JIT-compilation with JAX

Peter Fackeldey

7/21/25

Introduction

- JAX describes itself as “high performance array computing” in Python
- Looks & feels like NumPy (~drop-in replacement)
- Very popular in science, especially AI (~31k ⭐ on Github)
- Large ecosystem for science and AI:
  - Building AI Models: flax, equinox, keras
  - Optimizers: optax, optimistix, lineax
  - …many more
- Docs 📄 https://docs.jax.dev/en/latest/
- Awesome JAX: https://github.com/n2cholas/awesome-jax

Drop-in replacement for NumPy

```
import numpy as np

def quadratic_formula(a, b):
  return (-b + np.sqrt(b**2 - 4*a)) / (2*a)
```

only change: import

```
import jax.numpy as jnp

def quadratic_formula(a, b):
  return (-b + jnp.sqrt(b**2 - 4*a)) / (2*a)
```

When NumPy is not enough?

- Performance
- Gradients
- (Multi-)accelerator support

When NumPy is not enough?

We'll focus on these two today

Performance

Gradients

(Multi-)accelerator support

Performance: JIT-compilation

- JIT (just-in-time) compilation with **jax.jit**

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

@jax.jit
def quadratic_formula(a, b):
  return (-b + jnp.sqrt(b**2 - 4*a)) / (2*a)
```

- Let’s have a look at an example to see what JIT-compilation is good for

Performance: JIT-compilation

- JIT (just-in-time) compilation with **jax.jit**

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

@jax.jit
def quadratic_formula(a, b):
  return (-b + jnp.sqrt(b**2 - 4*a)) / (2*a)
```

- Let's have a look at an example to see what JIT-compilation is good for
- It's fast because: JAX executes highly optimized machine code with the help of a chain of compiler steps: XLA & LLVM

How to do JIT compilation in such a dynamic language as Python?

JIT-compilation in Python: two possibilities

Two ways to implement JIT compilation:

Tracing	Bytecode Analysis
JAX	**Numba**

JIT-compilation: Tracing

- What does this function do?

```python
def fun(x):
  return x + 1
```

JIT-compilation: Tracing

Ref: Intro to JAX: Accelerating Machine Learning research (adapted)

- What does this function do?

```
def fun(x):
  return x + 1


class IncreaseBrightness:
  def __add__(self, x)
    subprocess.call(["ssh", "my@home", ...])


fun(IncreaseBrightness())
```

- Python can literally do *anything*
- JAX uses *tracers* to find out **which *array* computations happen inside *fun***

JIT-compilation: Tracing

Ref: Intro to JAX: Accelerating Machine Learning research (adapted)

- How does a tracer work?

```
def fun(x):
  return x + 1


class Tracer:
  recording = []

  def __init__(self, shape, dtype):
    self.shape = shape
    self.dtype = dtype

  def __add__(self, other)
    self.recording.append("add", self, other)
    return Tracer(self.shape, self.dtype)


fun(Tracer(shape=(10,), dtype="float"))
```

JIT-compilation: Tracing

Ref: Intro to JAX: Accelerating Machine Learning research (adapted)

