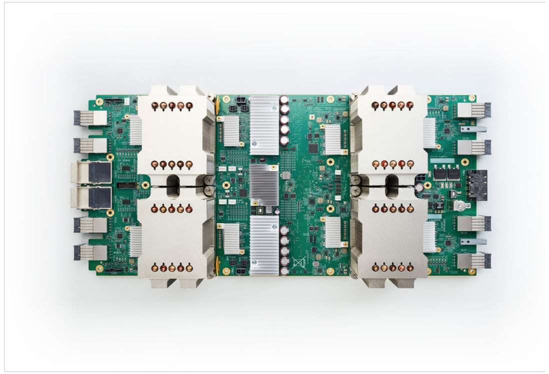
# Scientific computing with JAX and Dex

Adam Paszke
on behalf of the JAX team

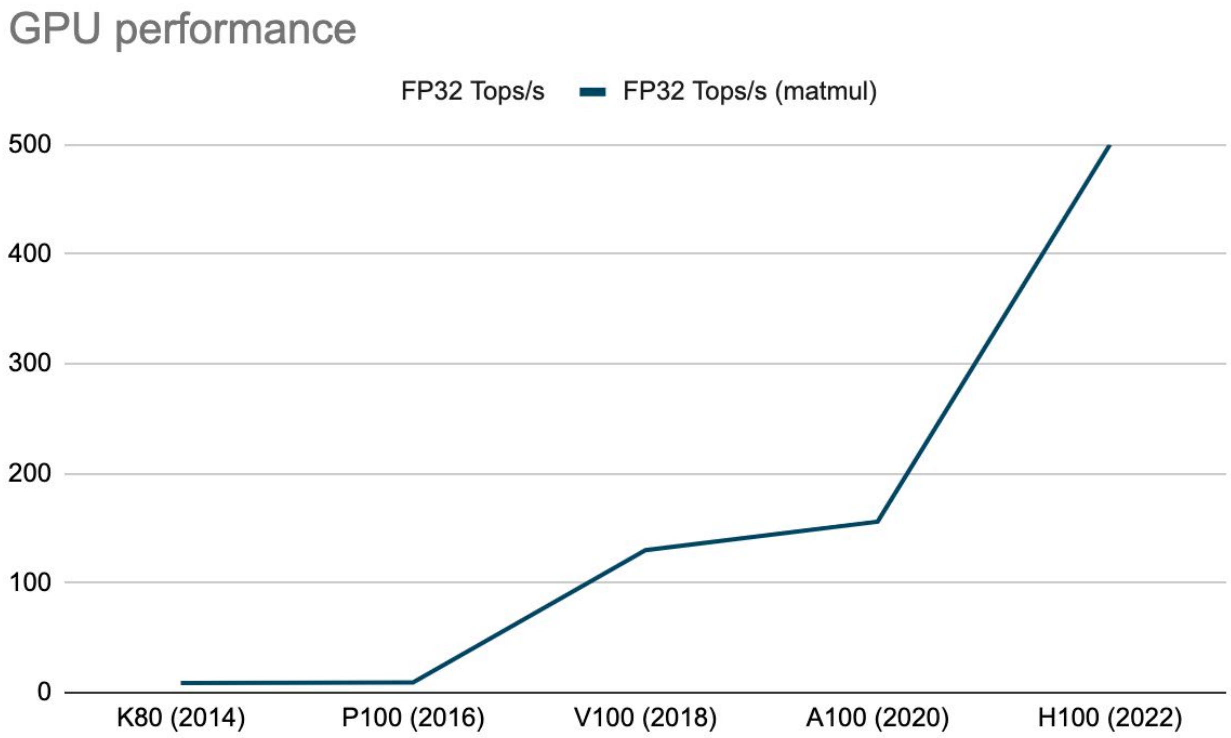
Google Research

# The workhorse of modern HPC

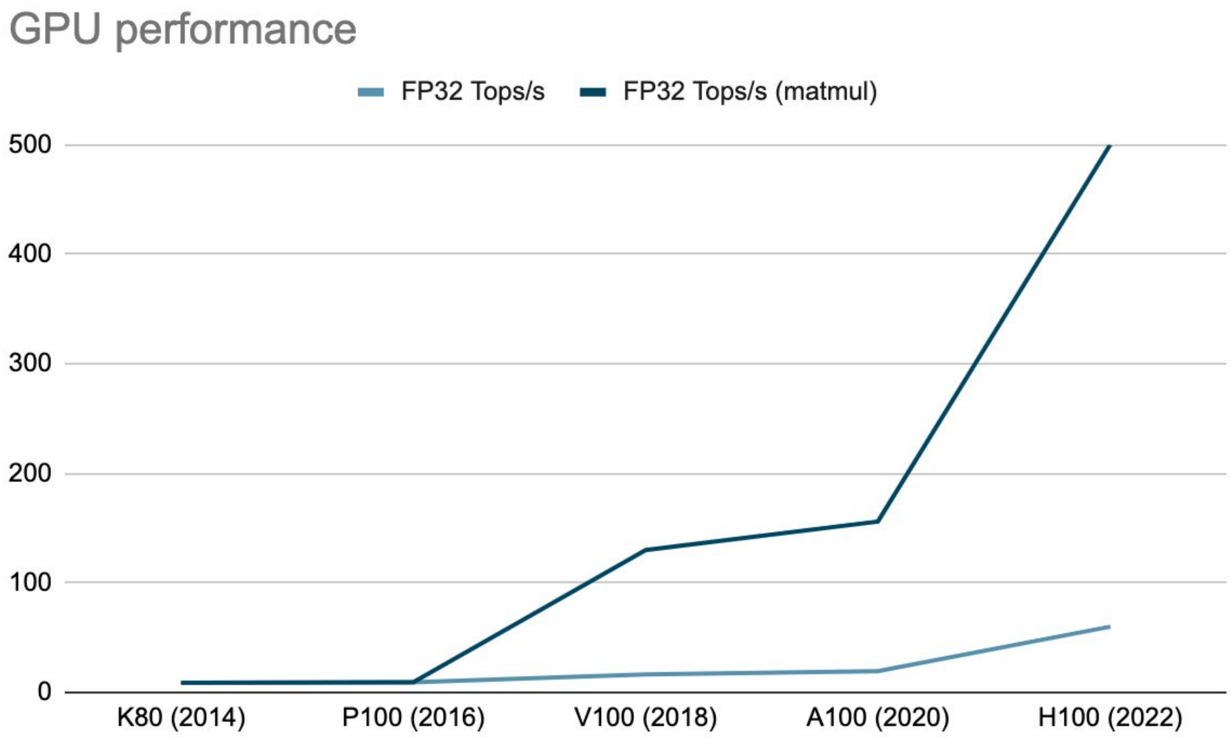
## Accelerators



# The evolution of accelerator hardware



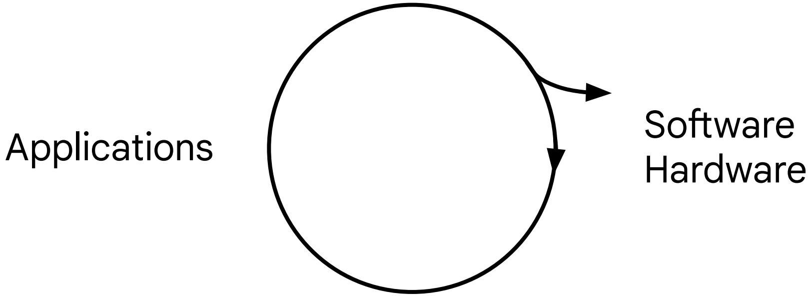
# The evolution of accelerator hardware



# The hardware lottery

Hardware and software shapes research!

If you work on problems resembling matrix multiplication
you will be >5x more productive!

