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

Nestor Demeure*, Theodore Kisner†, Reijo Keskitalo†, Rollin Thomas*, Julian Borrill†, Wahid Bhimji*

* National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, California, U.S.A.
{ndemeure,rcthomas,wbhimji}@lbl.gov

† Computational Cosmology Center
Lawrence Berkeley National Laboratory
Berkeley, California, U.S.A.
{tskisner,rtkeskitalo,jdborrill}@lbl.gov

***Abstract*—In recent years, a common pattern has emerged where numerical software is designed around a Python interface calling high-performance kernels written in a lower level language. With the advent of general-purpose graphics processing units (GPUs), many of those kernels now need to be rewritten, a task which can seem daunting to those new to GPU programming. Furthermore, developers also need to ensure that their code will be both portable to future GPU architectures and flexible enough to evolve with their needs. Higher-level approaches that abstract away system architecture details may meet these needs, with some assumed performance trade-off, and in recent years several such frameworks have been proposed or developed. This paper is a side-by-side comparative case study of using two such higher-level frameworks, JAX and OpenMP target offload, to produce straightforward and portable code while achieving good GPU performance in a real science application. JAX is a Python library that allows us to write our kernels in pure Python, while OpenMP target offload is a directive-based strategy that integrates seamlessly with our already OpenMP-accelerated C++ kernels. The science application we consider is TOAST, a simulation and analysis framework for studying the cosmic microwave background that is designed to take full advantage of a supercomputer. We ported a dozen TOAST kernels to both frameworks in order to compare development cost, run times and to study whether they can be used to port a given a complex numerical code.**

***Index Terms*—Graphics processing unit, Application software**

## I. Introduction

It is now common for scientific software to be designed with a high-level, user-friendly interface written in Python that leverages high-performance kernels written in a lower-level, compiled language such as C, C++, or Fortran. This structure enables domain scientists to interact with applications through a familiar and popular programming language without having to become experts in high-performance computing (HPC). It also enables those scientists or developers with the requisite HPC skills and experience to accelerate these applications by identifying and acting on opportunities for thread-level parallelism and otherwise tuning performance in the lower-level code. A consequence is that domain scientists who mostly know Python end up depending on expert HPC software engineers when they need the application to do something new at a high level of performance. As an application's user base grows, this creates a bottleneck where science depends on a typically small team of developers with very specific skills.

The increasing adoption of general-purpose graphics processing units (GPUs) in HPC adds further motivation and complexity to this paradigm. GPUs offer lower energy consumption, allowing supercomputers to scale further, and promise significant performance improvements for those who can harness their potential. However, developers now face the task of refactoring many of their lower-level high-performance kernels for GPU architectures, a task even more daunting to those unfamiliar with GPU programming. Moreover, those scientific applications must be portable across different and future GPU architectures, flexible enough to tackle new kinds of problems, and maintainable by a broader cross-section of scientists and developers.

To explore these issues, we investigate two alternative approaches to porting a case-study Python-based HPC science application to GPU architectures through higher-level frameworks. One approach sets aside the two-language paradigm through the adoption of JAX [1] which brings just-in-time compilation, vectorization, and automatic parallelization to Python. The other, more traditional approach we consider is to retain the two-language model but use OpenMP target offload [2] to port lower-level kernels to GPUs. The goal is to evaluate the possibility of obtaining good GPU performance without sacrificing portability and readability, producing kernels that can be maintained and updated by domain experts as the code evolves.

We used TOAST [3] (Time Ordered Astrophysics Scalable Tools), a software framework for simulating and processing timestream data collected by microwave telescopes in order to measure the Cosmic Microwave Background (CMB), as our case study application. TOAST provides simulation and analysis capabilities for experiments like CMB-S4 [4], Simons Observatory [5] and the associated international CMB research community including university faculty, early career scientists, and even students. As such, it is expected to run both on a laptop, where students might use it, as well as on a supercomputer at full scale using MPI [6]. The challenge is to achieve a balance of both high performance and high productivity. These competing concerns, and the established preference for

Python among CMB domain scientists, guided the developers to a hybrid Python/C++ model. TOAST is written in Python and composed of pipelines that simulate or process data with a series of kernels, usually written in C++ and parallelized with OpenMP. CMB scientists who have limited HPC experience can code custom modules in Python that interface with the lower-level high-performance C++ kernels. The kernels make use of linear algebra, fast Fourier transformations and random number generation, covering a wide range of commonly used numerical building blocks.

Since TOAST already uses OpenMP to parallelize loops, it was natural to consider OpenMP target offload [2], a set of constructs introduced in OpenMP 4.0 [7] and further updated since. Available in the major C++ compilers (such as Clang [8] and GCC [9]), OpenMP target offload enables parallelism on heterogeneous systems and in particular portable GPU parallelism (the aforementioned compilers support both Nvidia and AMD hardware). Following the OpenMP philosophy, the instructions are relatively high level, providing basic building blocks to control data movement and split work, and introduced into normal C++ code via pragma directives. OpenMP target offload is of particular interest to us because it integrates seamlessly with our already OpenMP-accelerated C++ kernels and requires no additions to the compiler tool chain.

However, as an alternative to the two-language model, we also explore porting TOAST to JAX [1], a Python library developed by Google Brain as a building block for deep-learning frameworks and now seeing wider use in numerical applications such as molecular dynamics [10], computational fluid dynamics [11], [12] and ocean simulation [13]. It provides an array oriented library (similar to NumPy [14]) coupled with a just-in-time compiler to optimize the code for the hardware available: CPU, GPU (Nvidia as well as AMD) and more exotic architectures such as TPU. JAX came to our attention as a way to have a single-source implementation written in Python (with the associated productivity benefits) while targeting both CPU and GPU.

