Introduction to JAX Workshop

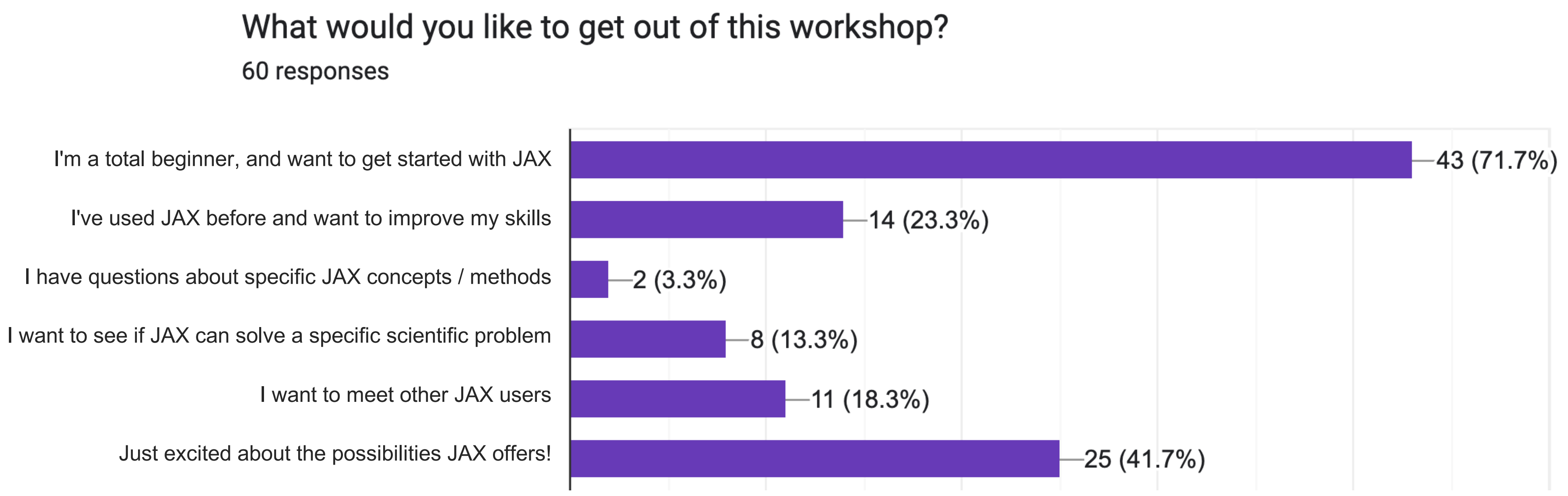
June 25th 2024

Ben Moseley
Paweł Czyż

ETH zürich

ETH AI CENTER

Workshop overview



Workshop overview

14:00-15:30 Introduction to JAX (Ben)

15:30-15:45 Break

15:45-16:45 Advanced concepts in JAX (Pawel)

16:45 onwards
Group discussions
on JAX

Workshop overview

14:00-15:30 Introduction to JAX (Ben)

15:30-15:45 Break

15:45-16:45 Advanced concepts in JAX (Pawel)

16:45 onwards
Group discussions
on JAX

Goals:

1) introduce you to JAX
2) help build a **community** of JAX users at ETH
3) help you **solve** any JAX problems you have in your own work

What is JAX?

JAX = accelerated array computation + program transformation

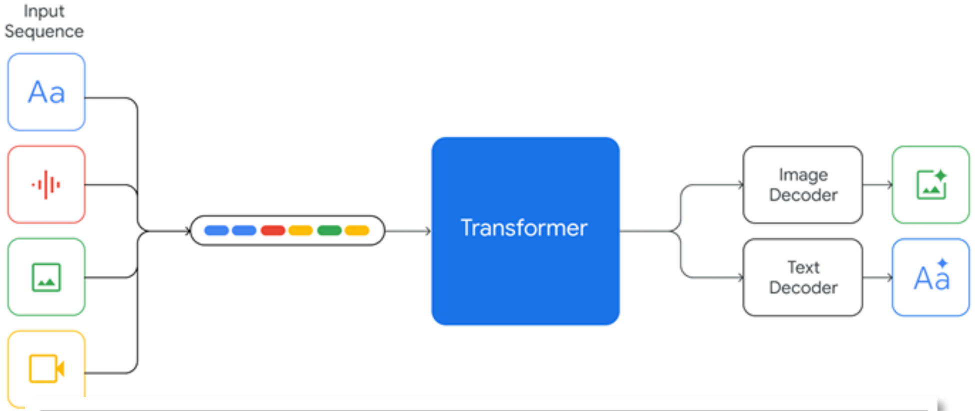
.. Which is incredibly useful for high-performance numerical computing and large-scale (Sci)ML

JAX in ML

Google DeepMind

Gemini: A Family of Highly Capable Multimodal Models

Gemini Team, Google[1]

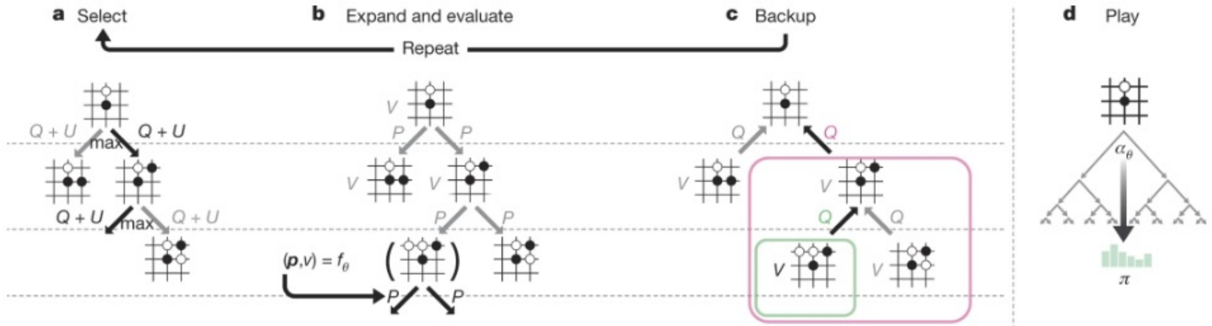


Implementation Frameworks	
Hardware & Software	Hardware: Training was conducted on TPUv4 and TPUv5e (Jouppi et al., 2020, 2023). Software: JAX (Bradbury et al., 2018), ML Pathways (Dean, 2021). JAX allows researchers to leverage the latest generation of hardware, including TPUs, for faster and more efficient training of large models.

Google/ DeepMind, Gemini: A Family of Highly Capable Multimodal Models, ArXiv (2023)

Figure 2: MCTS in AlphaGo Zero.

From: Mastering the game of Go without human knowledge



.pylintrc	Update .pylintrc.	last year
CONTRIBUTING.md	Initial commit.	2 years ago
LICENSE	Initial commit.	2 years ago
MANIFEST.in	Initial commit.	2 years ago
README.md	Add a link to mctx-az.	4 months ago
setup.py	Drop support for python<3.9.	6 months ago
test.sh	Drop support for python<3.9.	6 months ago