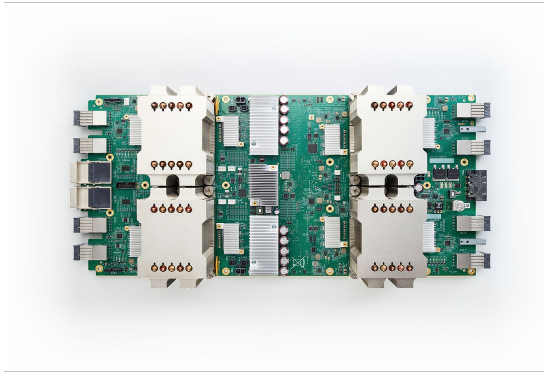


🏆 Where do I claim my winning ticket?

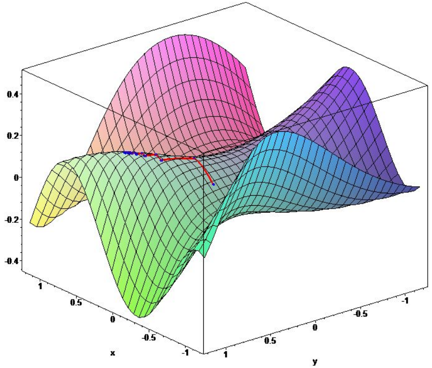
JAX

# Background: the success of first-order array libraries

## Accelerators



## Autodiff

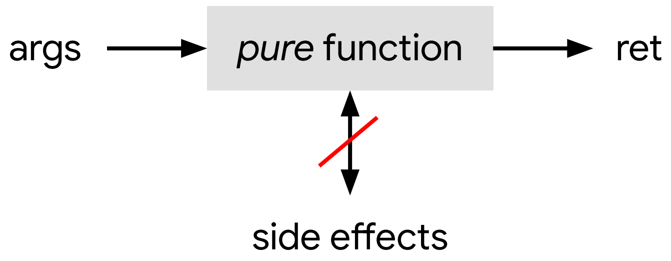


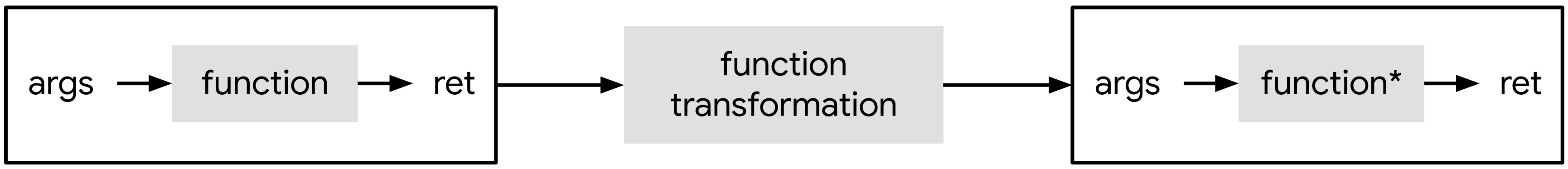

Autodiff only sees *and outputs* a sequential composition of *opaque* parallel programs.

🤯 Standard sequential autodiff *gives us efficient parallel programs out of the box!*

JAX is an extensible system for
composable function transformations
of Python + Numpy code.

