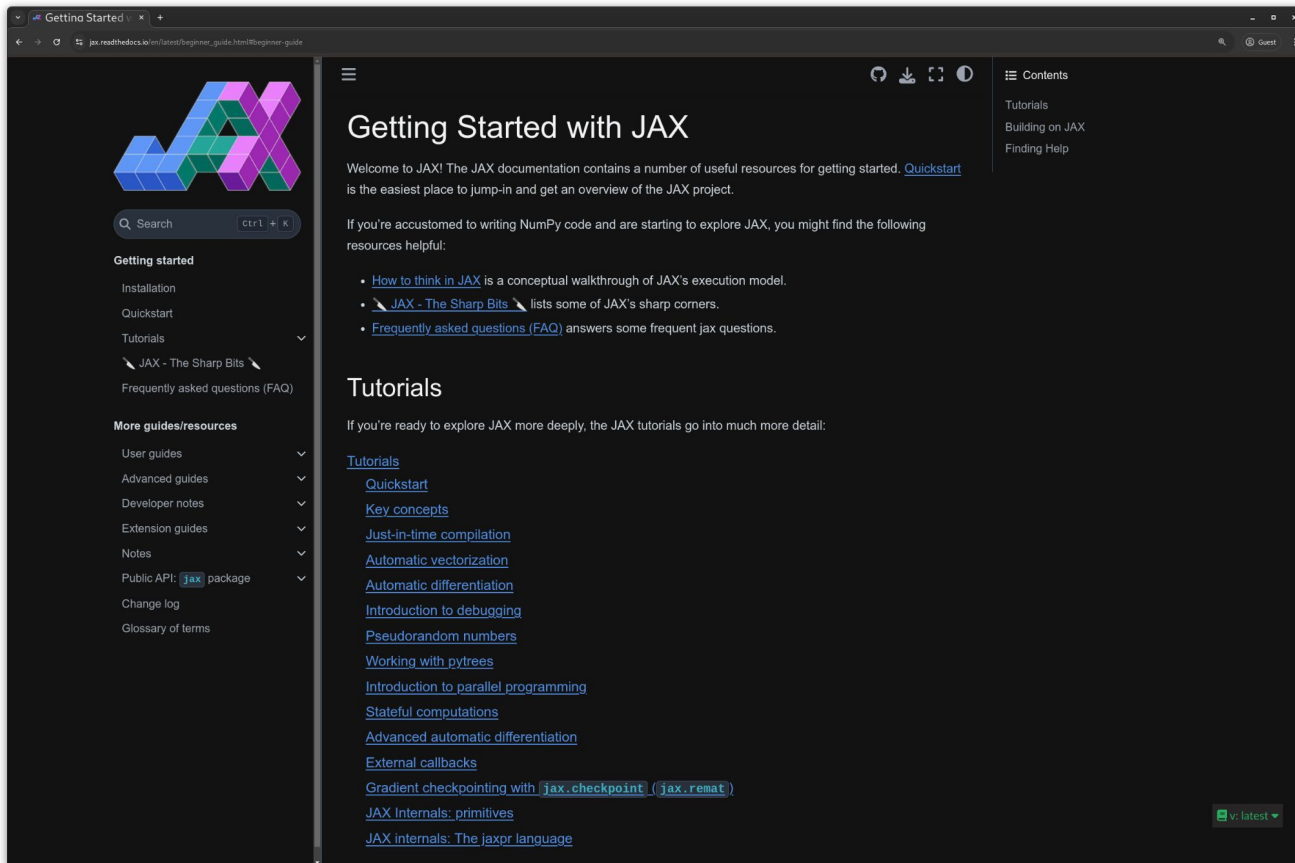
3. Record jaxpr
(JAX expression)

```
{ 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



The screenshot shows the JAX documentation website in a dark-themed browser window. The page title is 'Getting Started with JAX'. The left sidebar contains a navigation menu with sections: 'Getting started' (including Installation, Quickstart, Tutorials, JAX - The Sharp Bits, and Frequently asked questions (FAQ)), and 'More guides/resources' (including User guides, Advanced guides, Developer notes, Extension guides, Notes, Public API: jax package, Change log, and Glossary of terms). The main content area has a large JAX logo at the top left. Below it, a search bar is visible. The main heading is 'Getting Started with JAX'. The text below the heading says: 'Welcome to JAX! The JAX documentation contains a number of useful resources for getting started. [Quickstart](#) is the easiest place to jump-in and get an overview of the JAX project.' Below this, a paragraph states: 'If you're accustomed to writing NumPy code and are starting to explore JAX, you might find the following resources helpful:' followed by a bulleted list of links: 'How to think in JAX', 'JAX - The Sharp Bits', and 'Frequently asked questions (FAQ)'. The next section is 'Tutorials', with a paragraph stating: 'If you're ready to explore JAX more deeply, the JAX tutorials go into much more detail.' Below this, a list of tutorial links is provided: 'Quickstart', 'Key concepts', 'Just-in-time compilation', 'Automatic vectorization', 'Automatic differentiation', 'Introduction to debugging', 'Pseudorandom