In this case study, we focus on a satellite telescope simulation, as it exercises a wide range of kernels, drawing on TOAST's ability to simulate and analyze CMB data across various scenarios. Our target hardware for this study is the Perlmutter supercomputer [15]. Equipped with Slingshot interconnect, 6,159 Nvidia A100 GPUs and approximately 1,500 AMD Milan CPUs with 64 processors each: it can deliver about 60 petaflops of peak double-precision (FP64) performance. Through the course of our analysis, we observed several factors, including the number of lines of code, the appearance and structure of the code, and the runtime performance. We also examined the effects of varying the number of processes on a single node and assessed the impact on realistic problem sizes.

The first section of this paper introduces TOAST and outlines the requirements that guided our study. Sections Two and Three focus on the OpenMP target offload and JAX ports, respectively, discussing their advantages, drawbacks, and the efforts made to overcome their limitations. In Section Four, we cover the porting of the codebase, and finally, in Section Five, we examine the performance of the resulting implementations.

## II. The TOAST codebase

TOAST [3] (Time Ordered Astrophysics Scalable Tools) is a software framework for simulating and processing time stream data collected by telescopes to estimate the Cosmic Microwave Background (CMB), which consists of faint, primordial photons created in the Big Bang. Fluctuations in the observed temperature and polarization of CMB photons encode fundamental parameters of cosmology and physics, making CMB measurements a valuable tool for cosmologists. To measure these exceedingly tiny fluctuations, CMB experiments gather, manage, analyze, and simulate enormous volumes of multi-frequency time-stream telescope data. This has made HPC essential to manage the exponential growth in CMB data volumes. TOAST provides simulation and analysis capabilities for experiments like the proposed "Stage 4" experiment CMB-S4 [4] and the associated international CMB research community.

A typical TOAST simulation workflow includes the following steps:

1) Simulate the telescope scanning, involving the telescope's position and orientation (boresight) as a function of time.
2) Simulate detector time streams acquired during telescope scanning, which may comprise sky signal, instrumental noise, and/or instrumental systematics. Modern telescopes host tens of thousands of detectors recording time-ordered data as the telescope scans the sky.
3) Reduce the time streams into pixelized sky maps, involving noise mitigation through a process called *destriping* [16], and solving for intensity and polarization in each of the observed sky pixels.

### A. The TOAST codebase

TOAST is a large Python software framework used to develop workflows (Python programs) for simulating and reducing CMB data. Workflows are composed of *pipelines* that call sequences *operators* one after the other on subsets of data. It is distributed using MPI at the pipeline level, meaning that we can ignore MPI while porting individual operators.

Each *operator* calls one or more computation kernels written in C++, parallelized with OpenMP and bound to Python using pybind11 [17]. This means that the computation-intensive operations are compartmentalized and can be replaced one at a time. These kernels make use of a wide variety of numerical components including linear algebra, fast Fourier transform and random number generation.

Our immediate goal is to replace enough of the kernels so that the dominant pipelines can run a sequence of operators with intermediate data kept on the GPU. Furthermore, the code should still be able to run on the CPU depending on the hardware available to the user. For the satellite simulation considered here, there are two large pipelines run for each iteration of a conjugate gradient solver. Our eventual goal is to

achieve performance portability across GPU architectures from the dominant vendors, as well as improved CPU performance on manycore traditional architectures. The TOAST package includes hundreds of unit tests and an integrated benchmark suite. These enable us to test our kernel implementations for correctness in a variety of use cases and compiler / Python environments.

### B. The test kernels

We ported the following 10 kernels, careful to preserve the API of the original code, 8 of which will be used in our benchmark:

- `build_noise_weighted` – accumulate noise-weighted timestreams onto a sky map,
- `noise_weight` – scale timestreams with noise weights,
- `pixels_healpix` – translate detector pointing angles into HEALPix [18] pixel numbers,
- `pointing_detector` – expand boresight pointing into detector pointing angles,
- `scan_map` – scan a pixelized sky map onto a timestream,
- `stokes_weights_I` – return a trivial weight vector,
- `stokes_weights_IQU` – compute detector response (weight) to intensity (I) and linear polarization (Q, U) on the sky at the time of each time sample,
- `template_offset_add_to_signal` – scan a step-wise noise offset solution onto a timestream,
- `template_offset_project_signal` – compute the dot product between a set of noise offset steps and a timestream,
- `template_offset_apply_diag_precond` – apply a diagonal preconditioner matrix to a noise offset problem (a sparse linear system).

### C. GPU computing in Python

Previous studies on hybrid language codes have demonstrated the feasibility and benefits of porting Python applications to GPUs. For example, the DESI project [19] ported its Python-based data processing pipeline to run on GPU devices, achieving a 20x improvement in per-node throughput compared to a CPU-only implementation. Another notable case is the PyFR project [20], a high-performance CFD framework that uses code-generation for solving problems on unstructured grids, targeting clusters of CPUs and NVIDIA GPUs.

Looking at previous studies and available libraries, we found four main options to write GPU code in Python:

*1) Using off-the-shelf kernels:* When considering porting a codebase to GPU, a common approach is to use off-the-shelf kernels provided by libraries such as CuPy [21] and RAPIDS [22]. CuPy provides an interface designed to be as similar as possible to NumPy [14] and SciPy [23] while supporting both NVIDIA and AMD GPUs (with experimental ROCm support). RAPIDS, built on CUDA, offers GPU-accelerated data science workflows, allowing the replacement of Pandas [24] with RAPIDS CuDF and Scikit-learn [25] with RAPIDS CuML.