# Function transformations




```python
def half(f):
  return lambda *args: f(*args) / 2

def add(x, y):
  return x + y
```

```python
>>> add(2, 4)
6.0
>>> half(add)(2, 4)
3.0
>>> half(half(add))(2, 4)
1.5
```

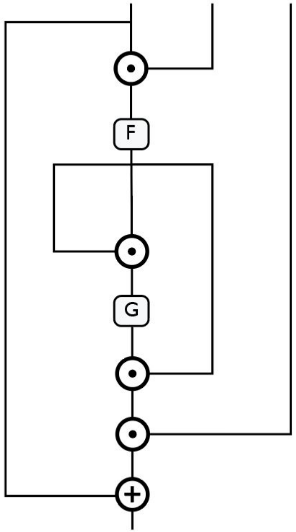
Decorator syntax:

```python
@half
def add(x, y):
  return x + y
```

# JAX features / transforms

| | |
|---|---|
| **Acceleration** | `jax.numpy` |
| **Automatic differentiation** | `jax.grad`, `jax.jet`, `jax.jacfwd`, `jax.jacbwd`, `jax.hessian`, `jax.checkpoint` |
| **Batching** | `jax.vmap`, `jax.xmap` |
| **Acceleration / JIT-compilation** | `jax.jit` |
| **Scaling** | `jax.pjit`, `jax.xmap` |

# Scaling

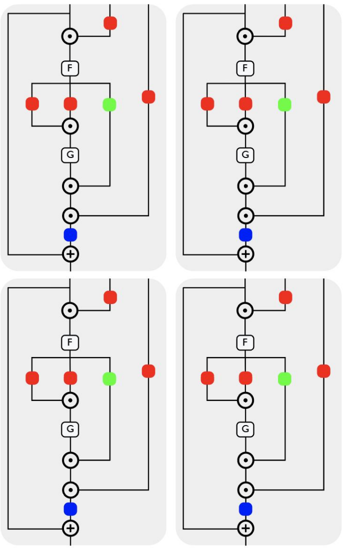



Single device program

+


Input/output device assignment

→

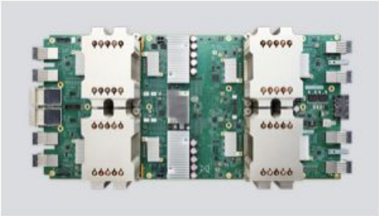



Distributed program

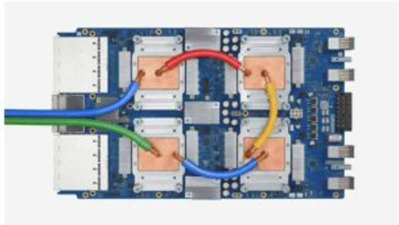
**Collective operations**

# Tensor Processing Units

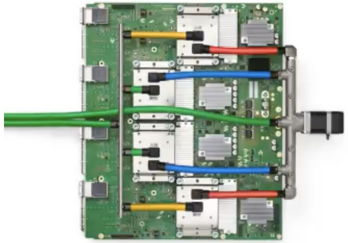
https://cloud.google.com/tpu



TPU v2
180 TeraFlop, 64 GB

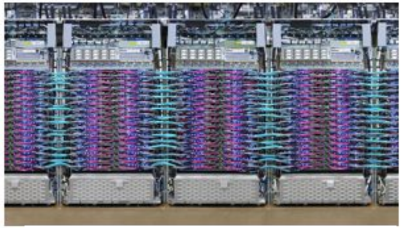


TPU v3
420 TeraFlop, 128 GB

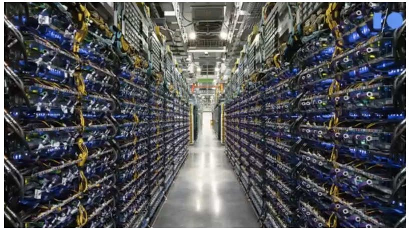


TPU v4

TPU v2 Pod
11.5 PetaFlop



TPU v3 Pod
100+ PetaFlop

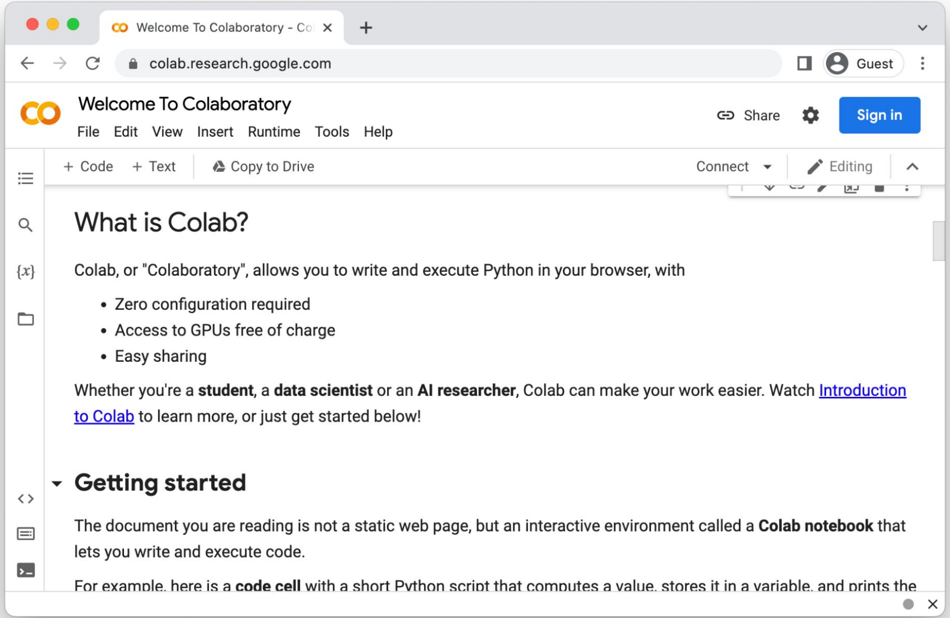


TPU v4 Pod
1000+ PetaFlop

Google Research

# Scaling (a little less)

Colab → TPU runtime → 8 devices!



