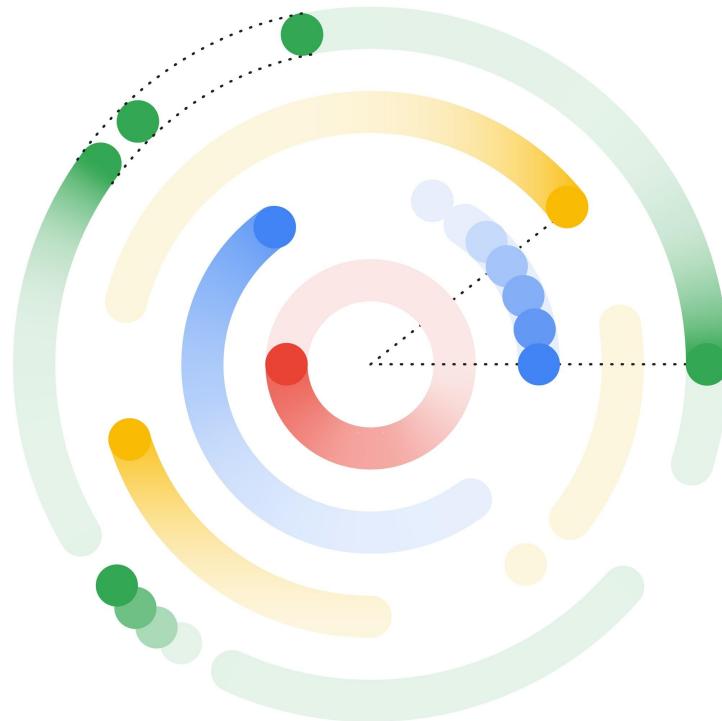


What JAX and Colab Can Offer the Data Science Community

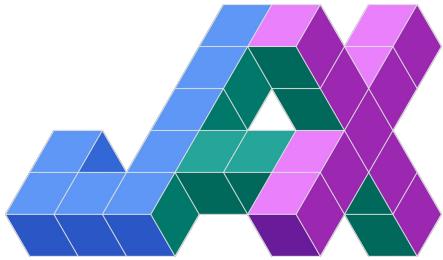
Google Cloud Applied ML Summit
Solving for the future.

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

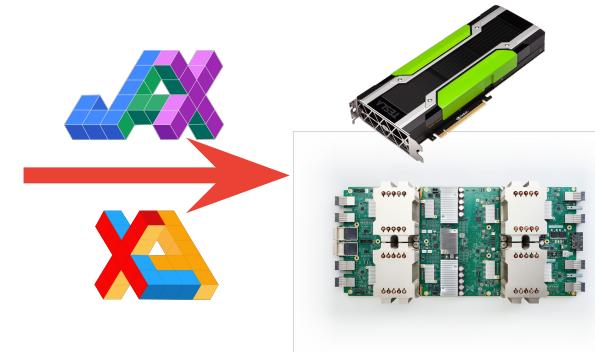
```
import jax.numpy as np

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\n\n\ndef predict(params, inputs):\n    for W, b in params:\n        outputs = np.dot(inputs, W) + b\n        inputs = np.tanh(outputs)\n    return outputs\n\n\ndef loss(params, batch):\n    inputs, targets = batch\n    preds = predict(params, inputs)\n    return np.sum((preds - targets) ** 2)
```