The primary advantage of using off-the-shelf kernels is their ease of use and suitability for cases where the required functionality is already available. However, there are some drawbacks: the inability to write custom kernels tailored to specific needs and potential performance losses due to the allocation of intermediate values and increased data transfer. These limitations may lead to suboptimal performance compared to a fully optimized, custom-built GPU kernel.

*2) Writing a kernel in a low-level language:* Another approach for porting a codebase to GPU is to write a kernel in a low-level language, such as CUDA [26], OpenCL [27], HIP [28], SYCL [29], or OpenMP target offload [2], and then link it to Python. Libraries like PyOpenCL [30] or PyCuda [30] can be employed to establish the connection between the kernel and the Python code.

This method offers perfect control of performance and allows developers to fine-tune their code for optimal results. However, there are several downsides: it is difficult to reuse numerical building blocks (such as random number generators, fast Fourier transform and linear algebra), it requires a significant level of expertise to write performant and correct code, and compiling and linking the result into Python can be challenging. Additionally, some of these options lack portability across different hardware architectures.

OpenMP target offload is of particular interest to us because it maintains a relatively high level of abstraction and produces portable code. Plus, since OpenMP kernels are already part of the codebase, the linking work is completed, and developers mainly need to add the device code to enable GPU acceleration.

*3) Writing a kernel in Python:* For a more accessible approach to writing GPU kernels, Python-based options such as Numba [31] (applying their JIT compiler to a low-level CUDA-like syntax), Taichi [32] (focussing on graphics), and Triton [33] (offering a unique low-level syntax, diverging from the CUDA model) can be used.

One significant advantage of these options is the ability to maintain a full Python codebase, which simplifies code management and reduces the learning curve for developers. This streamlined approach can lead to a more maintainable and accessible codebase, particularly for those with a strong Python background. However, those options have in common the fact that they drop down to a very low-level syntax, which is a loss for accessibility, and they provide limited (or non-existent) support for libraries and common numerical operations.

*4) Using a Deep-Learning library:* Leveraging deep learning libraries such as PyTorch [34], TensorFlow [35], or JAX [1] is an attractive solution as it is convenient, modern, and well-documented approach to writing GPU kernels. Furthermore, these libraries offer great support for most numerical building blocks, including linear algebra, random numbers, solvers, fast Fourier transforms, and complex numbers, which can result in an ergonomic and easy-to-use Python kernel.

However, deep learning libraries are often disappointing when used for straight numerical computations as they opti-

mize for training speed and come with a large overhead mostly due to gradient computation and intermediate values.

JAX is an exception in this regard, as it builds on code transformation and JIT compilation. Letting us avoid gradient computation costs when we are not using the functionality and simplifying intermediate values at compile time. This makes it an attractive solution for GPU kernel development.

After considering various options for GPU kernel development, we decided to evaluate JAX and OpenMP target offload as the most promising candidates for our needs. OpenMP target offload holds significant appeal due to its seamless integration with our existing OpenMP-accelerated C++ kernels and the absence of any additional compiler tool chain requirements. On the other hand, JAX presents an opportunity to maintain a single-source implementation in Python, which can provide productivity benefits while targeting both CPU and GPU platforms. By exploring these two approaches, we aim to find a balance between performance, portability, and ease of development in our codebase.

## III. OpenMP Target Offload

The OpenMP specification [36] has supported a mechanism for offloading data or computation to "target" computing devices, such as accelerators and GPUs, since version 4.0. As TOAST already utilizes OpenMP for CPU parallelism, considering OpenMP target offload for porting TOAST to GPUs was a natural choice.

### A. Design

To use OpenMP target offload, compute-intensive code kernels are labeled with directives (`#pragma omp target` in C++) and enclosed in a structured block. Upon execution, instructions in the code region are mapped to the target device and executed. "Clauses" in the OpenMP directive allow control over various aspects such as execution, data privacy, and synchronization.

As demonstrated in other case studies [37], OpenMP target offload is particularly attractive because it is designed to abstract hardware details, enabling the same code to execute across multiple target devices without modification. Furthermore, it can be combined with vendor-optimized GPU libraries for BLAS, FFT, and other operations by passing device memory pointers directly as their input.

### B. OpenMP Target Offload Limitations

Although OpenMP target offload theoretically enables developers to port existing compiled code with minimal changes and achieve performance gains across devices, such effortless porting is only possible for simply structured and compute-intensive code. Allocating device memory and moving large buffers can be slow, and porting modular (unfused) lightweight kernels requires careful data movement management outside individual kernel blocks.

Although some compiler implementations (e.g. LLVM) do attempt to use asynchronous data movement and kernel submission internally, achieving good overlap of kernel submission and execution often requires manually specifying data dependencies. Advanced OpenMP techniques can push the limits of currently available compiler support, but real-world examples of such patterns are scarce.

### C. The Porting Process

TOAST's existing codebase contains OpenMP threaded blocks in C++ for the most expensive parts of a typical workflow. TOAST data simulation and processing operators are intentionally modular and lightweight, allowing science domain experts with minimal coding experience to drop-in customizations for particular tasks while leveraging the rest of the framework. When moving these kernels to target offload, maintaining modularity was a goal.

TOAST `Operator` classes track required data inputs and outputs. A C++ singleton class was implemented to manage device memory buffers allocated with `omp_target_alloc()` and their host memory counterparts. This class provides methods for host-to-device and device-to-host updates, creating, deleting and zeroing device memory, and other helper functions. Higher-level TOAST code (`Pipeline` class) can then use operator dependency information to move data to/from the device after a sequence of lightweight kernels. Furthermore, the largest memory objects (containing detector timestreams) support buffer re-use, enabling repeated execution of kernel sequences for different inputs while re-using intermediate data objects.