- How does a tracer work?

```
def fun(x):
  return x + 1
```

Let's see how this looks in real code

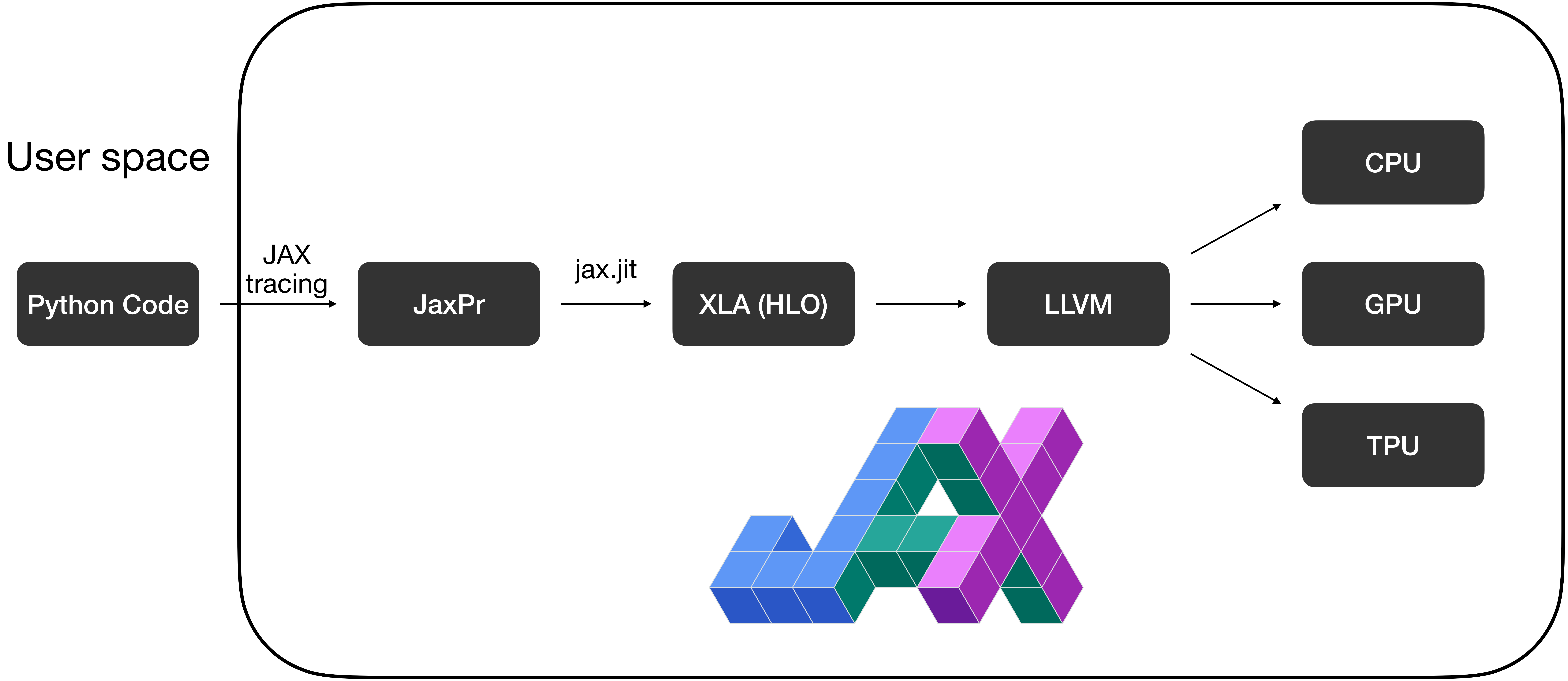
```
  def __add__(self, other)
    self.recording.append("add", self, other)
    return Tracer(self.shape, self.dtype)


fun(Tracer(shape=(10,), dtype="float"))
```

JIT-compilation: step-by-step

1. Replace all arrays with tracers
2. Record all (recordable) array computations by running tracers through the function
3. Write a new program that only contains the recorded computations in a new intermediate representation: *JaxPr*
4. Translate JaxPr to another intermediate representation for XLA compiler (HLO)
5. Run XLA compiler optimization (hardware-agnostic, e.g. operation fusion)
6. Pass the optimized code to LLVM
7. Run LLVM toolchain (another round of optimizations, hardware-“aware”)
8. Emit highly-optimized machine code for certain hardware

JIT-compilation: step-by-step (simplified)



JIT-compilation: step-by-step (simplified)

User space

Python Code → (JAX tracing) → **JaxPr** → (jax.jit) → XLA (HLO) → LLVM → CPU / GPU / TPU

What's this for?

JAX: sharp bits

- Tracing:
  - Can not work with numerical values during tracing (no printing)
  - Only one if-branch is traced, the others are not recorded!
  - JAX needs to know about “static” (non-array) arguments, so it can properly re-compile if needed (needs to be provided by the user)
  - Python loops are unrolled → can lead to very long compile-times
- NaN-handling/debugging can be hard
- JAX’s JIT needs to know all shapes at compile time, you can’t work with dynamic shapes, i.e. **array[array>0]** is not possible
- Impure functions may lead to wrong results

When NumPy is not enough?

Gradients (and other "transformations")

Transformations of JaxPr: Gradients

- JAX has a lookup table that maps each recordable function to its gradient
- We need to record all operations and apply rules of differentiation with the help of this lookup table

  …remember we have already the recording of all operations → JaxPr

- All we have to do is to rewrite (or “transform”) the JaxPr into a JaxPr that resembles the gradient: $f \to f'$

```python
def fun(x):
  return 2 + jnp.sin(x)

jax.make_jaxpr(fun)(jnp.array(1.0))
# { lambda ; a:f32[]. let
#     b:f32[] = sin a
#     c:f32[] = add 2.0 b
#   in (c,) }
```

jax.grad →

```python
def fun(x):
  return 2 + jnp.sin(x)

jax.make_jaxpr(jax.grad(fun))(jnp.array(1.0))
# { lambda ; a:f32[]. let
#     b:f32[] = sin a
#     c:f32[] = cos a
#     _:f32[] = add 2.0 b
#     d:f32[] = mul 1.0 c
#   in (d,) }
```