GPU/TPU

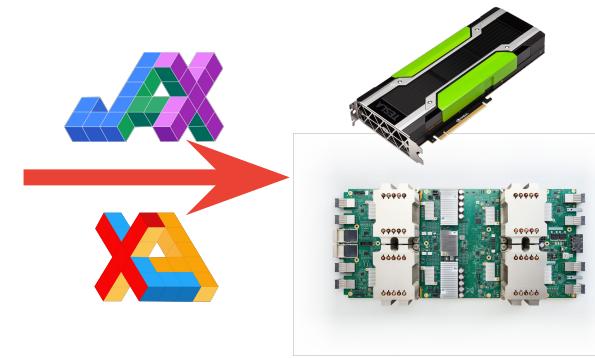
Motivating JAX

```
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

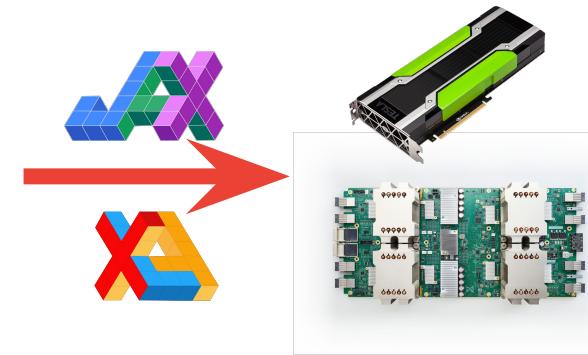
Motivating JAX

```
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

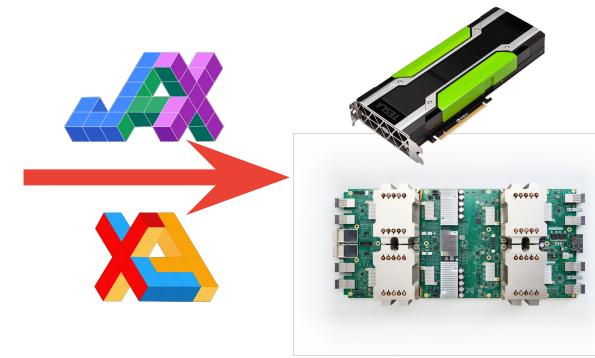
Motivating JAX

```
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

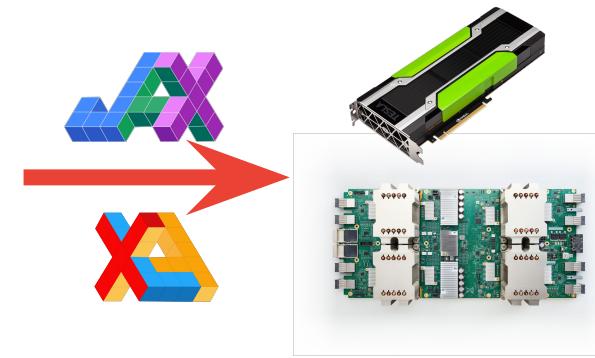
Motivating JAX

```
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 its 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

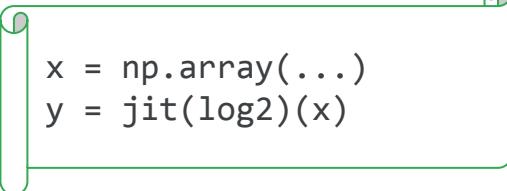
def log2(x):
    ln_x = lax.log(x)
    ln_2 = lax.log(2)
    return ln_x / ln_2
```

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
```

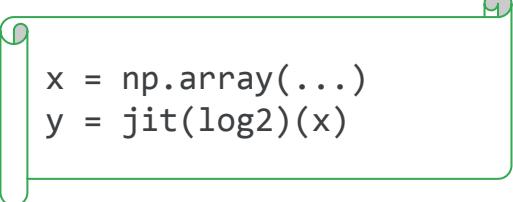


```
x = np.array(...)
y = jit(log2)(x)
```

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
```

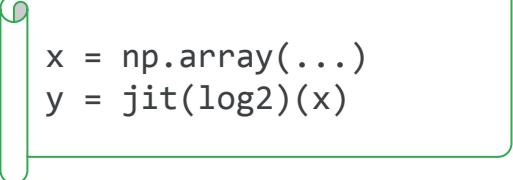
x = np.array(...)
y = jit(log2)(x)

