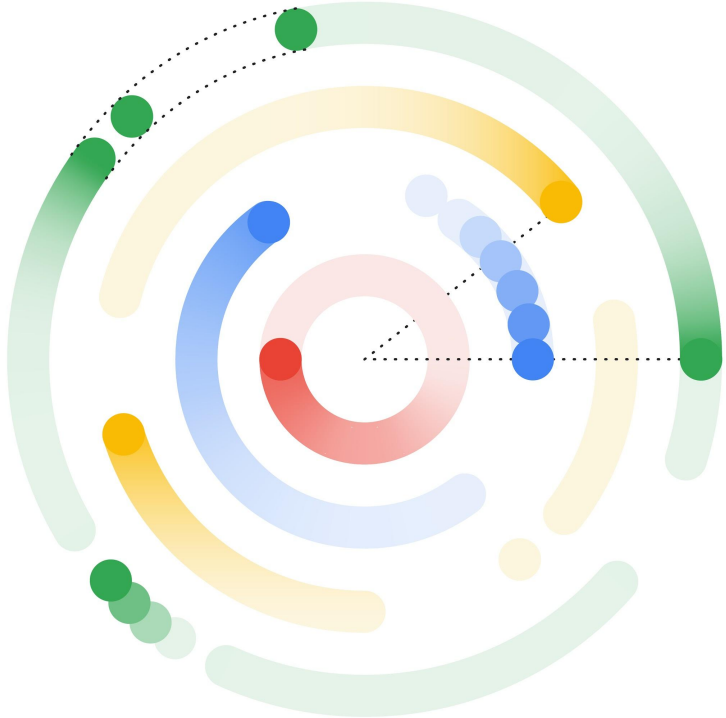
Google Cloud

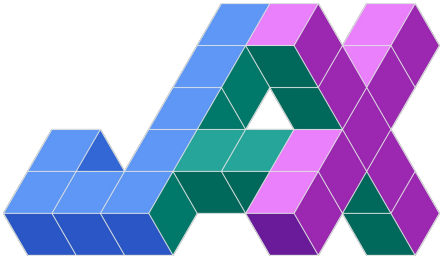
What JAX and Colab Can Offer the Data Science Community

Google Cloud Applied ML Summit
Solving for the future.

06/10/21



Jake VanderPlas
Software Engineer
Google



Accelerated machine-learning research via composable function transformations in Python

mattjj@ frostig@ leary@ dougalm@

phawkins@ skyewm@ jekbradbury@

necula@ apaszke@ jakevdp@

Motivating JAX

How might you implement a **performant** & **scalable** deep neural network from scratch in Python?

Array Computing in Python: NumPy

- Developed since the 1990s for **array computing** in Python
- Efficient & powerful language for manipulating arrays
- Highly extensible; core of the scientific Python ecosystem

Deep learning in NumPy

```
import numpy as np


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

Deep learning in NumPy

```
import numpy as np


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

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

Deep learning in NumPy

```
import numpy as np


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

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

What's missing?

- Running on accelerated hardware (**GPU/TPU**)
- Fast optimization via **autodiff**
- **Compilation**/Fusing of operations
- **Parallelization** of data & computation

Motivating JAX

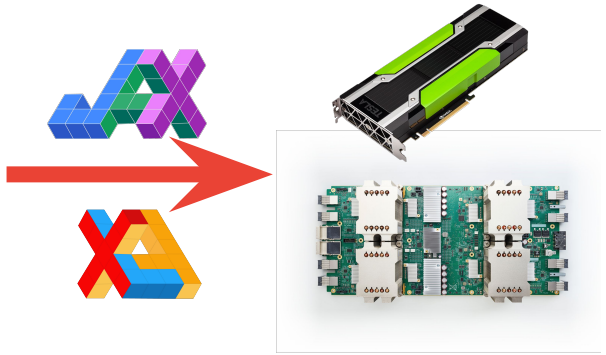
```
import numpy as np