# It's all composable!

Solving dy/dt= f(y(t)) with an ODE solver uses a while loop over steps:

$y_0 \rightarrow$ step $\rightarrow y_1 \rightarrow$ step $\rightarrow y_2 \rightarrow$ step $\rightarrow \dots \rightarrow$ step $\rightarrow y_{n-1} \rightarrow$ step $\rightarrow y_n$

Each step (of an implicit solver) computes a Jacobian of f:

$y_k \rightarrow$

$$z_0 = y_0$$

$$\dots$$

$$z_{k+1} = z_k - (I + Jac(f)(z_k))^{-1}(z_k + f(z_k) - y_0)$$

$$\dots$$

$$y_1 = z_m$$

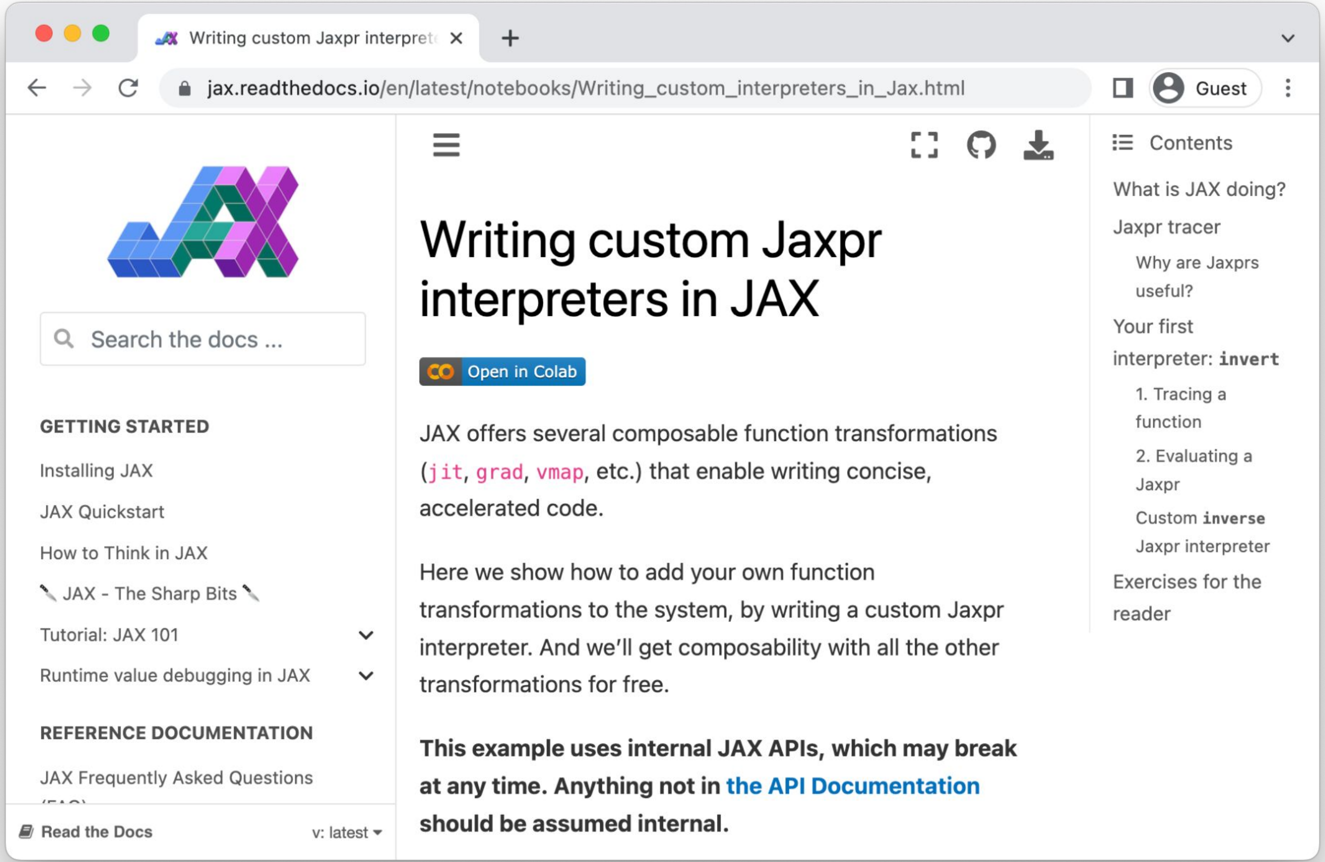
$\rightarrow y_{k+1}$

$f = \nabla \phi$

(f is often defined as the gradient of a potential)

Implies: we need *third-order* autodiff!

```
pjit(grad(vmap(diffeqsolve(jacfwd(grad(phi))))))
```


Writing custom Jaxpr interprete ×

jax.readthedocs.io/en/latest/notebooks/Writing_custom_interpreters_in_Jax.html

Guest

Search the docs ...