```
{ 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
```



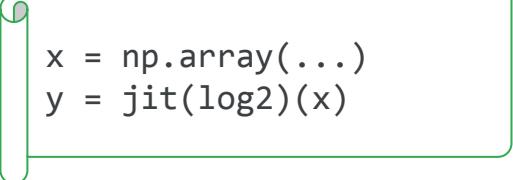
```
x = np.array(...)
y = jit(log2)(x)
```

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

Python function → JAX Intermediate Representation

```
from jax import lax

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



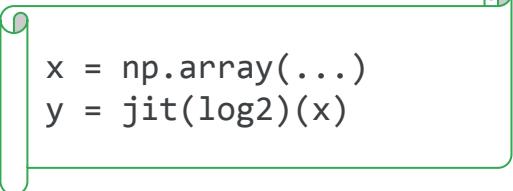
```
x = np.array(...)
y = jit(log2)(x)
```

```
{ 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
```



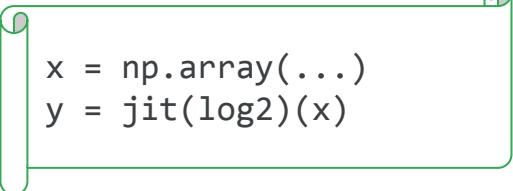
```
x = np.array(...)
y = jit(log2)(x)
```

```
{ 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)
```

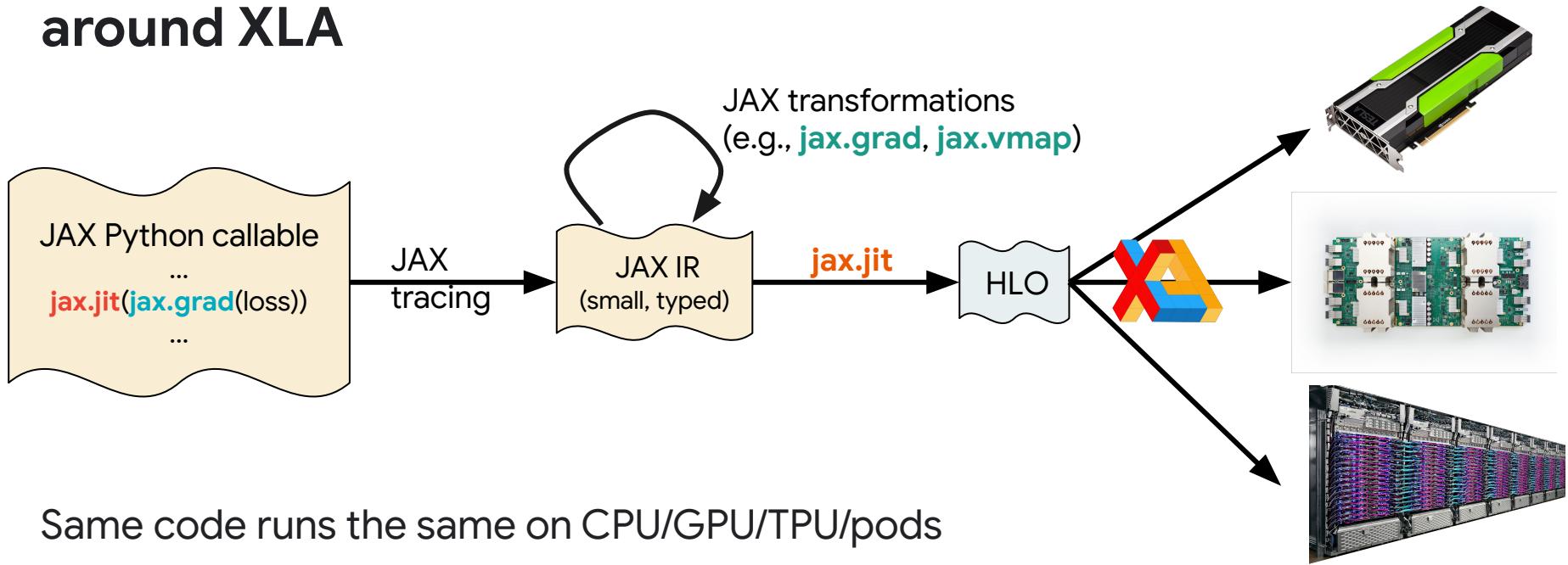
```
{ lambda ; a.
    let b = log a
        c = log 2.0
        d = div b c
    in [d] }
```