29 watching
172 forks
Report repository

Releases 4
mctx 0.0.5 Latest
on Nov 24, 2023
+ 3 releases

Contributors 9

Languages
Python 97.8% Shell 2.2%

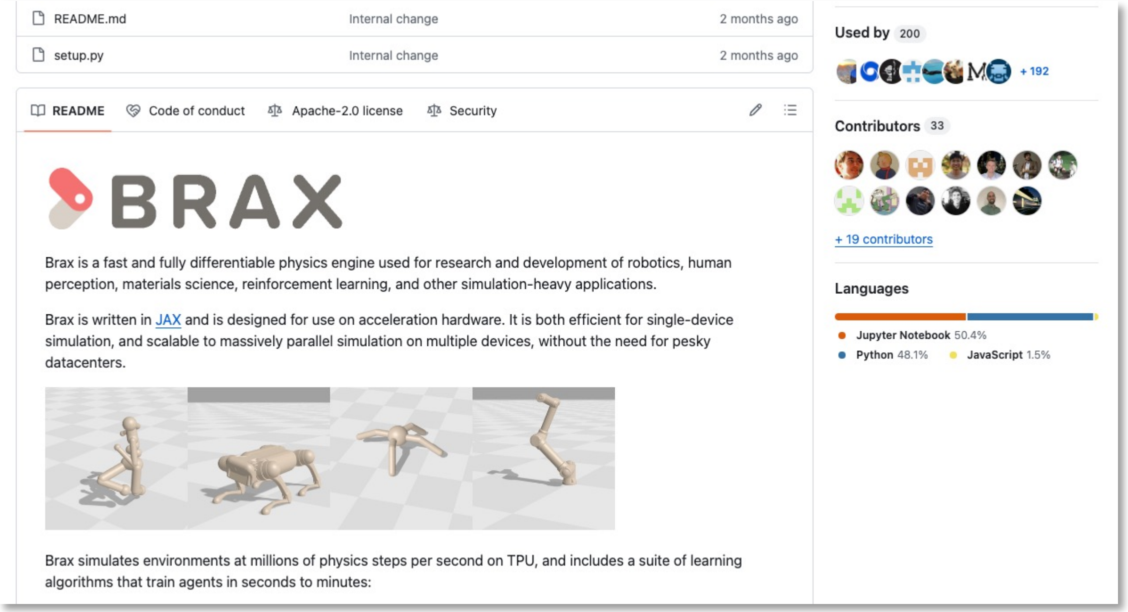
README Apache-2.0 license

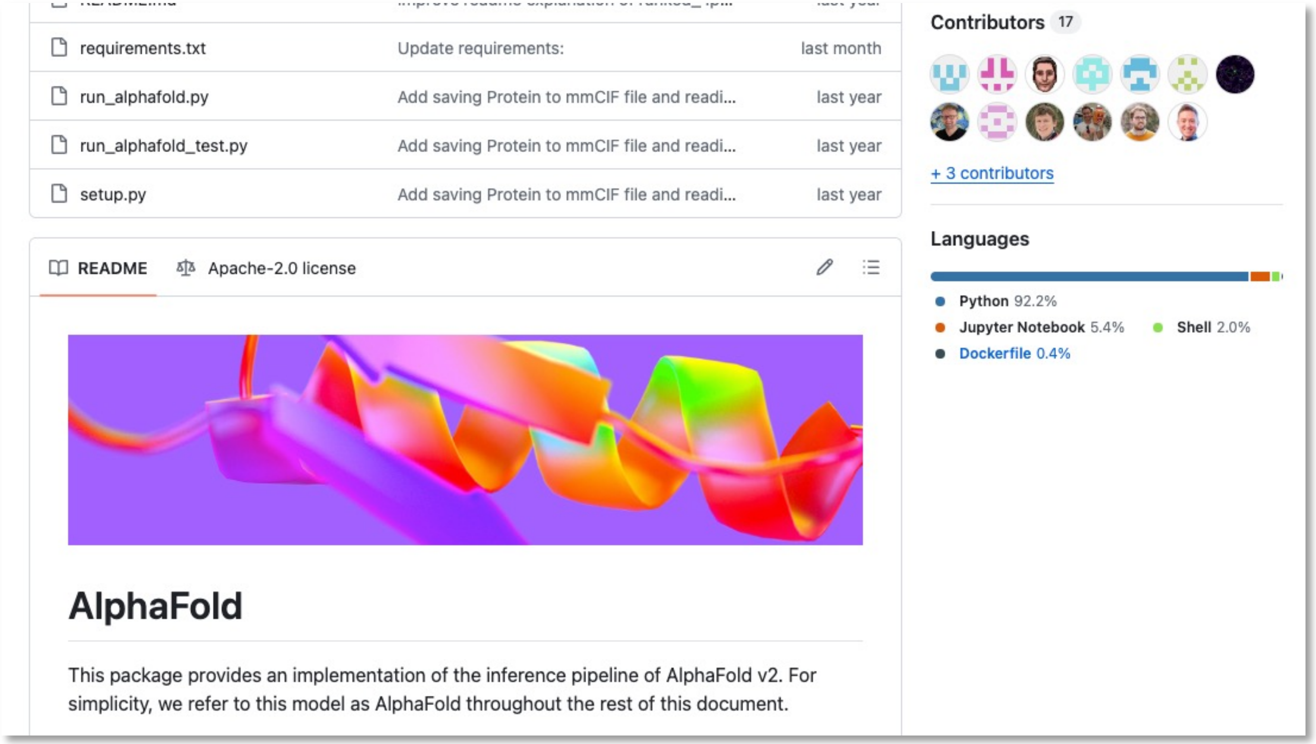
Mctx: MCTS-in-JAX

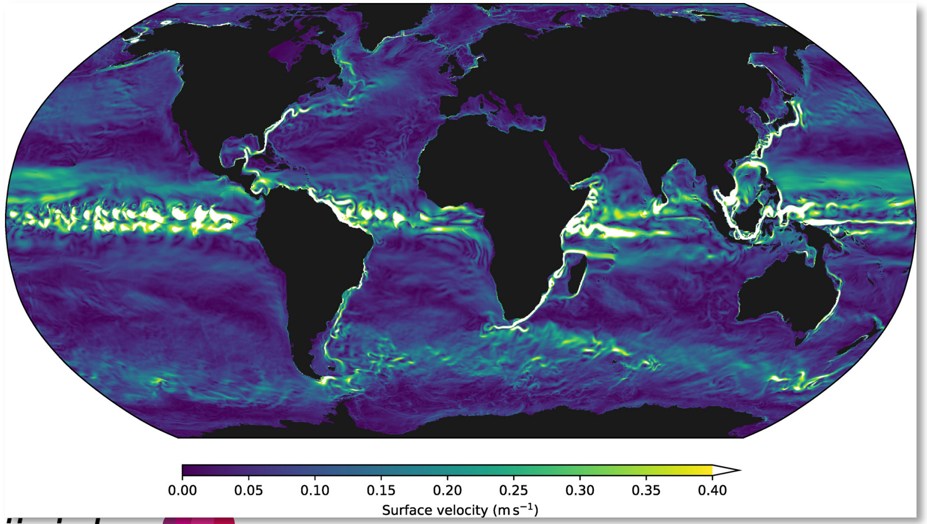
Mctx is a library with a JAX-native implementation of Monte Carlo tree search (MCTS) algorithms such as AlphaZero, MuZero, and Gumbel MuZero. For computation speed up, the implementation fully supports JIT-compilation. Search algorithms in Mctx are defined for and operate on batches of inputs, in parallel. This allows to make the most of the accelerators and enables the algorithms to work with large learned environment models parameterized by deep neural networks.

Silver et al, Mastering the game of Go without human knowledge, Nature (2017)

JAX in scientific computing







← Ocean surface velocity, simulated in 24 hr using 16 NVIDIA A100 GPUs

Hafner et al, Fast, Cheap, and Turbulent - Global Ocean Modeling With GPU Acceleration in Python, Journal of Advances in Modeling Earth Systems (2021)

What is JAX?

JAX = accelerated array computation + program transformation

```
import jax.numpy as jnp
```

- JAX is NumPy on the CPU and GPU!
- JAX uses XLA (Accelerated Linear Algebra) to compile and run NumPy code, *lightning fast*

What is JAX?