Because we are not yet using explicit data dependencies inside kernel sequences, each sequence of kernels is submitted to the default CUDA context and each kernel is run synchronously, waiting for its completion before running the next kernel. To simulate the performance that would be obtained with asynchronous kernel dispatch, we are relying on multiple MPI processes submitting kernels to each GPU concurrently. Initially this over-subscription failed to improve performance. This was due to the default behavior of the CUDA driver which context-switches between processes, putting a ceiling of one process per device on the performance. This was fixed by introducing NVIDIA Multi-Process Service (MPS) [38], automatically allocating a single set of storage and scheduling resources on each device and funnels kernel submission from multiple processes. Using MPS, we were able to effectively oversubscribe the GPUs keep scaling our performance proportionally with the number of process (see section VI-A).

The TOAST package already contained a compiled Python extension using OpenMP threading. For OpenMP target offload, additions were made to the build system to enable target offload support at compile time when using a supported compiler. After evaluating several compilers (NVIDIA NVC, Clang, ROCmCC, GCC), we settled on NVIDIA NVC as it has good support on Perlmutter and covers all of the functionalities we needed for our kernels.

## IV. The JAX library

JAX [1] is a Python library that provides a high level interface designed to be as similar as possible to NumPy [14] and SciPy [23] (similarly to CuPy [21]) coupled with a JIT compiler to fuse kernels and elide intermediate results. It is available for CPU, GPU (both Nvidia and an experimental support for AMD) and some deep-learning focused alternative architectures (TPU and specialized hardware). JAX is purposefully high-level, it restricts the freedom of the developer (see section IV-B) but, in exchange, separates the semantics of the code (left to the developer) from its optimization (left to the compiler).

### A. Design

JAX requires kernels to be *pure and statically composed* meaning that they do not have side effects and that the computation can be expressed as a static data dependency graph whose nodes are taken from a set of primitives.

To do so, it provides a set of primitives as well as NumPy- and SciPy-like interfaces. One can write a program using those operations and they will run individually, as CuPy operations would. However, one can also JIT-compile a function written using JAX primitives. When a JIT compiled function is called, it is traced then transcribed into the "High Level Operations" (HLO) intermediate representation[1] to be compiled by the XLA [39] compiler which will then run it on the target architecture (as seen on figure 1). Subsequent runs will reuse the compiled function.


Fig. 1. JAX workflow.

Due to its NumPy and SciPy interface, it comes with out-of-the-box support for linear algebra, fast Fourier transform and random number generation (plus library support for MPI via the MPI4jax library [40]) all of which are used within TOAST.

### B. JAX specific limitations

The assumption that the functions are pure and statically composed introduces four strong limitations when writing JAX code, some of which mattered to us as we are starting from an existing codebase and striving to preserve the API of the existing kernels:

[1]Interestingly, this representation encapsulates information on the shape of the inputs and intermediate results meaning that the compiler can work with full knowledge of the size of the tensors in play. In theory, this means that the compiler could pick different loops to be parallelized depending on the problem size.

*1) Purity:* Having all operation be pure means that common operations such as updating an element of an array in place are forbidden (as it would be, by definition, a side effect). JAX provides alternative operations that create a new updated array, such as `x.at[idx].set(y)`, that cover most use cases.

However, all of our kernels have output arguments that should be modified in place. To preserve the kernel API we had introduced a `MutableJaxArray` class that boxes a JAX array such that one can read and set slices of data as one would with a NumPy array. Internally, our kernels pass the data inside the `MutableJaxArray` to *pure* JIT compiled functions (using the `donate_argnums` option to reuse memory when possible) that produce an output that is then put back into the `MutableJaxArray`.

*2) Conditional:* Traced values are considered unknown at tracing time, meaning that one cannot have a conditional dependence on a value that will be traced.

JAX provide some primitives as well as the possibility to mark an input as static (considered constant across calls to the function and thus elided at tracing time) to deal with this limitation.

In practice we have never felt strongly impacted by the particular limitation.

*3) Loops:* The fact that conditionals cannot depend on traced values also restricts the looping behavior but, similarly to conditionals, JAX comes with primitives to work around it and, in practice, it has not been a direct problem for us (see IV-B4).

However, one needs to be aware of the fact that JAX will unroll loops whose number of iterations is known at tracing time. While easily avoided, this can lead to large accidental increase of the compilation time (which is not desirable as the compilation happens while the program is running).

*4) Static array shapes:* JAX expects to be able to know all array shapes at tracing time, meaning that an array size cannot be a function of data. For example, a programmer cannot pass the beginning and end of an interval to extract it into a kernel. This presented a problem to us as a large number of our kernels consist of a loop over irregularly sized intervals

The simplest fix is to JIT only the body of the loop (where the shape *is* known). This gave us adequate performance but meant that some JAX operations would be performed outside of a JIT compiled section, where they are significantly more expensive, and that we would be less likely to saturate the GPU with computation.

A better fix is to introduce padding and/or masking or to find a way to make an array size deducible from statically known elements. We thus decided to introduce a `JaxIntervals` type that can be built from arrays of interval starting points, ending points and a statically known maximum interval size (meaning that the size of the padded array was known at tracing time). This type can be used to extract intervals as a single block, padded to the maximum interval size (using either values from the array being sliced or user-provided values when it matters to the subsequent algorithm), and to update several intervals at once with new data (masking

data that is out of the interval). While this increased code complexity, it also reduced our computing time significantly.

### C. The porting process

The port was done in two steps, first from C++ to NumPy, staying as close as possible to the original. Then, exploiting the similarity between NumPy and JAX, from NumPy to JAX turning loops into calls to `vmap` or `xmap` (vectorizing the loop body, see section V-B and Appendix B for an example) and removing side effects. Thorough the process, we used the extensive unit tests available in the codebase to validate our work after each step.