numbers', 'Working with pytrees', 'Introduction to parallel programming', 'Stateful computations', 'Advanced automatic differentiation', 'External callbacks', 'Gradient checkpointing with jax.checkpoint (jax.remat)', 'JAX Internals: primitives', and 'JAX Internals: The jaxpr language'. A 'v: latest' dropdown menu is located in the bottom right corner of the page.

Getting Started with JAX

Welcome to JAX! The JAX documentation contains a number of useful resources for getting started. [Quickstart](#) is the easiest place to jump-in and get an overview of the JAX project.

If you're accustomed to writing NumPy code and are starting to explore JAX, you might find the following resources helpful:

- [How to think in JAX](#) is a conceptual walkthrough of JAX's execution model.
- [JAX - The Sharp Bits](#) lists some of JAX's sharp corners.
- [Frequently asked questions \(FAQ\)](#) answers some frequent jax questions.

Tutorials

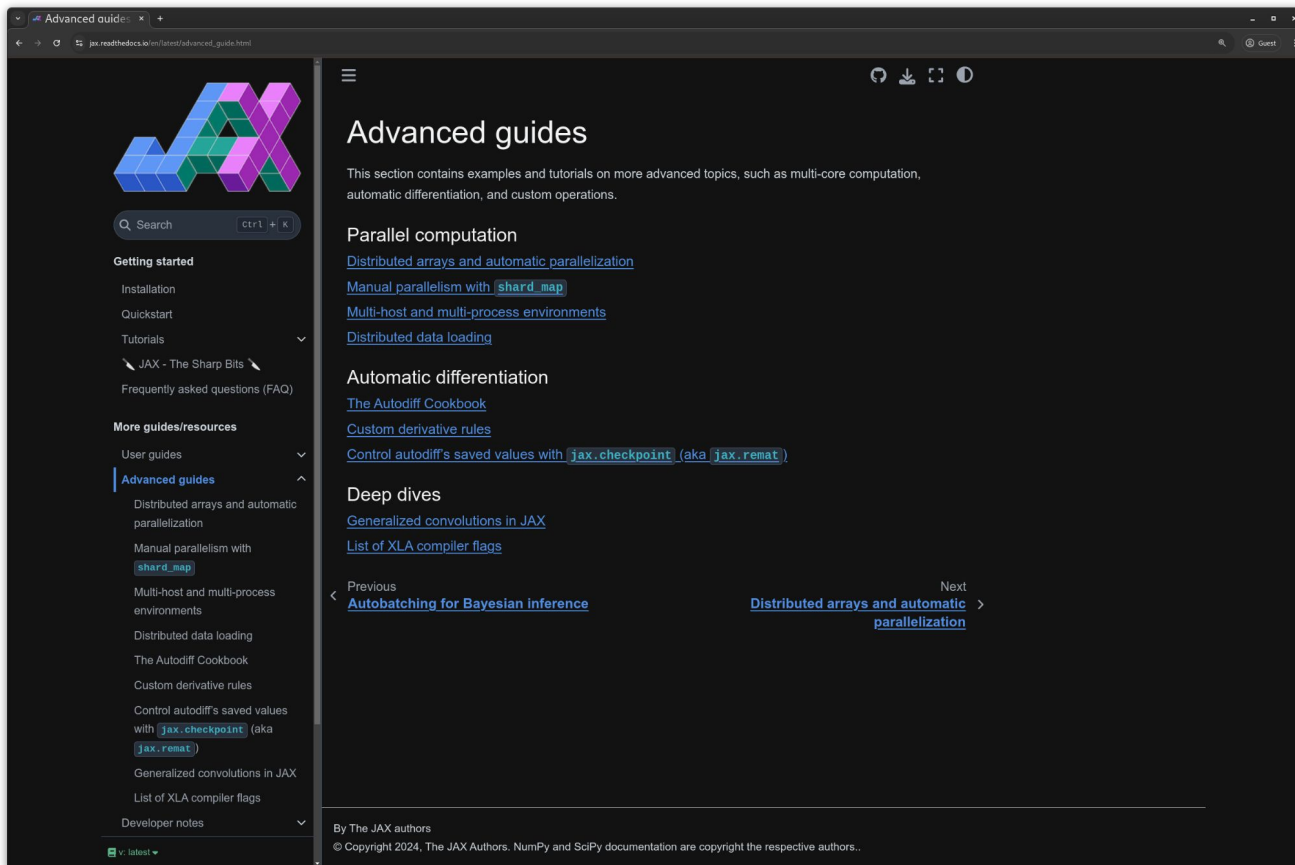
If you're ready to explore JAX more deeply, the JAX tutorials go into much more detail:

[Tutorials](#)

- [Quickstart](#)
- [Key concepts](#)
- [Just-in-time compilation](#)
- [Automatic vectorization](#)
- [Automatic differentiation](#)
- [Introduction to debugging](#)
- [Pseudorandom numbers](#)
- [Working with pytrees](#)
- [Introduction to parallel programming](#)
- [Stateful computations](#)
- [Advanced automatic differentiation](#)
- [External callbacks](#)
- [Gradient checkpointing with `jax.checkpoint` \(`jax.remat`\)](#)
- [JAX Internals: primitives](#)
- [JAX Internals: The jaxpr language](#)

v: latest

Getting Started



