Introduction to porting Python to GPU with JAX.

NERSC Data Day 2022

Nestor Demeure[1]
[1] National Energy Research Scientific Computing Center, Berkeley CA, United-states

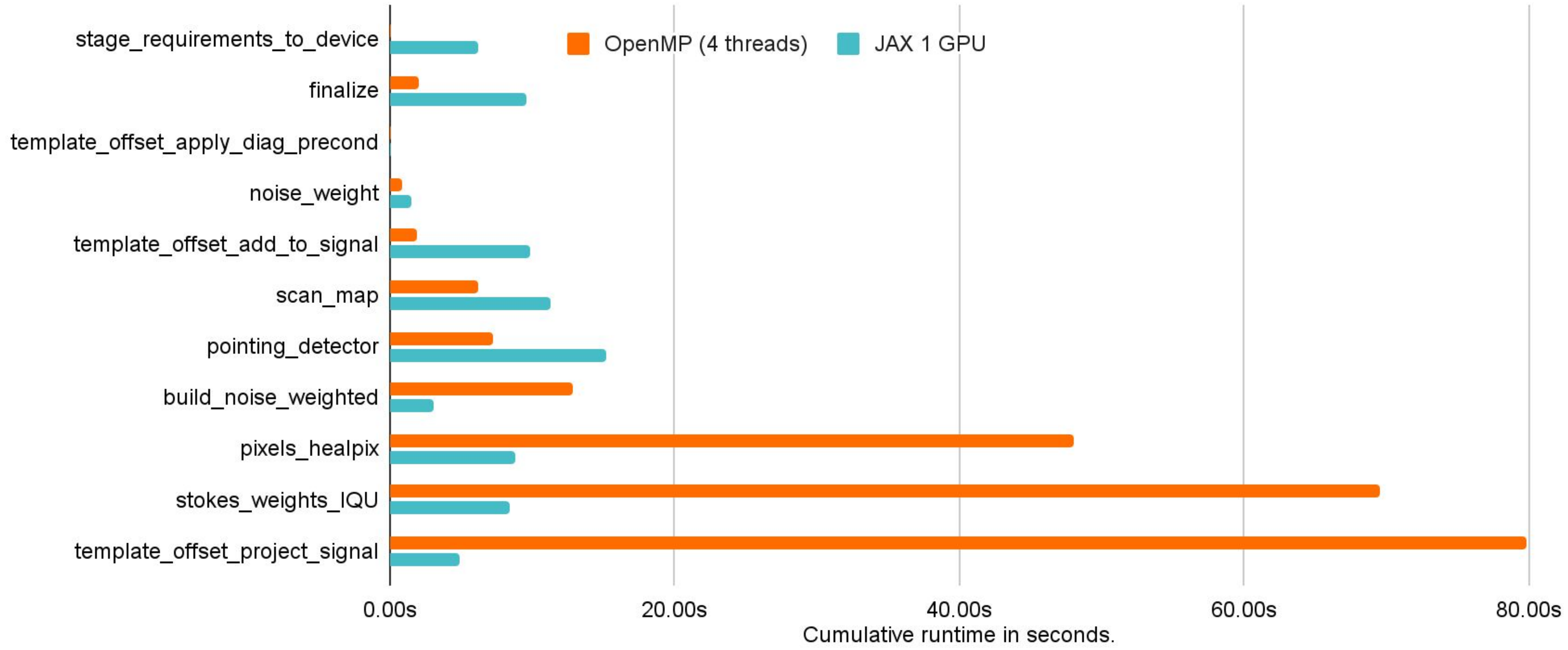
Who am I?

I am a **NESAP Postdoctoral Researcher at NERSC** with a focus on high performance computing, numerical accuracy and artificial intelligence.

I specialize in helping teams of researchers make use of high performance computing environments.

I am currently working to help port the [TOAST software framework] to the new Perlmutter supercomputer and, in particular, port it to graphic processors (GPU).