Interestingly, MPS was not needed to use several process per device efficiently with JAX (contrary to OpenMP Offload as seen in section III-C). Likely because JAX uses the NVIDIA Collective Communication Library (NCCL) [41] natively to deal with multi-device connections. The one modification we did in order to oversubscribe devices smoothly is to deactivate device memory preallocation. By default, JAX will preallocate most of the device memory when started in order to construct a memory pool but, it can results in out of memory errors if one is not careful with allocation size when several processes share the sane device. Deactivating the preallocation was the, recommended, easiest way to deal with the problem and our measurements shown no performance degradation (likely because we batch most of our allocations at the beginning and end of the pipelines).

Overall, writing the JAX code felt very productive, its use of interfaces designed to be as close as possible to NumPy made it extremely easy to find help implementing functionality (one can search for help implementing any functionality in NumPy first) and the design of the library let us focus on the semantics of the operations and readability of the implementation, knowing that the compiler would be able to deal with missed optimization and wasteful copies.

On the flip side, this meant that there was very little scope for optimization inside a JIT compiled function, as optimization ends up pushed to the interface. If a JIT compiled function does not perform as fast as desired, the main fixes are pushing more operations into the JIT compiled section (to remove inefficiencies at the interface), trying to keep the data on GPU or trying to reduce data movement. The interface between the JIT compiled function and the rest of the code is where we found *all* of our performance bugs.

Overall we found it easy to deal with loops and conditionals and, while it took some workarounds to deal with dynamic shapes and mutability, the port was mostly straightforward. Furthermore, it is our belief that, had the code been designed with JAX in mind from the beginning, most of those workarounds could have been avoided.

## V. Porting the TOAST codebase

### A. Framework-Agnostic Modifications

To ensure flexibility in our codebase and facilitate testing of new GPU technologies, we developed a framework-agnostic approach. This method introduces additional layers of indirection but offers the advantage of allowing easy extension to future accelerators, a crucial guarantee for long-running CMB projects.

*1) Abstraction Layers:* We designed a runtime dispatch system over kernels, enabling the selection of specific implementations for the entire code, individual pipelines, and kernels. Additionally, we created an abstraction layer for memory operations, including allocation, deallocation, and data transfer between devices. This layer allows us to manage data movement within our pipeline in a framework-agnostic manner.

*2) Hybrid Pipelines:* Each operator includes information regarding GPU support and a list of input and output data it handles. This information allows us to implement data movement logic within our pipeline. By default, all GPU-enabled operators are executed on the GPU. When an operator is called, we ensure that the required data is in the correct location (GPU or CPU). At the end of the pipeline, the final output is transferred back to the CPU, and any leftover data on the GPU is deleted.

This approach enables transparent handling of various TOAST looping patterns (looping on detectors, then operators; on operators, then detectors; on specific detectors only, etc.). Additionally, it allows us to easily run only a subset of operators on the GPU for testing and debugging purposes. Lastly, it significantly reduces data movement compared to the naive approach of transferring data to/from the GPU whenever a GPU kernel is called (something that both JAX and OpenMP Target Offload are able to do). In early tests, this optimization resulted in a 40% speedup compared to a naive implementation (confirming the well-known fact that data movement is expensive).

*3) Profiling:* TOAST already included support for collecting coarse timing information of functions (including gathering that information across an MPI job) through the use of a custom python decorator. This timing information can then be dumped to a CSV file. We expanded on this functionality with a script that merges several CSV files into a comparative spreadsheet. This has been a tremendously useful and simple tool to identify operations where our updated code spent a suspect amount of time, allowing us to address performance issues effectively.

### B. Example kernel

The `stokes_weights_IQU` kernel is a good example of the patterns found and used in the port.

Appendix A shows the OpenMP target offload version of the code, omitting the code relevant to binding C++ to Python, the CPU version of the code (a duplicate of the main loop without the `#pragma omp target` pragmas) and some edge cases (also omitted from the JAX code). It consists of a loop over detectors, intervals and samples within intervals. The loop body (`stokes_weights_IQU_inner`) extracts a piece of data using some indexing information, computes a result then stores it in an output variable (`weights`).

*1) OpenMP target offload:* The OpenMP target offload parallelization is fairly straightforward, we get pointers to the device data, then parallelize a triply nested loop with pragmas. The first two loops are distributed over GPU teams while the inner loop is parallelized directly, as one would do on CPU.

*2) JAX:* The JAX port of the code (Appendix B) might seem less intuitive but it illustrates several layers of indirection present in all our JAX kernels. The `stokes_weights_IQU_inner` function is a fairly straightforward port that reads nearly identical to our NumPy implementation. It is parallelized over detectors, intervals and samples using `jax.xmap`, vectorizing the loop body over the loop axes (a common pattern in JAX code). The function is called by a `stokes_weights_IQU_interval` function whose sole purpose is to extract and update intervals into a padded representation, using our `JaxIntervals` class. This is JIT-compiled using `jax.jit`, specifying that some inputs are static (such as the maximum size of the intervals) and that JAX is allowed to recycle the memory of the output parameter. Finally, `stokes_weights_IQU` is our outer layer, reproducing the C++ interface. It uses `MutableJaxArray.to_array` to insure that all inputs are JAX or NumPy array (this is only needed when calling the JAX kernels on CPU, to deal with TOAST specific datatypes such as `MPIShared` arrays that deal with distributed data).

### C. Analysis

Compared to our original OpenMP (CPU) kernels, JAX kernels are on average 1.2 times *shorter* while OpenMP target offload are on average 1.8 times *longer*[2] as illustrated in figure 2.