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

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

Motivating JAX

```
import jax.numpy as np


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

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

Motivating JAX

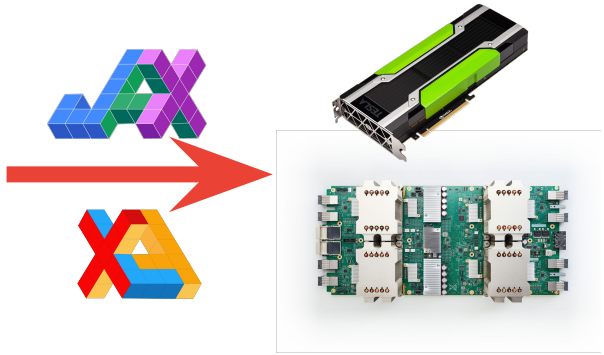
```
import jax.numpy as np


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

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



GPU/TPU

Motivating JAX

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

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

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


gradient_fun = grad(loss)
```

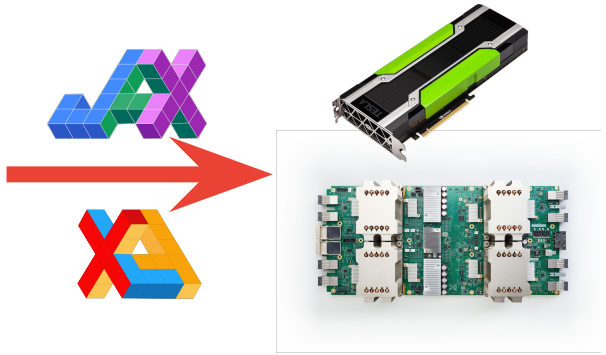


GPU/TPU

autodiff

Motivating JAX

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

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

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


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

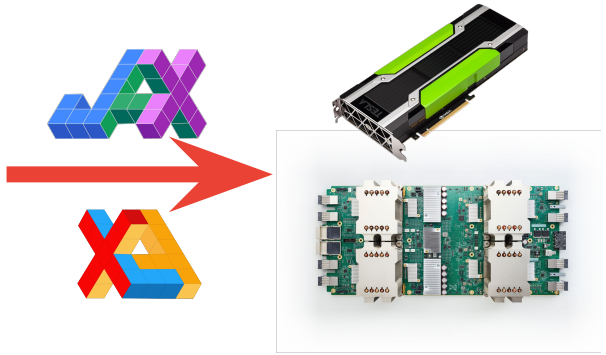


GPU/TPU

autodiff

Motivating JAX

```
import jax.numpy as np
from jax import grad, pmap, jit

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

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


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



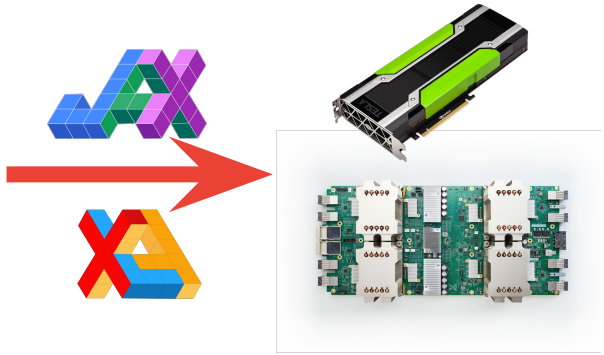
GPU/TPU

autodiff

compilation

Motivating JAX

```
import jax.numpy as np
from jax import grad, pmap, jit

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

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


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



GPU/TPU

autodiff

compilation

parallelization

Key Ideas:

- Python code traced to an IR* that can be transformed (e.g. automatic differentiation)
- Same IR enables domain-specific compilation (XLA)
- Familiar user-facing API (numpy + scipy)
- Powerful set of transforms (grad, jit, vmap, pmap)

*IR = Intermediate Representation

How does JAX work?

Thinking about JIT compilation.

What does this function do?

```
def f(x):
  return x + 2
```

Thinking about JIT compilation.

What does this function do?