The screenshot shows a web browser displaying the JAX Advanced guides page. The page has a dark theme and a sidebar on the left. The sidebar contains a search bar, a 'Getting started' section with links to Installation, Quickstart, Tutorials, and JAX - The Sharp Bits, and a 'More guides/resources' section with links to User guides, Advanced guides (highlighted), Distributed arrays and automatic parallelization, Manual parallelism with shard_map, Multi-host and multi-process environments, Distributed data loading, The Autodiff Cookbook, Custom derivative rules, Control autodiff's saved values with jax.checkpoint (aka jax.remat), Generalized convolutions in JAX, List of XLA compiler flags, and Developer notes. The main content area is titled 'Advanced guides' and contains a paragraph about the section's purpose. It lists three categories: 'Parallel computation' with links to Distributed arrays and automatic parallelization, Manual parallelism with shard_map, Multi-host and multi-process environments, and Distributed data loading; 'Automatic differentiation' with links to The Autodiff Cookbook, Custom derivative rules, and Control autodiff's saved values with jax.checkpoint (aka jax.remat); and 'Deep dives' with links to Generalized convolutions in JAX and List of XLA compiler flags. At the bottom, there are navigation links for 'Previous' (Autobatching for Bayesian inference) and 'Next' (Distributed arrays and automatic parallelization). The footer includes the text 'By The JAX authors' and '© Copyright 2024, The JAX Authors. NumPy and SciPy documentation are copyright the respective authors..'

Advanced guides