JAX = accelerated array computation + program transformation

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

- JAX is NumPy on the CPU and GPU!
- JAX uses XLA (Accelerated Linear Algebra) to compile and run NumPy code, *lightning fast*

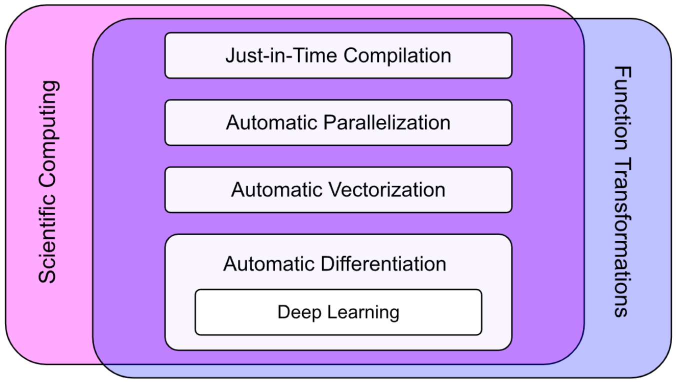


Image credit: AssemblyAI

- JAX can automatically *differentiate* and *parallelise* native Python and NumPy code

JAX = accelerated array computation

JAX is NumPy on the GPU

```python
import numpy as np

A = np.array([[1., 2., 3.],
              [1., 2., 3.],
              [1., 2., 3.],])

x = np.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

```python
import jax.numpy as jnp

A = jnp.array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.],])

x = jnp.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

JAX is NumPy on the GPU

```python
import numpy as np

A = np.array([[1., 2., 3.],
              [1., 2., 3.],
              [1., 2., 3.],])

x = np.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

```python
import jax.numpy as jnp

A = jnp.array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.],])

x = jnp.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

(10,000 x 10,000) (10,000 x 10,000)
NumPy on CPU (Apple M1 Max):
7.22 s ± 109 ms

(10,000 x 10,000) (10,000 x 10,000)
JAX on GPU (NVIDIA RTX 3090):
56.9 ms ± 222 μs (**126x** faster)

JAX is NumPy on the GPU

```python
import numpy as np

A = np.array([[1., 2., 3.],
              [1., 2., 3.],
              [1., 2., 3.],])

x = np.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

(10,000 x 10,000) (10,000 x 10,000)
NumPy on CPU (Apple M1 Max):
7.22 s ± 109 ms

```python
import jax.numpy as jnp

A = jnp.array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.],])

x = jnp.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

(10,000 x 10,000) (10,000 x 10,000)
JAX on GPU (NVIDIA RTX 3090):
56.9 ms ± 222 µs (**126x** faster)

Why is this operation faster on the GPU?

JAX is NumPy on the GPU

```python
import numpy as np

A = np.array([[1., 2., 3.],
              [1., 2., 3.],
              [1., 2., 3.],])

x = np.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

(10,000 x 10,000) (10,000 x 10,000)
NumPy on CPU (Apple M1 Max):
7.22 s ± 109 ms

```python
import jax.numpy as jnp

A = jnp.array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.],])

x = jnp.array([4.,5.,6.])

b = A @ x
print(b)

---
[32. 32. 32.]
```

(10,000 x 10,000) (10,000 x 10,000)
JAX on GPU (NVIDIA RTX 3090):
56.9 ms ± 222 μs (**126x** faster)

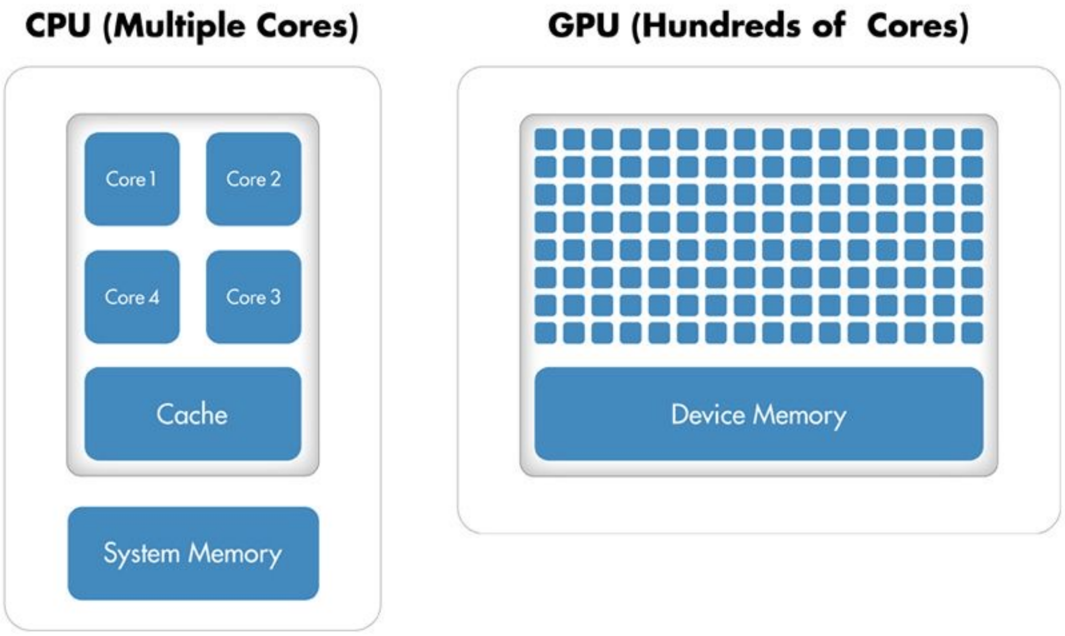


Image credit: MathWorks

Low latency
Ideal for serial processing

High throughput
Ideal for parallel processing

Wave simulation

```python
import numpy as np


def forward(velocity, density, source_i, f0, NX, NY, NSTEPS, DELTAX, DELTAY, DELTAT):

    assert velocity.shape == density.shape == (NX, NY)
    assert source_i.shape == (2,)

    pressure_present = np.zeros((NX, NY))
    pressure_past = np.zeros((NX, NY))

    kronecker_source = np.zeros((NX, NY))
    kronecker_source[source_i[0], source_i[1]] = 1.

    # precompute some arrays
    t0 = 1.2 / f0
    factor = 1e-3
    kappa = density*(velocity**2)
    density_half_x = np.pad(0.5 * (density[1:NX,:]+density[:NX-1,:]), [[0,1],[0,0]], mode="edge")
    density_half_y = np.pad(0.5 * (density[:,1:NY]+density[:,:NY-1]), [[0,0],[0,1]], mode="edge")

    carry = pressure_past, pressure_present

    def single_step(carry, it):
        pressure_past, pressure_present = carry

        t = it*DELTAT

        # compute the first spatial derivatives divided by density
        value_dpressure_dx = np.pad((pressure_present[1:NX,:]-pressure_present[:NX-1,:]) / DELTAX, [[0,1],[0,0]], mode="constant", constant_values=0.)
        value_dpressure_dy = np.pad((pressure_present[:,1:NY]-pressure_present[:,:NY-1]) / DELTAY, [[0,0],[0,1]], mode="constant", constant_values=0.)

        pressure_xx = value_dpressure_dx / density_half_x
        pressure_yy = value_dpressure_dy / density_half_y

        # compute the second spatial derivatives

        value_dpressurexx_dx = np.pad((pressure_xx[1:NX,:]-pressure_xx[:NX-1,:]) / DELTAX, [[1,0],[0,0]], mode="constant", constant_values=0.)
        value_dpressureyy_dy = np.pad((pressure_yy[:,1:NY]-pressure_yy[:,:NY-1]) / DELTAY, [[0,0],[1,0]], mode="constant", constant_values=0.)

        dpressurexx_dx = value_dpressurexx_dx
        dpressureyy_dy = value_dpressureyy_dy

        # add the source (pressure located at a given grid point)
        a = (np.pi**2)*f0*f0

        # Ricker source time function (second derivative of a Gaussian)
        source_term = factor * (1 - 2*a*(t-t0)**2)*np.exp(-a*(t-t0)**2)

        pressure_future =   - pressure_past \
                            + 2 * pressure_present \
                            + DELTAT*DELTAT*(dpressurexx_dx+dpressureyy_dy)*kappa

        pressure_future += DELTAT*DELTAT*(4*np.pi*(velocity**2)*source_term*kronecker_source)# latest seismicCPML

        wavefield = pressure_future

        # move new values to old values (the present becomes the past, the future becomes the present)
        pressure_past = pressure_present
        pressure_present = pressure_future

        carry = pressure_past, pressure_present
        return carry, wavefield

    wavefields = np.zeros((NSTEPS, NX, NY), dtype=float)
    for it in range(NSTEPS):
        carry, w = single_step(carry, it)
        wavefields[it] = w.copy()

    return wavefields
```

Wave simulation

Lots of (element-wise) matrix operations!