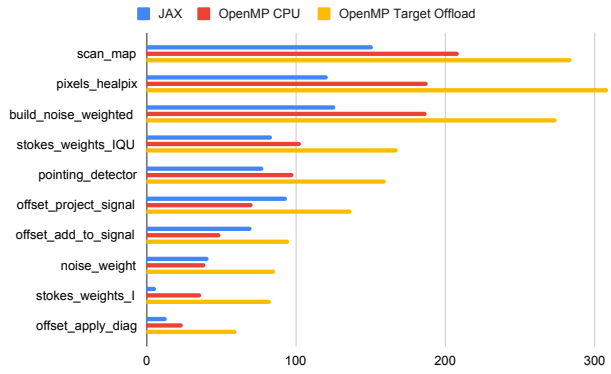


Fig. 2. Number of lines of code per kernel.

The OpenMP target offload code shares inner function and dependencies with the original code. Thus, all additional lines of code inside kernels are due to the duplication of the main loops and addition of the GPU specific pragmas and statements. This means, as a very rough rule of thumb, that the GPU specific logic took almost as many lines of code as the actual inner function (doing the numerical computations inside our loops).

Meanwhile, the fact that the JAX code is shorter than our original code, despite producing code that is *both* CPU and GPU compatible and adding various layers of indirection, boils down to JAX letting use a NumPy like syntax and high level operations in Python. This leads to a large number of code simplifications such as the fact that we do not have to compute indices within multidimensional array manually.

Furthermore, looking at the code that is not a kernel (seen in the difference between the two bars of figure 3) we see that the JAX code used to implement dependencies and GPU operations is significantly (3 times) shorter than the corresponding OpenMP target offload code. This is explained by the fact that we both require less dependencies (being able to reuse Python libraries) and are implementing the logic in a higher level language.

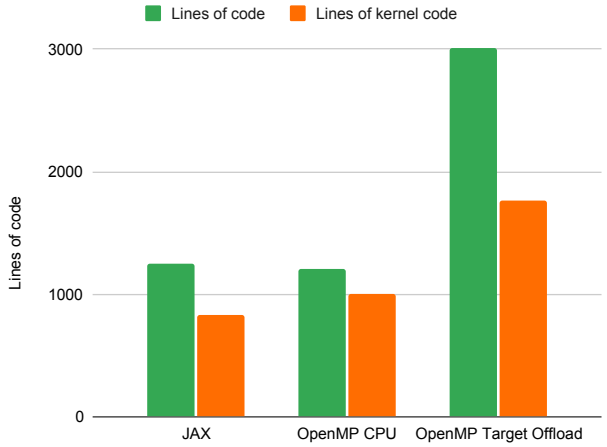


Fig. 3. Number of lines of code per implementation. *Lines of kernel code* is the number of lines used strictly in the kernel implementations while *Lines of code* includes the dependencies of the kernels and the accelerator (data movement, GPU related types, etc.) related code.

## VI. Results

The following measurements were made on GPU nodes of the Perlmutter supercomputer. Each GPU node is equipped with four 40 GB NVIDIA A100 GPUs, with 256 GB of CPU memory, and a single AMD Milan CPU with 64 processors.

Measurements have been made on various problem sizes of a satellite telescope simulation. This benchmark workflow simulates the characteristic scanning motion of a space-based CMB telescope and uses a typical instrument configuration with a couple thousand detectors observing a simulated sky. The data from each detector includes simulated signal from the sky, simulated realistic noise, and other typical features of the detector response:

- *medium size* – for all the single node runs, which uses $5 \times 10^9$ samples (roughly one terabyte of data)
- *large size* – for the 8 nodes run, which uses $5 \times 10^{10}$ samples (roughly ten terabytes of data)

[2]Measured with `cloc v1.82` [42], not counting empty lines and comments.

The runtime includes everything from the time needed to load the data to the time needed to export the outputs, including the JIT compiling time for the JAX version of the code and the MPI communication cost.

### A. Number of process

Figure 4 gives us the evolution of the runtime when increasing the number of processes. Note that the number of threads per process is decreased proportionally as we increase the number of processes: we go from one process using 64 threads to 64 processes using one thread each.

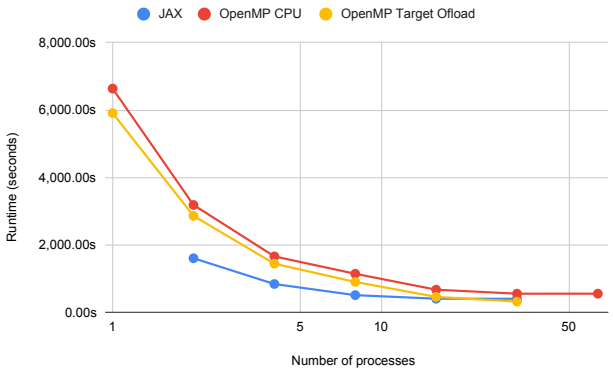


Fig. 4. Runtime as a function of the number of processes, running the medium problem size on 1 node.

The OpenMP CPU runtime decreases proportionally to the number of process. While one might expect that it would stay constant (as the CPU processing power stays constant, threads decrease being balanced out by process increase), the decrease is explained by the fact that a large number of operations are serial within a process and are parallelized by the addition of more processes at the price of increased memory use.

The JAX plot shows a similar behavior (note that it does not fit on GPU memory when running with one and 64 processes). Interestingly, it shows a 2x speedup up to 8 processes (two process per GPU), demonstrating a benefit to oversubscribing the GPUs. The speedup then slowly decreases to 1.7x (16 processes) then 1.4x (32 processes) as we progressively lose the oversubscription benefit.