```
def f(x):
  return x + 2


class EspressoDelegator(object):

  def __add__(self, num_espressos):
    subprocess.popen(["ssh", ...])
```

Thinking about JIT compilation.

What does this function do?

```
def f(x:float32):
  return x + 2
```

Python's dynamism means a function can do almost *anything* depending on it's input.

Thinking about JIT compilation.

What does this function do?

```
def f(x:ShapedArray[float32, (2, 2)]):
  return x + 2
```

Python's dynamism means a function can do almost *anything* depending on it's input.

JAX takes advantage of this and evaluates a function's behavior by calling it on a tracer value.

Python function ⟶ JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

Python function → JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

Calls to JAX **primitive operations,** the elementary operations jax knows how to transform.

Python function → JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

Python function ⟶ JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

Replace argument **x** with a special tracer object

Python function →
JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

```
{ lambda  ; a.
  let b = log a
```

Python function →
JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2.0)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

```
{ lambda  ; a.
  let b = log a
      c = log 2.0
```

Python function →
JAX Intermediate Representation

```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2.0)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

```
{ lambda  ; a.
  let b = log a
      c = log 2.0
      d = div b c
```

Python function →
JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2.0)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

```
{ lambda  ; a.
  let b = log a
      c = log 2.0
      d = div b c
  in [d] }
```

Python function →
JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2.0)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

```
{ lambda  ; a.
  let b = log a
      c = log 2.0
      d = div b c
  in [d] }
```

Python function →
JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2.0)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

```
{ lambda  ; a.
  let b = log a
      c = log 2.0
      d = div b c
  in [d] }
```

`grad` Automatic differentiation

Python function →
JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2.0)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

```
{ lambda  ; a.
  let b = log a
      c = log 2.0
      d = div b c
  in [d] }
```

`grad` Automatic differentiation

`vmap/pmap` Batching & parallelization

Python function →
JAX Intermediate Representation

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2.0)
  return ln_x / ln_2
```