GETTING STARTED

Installing JAX

JAX Quickstart

How to Think in JAX

🔪 JAX - The Sharp Bits 🔪

Tutorial: JAX 101

Runtime value debugging in JAX

REFERENCE DOCUMENTATION

JAX Frequently Asked Questions

Read the Docs v: latest

# Writing custom Jaxpr interpreters in JAX

Open in Colab

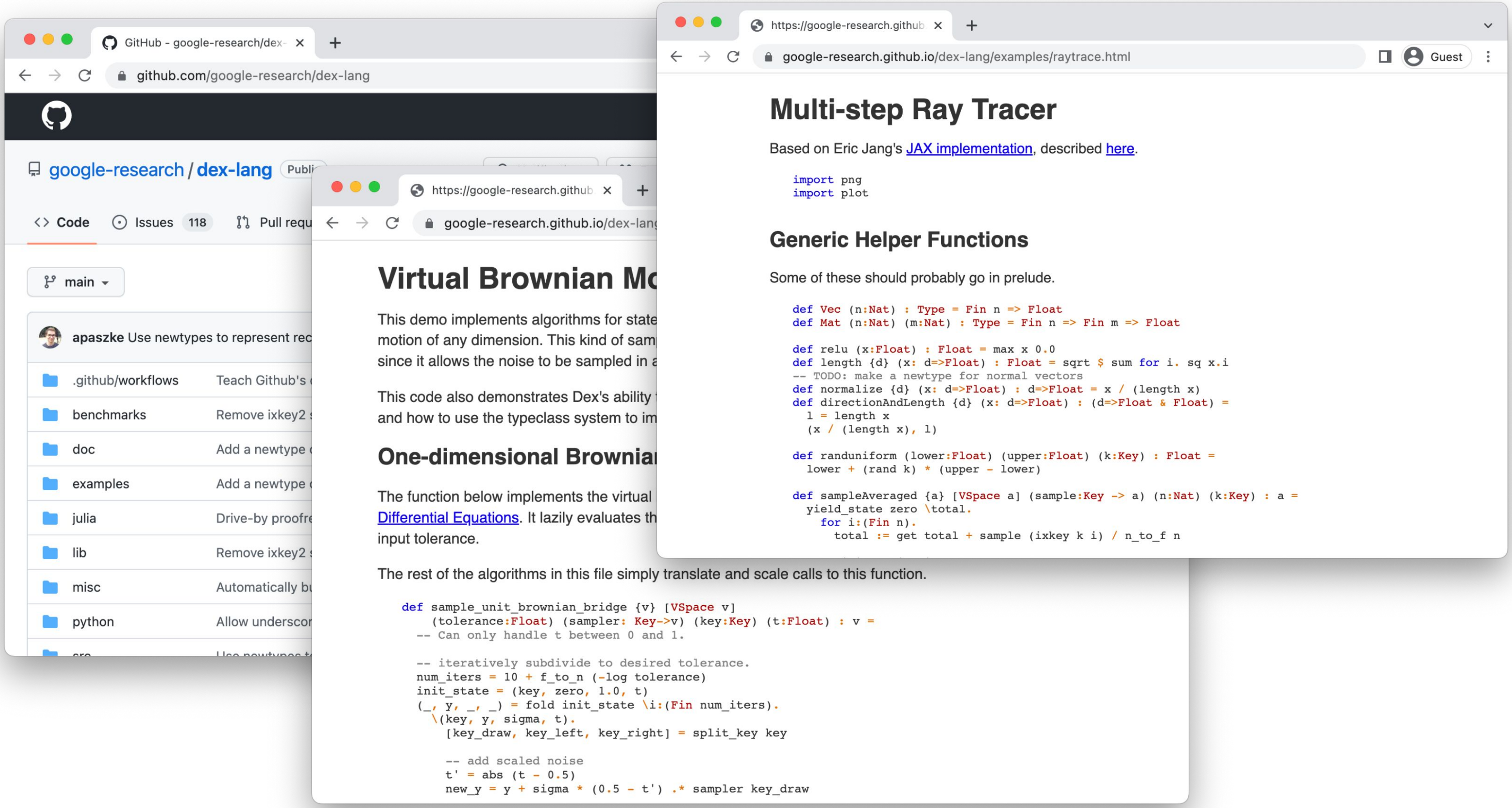
JAX offers several composable function transformations (`jit`, `grad`, `vmap`, etc.) that enable writing concise, accelerated code.

Here we show how to add your own function transformations to the system, by writing a custom Jaxpr interpreter. And we'll get composability with all the other transformations for free.

**This example uses internal JAX APIs, which may break at any time. Anything not in the API Documentation should be assumed internal.**

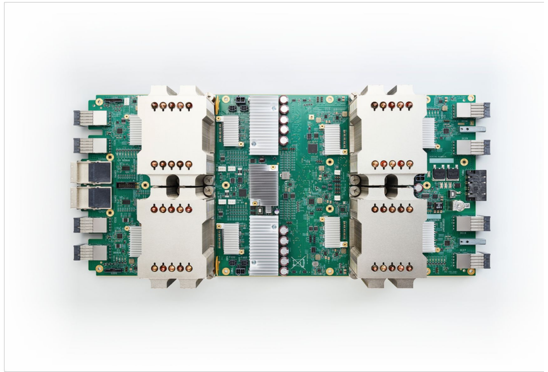
Contents

🏅 And what's the consolation prize?

