Porting a large cosmology code to GPU, a case study examining JAX and OpenMP.

Cray User Group 2023

Nestor Demeure[1], Theodore Kisner[2], Reijo Keskitalo[2], Rollin Thomas[1], Julian Borrill[2], and Wahid Bhimji[1]
[1] National Energy Research Scientific Computing Center, Berkeley CA, United-States
[2] Computational Cosmology 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).

Can we have good GPU performance, portability and productivity?

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.

Writing a kernel in a low-level language

Write a kernel in **CUDA** / **HIP** / **SYCL** / **OpenMP Target Offload** / etc and link it in Python.

You can use [PyOpenCL]() or [PyCuda]() to link your kernel.

- Perfect control of performance,
- hard to reuse numerical building blocks (PRNG, FFT, linear algebra),
- *usually* requires a lot of expertise:
  - to write **correct** code,
  - to write code that is *actually* **performant**,
  - 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,
- Triton
  - no library support,
  - low-level unique syntax.

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

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.

Can we have good GPU performance, portability and productivity?

Examining **OpenMP Target Offload** and **Jax**.

Introducing OpenMP Target Offload

High-level introduction to OpenMP Target Offload

What is OpenMP Target Offload?

OpenMP is ***the* classical shared memory parallelism framework**. Since version 4.0 it includes `target` commands to run code on device.

It promises:

- **Portability**,
- **high-level code**,
- building on existing OpenMP **expertise** and infrastructures.

OpenMP Target Offload's limitations

- Limited compiler support,
- reduced access to optimization,
- default, automatic, data management can be costly.

Lower level than might appear at first.

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,
- cosmology.

What does JAX look like?

It has a Numpy-like interface:

```python
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

JAX’s strengths

- 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.

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 **10 kernels to GPU**.

Porting the code: OpenMP Target offload

Settled on the **NVIDIA NVC compiler**.

Duplicated the kernel's main loops into CPU and GPU versions.

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

Porting the code: OpenMP Target offload

```
void stokes_weights_IQU(int32_t const* quat_index, double const* quats, int32_t const* weight_index,
                        double* weights, double const* hwp, double const* epsilon, double cal,
                        Interval const* intervals, int64_t const n_det, int64_t const n_samp,
                        int64_t const n_view) {

  auto& omgr = OmpManager::get();
  int dev = omgr.get_device();

  double* dev_quats = omgr.device_ptr(quats);
  double* dev_weights = omgr.device_ptr(weights);
  Interval* dev_intervals = omgr.device_ptr(intervals);
  double* dev_hwp = omgr.device_ptr(hwp);