OpenMP target offload starts slightly faster than OpenMP CPU for up to 4 processes (about 1.1x times faster) but it plateaus slower than OpenMP CPU, reaching a speedup of 1.5x (16 processes) then 1.8x (32) processes, at which point it becomes faster than JAX. This i slikely a sign that, while OpenMP target offload does not make the kernels as fast as JAX (as seen in the section VI-B), it has a lower overhead (JAX overhead includes JIT compiling and check if a given function has already been compiled for a given problem size) which benefits it as the work get spread between processes.

We will use 16 process as our default for the rest of this study as it is the default in most TOAST simulations.

### B. Full benchmark

Figure 5 illustrates a run with a realistic problem size (about 10 terabytes of data spread over 8 nodes, using 16 process per node). We see about similar behavior as when running a smaller problem on one node and 16 processes: compared to the CPU reference, JAX is 1.67x times faster and OpenMP target offload is 1.47x times faster.

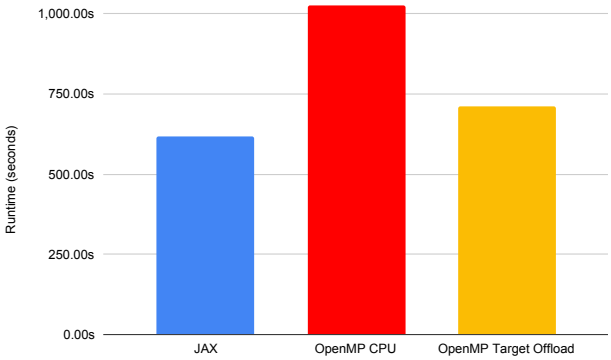


Fig. 5. Runtime as a function of the kernel implementation, running the large problem size on 8 node, with 16 processes per node.

Examining per-kernel performance (figure 6), we see that JAX goes from a slight speed up (as low as 1.3x times faster for `offset_add_to_signal`, a kernel doing very little computation) to 12x and 42x speedups on some of the kernels that were most expensive to run on CPU (`stokes_weights_IQU` and `offset_project_signal` respectively). This is in line with the idea that we benefit further when we are able to push more work onto the GPU (something confirmed by the fact that we see benefits to oversubscribing the GPU). `pixels_healpix` is the expensive kernel that benefits least (a 9x speedup), this was however expected as this kernel has a lot of branches, with dozen of variables declared per branches, something which is known to be expensive on GPU.

Looking at OpenMP target offload, we see that it is slower than our CPU baseline (1.9x times slower on average) on all but the three most computationally expensive kernels: `stokes_weights_IQU` (1.6x times faster), `pixels_healpix` (2.3x times faster) and `offset_project_signal` (10x times faster). Interestingly, those kernels are expensive enough to make OpenMP target offload faster than our CPU baseline.

Finally, while data movement to and from the device is expensive, our pipelining infrastructure seems to be doing a good job at minimizing this cost as most of the data operations barely register on the plot. One noticeable exception is `accel_data_create` (which allocates GPU memory *and* moves data from host to device) being significantly slower with JAX, taking about 15% of the JAX runtime. Looking into it, we found a divergence in behavior between both of our implementations (the OpenMP target offload implementation

recycles existing buffers, setting them to 0) which, we believe, is fixable.

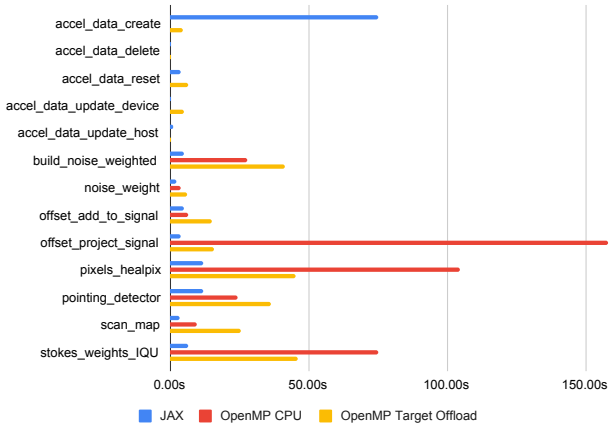


Fig. 6. Total runtime for each kernel, running the medium problem size with 16 processes on 1 node. `accel_data` functions represent data movement operations happening out of the kernels.

## VII. Discussion

Our experience has left us with the belief that high-level frameworks can fulfill a niche in the Pareto front by committing to very high-level abstractions. OpenMP Target Offload, occupying a somewhat intermediate position, appears to be limited in its ability to provide access to low-level functionalities that could result in improved performance, while still granting enough leeway for performance pitfalls. We believe it is best used on *codebases with large preexisting OpenMP kernels*, as it allows for progressive updates to existing code and can serves as a gateway to GPU computing. With further effort, we might be able to achieve parity with the JAX implementation. However, making the code run correctly and performantly has proven to be time-consuming, especially given the sparse compiler support, documentation, and usage examples currently available.

On the other hand, we found JAX's concept of imposing restrictions on users to empower its compiler to be surprisingly beneficial. These constraints did not prove problematic, even when preserving the idioms of an existing mutability-oriented codebase, and the JAX compiler excelled at generating performant GPU code, with delegating larger portions of the code to it consistently improving performance. This approach enabled us to write efficient GPU code very productively, with performance challenges primarily arising at the interface with the JIT and during data conversion. JAX feels particularly well suited for *new Python projects* which can be designed around it: focusing on immutability and static sizes, opening the door to code simplifications and JIT compiling as many things as possible.