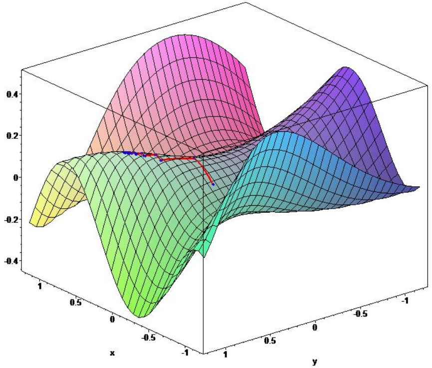

GitHub - google-research/dex-
github.com/google-research/dex-lang
google-research / dex-lang
Code
Issues 118
main
apaszke Use newtypes to represent rec
.github/workflows
Teach Github's
benchmarks
Remove ixkey2
doc
Add a newtype
examples
Add a newtype
julia
Drive-by proofre
lib
Remove ixkey2
misc
Automatically bu
python
Allow underscor
https://google-research.github
google-research.github.io/dex-lang
Virtual Brownian Mo
This demo implements algorithms for state
motion of any dimension. This kind of sam
since it allows the noise to be sampled in a
This code also demonstrates Dex's ability
and how to use the typeclass system to im
One-dimensional Brownia
The function below implements the virtual
Differential Equations. It lazily evaluates th
input tolerance.
The rest of the algorithms in this file simply translate and scale calls to this function.
def sample_unit_brownian_bridge {v} [VSpace v]
(tolerance:Float) (sampler: Key->v) (key:Key) (t:Float) : v =
-- Can only handle t between 0 and 1.
-- iteratively subdivide to desired tolerance.
num_iters = 10 + f_to_n (-log tolerance)
init_state = (key, zero, 1.0, t)
(_, y, _, _) = fold init_state \i:(Fin num_iters).
\(key, y, sigma, t).
[key_draw, key_left, key_right] = split_key key
-- add scaled noise
t' = abs (t - 0.5)
new_y = y + sigma * (0.5 - t') .* sampler key_draw
https://google-research.github
google-research.github.io/dex-lang/examples/raytrace.html
Guest
Multi-step Ray Tracer
Based on Eric Jang's JAX implementation, described here.
import png
import plot
Generic Helper Functions
Some of these should probably go in prelude.
def Vec (n:Nat) : Type = Fin n => Float
def Mat (n:Nat) (m:Nat) : Type = Fin n => Fin m => Float
def relu (x:Float) : Float = max x 0.0
def length {d} (x: d=>Float) : Float = sqrt $ sum for i. sq x.i
-- TODO: make a newtype for normal vectors
def normalize {d} (x: d=>Float) : d=>Float = x / (length x)
def directionAndLength {d} (x: d=>Float) : (d=>Float & Float) =
l = length x
(x / (length x), l)
def randuniform (lower:Float) (upper:Float) (k:Key) : Float =
lower + (rand k) * (upper - lower)
def sampleAveraged {a} [VSpace a] (sample:Key -> a) (n:Nat) (k:Key) : a =
yield_state zero \total.
for i:(Fin n).
total := get total + sample (ixkey k i) / n_to_f n

# The three pillars

## Accelerators



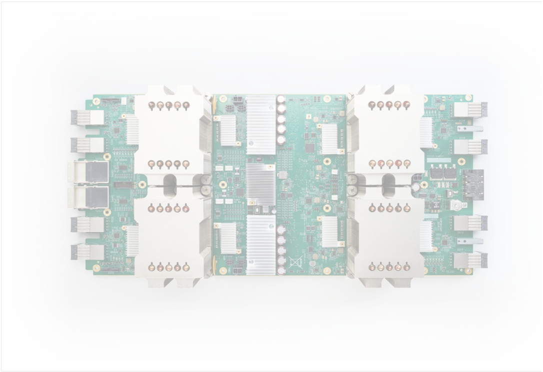
## Autodiff



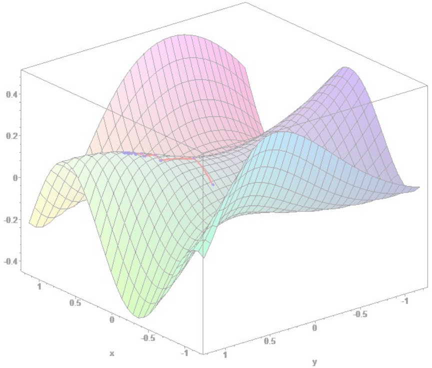

## Expressiveness

# The three pillars

Accelerators



Autodiff




Expressiveness

# Function types, dually

| | Function | Array |
|---|---|---|
| **Type** | `a -> b` | `a=>b` |
| **Introduction** | `\x:ty. expr` | `for x:ty. expr` |
| **Elimination** | `f expr` | `f.expr` |
| **Reduction** | `(\x. e) u ↦ e[x/u]` | `(for x:ty. e).u ↦ e[x/u]` |
| **Construction** | Cheap | Expensive |
| **Application** | Expensive | Cheap |
| **Domain** | Arbitrary | Finite (ordered) |

# Syntax benchmark: matrix multiply

**SOAC**

```
combinator_matrix_multiply = \x y.
  yt = transpose y
  dot = \x y. sum (map (uncurry (*)) (zip x y))
  map (\xr. map (\yc. dot xr yc) yt) x
```

**NumPy**

```
matmul = lambda x, y: np.einsum('ik,kj->ij', x, y)
```

**Dex**

```
for i:(Fin n). for j:(Fin m). sum (for k:(Fin q). x.i.k * y.k.j)
for i:(Fin n) j:(Fin m). sum (for q:(Fin k). x.i.k * y.k.j)
for i j. sum (for q. x.i.k * y.k.j)
for i j. sum for q. x.i.k * y.k.j
```

# Dex by example — matrix multiplication

```
def matmul (l : n=>k=>Float) (r : k=>m=>Float) : n=>m=>Float =
  for i j. sum for u. l.i.u * r.u.j
```

No need to spell out loop bounds (but you can if you'd like)!

```
def matmul (l : n=>k=>Float) (r : k=>m=>Float) : n=>m=>Float =
  for i j. sum for u. l.u.i * r.u.j
> Type error:
> Expected: k
>   Actual: n
>
>   for i j. sum for u. l.u.i * r.u.j
>                             ^^
```

Expressive array types prevent errors and make code more accessible to readers

```
def matmul [Semiring a] (l : n=>k=>a) (r : k=>m=>a) : n=>m=>a =
  for i j. sum for u. l.i.u * r.u.j
```

Zero-cost generics/type-classes/traits make it easy to write reusable libraries

Can array programming be liberated
from integer indices?