  #pragma omp target data device(dev) map(to: weight_index[0:n_det], quat_index[0:n_det], epsilon[0:n_det], cal, n_view, n_det, n_samp)
  #pragma omp target teams distribute collapse(2) is_device_ptr(dev_weights, dev_quats, dev_hwp, dev_intervals)
  for (int64_t idet = 0; idet < n_det; idet++) {
    for (int64_t iview = 0; iview < n_view; iview++) {
        #pragma omp parallel for default(shared)
        for (int64_t isamp = dev_intervals[iview].first; isamp <= dev_intervals[iview].last; isamp++) {
          stokes_weights_IQU_inner(cal, raw_quat_index, raw_weight_index, dev_quats, dev_hwp,
                                   raw_epsilon, dev_weights, isamp, n_samp, idet);
}}}}
```

Porting the code: JAX

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.

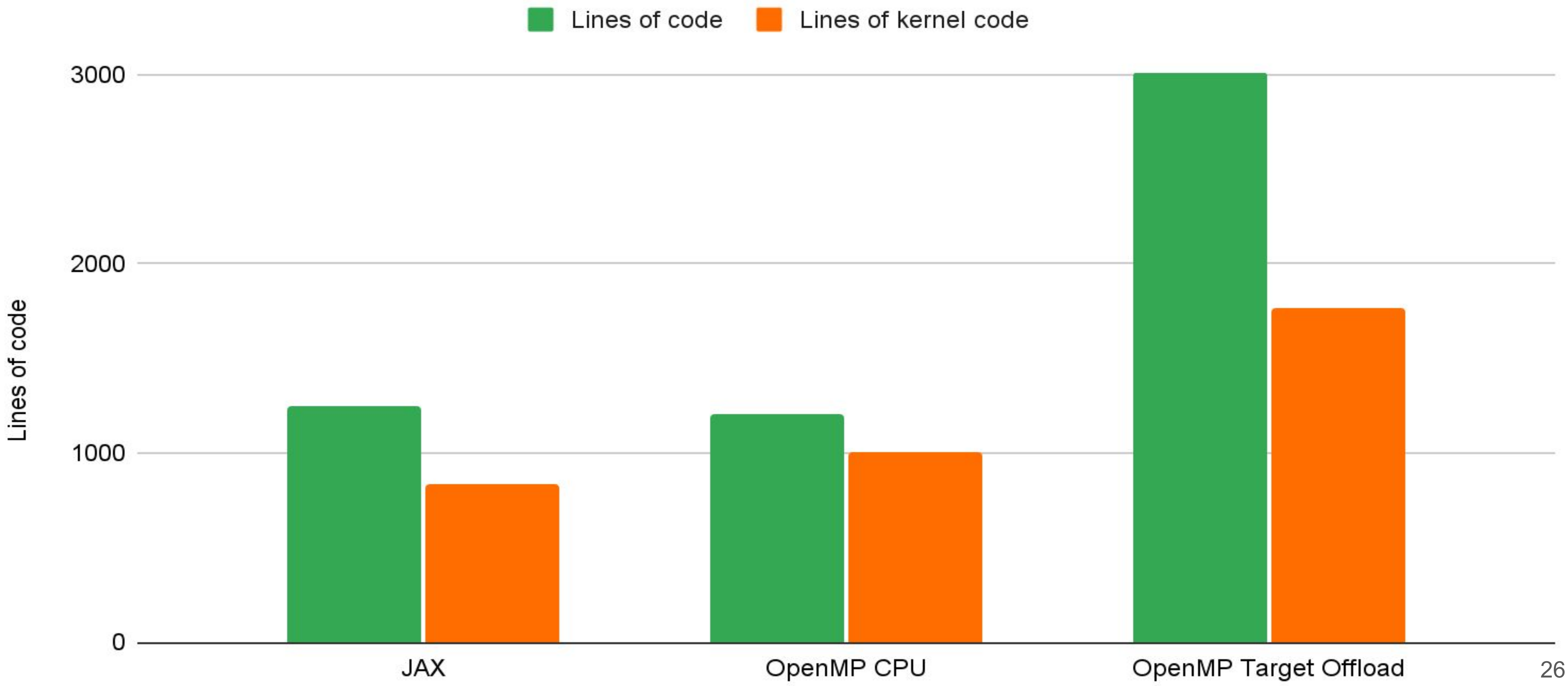
Porting the code: JAX

```python
def stokes_weights_IQU_jax(quat_index, quats, weight_index, weights,
                           hwp, intervals, epsilon, cal):
    # prepares inputs
    intervals_max_length = INTERVALS_JAX.compute_max_intervals_length(intervals)
    quat_index_input = MutableJaxArray.to_array(quat_index)
    quats_input = MutableJaxArray.to_array(quats)
    weight_index_input = MutableJaxArray.to_array(weight_index)
    weights_input = MutableJaxArray.to_array(weights)
    hwp_input = MutableJaxArray.to_array(hwp)
    epsilon_input = MutableJaxArray.to_array(epsilon)

    # runs computation
    weights[:] = stokes_weights_IQU_interval(quat_index_input,  quats_input, weight_index_input, weights_input,
                                             hwp_input, epsilon_input, cal, intervals.first, intervals.last,  intervals_max_length)