Up to x16 speed-up from optimized C++ to JAX!



Porting a Python code to GPU

Pros and cons of the current approaches

Using off-the-shelf kernels

Call a library providing off-the-shelf kernels:

- Numpy ➡ Cupy
- Scipy ➡ Cupy
- Pandas ➡ RAPIDS CuDF
- Scikit-learn ➡ RAPIDS CuML

- Very easy to use,
- perfect if you find what you need,
- cannot write your own kernel,
- performance loss:
  - allocating intermediate values,
  - more data transfers to the GPU.

Using a deep-learning library

Use a deep-learning library:

- Pytorch
- Tensorflow
- JAX

- Great for deep-learning,
- easy to use and well documented,
- support for most numerical building blocks,
- *usually*, a large overhead:
  - gradient computation,
  - intermediate values.

Writing a kernel in a low-level language

Write a kernel in **CUDA** / **OpenCL** / **HIP** / **SYCL** / etc and link it in Python.

You can use PyOpenCL or PyCuda to link your kernel.

- Perfect control of performance,
- cannot reuse numerical building blocks (PRNG, FFT, linear algebra),
- requires a lot of expertise:
  - to write code that is *actually* **performant**,
  - to write **correct** code,
  - to **compile and link** the result into Python.

Writing a kernel in Python

Write a kernel in Python using:

- Numba,
  - limited Numpy support,
  - low-level CUDA-like syntax,
- Taichi
  - focus on graphics,
  - requires implementing most of the operations you need from scratch.

- Full Python codebase,
- can still be very low-level,
- very limited building blocks.

Can we have good GPU performance, portability and productivity?

Introducing JAX

High-level introduction to JAX

What is JAX?

JAX is a Python library to write code that can run in parallel on:

- CPU,
- GPU (Nvidia and AMD),
- TPU,
- etc.

Developed by Google as a building block for deep-learning frameworks. Seeing wider use in numerical applications including:

- Molecular dynamics,
- computational fluid dynamics,
- ocean simulation.

What does JAX look like?

It has a Numpy-like interface:

```
from jax import random
from jax import numpy as jnp

key = random.PRNGKey(0)
x = random.normal(key, shape=(3000, 3000), dtype=jnp.float32)
y = jnp.dot(x, x.T)  # runs on GPU if available
```

How does JAX work?

Calls a ***just-in-time compiler*** when you execute your function with a ***new problem size***:

JAX's limitations

- Compilation happens just-in-time, at runtime,
  easily amortized on a long running computation
- input sizes must be known to the tracer,
  padding, masking and recompiling for various sizes
- loops and tests are limited inside JIT sections,
  JAX provides replacement functions
- no side effects and no in-place modifications,
  one gets used to it, it actually helps with correctness
- focus on GPU optimizations rather than CPU.
  there is growing attention to the problem

How do we use it?

Using JAX

Writing JAX code

Numpy-like syntax

If you know Numpy you are 90% of the way there.

```
import jax.numpy as jnp

x = jnp.ones(shape=(1000,1000))
y = 2 * jnp.zeros(1000)

z = jnp.dot(x, jnp.cos(y))
y2 = jnp.linalg.solve(x, z)
```

Mutability

JAX arrays are **immutable** but, you can use shadowing and .at[] functions:

```
# arr += 1
arr = arr + 1

# arr[index] = 1
# WARNING: this produces a new array
arr = arr.at[index].set(1)

# arr[index] += 1
# NOTE: this operation is atomic
arr = arr.at[index].add(1)
```

Just-in-time compilation

JAX will be *slow* unless you compile your code:

```
from jax import jit

def f(x):
   print("Tracing right now!")
   return x*2

f_jitted = jit(f)
y = f_jitted(x)
```

- Recompile when the **static inputs** (including problem size) are changed,
- inputs can be built-in types, arrays, lists, dictionaries, struct, etc.