[grad](#) Automatic differentiation

[vmap/pmap](#) Batching & parallelization

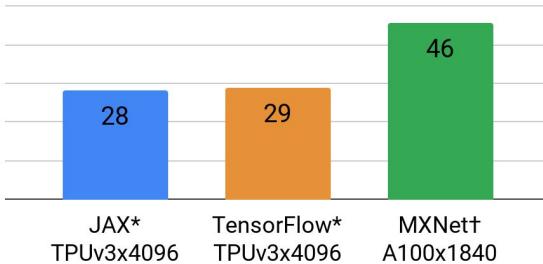
[jit](#) Just-In-Time compilation

JAX is designed from ground-up around XLA

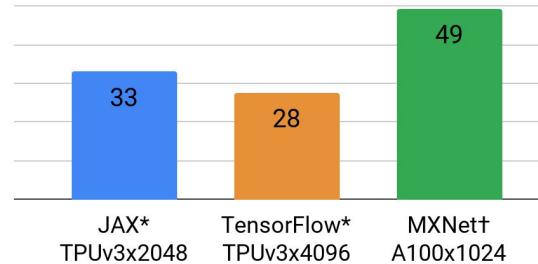


JAX is Fast: MLPerf-2020 results

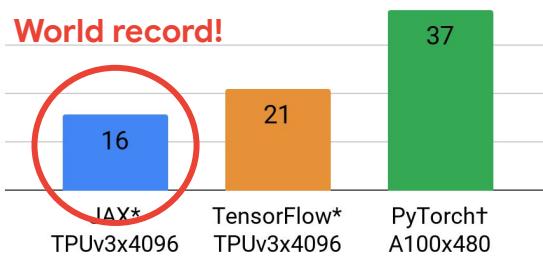
ResNet



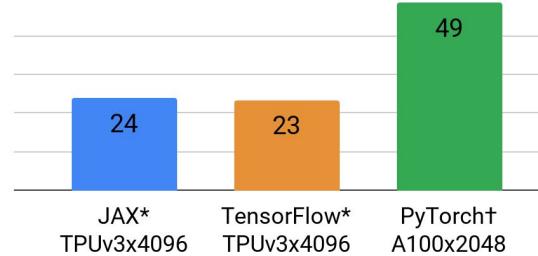
SSD



Transformer



BERT



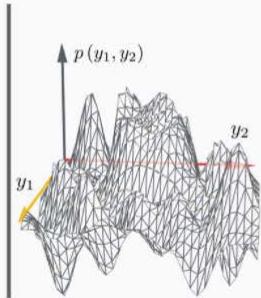
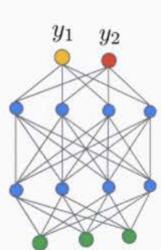
* Google, Research category

† 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.

Google Cloud

Neural Tangent Kernel



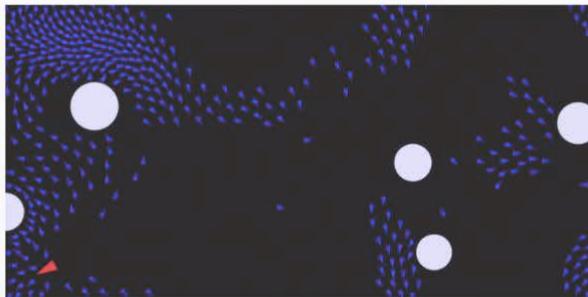
<https://ai.googleblog.com/2020/03/fast-and-easy-infinitely-wide-networks.html>

Robots!