# Rich index sets

In Dex, any type *conforming to Ix* can be an array index:

```
interface Ix n where
  size n            : Int
  toOrdinal         : n -> Int
  unsafeFromOrdinal : Int -> n
```

size

& isomorphism with a
prefix of natural numbers

```
def fromOrdinal {n} [Ix n] (o:Int) : n =
  case 0 <= o && o < size n of
    True  -> unsafeFromOrdinal o
    False -> error …
```

Basic shape arithmetic can be done using standard type constructors:

| | |
|---|---|
| **Products** | `(n & m)` |
| **Sums** | `(n \| m)` |
| **Exponentials** | `(n=>m)` |

# Basic examples

| | | |
|---|---|---|
| **Reshapes** | `reshape (2, -1, 4) x` | (n & m)-typed binder<br>`for i (j, k) l. x.i.j.k.l` |
| **Concatenation** | `concatenate x y` | (n \| m)-typed binder<br>`for ci. case ci of`<br>`  Left  xi -> x.xi`<br>`  Right yi -> y.yi` |
| **Named axes** | `image[h, w]` or `image[w, h]`? | `image.{height=h, width=w}`<br>`image.{width=w, height=h}` |
| **Boundary conditions** | `x: (Fin (1 + n))=>a`<br>`x[0] vs x[1 + i]` | `x: (Unit\|n)=>a`<br>`x.(Left ()) vs x.(Right i)` |

# Indexing lemmas

**Array reversal**

```
def reflect {n} (i:n) : n =
  unsafeFromOrdinal n (size n - 1 - ordinal i)
```

```
sequence : (Fin s)=>Int = …
for i in range(len(sequence)).
  sequence[len(sequence) - 1 - i]
```

Correctness reasoning requires non-local context (e.g. range of i)

```
sequence : n=>Int = …
for i.
  sequence.(reflect i)
```

**Dynamic programming**

```
def prev (i:n) : (Unit|n) =
  unsafeFromOrdinal _ (ordinal i)
```

```
x : (Fin s)=>Int = …
sumWithPrev = for i in range(len(x)).
  if i == 0
    then x[i]
    else x[i - 1] + x[i]
```

Easy to forget about the base case and read out of bounds!

```
x : (Unit|n)=>Int = …
sumWithPrev = for i.
  case i of
    Left  () -> x.i
    Right i' -> x.(prev i') + x.i
```

# Index sets are user-definable

```
data RGB = Red | Green | Blue
instance Ix RGB
  size = 3
  toOrdinal = \x. case x of
    Red   -> 0
    Green -> 1
    Blue  -> 2
  unsafeFromOrdinal = …

data HSV = Hue | Saturation | Value
instance Ix HSV …

Image = \h w colorSpace. { height: (Fin h) & width: (Fin w) }=>colorSpace=>UInt8

imgRGB : Image 200 200 RGB = loadKnownSizeJPG "doggo.jpg"
imgHSV : Image _   _   HSV = RGBtoHSV imgHSV
hues = for h w. imgHSV.{height=h, width=w}.Hue
```

Arrays can function as *named tuples*

# Array type zoo

🤔 If we have dependent functions… why don't we try dependent arrays?

**Homogeneous**

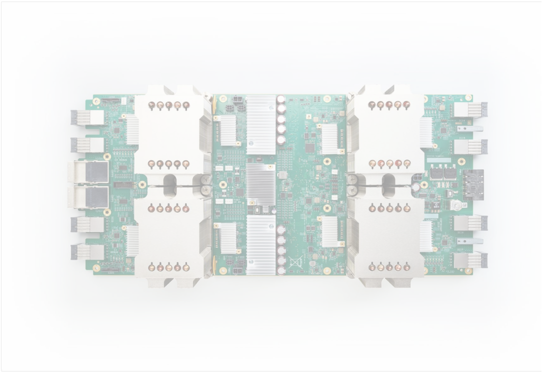
**Heterogeneous**

| **Array kind** | **Example type** |
|---|---|
| Static | `(Fin 10)=>(Fin 20)=>Float` |
| Dynamic | `(Fin n)=>(Fin m)=>Float` |
| Structured ragged | `(i:Fin 10)=>(...i)=>Float` |
| Ragged | `(i:Fin 10)=>(Fin lengths.i)=>Float` |
| Jagged | `(Fin 10)=>List Float` |

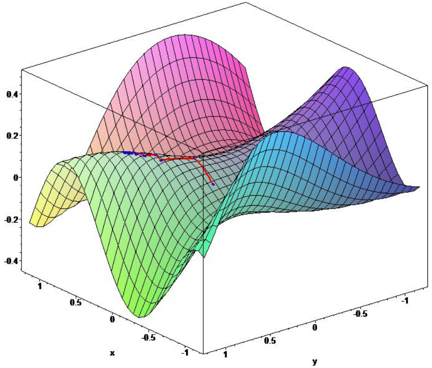
Also:
Position-dependent arrays and their application for high performance code generation, F. Pizzuti et al.
Generating High Performance Code for Irregular Data Structures using Dependent Types, F. Pizzuti et al.