This section contains examples and tutorials on more advanced topics, such as multi-core computation, automatic differentiation, and custom operations.

Parallel computation

- [Distributed arrays and automatic parallelization](#)
- [Manual parallelism with `shard_map`](#)
- [Multi-host and multi-process environments](#)
- [Distributed data loading](#)

Automatic differentiation

- [The Autodiff Cookbook](#)
- [Custom derivative rules](#)
- [Control autodiff's saved values with `jax.checkpoint` \(aka `jax.remat`\)](#)

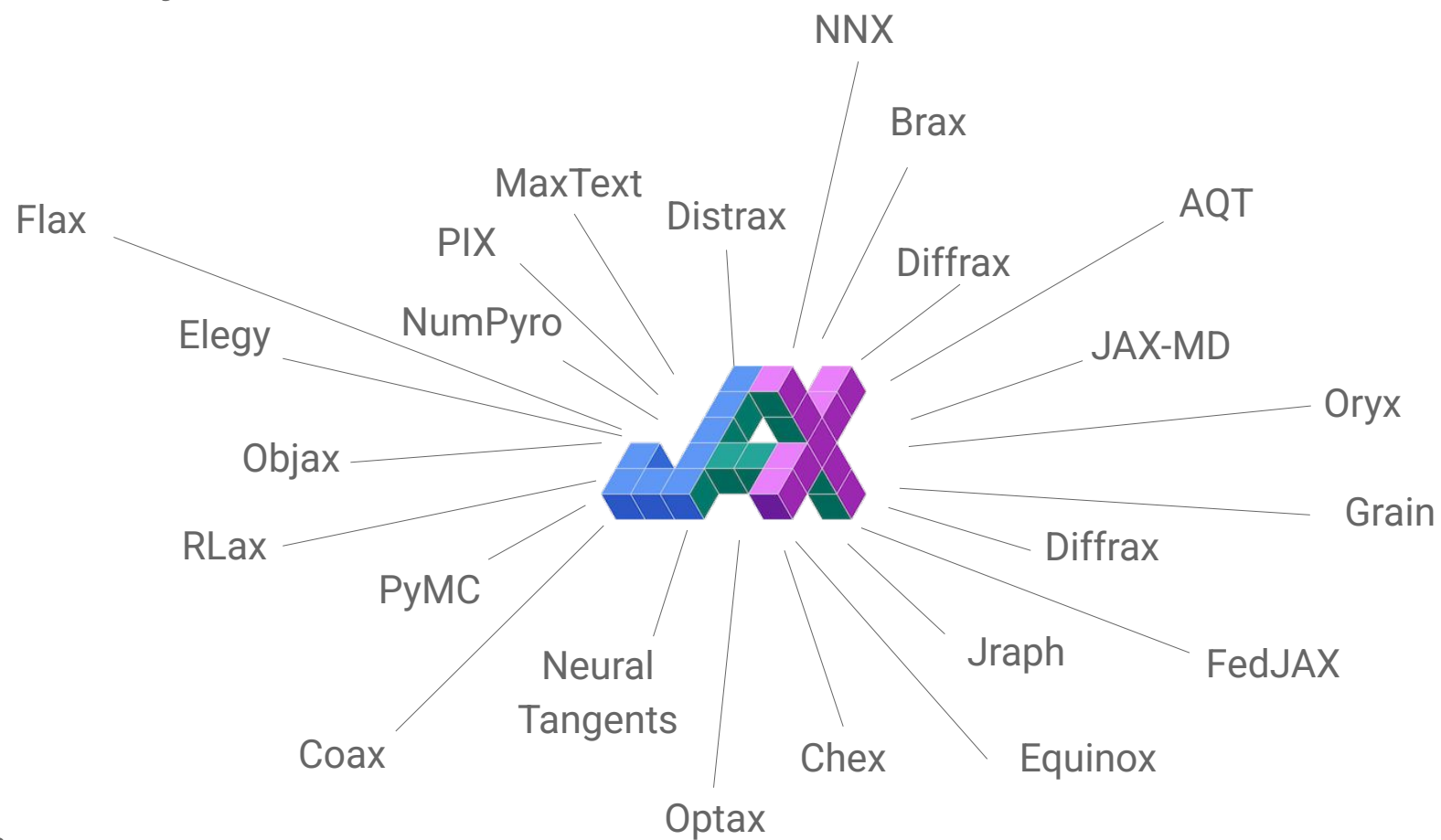
Deep dives

- [Generalized convolutions in JAX](#)
- [List of XLA compiler flags](#)


Previous: [Autobatching for Bayesian inference](#) | Next: [Distributed arrays and automatic parallelization](#)

By The JAX authors
© Copyright 2024, The JAX Authors. NumPy and SciPy documentation are copyright the respective authors..

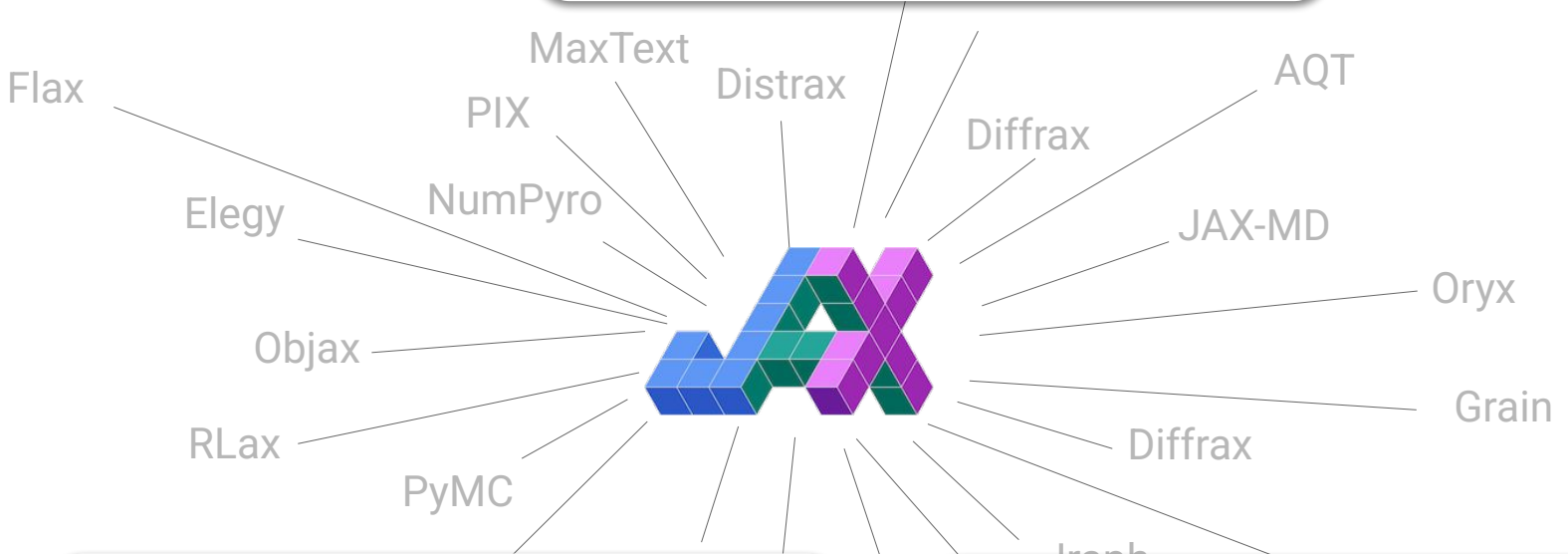
JAX Ecosystem

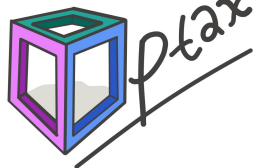


JAX Ecosystem



NNX
Object Oriented Neural Nets



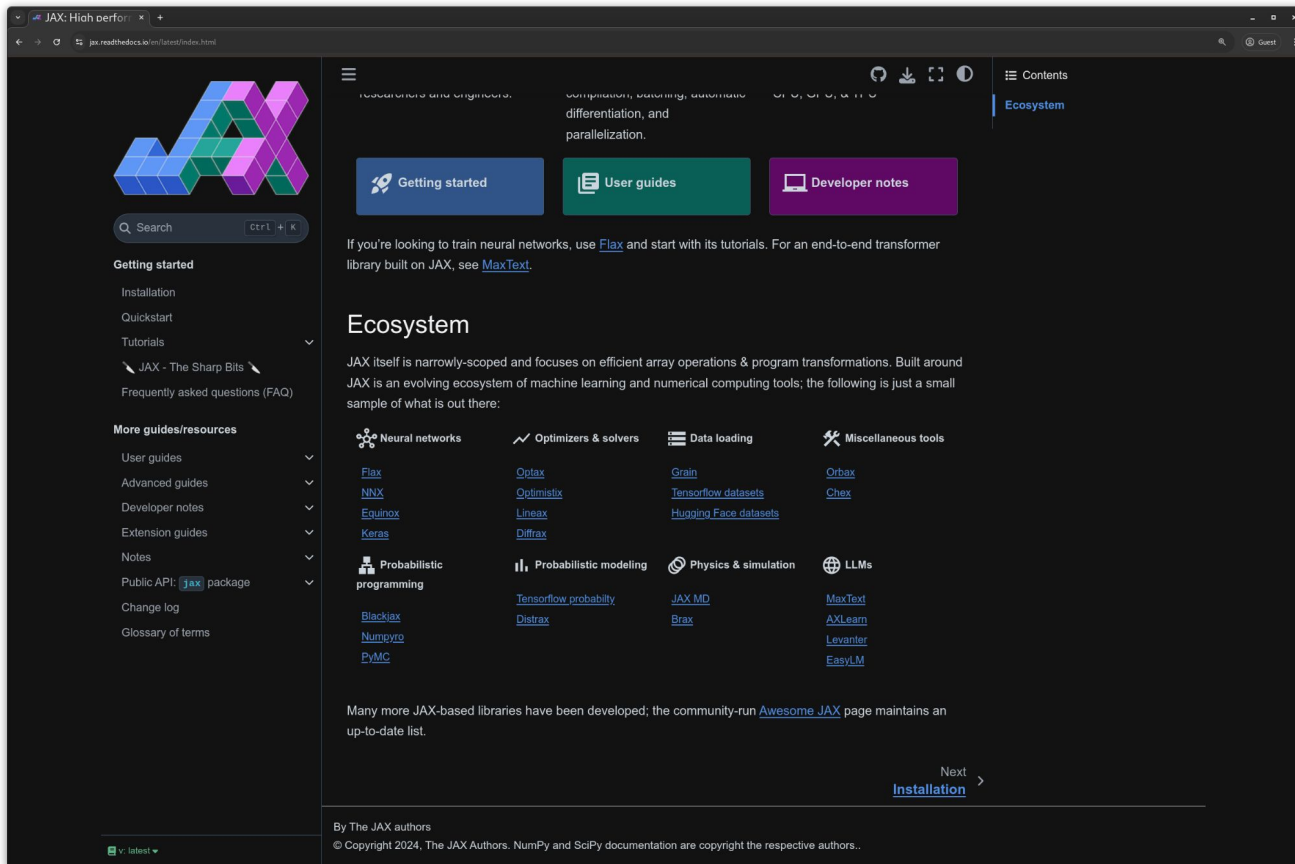


Optax
Fast Optimizers in JAX

Community List:

<https://github.com/n2cholas/awesome-jax>

Up-to-date list on jax.dev



The screenshot shows the JAX High Performance Computing website. The header features the JAX logo, a search bar, and navigation links for 'Contents' and 'Ecosystem'. The main content area is divided into sections: 'Getting started' (Installation, Quickstart, Tutorials, JAX - The Sharp Bits, Frequently asked questions (FAQ)), 'More guides/resources' (User guides, Advanced guides, Developer notes, Extension