# jit compiling
stokes_weights_IQU_interval = jax.jit(stokes_weights_IQU_interval,
                                      static_argnames=["cal", "intervals_max_length"],
                                      donate_argnums=[3]) # donating weights
```

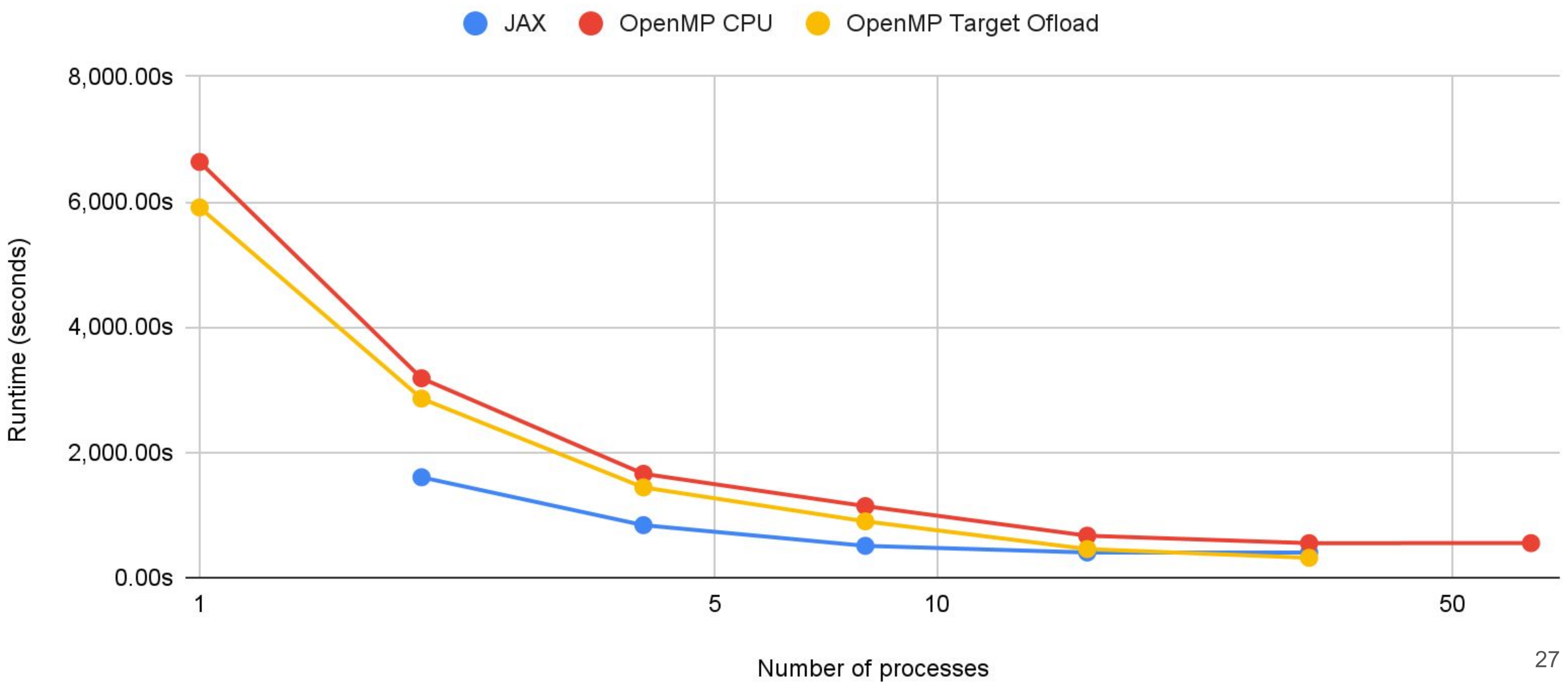
Porting the code: JAX

```
def stokes_weights_IQU_interval(quat_index, quats, weight_index, weights, hwp, epsilon, cal,
                                interval_starts, interval_ends, intervals_max_length):
    # extract interval slices
    intervals = JaxIntervals(interval_starts, interval_ends+1, intervals_max_length) # end+1 as the interval is inclusive
    quats_interval = JaxIntervals.get(quats, (quat_index,intervals,ALL)) # quats[quat_index,intervals,:]
    hwp_interval = JaxIntervals.get(hwp, intervals) # hwp[intervals]

    # does the computation
    new_weights_interval = stokes_weights_IQU_inner(epsilon, cal, quats_interval, hwp_interval)

    # updates results and returns
    # weights[weight_index,intervals,:] = new_weights_interval
    weights = JaxIntervals.set(weights, (weight_index, intervals, ALL), new_weights_interval)
    return weights