# The three pillars

## Accelerators

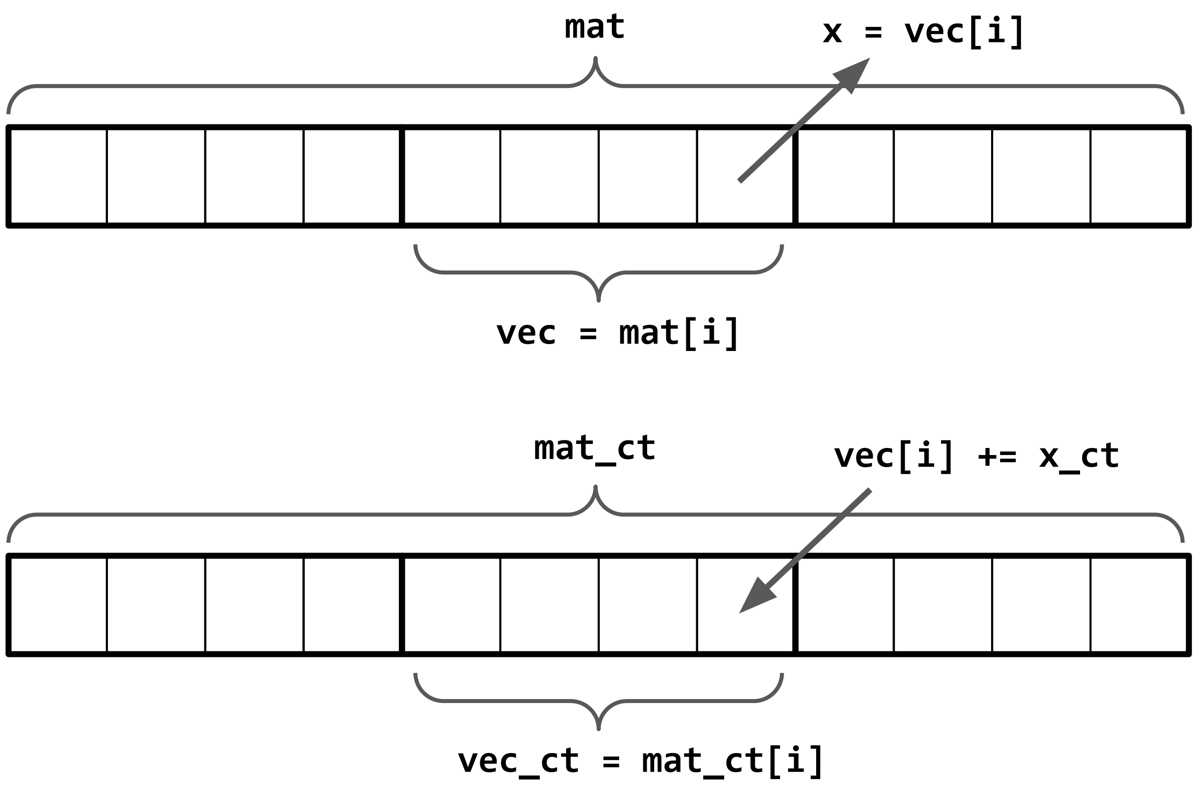


## Autodiff




## Expressiveness

# Cost model: indexing is aliasing




We need to alias writes like we alias reads!

# Efficient AD as a language design benchmark

*There exists a constant c such that for every program P the cost of evaluating P' (P' being derived using forward- or reverse-mode AD from P) is at most c times larger than the cost of evaluating P.*

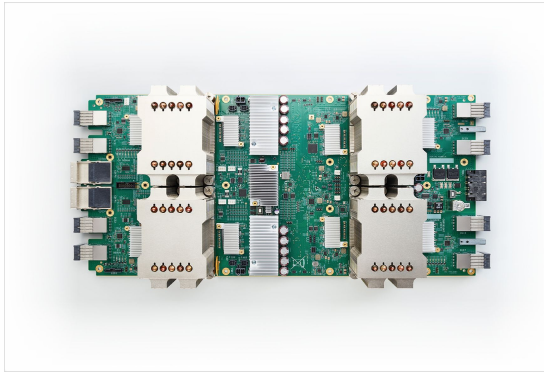
**Good reverse-mode autodiff support requires:**

1️⃣ Closure under partial evaluation

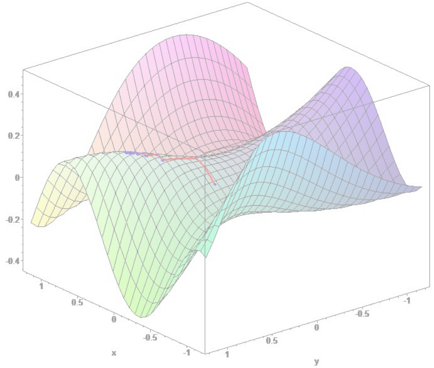
2️⃣ Closure under data-flow duality

# The three pillars

## Accelerators



## Autodiff




## Expressiveness

# Acceleration

### ✅ No effects

```
for i. x.i + 1
```

### ⚠️ Reductions

```
yieldAccum \ref.
  for i. ref += x.i
```

### ❌ Arbitrary state

```
yieldState \ref.
  for i.
    ref := f (get ref)
```

# Thank you!

apaszke@google.com