Just-in-time compilation: static values

Numbers, booleans and user defined struct can be marked as **static**:

```
from jax import jit

def f(x, should_double):
    return (x*2) if should_double else x

# specify static inputs
f_jitted = jit(f, static_argnames=["should_double"])
```

- Useful to **help optimizer** and **workaround limitations** in tests and loops,
- value needs to be **hashable** (does not apply to lists and arrays),
- will **trigger recompilation** if the value is changed.

Just-in-time compilation: donate input

Inputs can be **donated**:

```python
from jax import jit

def f(x):
    return 2*x

# specify donated inputs
f_jitted = jit(f, donate_argnums=[0])
```

- useful to **reduce allocations**,
- does **not currently apply to CPU**.

Conditionals

In jitted sections, you can only perform tests on static values, instead:

- Use where to combine inexpensive computations with a mask,
- use cond to run expensive computations depending on a boolean.

```
import jax

# where
y = jax.numpy.where(is_true, y_true, y_false)

# cond
y = jax.lax.cond(is_true, f_true, f_false, x)
```

Loops and vectorisation

In jitted sections, loop conditions are restricted to static values and will be **unrolled**:

- JAX provides control flow operators including while_loop and fori_loop,
- JAX let you vectorise your function with vmap, pmap and xmap.

```
from jax.experimental.maps import xmap
from jax import vmap

#for i in range(nb_i):
#   for j in range(nb_j):
#       result[i,j] = f_body(x[i,j,:], y)

f_vmap_j = vmap(f_body, in_axes=(0,None), out_axes=0)
f_vmap_ij = vmap(f_vmap_j, in_axes=(0,None), out_axes=0)

f_xmap_ij = xmap(f_body, in_axes=[['i','j',...],[...]], out_axes=['i','j'])
```

Pseudo random number generation

JAX uses [its own PRNG] tailored for parallelism and reproducibility:

```python
from jax import random

# initialize PRNG
seed = 1701
key = random.PRNGKey(seed)

# generates random numbers
key, subkey = random.split(key)
x = random.normal(subkey, shape=( 3000, 3000))
```

Automatic differentiation

JAX does automatic differentiation by code transformation:

```python
from jax import grad

# computes the derivative of the function f
df = grad(f)

# gets a result and its derivative
y = f(x)
dx = df(x)
```

- Can be applied repeatedly for **higher order derivation**,
- overhead **similar to analytic solution**,
- **no overhead** to function that are not differentiated,
- some operations cannot be differentiated.

Performance tricks

You can do three things to improve performance significantly:

- Minimise the number of **recompilations**,
- put a maximum of your **code inside a jitted section**,
- **keep the data on GPU**, inside JAX arrays.

Useful libraries

The Awesome JAX repository has a *lot* of good references including:

- MPI4JAX: MPI support for JAX,
- Chex: testing utilities for JAX,
- JAXopt: optimizers written in JAX,
- Einshape: an alternative reshaping syntax,
- deep learning frameworks built upon JAX:
  - FLAX: widely used and flexible,
  - Equinox: focus on simplicity,
  - etc.

Is it worth it?

Case study

Porting the TOAST codebase to GPU

TOAST

TOAST is a large Python application used to study the **cosmic microwave background**.

It is made of pipelines distributed with MPI and composed of **C++ kernels parallelized with OpenMP**.

Kernels use a **wide variety of numerical methods** including random number generation, linear algebra and fast fourier transforms.

We ported **two pipelines to GPU**.