guides, Notes, Public API: jax package, Change log, Glossary of terms), and 'Ecosystem' (Neural networks, Optimizers & solvers, Data loading, Miscellaneous tools, Probabilistic programming, Probabilistic modeling, Physics & simulation, LLMs). The 'Ecosystem' section lists various libraries and tools, including Flax, NNX, Equinox, Keras, Optax, Optimistix, Lineax, Diffax, Grain, Tensorflow datasets, Hugging Face datasets, Orbx, Chex, TensorFlow probability, Distrax, JAX MD, Brax, MaxText, AXLearn, Levanter, and EasyLM. A 'Next Installation' link is visible at the bottom right.

JAX: High performance computing

Installation, Quickstart, Tutorials, JAX - The Sharp Bits, Frequently asked questions (FAQ)

More guides/resources

User guides, Advanced guides, Developer notes, Extension guides, Notes, Public API: [jax](#) package, Change log, Glossary of terms

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	Optimizers & solvers	Data loading	Miscellaneous tools
Flax	Optax	Grain	Orbx
NNX	Optimistix	Tensorflow datasets	Chex
Equinox	Lineax	Hugging Face datasets	
Keras	Diffax		

Probabilistic programming	Probabilistic modeling	Physics & simulation	LLMs
Blackjax	Tensorflow probability	JAX MD	MaxText
NumPyro	Distrax	Brax	AXLearn
PyMC			Levanter
			EasyLM

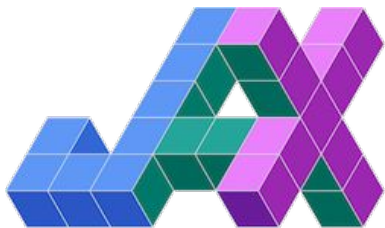
Many more JAX-based libraries have been developed; the community-run [Awesome JAX](#) page maintains an up-to-date list.

Next [Installation](#)

By The JAX authors
© Copyright 2024, The JAX Authors. NumPy and SciPy documentation are copyright the respective authors.

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)
```