```python
import numpy as np


def forward(velocity, density, source_i, f0, NX, NY, NSTEPS, DELTAX, DELTAY, DELTAT):

    assert velocity.shape == density.shape == (NX, NY)
    assert source_i.shape == (2,)

    pressure_present = np.zeros((NX, NY))
    pressure_past = np.zeros((NX, NY))

    kronecker_source = np.zeros((NX, NY))
    kronecker_source[source_i[0], source_i[1]] = 1.

    # precompute some arrays
    t0 = 1.2 / f0
    factor = 1e-3
    kappa = density*(velocity**2)
    density_half_x = np.pad(0.5 * (density[1:NX,:]+density[:NX-1,:]), [[0,1],[0,0]], mode="edge")
    density_half_y = np.pad(0.5 * (density[:,1:NY]+density[:,:NY-1]), [[0,0],[0,1]], mode="edge")

    carry = pressure_past, pressure_present

    def single_step(carry, it):
        pressure_past, pressure_present = carry

        t = it*DELTAT

        # compute the first spatial derivatives divided by density
        value_dpressure_dx = np.pad((pressure_present[1:NX,:]-pressure_present[:NX-1,:]) / DELTAX, [[0,1],[0,0]], mode="constant", constant_values=0.)
        value_dpressure_dy = np.pad((pressure_present[:,1:NY]-pressure_present[:,:NY-1]) / DELTAY, [[0,0],[0,1]], mode="constant", constant_values=0.)

        pressure_xx = value_dpressure_dx / density_half_x
        pressure_yy = value_dpressure_dy / density_half_y

        # compute the second spatial derivatives

        value_dpressurexx_dx = np.pad((pressure_xx[1:NX,:]-pressure_xx[:NX-1,:]) / DELTAX, [[1,0],[0,0]], mode="constant", constant_values=0.)
        value_dpressureyy_dy = np.pad((pressure_yy[:,1:NY]-pressure_yy[:,:NY-1]) / DELTAY, [[0,0],[1,0]], mode="constant", constant_values=0.)

        dpressurexx_dx = value_dpressurexx_dx
        dpressureyy_dy = value_dpressureyy_dy

        # add the source (pressure located at a given grid point)
        a = (np.pi**2)*f0*f0

        # Ricker source time function (second derivative of a Gaussian)
        source_term = factor * (1 - 2*a*(t-t0)**2)*np.exp(-a*(t-t0)**2)

        pressure_future =   - pressure_past \
                            + 2 * pressure_present \
                            + DELTAT*DELTAT*(dpressurexx_dx+dpressureyy_dy)*kappa

        pressure_future += DELTAT*DELTAT*(4*np.pi*(velocity**2)*source_term*kronecker_source)# latest seismicCPML

        wavefield = pressure_future

        # move new values to old values (the present becomes the past, the future becomes the present)
        pressure_past = pressure_present
        pressure_present = pressure_future

        carry = pressure_past, pressure_present
        return carry, wavefield

    wavefields = np.zeros((NSTEPS, NX, NY), dtype=float)
    for it in range(NSTEPS):
        carry, w = single_step(carry, it)
        wavefields[it] = w.copy()

    return wavefields
```

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


def forward(velocity, density, source_i, f0, NX, NY, NSTEPS, DELTAX, DELTAY, DELTAT):

    assert velocity.shape == density.shape == (NX, NY)
    assert source_i.shape == (2,)

    pressure_present = jnp.zeros((NX, NY))
    pressure_past = jnp.zeros((NX, NY))

    kronecker_source = jnp.zeros((NX, NY))
    kronecker_source = kronecker_source.at[source_i[0], source_i[1]].set(1.)

    # precompute some arrays
    t0 = 1.2 / f0
    factor = 1e-3
    kappa = density*(velocity**2)
    density_half_x = jnp.pad(0.5 * (density[1:NX,:]+density[:NX-1,:]), [[0,1],[0,0]], mode="edge")
    density_half_y = jnp.pad(0.5 * (density[:,1:NY]+density[:,:NY-1]), [[0,0],[0,1]], mode="edge")

    carry = pressure_past, pressure_present

    def single_step(carry, it):
        pressure_past, pressure_present = carry

        t = it*DELTAT

        # compute the first spatial derivatives divided by density
        value_dpressure_dx = jnp.pad((pressure_present[1:NX,:]-pressure_present[:NX-1,:]) / DELTAX, [[0,1],[0,0]], mode="constant", constant_values=0.)
        value_dpressure_dy = jnp.pad((pressure_present[:,1:NY]-pressure_present[:,:NY-1]) / DELTAY, [[0,0],[0,1]], mode="constant", constant_values=0.)

        pressure_xx = value_dpressure_dx / density_half_x
        pressure_yy = value_dpressure_dy / density_half_y

        # compute the second spatial derivatives

        value_dpressurexx_dx = jnp.pad((pressure_xx[1:NX,:]-pressure_xx[:NX-1,:]) / DELTAX, [[1,0],[0,0]], mode="constant", constant_values=0.)
        value_dpressureyy_dy = jnp.pad((pressure_yy[:,1:NY]-pressure_yy[:,:NY-1]) / DELTAY, [[0,0],[1,0]], mode="constant", constant_values=0.)

        dpressurexx_dx = value_dpressurexx_dx
        dpressureyy_dy = value_dpressureyy_dy

        # add the source (pressure located at a given grid point)
        a = (jnp.pi**2)*f0*f0

        # Ricker source time function (second derivative of a Gaussian)
        source_term = factor * (1 - 2*a*(t-t0)**2)*jnp.exp(-a*(t-t0)**2)

        pressure_future =   - pressure_past \
                            + 2 * pressure_present \
                            + DELTAT*DELTAT*(dpressurexx_dx+dpressureyy_dy)*kappa

        pressure_future += DELTAT*DELTAT*(4*jnp.pi*(velocity**2)*source_term*kronecker_source)# latest seismicCPML

        wavefield = pressure_future

        # move new values to old values (the present becomes the past, the future becomes the present)
        pressure_past = pressure_present
        pressure_present = pressure_future

        carry = pressure_past, pressure_present
        return carry, wavefield

    _, wavefields = jax.lax.scan(single_step, carry, jnp.arange(NSTEPS))

    return wavefields
```

```python
import numpy as np


def forward(velocity, density, source_i, f0, NX, NY, NSTEPS, DELTAX, DELTAY, DELTAT):

    assert velocity.shape == density.shape == (NX, NY)
    assert source_i.shape == (2,)

    pressure_present = np.zeros((NX, NY))
    pressure_past = np.zeros((NX, NY))

    kronecker_source = np.zeros((NX, NY))
    kronecker_source[source_i[0], source_i[1]] = 1.

    # precompute some arrays
    t0 = 1.2 / f0
    factor = 1e-3
    kappa = density*(velocity**2)
    density_half_x = np.pad(0.5 * (density[1:NX,:]+density[:NX-1,:]), [[0,1],[0,0]], mode="edge")
    density_half_y = np.pad(0.5 * (density[:,1:NY]+density[:,:NY-1]), [[0,0],[0,1]], mode="edge")

    carry = pressure_past, pressure_present

    def single_step(carry, it):
        pressure_past, pressure_present = carry

        t = it*DELTAT

        # compute the first spatial derivatives divided by density
        value_dpressure_dx = np.pad((pressure_present[1:NX,:]-pressure_present[:NX-1,:]) / DELTAX, [[0,1],[0,0]], mode="constant", constant_values=0.)
        value_dpressure_dy = np.pad((pressure_present[:,1:NY]-pressure_present[:,:NY-1]) / DELTAY, [[0,0],[0,1]], mode="constant", constant_values=0.)

        pressure_xx = value_dpressure_dx / density_half_x
        pressure_yy = value_dpressure_dy / density_half_y

        # compute the second spatial derivatives

        value_dpressurexx_dx = np.pad((pressure_xx[1:NX,:]-pressure_xx[:NX-1,:]) / DELTAX, [[1,0],[0,0]], mode="constant", constant_values=0.)
        value_dpressureyy_dy = np.pad((pressure_yy[:,1:NY]-pressure_yy[:,:NY-1]) / DELTAY, [[0,0],[1,0]], mode="constant", constant_values=0.)

        dpressurexx_dx = value_dpressurexx_dx
        dpressureyy_dy = value_dpressureyy_dy

        # add the source (pressure located at a given grid point)
        a = (np.pi**2)*f0*f0

        # Ricker source time function (second derivative of a Gaussian)
        source_term = factor * (1 - 2*a*(t-t0)**2)*np.exp(-a*(t-t0)**2)

        pressure_future =   - pressure_past \
                            + 2 * pressure_present \
                            + DELTAT*DELTAT*(dpressurexx_dx+dpressureyy_dy)*kappa

        pressure_future += DELTAT*DELTAT*(4*np.pi*(velocity**2)*source_term*kronecker_source)# latest seismicCPML

        wavefield = pressure_future

        # move new values to old values (the present becomes the past, the future becomes the present)
        pressure_past = pressure_present
        pressure_present = pressure_future

        carry = pressure_past, pressure_present
        return carry, wavefield

    wavefields = np.zeros((NSTEPS, NX, NY), dtype=float)
    for it in range(NSTEPS):
        carry, w = single_step(carry, it)
        wavefields[it] = w.copy()

    return wavefields
```