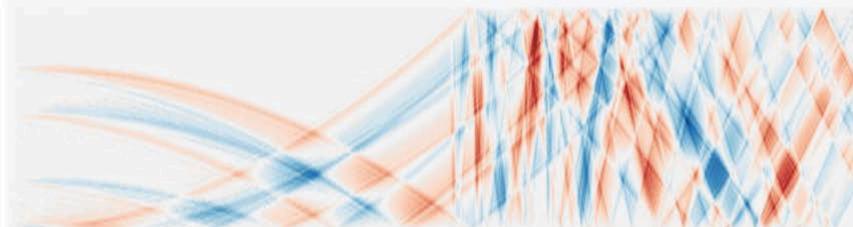
<https://arxiv.org/abs/1907.03613>

Boids! MD sim!



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

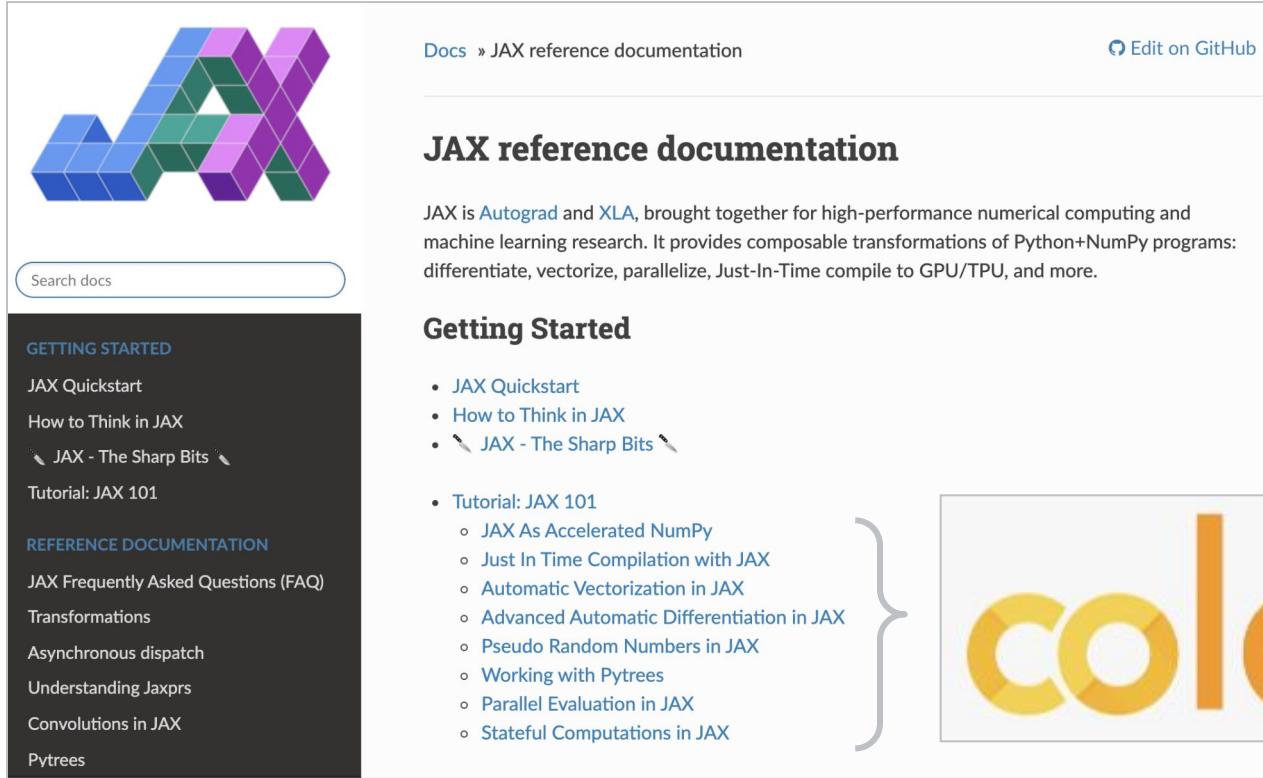
Physics!