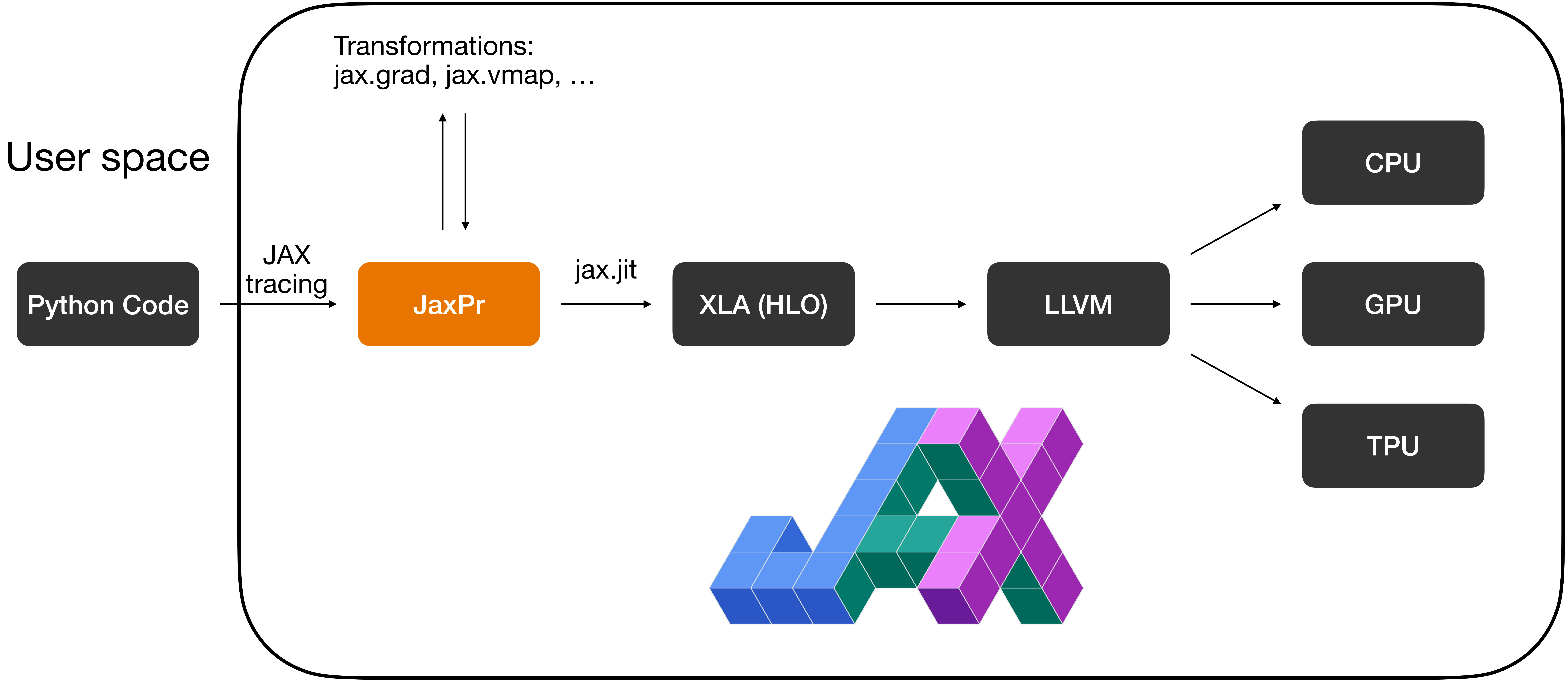
Transformations of JaxPr

- We know already of: **jax.jit** & **jax.grad**
- JAX provides a few more transformations:
  - **jax.vmap**: *real* vectorization along axes (unlike np.vectorize)
  - **jax.pmap (or jax.make_mesh)**: parallelization across devices
  - …and a few more for auto-diff (jax.jacobian, jax.jvp, …)
- All of these transformations are **composable!**

```
def fun(x):
  return 2 + jnp.sin(x)

jax.jit(jax.vmap(jax.grad(fun)))(jnp.array([1.0, 2.0, 3.0]))
# Array([ 0.5403023 , -0.41614684, -0.9899925 ], dtype=float32)
```

JAX: under-the-hood (final)



Let’s move on the some hands-on examples

References & links

- If you want to know more about tracing & JaxPr internals: https://docs.jax.dev/en/latest/autodidax.html
- If you want to know more about auto-differentiation: https://docs.jax.dev/en/latest/notebooks/autodiff_cookbook.html
- If you're interested in scaling to multi-host (e.g. GPU) systems: https://jax-ml.github.io/scaling-book/
- AI tutorials with JAX: https://docs.jaxstack.ai/en/latest/index.html
- Join our Princeton JAX mailing list: https://lists.princeton.edu/cgi-bin/wa?A0=JAX-user-group