Wave simulation

NumPy on **CPU** (Apple M1 Max): 8.06 s ± 54.7 ms

JAX (jit compiled) on **CPU** (Apple M1 Max): 1.58 s ± 11.6 ms
(**5x** faster)

JAX (jit compiled) on **GPU** (NVIDIA RTX 3090): 65.5 ms ± 30.2 μs
(**123x** faster)

Live coding examples

Follow along here:

SCAN ME

What is JAX?

JAX = accelerated array computation + program transformation

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

- JAX is NumPy on the CPU and GPU!
- JAX uses XLA (Accelerated Linear Algebra) to compile and run NumPy code, *lightning fast*

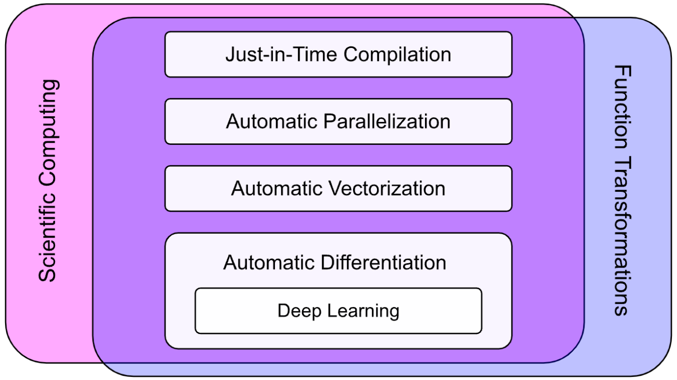


Image credit: AssemblyAI

- JAX can automatically *differentiate* and *parallelise* native Python and NumPy code

JAX = program transformation

What is a program transformation?

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

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

What is a program transformation?

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

def f(x):
    return x**2

dfdx = jax.grad(f)# this returns a python function!
```

What is a program transformation?

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

def f(x):
    return x**2

dfdx = jax.grad(f)# this returns a python function!

x = jnp.array(10.)

print(x)
print(dfdx(x))

---
10.0
20.0
```

What is a program transformation?

```
import jax
import jax.numpy as jnp

def f(x):
    return x**2

dfdx = jax.grad(f)# this returns a python function!

x = jnp.array(10.)

print(x)
print(dfdx(x))

---
10.0
20.0
```

Step 1: convert Python function into a simple intermediate language (jaxpr)

```
print(jax.make_jaxpr(f)(x))

---
{ lambda ; a:f32[]. let b:f32[] = integer_pow[y=2] a in (b,) }
```

What is a program transformation?

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

def f(x):
    return x**2

dfdx = jax.grad(f)# this returns a python function!

x = jnp.array(10.)

print(x)
print(dfdx(x))

---
10.0
20.0
```

Step 1: convert Python function into a simple intermediate language (jaxpr)

```python
print(jax.make_jaxpr(f)(x))

---
{ lambda ; a:f32[]. let b:f32[] = integer_pow[y=2] a in (b,) }
```

Step 2: apply transformation (e.g. return the corresponding gradient function)

```python
print(jax.make_jaxpr(dfdx)(x))

---
{ lambda ; a:f32[]. let
    _:f32[] = integer_pow[y=2] a
    b:f32[] = integer_pow[y=1] a
    c:f32[] = mul 2.0 b
    d:f32[] = mul 1.0 c
  in (d,) }
```

What is a program transformation?

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

def f(x):
    return x**2

dfdx = jax.grad(f)# this returns a python function!

x = jnp.array(10.)

print(x)
print(dfdx(x))

---
10.0
20.0
```

Program transformation =

Transform one **program** to another **program**

- Treats programs as **data**
- Aka **meta-programming**

Program transformations are composable

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

def f(x):
    return x**2

dfdx = jax.grad(f)# this returns a python function!

d2fdx2 = jax.grad(dfdx)# transformations are composable!

x = jnp.array(10.)

print(x)
print(d2fdx2(x))

---
10.0
2.0
```

- We can **arbitrarily compose** program transformations in JAX!
- This allows highly **sophisticated** workflows to be developed

Autodifferentiation in JAX

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

def f(x):
    return jnp.sum(x**2)

x = jnp.arange(5.)

g = jax.grad(f)# returns function which computes gradient
j = jax.jacfwd(f)# returns function which computes Jacobian
j = jax.jacrev(f)# returns function which computes Jacobian
h = jax.hessian(f)# returns function which computes Hessian

print(g(x))
print(h(x))

# vector-Jacobian product
fval, vjp = jax.vjp(f, x)# returns function output and function which computes vjp at x
vjp_val = vjp(1.)

# Jacobian-vector product
v = jnp.ones_like(x)
fval, jvp_val = jax.jvp(f, (x,), (v,))# returns function output and jvp at x

---
[0. 2. 4. 6. 8.]

[[2. 0. 0. 0. 0.]
 [0. 2. 0. 0. 0.]
 [0. 0. 2. 0. 0.]
 [0. 0. 0. 2. 0.]
 [0. 0. 0. 0. 2.]]
```

- JAX has many autodifferentiation capabilities
- **all** are based on compositions of **vjp** and **jvp** (i.e. reverse- and forward- mode autodiff)

Other function transformations

$f(x) \rightarrow dfdx(x)$ is not the only function transformation we could make!