<https://arxiv.org/abs/2003.04630>

https://github.com/google/jax/blob/master/cloud_tpu_colabs/Wave_Equation.ipynb

How to get started?



The screenshot shows the JAX reference documentation page. At the top left is a stylized logo of the letters 'JAX' composed of colored 3D blocks (blue, green, purple). To the right of the logo is a search bar containing the placeholder 'Search docs'. On the far right of the header are two links: 'Docs » JAX reference documentation' and 'Edit on GitHub'. Below the header, the main title 'JAX reference documentation' is centered. A brief description follows: 'JAX is Autograd and XLA, brought together for high-performance numerical computing and machine learning research. It provides composable transformations of Python+NumPy programs: differentiate, vectorize, parallelize, Just-In-Time compile to GPU/TPU, and more.' Underneath this, a large section titled 'Getting Started' is shown. It includes a list of links: 'JAX Quickstart', 'How to Think in JAX', 'JAX - The Sharp Bits', 'Tutorial: JAX 101', and 'Tutorial: JAX 101' with its sub-sections: 'JAX As Accelerated NumPy', 'Just In Time Compilation with JAX', 'Automatic Vectorization in JAX', 'Advanced Automatic Differentiation in JAX', 'Pseudo Random Numbers in JAX', 'Working with Pytrees', 'Parallel Evaluation in JAX', and 'Stateful Computations in JAX'. A curly brace on the right side groups the 'Tutorial: JAX 101' section and its sub-sections. To the right of the main content area is a large 'colab' logo.

Docs » JAX reference documentation

Edit on GitHub

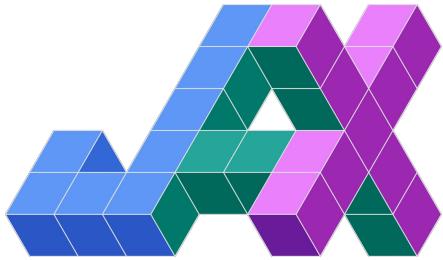
JAX reference documentation

JAX is [Autograd](#) and [XLA](#), brought together for high-performance numerical computing and machine learning research. It provides composable transformations of Python+NumPy programs: differentiate, vectorize, parallelize, Just-In-Time compile to GPU/TPU, and more.

Getting Started

- [JAX Quickstart](#)
- [How to Think in JAX](#)
- [JAX - The Sharp Bits](#)

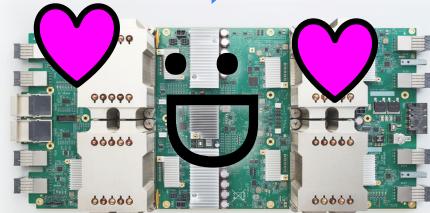
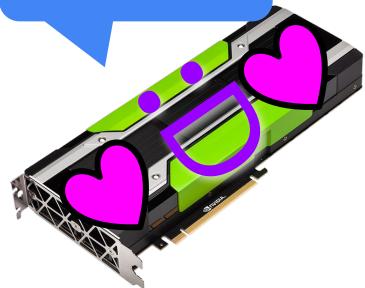
- [Tutorial: JAX 101](#)
 - [JAX As Accelerated NumPy](#)
 - [Just In Time Compilation with JAX](#)
 - [Automatic Vectorization in JAX](#)
 - [Advanced Automatic Differentiation in JAX](#)
 - [Pseudo Random Numbers in JAX](#)
 - [Working with Pytrees](#)
 - [Parallel Evaluation in JAX](#)
 - [Stateful Computations in JAX](#)



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

Thank you!

Thank you!



mattjj@



frostig@



leary@



dougalm@



phawkins@



skyewm@



jekbradbury@



necula@



apaszke@



jakevdp@



Thank you.