```
x = np.array(...)
y = jit(log2)(x)
```

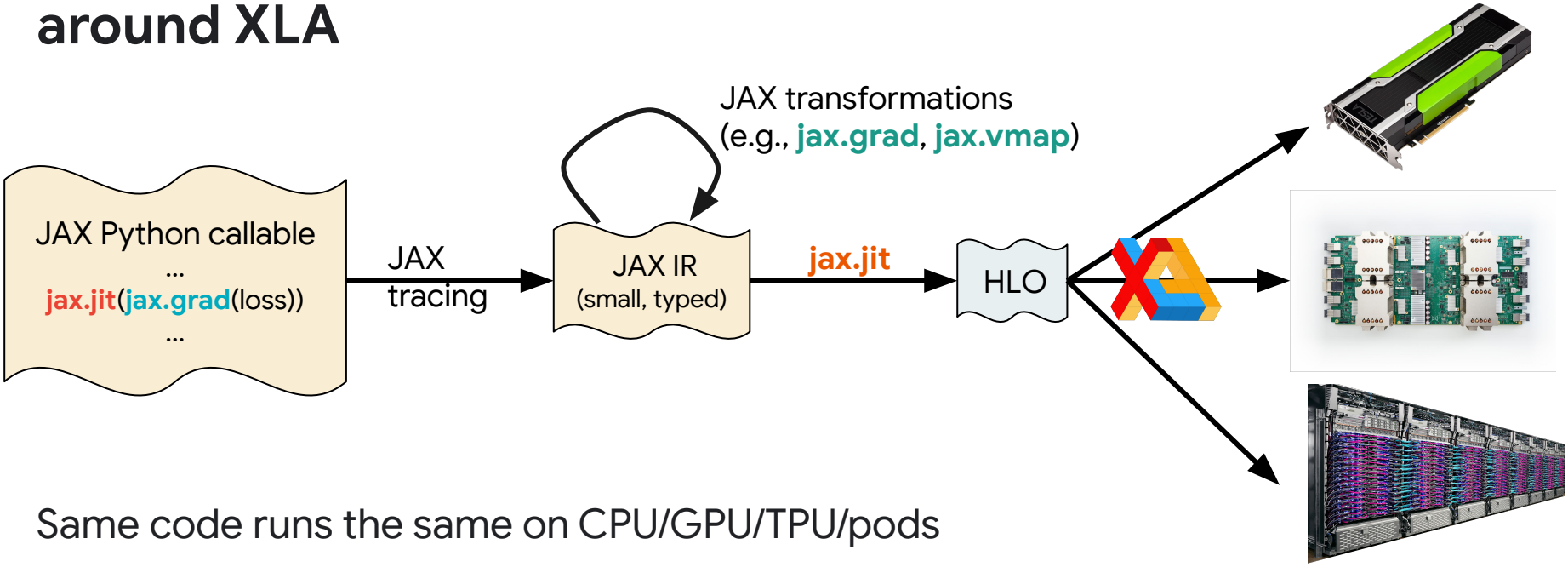
```
{ lambda  ; a.
  let b = log a
      c = log 2.0
      d = div b c
  in [d] }
```

`grad` Automatic differentiation

`vmap/pmap` Batching & parallelization

`jit` Just-In-Time compilation

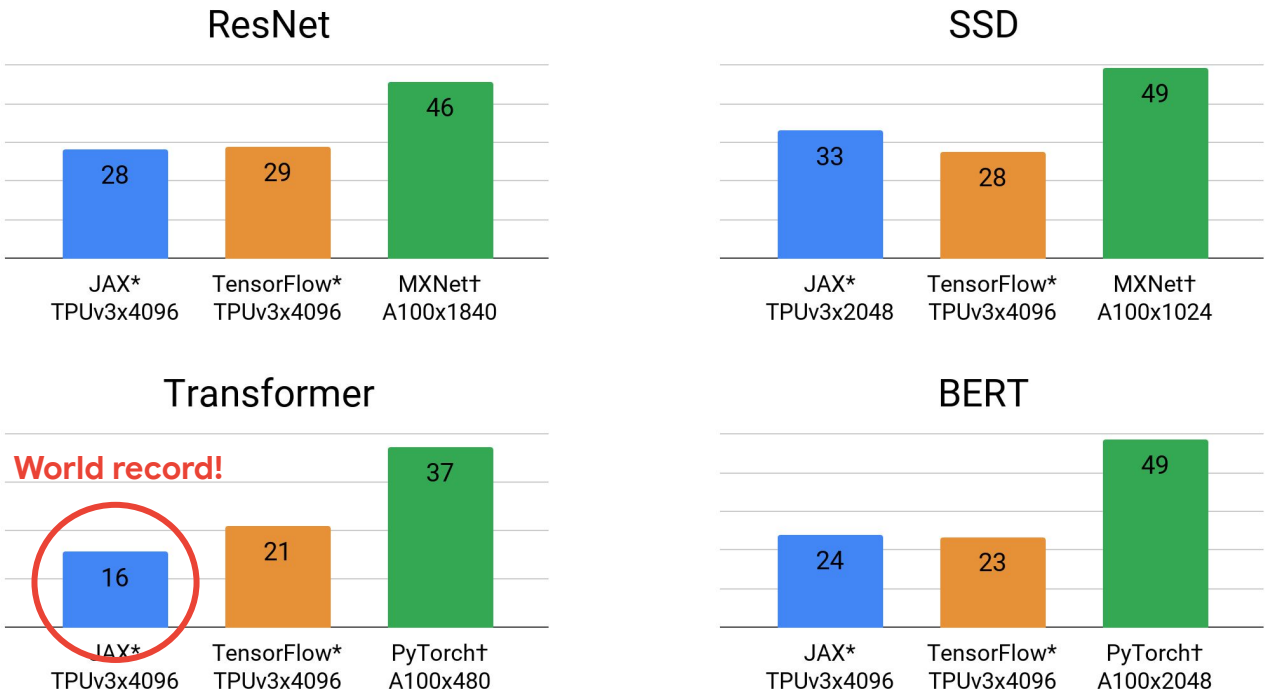
JAX is designed from ground-up around XLA



Same code runs the same on CPU/GPU/TPU/pods

User-control over compilation boundaries (jax.jit)

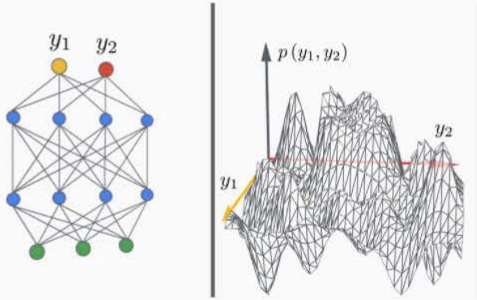
JAX is Fast: MLPerf-2020 results