# maps over samples, intervals and detectors
stokes_weights_IQU_inner = jax.xmap(stokes_weights_IQU_inner,
                                    in_axes=[['detectors'], # epsilon
                                             [...], # cal
                                             ['detectors','intervals','interval_size',...], # quats
                                             ['intervals','interval_size']], # hwp
                                    out_axes=['detectors','intervals','interval_size',...])
```

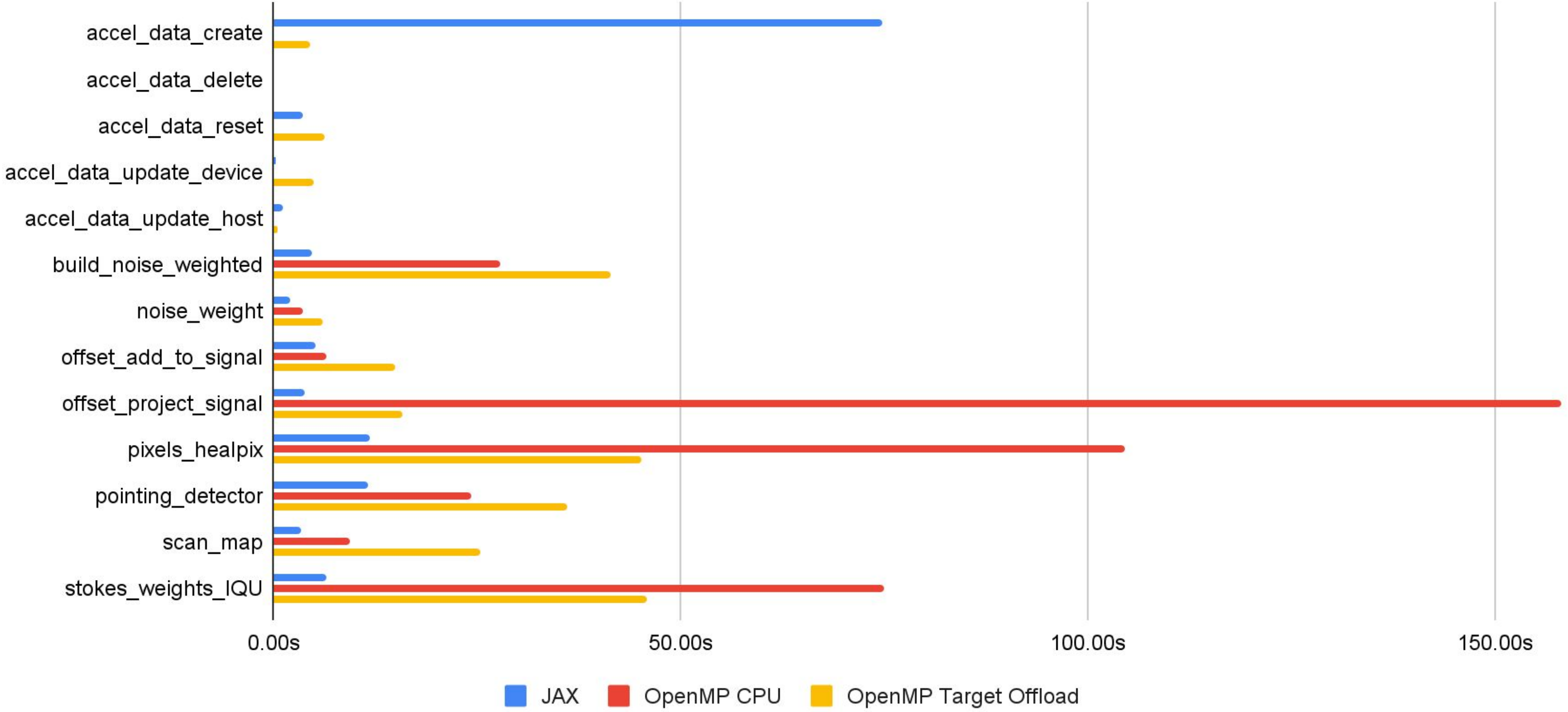
Porting the code: Code size



Performance per process



Performance per kernel (up to x42 speed-up)



Conclusion

Overview and Perspectives

Overview

- **High level framework are a *viable* path** to good GPU performance, portability and productivity.

- **OpenMP Target Offload** is best used on **large preexisting OpenMP kernels**:
  - update existing code progressively,
  - more work to get to performant code,
  - ***gateway to GPU computing for C++ programmers***.

- **JAX** is best used on **new Python project**:
  - *design* for simpler code,
  - possibility to JIT more / all code,
  - ***sweet spot for research and complex numerical codes***.

Perspectives

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

- Fix small **performance** bugs**,**
- **scale up to all kernels**, including the most complex ones,

- default to **JAX arrays** and **pure functions**,
- redesign pipelines to JIT them into **single GPU kernels**.

Perspectives

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

- Fix small **performance** bugs**,**
- **scale up to all kernels**, including the most complex ones,

- default to **JAX arrays** and **pure functions**,
- redesign pipelines to JIT them into **single GPU kernels**.

Trust our compilers and report their shortcomings as bug?

Thank you!

ndemeure@lbl.gov