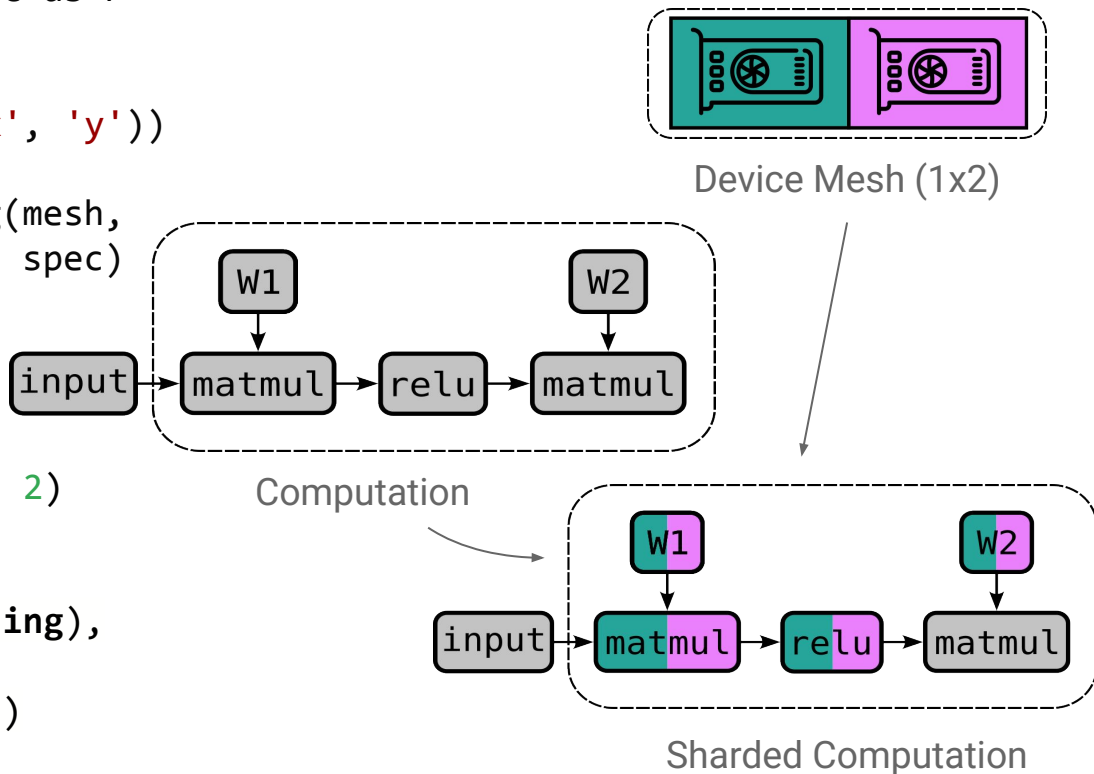
Automatic Parallel Computation with Array Sharding

```
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)
```