Porting the code

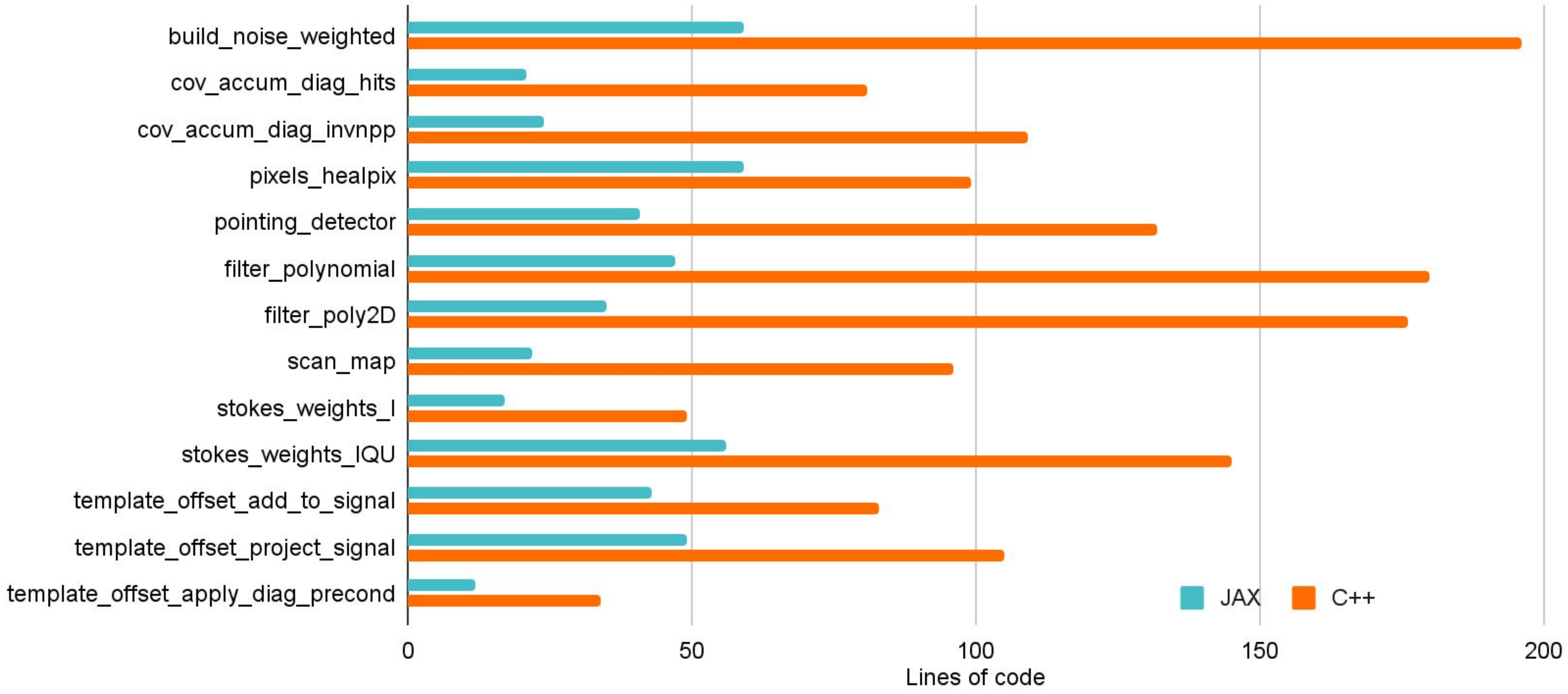
Kernels were ported **from C++ to Numpy to JAX** and validated using **unit tests**.

Kernels loop on irregular intervals, we introduced a **JaxIntervals** type to automate padding and masking.