* Google, Research category
† NVIDIA, Available On-Premise category.

MLPerf v0.7 Training, closed division. Retrieved from www.mlperf.org 1 December 2020, entries 0.7-64, 0.7-65, 0.7-67, 0.7-30, 0.7-33, 0.7-37, 0.7-38.
MLPerf name and logo are trademarks. See www.mlperf.org for more information.

Neural Tangent Kernel

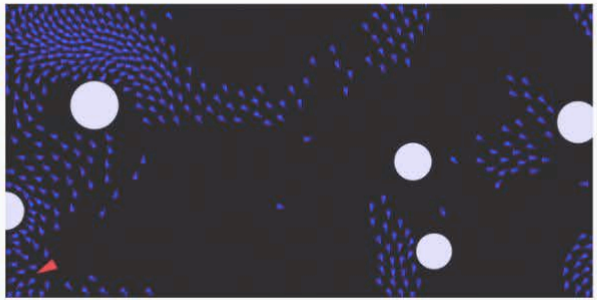


https://ai.googleblog.com/2020/03/fast-and-easy-infinitely-wide-networks.html

Robots!

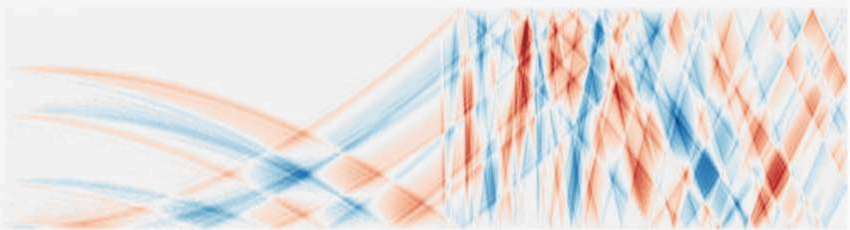