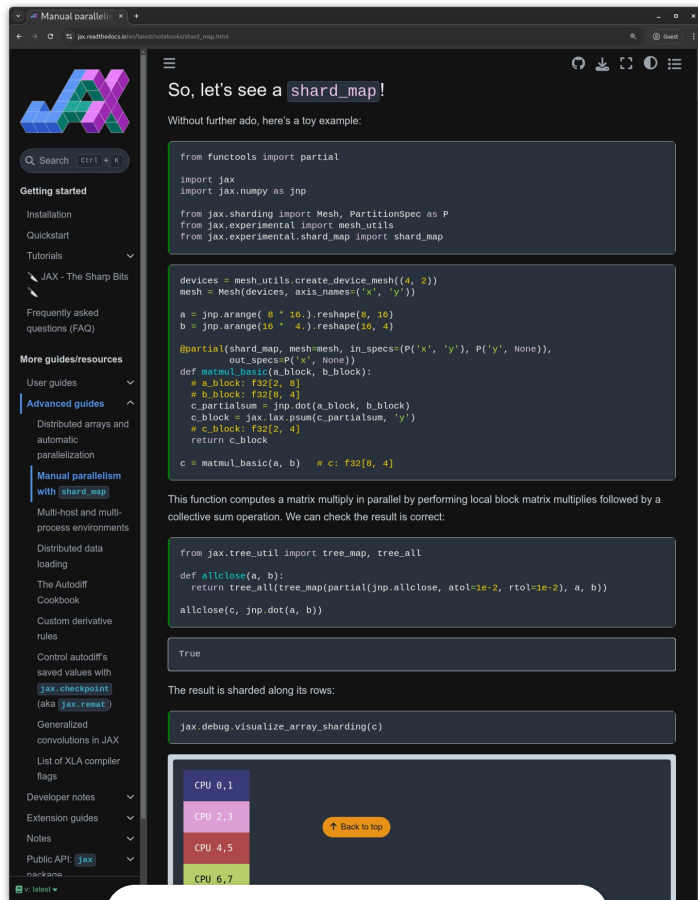
Manual Parallelization - Shard Map

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

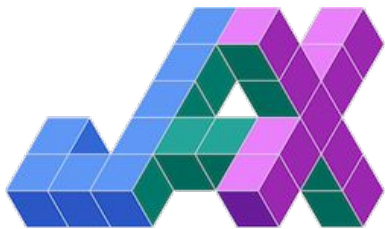
devices = mesh_utils.create_device_mesh((4, 2))
mesh = Mesh(devices, axis_names=('x', 'y'))

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_partials = jnp.dot(a_block, b_block)  # compute
    z_block = jax.lax.psum(z_partials, '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

```
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!

```
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

```
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))],
```

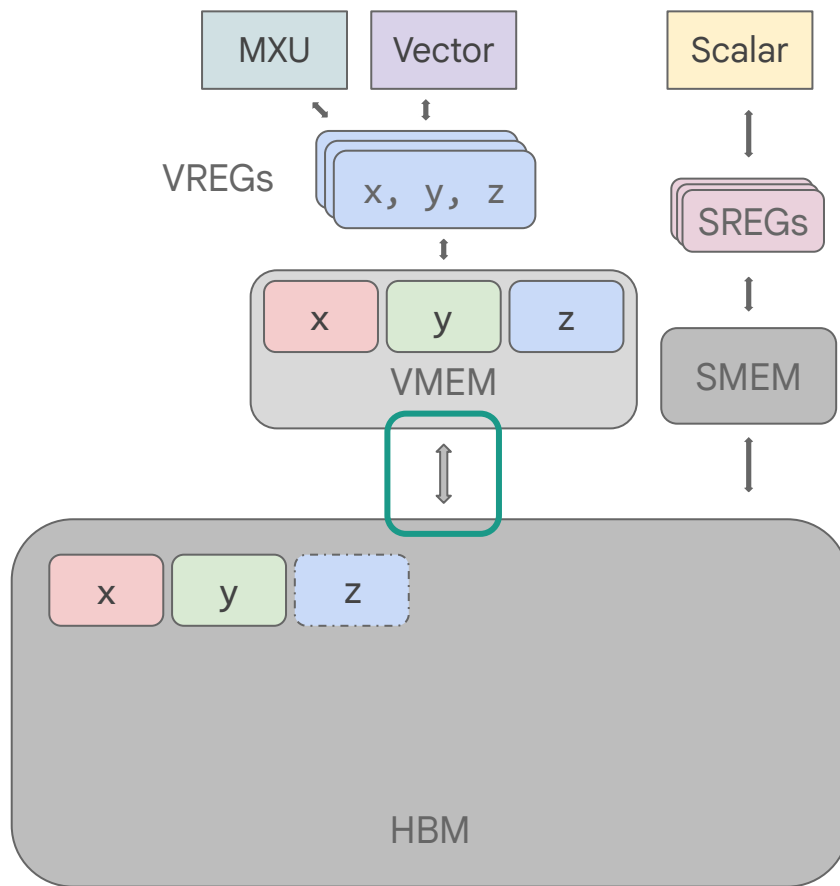
```
    out_specs=pl.BlockSpec(lambda i, j: (i, j), (m, n)),
```

```
    grid=(M, N),
```