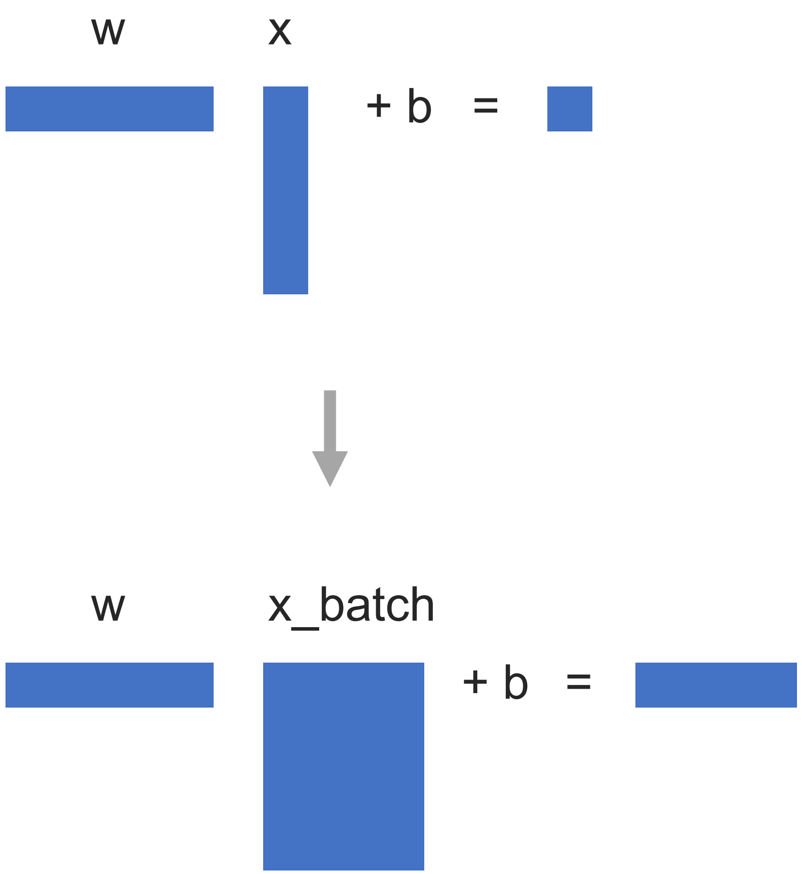
- What **other** function transformations can you imagine?

Automatic vectorisation

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

def f(w, b, x):
    y = jnp.dot(w, x) + b
    return y

x = jnp.array([1., 2.])
w = jnp.array([2., 4.])
b = jnp.array(1.)

print(f(w, b, x))
```

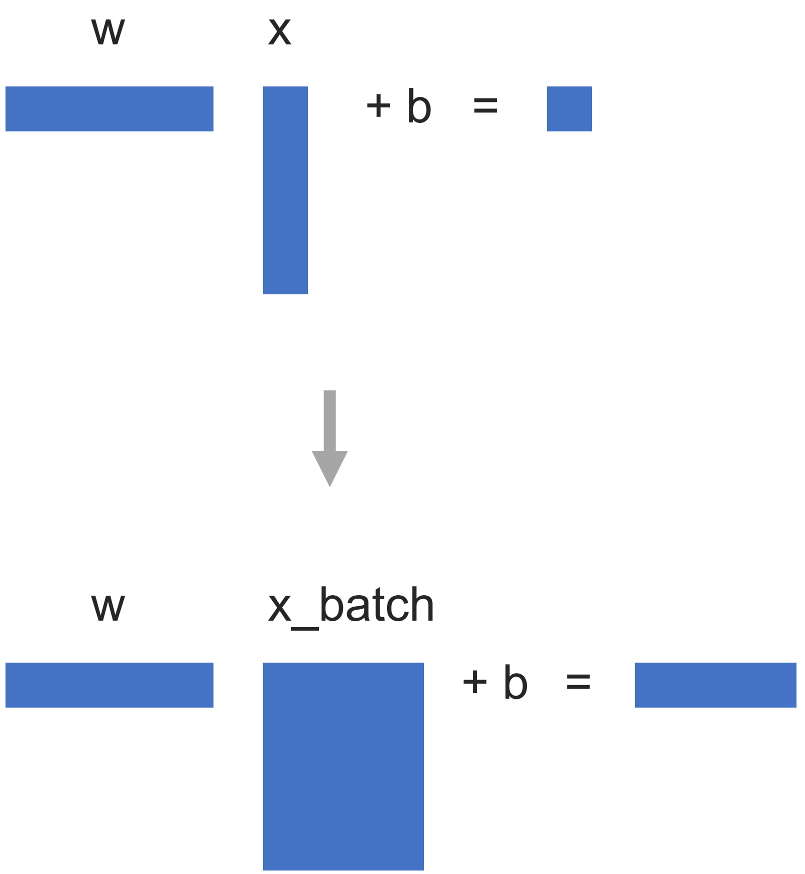
- **Vectorisation** is another type of function transformation

  = parallelise the function across many inputs (on a single CPU or GPU)



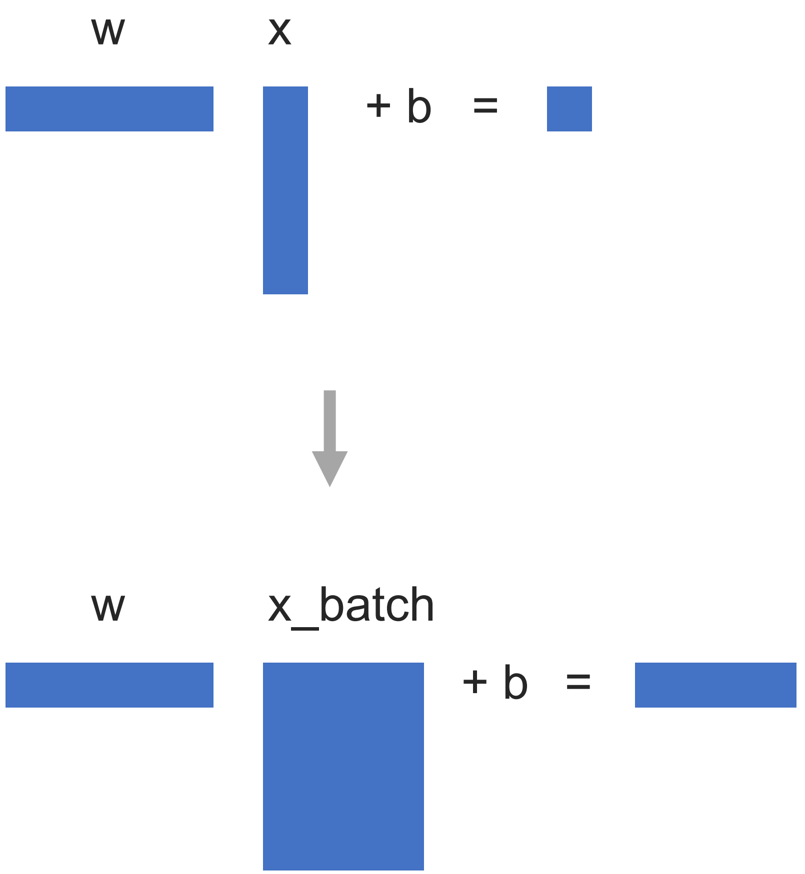
Automatic vectorisation

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

def f(w, b, x):
    y = jnp.dot(w, x) + b
    return y

x = jnp.array([1., 2.])
w = jnp.array([2., 4.])
b = jnp.array(1.)

print(f(w, b, x))

# vectorise function across first dimension of x
f_batch = jax.vmap(f, in_axes=(None, None, 0))
```



Automatic vectorisation

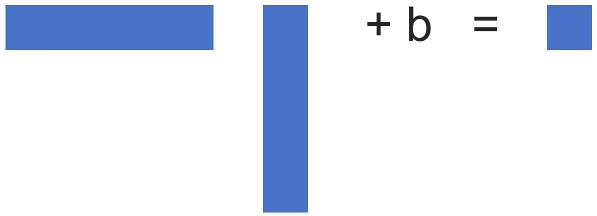
```python
import jax
import jax.numpy as jnp

def f(w, b, x):
    y = jnp.dot(w, x) + b
    return y

x = jnp.array([1., 2.])
w = jnp.array([2., 4.])
b = jnp.array(1.)

print(f(w, b, x))

# vectorise function across first dimension of x
f_batch = jax.vmap(f, in_axes=(None, None, 0))

x_batch = jnp.array([[1., 2.],
                     [3., 4.],
                     [5., 6.]])
print(f_batch(w, b, x_batch))

---
11.0
[11. 23. 35.]
```