https://arxiv.org/abs/1907.03613

Boids! MD sim!



https://github.com/google/jax-md

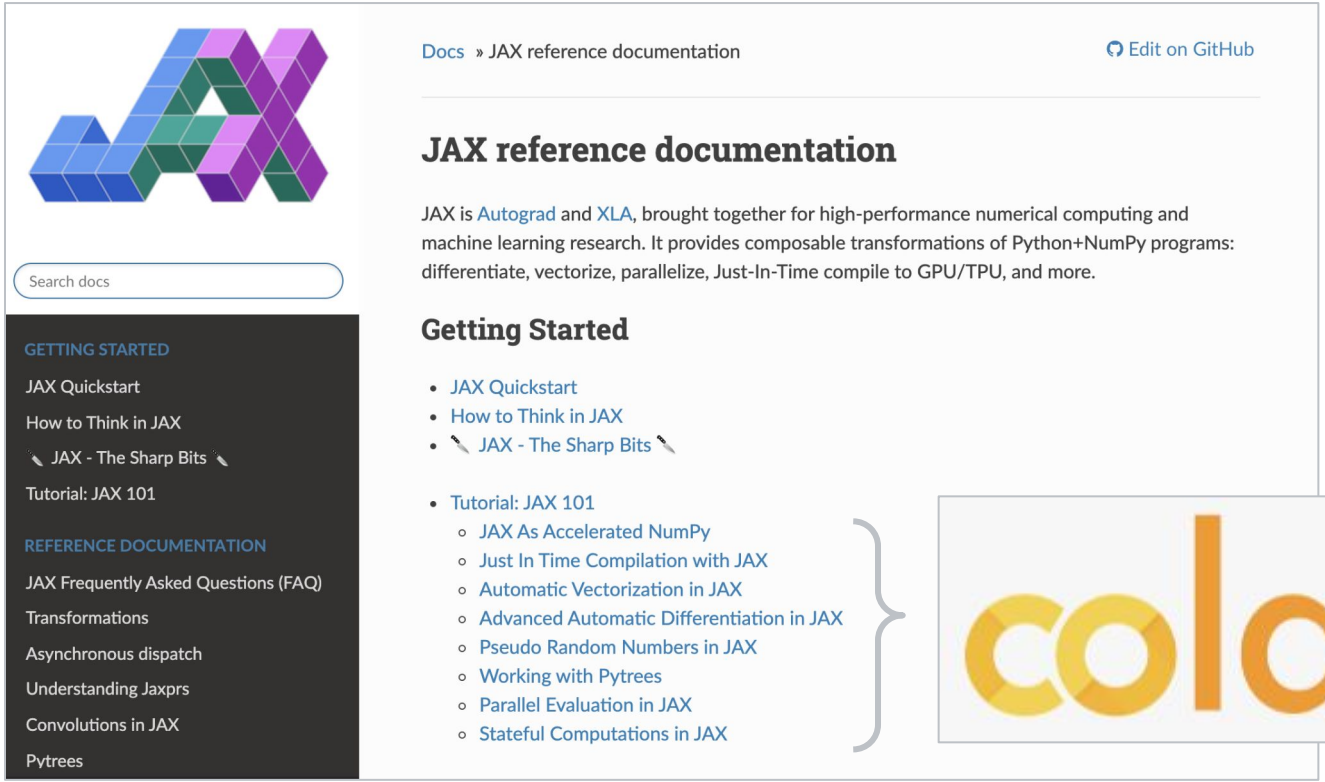
Physics!

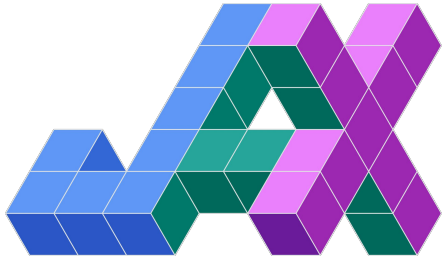


https://arxiv.org/abs/2003.04630
https://github.com/google/jax/blob/master/cloud_tpu_colabs/Wave_Equation.ipynb

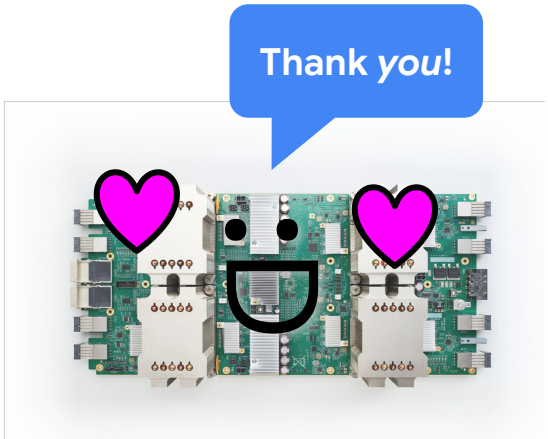
How to get started?

https://jax.readthedocs.io/





http://github.com/google/jax/



mattjj@ frostig@ leary@ dougalm@

phawkins@ skyewm@ jekbradbury@

necula@ apaszke@ jakevdp@

Google Cloud

Thank you.