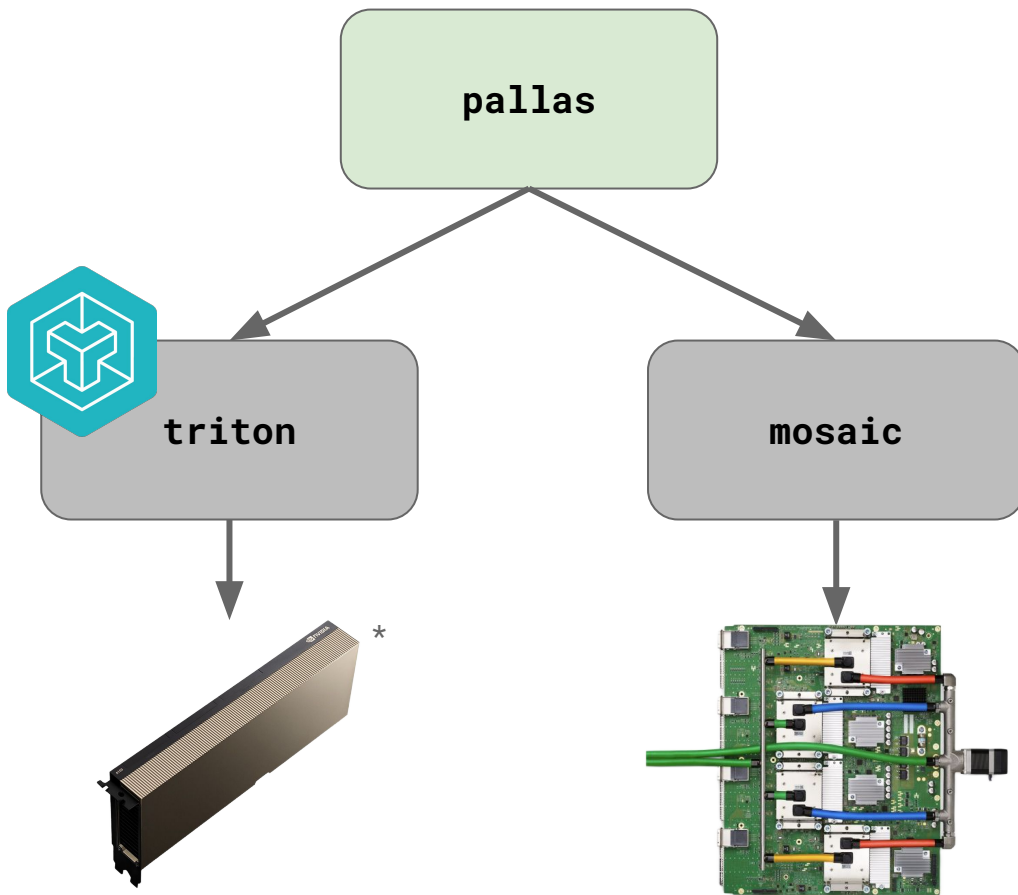
```
)
```

Pipeline

HBM ↔ VMEM



Pallas targets TPU and GPU





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

```
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)
```

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>

Foreign Function Interface (FFI) via C++

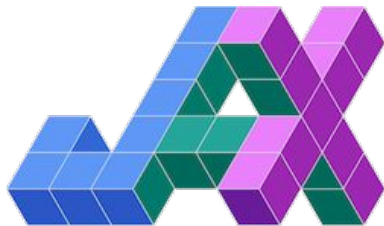
```
#include "xla/ffi/api/ffi.h" // no dependencies!
```

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

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

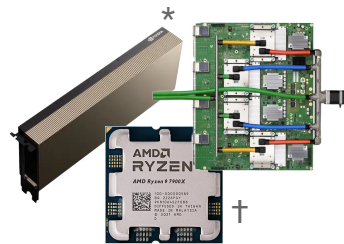
High-level, compiler based

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>

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