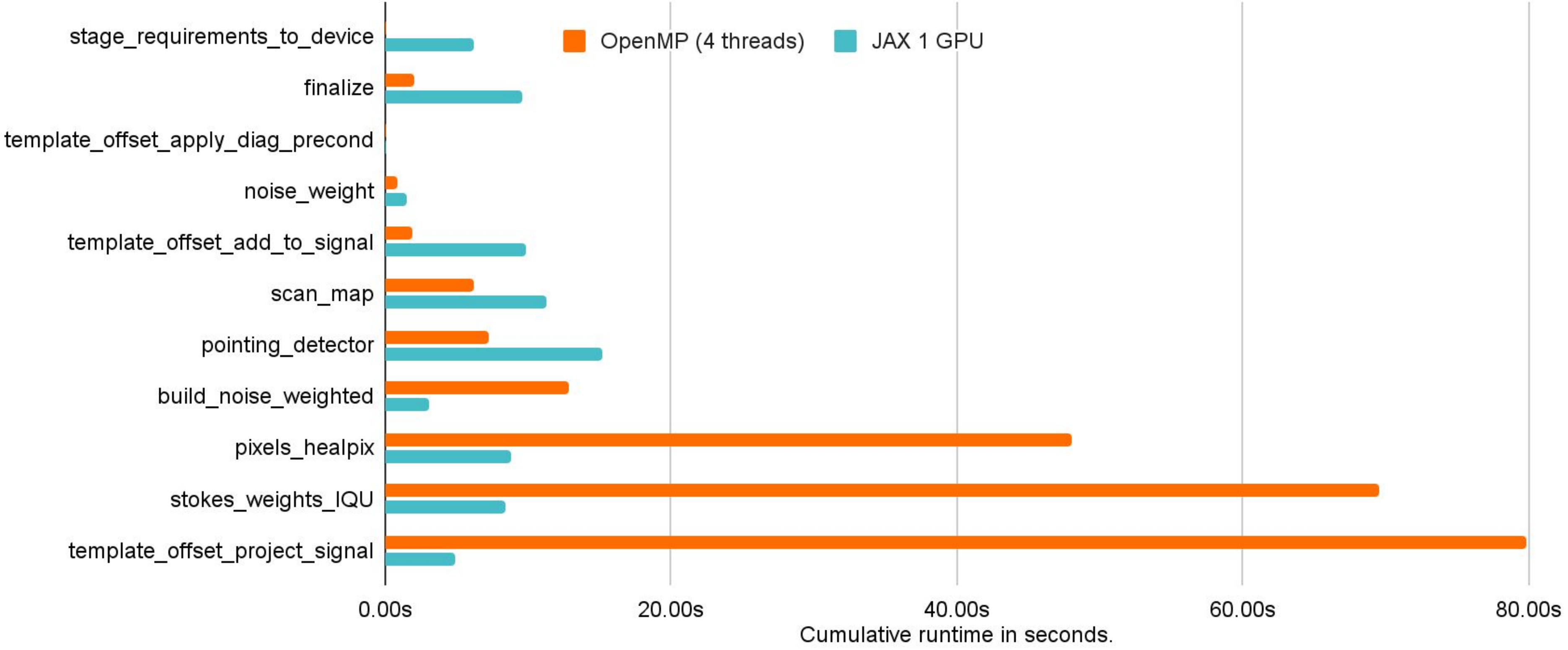
Kernels mutate output parameters, we introduced a **MutableJaxArray** type to box JAX arrays.

Data movement is expensive, we move data *once* at the beginning and end of each pipeline call.

Porting the code (x7 reduction in lines of code)



Performance per kernel (up to x16 speed-up)



Perspectives

This was a *proof of concept*, we can improve and simplify things significantly:

- **Reduce data movement**,
- remove C++ dependencies by **porting more kernels**,
- default to **JAX arrays** and **pure functions**,
- redesign pipelines to JIT them into **single GPU kernels**.

Overview

Should you use JAX in your project?

Should you use JAX?

- Your code is written in **Python**,
- your code can be written with **Numpy**,
- your array sizes are **not too dynamic**,
- single-thread CPU is an **acceptable fallback** in the absence of GPU.

JAX's strengths

I believe JAX is in a **sweet spot for research and complex numerical codes**:

- Focus on the semantic, leaves optimization to the compiler,
- single code base to deal with CPU and GPUs,
- immutable design is actually *nice* for correctness,
- easy to use numerical building blocks inside kernels.

Thank you!

ndemeure@lbl.gov

Exercises!

https://cutt.ly/tNf8N7w