Automatic vectorisation

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

def f(w, b, x):
    y = jnp.dot(w, x) + b
    return y

x = jnp.array([1., 2.])
w = jnp.array([2., 4.])
b = jnp.array(1.)

print(f(w, b, x))

# vectorise function across first dimension of x
f_batch = jax.vmap(f, in_axes=(None, None, 0))

x_batch = jnp.array([[1., 2.],
                     [3., 4.],
                     [5., 6.]])
print(f_batch(w, b, x_batch))

---
11.0
[11. 23. 35.]
```

```
{ lambda ; a:f32[2] b:f32[] c:f32[2]. let
    d:f32[] = dot_general[
      dimension_numbers=(([0], [0]), ([], []))
      preferred_element_type=float32
    ] a c
    e:f32[] = convert_element_type[new_dtype=float32 weak_type=False] b
    f:f32[] = add d e
  in (f,) }
```

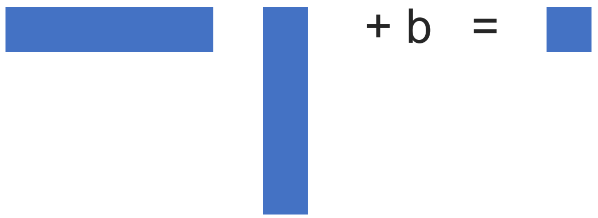



```
{ lambda ; a:f32[2] b:f32[] c:f32[3,2]. let
    d:f32[3] = dot_general[
      dimension_numbers=(([0], [1]), ([], []))
      preferred_element_type=float32
    ] a c
    e:f32[] = convert_element_type[new_dtype=float32 weak_type=False] b
    f:f32[3] = add d e
  in (f,) }
```

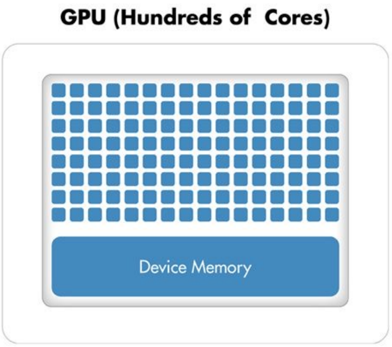
Automatic vectorisation

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

def f(w, b, x):
    y = jnp.dot(w, x) + b
    return y

x = jnp.array([1., 2.])
w = jnp.array([2., 4.])
b = jnp.array(1.)

print(f(w, b, x))

# vectorise function across first dimension of x
f_batch = jax.vmap(f, in_axes=(None, None, 0))

x_batch = jnp.array([[1., 2.],
                     [3., 4.],
                     [5., 6.]])
print(f_batch(w, b, x_batch))

---
11.0
[11. 23. 35.]
```

```
{ lambda ; a:f32[2] b:f32[] c:f32[2]. let
    d:f32[] = dot_general[
      dimension_numbers=(([0], [0]), ([], []))
      preferred_element_type=float32
    ] a c
    e:f32[] = convert_element_type[new_dtype=float32 weak_type=False] b
    f:f32[] = add d e
  in (f,) }
```




```
{ lambda ; a:f32[2] b:f32[] c:f32[3,2]. let
    d:f32[3] = dot_general[
      dimension_numbers=(([0], [1]), ([], []))
      preferred_element_type=float32
    ] a c
    e:f32[] = convert_element_type[new_dtype=float32 weak_type=False] b
    f:f32[3] = add d e
  in (f,) }
```



Much faster than a Python for loop!

Just-in-time compilation

```
import jax

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

- **Compilation** is another type of function transformation

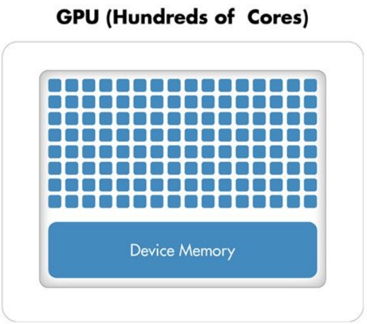
  = rewrite your code to be faster

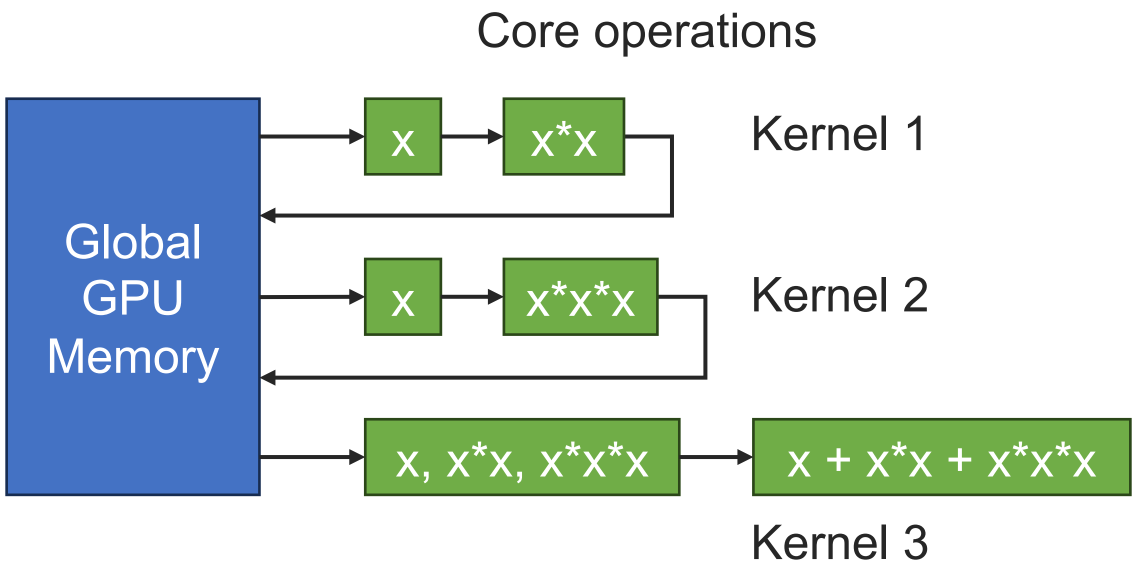
Just-in-time compilation

```python
import jax

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

- **Compilation** is another type of function transformation

  = rewrite your code to be faster



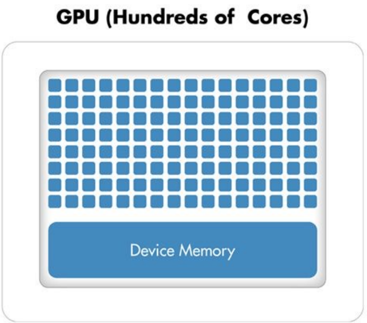


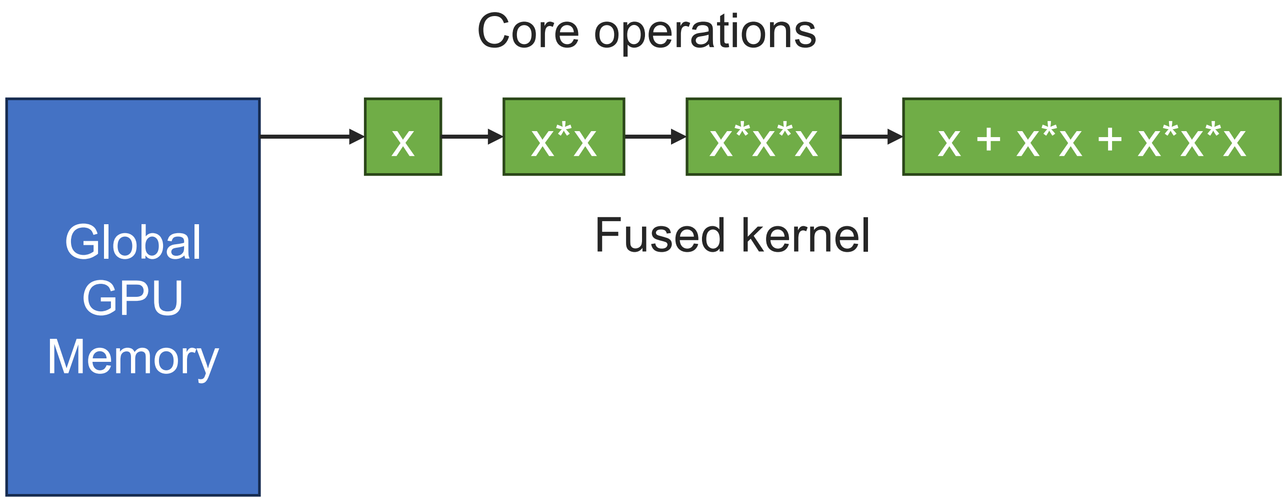
Just-in-time compilation

```python
import jax

def f(x):
    return x + x*x + x*x*x

jit_f = jax.jit(f)# compile function
```

- **Compilation** is another type of function transformation

  = rewrite your code to be faster





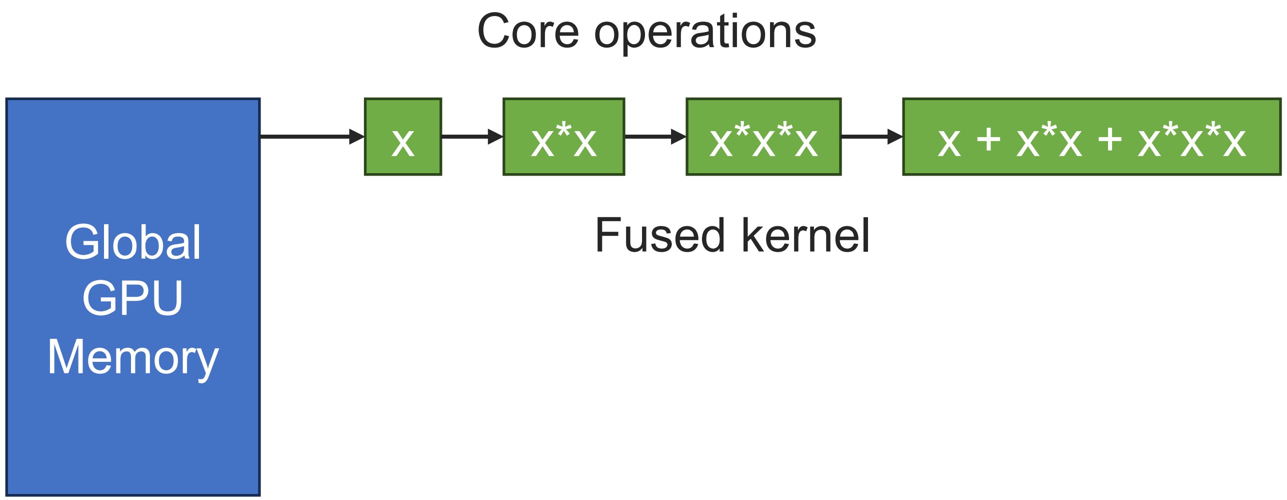
Just-in-time compilation

```
import jax

def f(x):
    return x + x*x + x*x*x

jit_f = jax.jit(f)# compile function

key = jax.random.key(0)
x = jax.random.normal(key, (1000,1000))
%timeit f(x).block_until_ready()
%timeit jit_f(x).block_until_ready()

---
870 µs ± 19.7 µs per loop
117 µs ± 253 ns per loop
```

8x faster!

- **Compilation** is another type of function transformation

  = rewrite your code to be faster



Just-in-time compilation

```
import jax

def f(x):
    return x + x*x + x*x*x

jit_f = jax.jit(f)# compile function

key = jax.random.key(0)
x = jax.random.normal(key, (1000,1000))
%timeit f(x).block_until_ready()
%timeit jit_f(x).block_until_ready()

---
870 µs ± 19.7 µs per loop
117 µs ± 253 ns per loop
```

8x faster!

- **Compilation** is another type of function transformation

  = rewrite your code to be faster

- XLA (accelerated linear algebra) is used for CPU / GPU compilation

- Function is compiled **first time it is called** (i.e. “just-in-time”)

  = upfront cost!

JAX ecosystem

Optimisation

Optax

JAXopt

Optimistix

Neural networks

Flax

Trax

Equinox

Jraph

ML

MaxText (LLMs)

RLax

EasyLM

Scenic

NumPyro

Scientific computing + ML

BRAX

JAX-MD

Diffrax

Lineax

jax.scipy

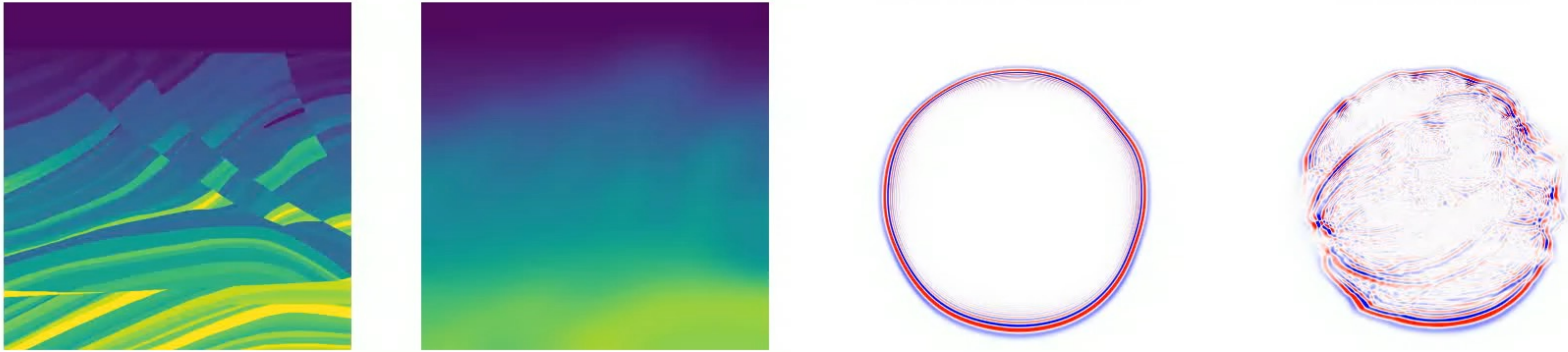
JAX-CFD

DeepXDE

For more: github.com/n2cholas/awesome-jax

Optimisation with Optax

True velocity | Estimated velocity | Estimated wavefield | True wavefield



```python
def loss(velocity, true_wavefield):
    estimated_wavefield = forward(velocity)
    return jnp.mean((estimated_wavefield-true_wavefield)**2)

# Initialize optimizer.
optimizer = optax.adam(learning_rate=1e-1)
opt_state = optimizer.init(velocity)

# A simple gradient descent loop.
for _ in range(10000):
    grads = jax.grad(loss)(velocity, true_wavefield)
    updates, opt_state = optimizer.update(grads, opt_state)
    velocity = optax.apply_updates(velocity, updates)
```

JAX - the sharp bits 🔪

🔪 **Pure functions**: JAX transforms are designed to work on pure functions

🔪 🔪 **Static shapes**: JAX transforms require all shapes to be known **in advance**

🔪 **Out-of-place updates**: JAX only allows out-of-place array updates

🔪 **Random numbers**: JAX requires us to handle RNG explicitly

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