Despite the advantages of working with JAX, we encountered two noticeable drawbacks that are worth mentioning. First, JAX's CPU performance is roughly comparable to single-threaded C++, making it too slow to run most of our benchmarks with JAX CPU. Consequently, for our application, we cannot currently rely on JAX for CPU runs. Second, JAX splits the code into ad hoc internal kernels that are difficult to map back to the JIT compiled function. This impedes the use of conventional tools like Nsight [43], which could have provided us with valuable per kernel information, such as roofline plots, to analyze the JAX accelerated code.

Finally, for all implementations, we found that *effective profiling tools are invaluable for performance optimization*. We used custom, ad-hoc tools and decorators to compare various code variants on an operation-by-operation basis, and this proved to be the most significant productivity boost throughout the project.

## VIII. Conclusion

In this case study, we investigated the porting of 10 kernels from OpenMP CPU to JAX and OpenMP target offload, analyzing the performance of the resulting application.

We observed a 1.47x times speedup on a realistic benchmark when using OpenMP target offload, getting individual kernels – in particular the most compute intensive ones – to run up to 10x times faster. The kernel code shares logic with the preexisting kernels, resulting in a codebase about 1.8 times longer than the reference while staying fairly straightforward to read by someone used to OpenMP and the original codebase. Overall, we believe that current compiler support and documentation for OpenMP target offload are insufficient (something also discussed in [37]), making the technology challenging to use to its full potential, despite OpenMP's strong presence in shared memory parallelism and its extensive user base.

On the other hand, JAX, a relatively new technology, provided a 1.67x times speedup and accelerated individual kernels by up to 42x times. The resulting code, despite containing several layers of indirection, is 1.2 times shorter than the reference, primarily due to Python and the NumPy API being higher-level than C++ (simplifying loop bodies). While JAX's CPU backend needs further development for JAX to become a comprehensive solution, it allowed us to concentrate on semantics and readability within JIT compiled sections, delegating performance considerations to the interface and data movements.

In the short term, we are interested in fixing the `accel_data_create` JAX performance and further exploring the optimization of the OpenMP target offload backend to determine the performance ceiling of each implementation. In the longer term, if JAX's CPU backend becomes competitive with OpenMP CPU, it would be worthwhile to attempt a complete removal of C++ dependencies from TOAST, simplifying the codebase and transforming it into a Python-only application. Moreover, we have not yet explored JAX's automatic differentiation capabilities, but since one of TOAST's purposes is experiment design and optimization, differentiating the code to fit specific experimental parameters could prove valuable.

From a design perspective, we believe that JAX's array-oriented approach, applied to pure and statically composed functions compiled with a JIT compiler, represents a promising paradigm for GPU computing in terms of both productivity and performance of the resulting application.

## ACKNOWLEDGMENT

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC awards ASCR-ERCAP0021861 and HEP-ERCAP0023125.

## REFERENCES

[1] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[2] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading support for openmp in clang and llvm," in *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2016, pp. 1–11.

[3] T. Kisner, R. Keskitalo, A. Zonca, J. R. Madsen, J. Savarit, M. Tomasi, K. Cheung, G. Puglisi, D. Liu, and M. Hasselfield, "hpc4cmb/toast: Update pybind11," Oct. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5559597

[4] K. Abazajian, G. Addison, P. Adshead, Z. Ahmed, S. W. Allen, D. Alonso, M. Alvarez, A. Anderson, K. S. Arnold, C. Baccigalupi *et al.*, "Cmb-s4 science case, reference design, and project plan," *arXiv preprint arXiv:1907.04473*, 2019.

[5] P. Ade, J. Aguirre, Z. Ahmed, S. Aiola, A. Ali, D. Alonso, M. A. Alvarez, K. Arnold, P. Ashton, J. Austermann *et al.*, "The simons observatory: science goals and forecasts," *Journal of Cosmology and Astroparticle Physics*, vol. 2019, no. 02, p. 056, 2019.

[6] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[7] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," Jul. 2013. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[8] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008, pp. 1–20.

[9] R. M. Stallman *et al.*, *Using and porting the GNU compiler collection*. Free Software Foundation, 1999, vol. 86.

[10] S. S. Schoenholz and E. D. Cubuk, "Jax m.d. a framework for differentiable physics," in *Advances in Neural Information Processing Systems*, vol. 33. Curran Associates, Inc., 2020. [Online]. Available: https://papers.nips.cc/paper/2020/file/83d3d4b6c9579515e1679aca8cbc8033-Paper.pdf

[11] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, "Machine learning–accelerated computational fluid dynamics," *Proceedings of the National Academy of Sciences*, vol. 118, no. 21, 2021. [Online]. Available: https://www.pnas.org/content/118/21/e2101784118

[12] D. A. Bezgin, A. B. Buhendwa, and N. A. Adams, "Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows," *arXiv preprint arXiv:2203.13760*, 2022.

[13] D. Häfner, R. L. Jacobsen, C. Eden, M. R. B. Kristensen, M. Jochum, R. Nuterman, and B. Vinter, "Veros v0.1 – a fast and versatile ocean simulator in pure Python," *Geoscientific Model Development*, vol. 11, no. 8, pp. 3299–3312, Aug. 2018. [Online]. Available: https://gmd.copernicus.org/articles/11/3299/2018/

[14] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in science & engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[15] C. Yang and J. Deslippe, "Accelerate science on perlmutter with nersc," *Bulletin of the American Physical Society*, vol. 65, 2020.

[16] E. Keihänen, H. Kurki-Suonio, and T. Poutanen, "Madam- a map-making method for CMB experiments," *Monthly Notices of the Royal Astronomical Society*, vol. 360, no. 1, pp. 390–400, jun 2005. [Online]. Available: https://doi.org/10.1111%2Fj.1365-2966.2005.09055.x

[17] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11–seamless operability between c++ 11 and python," *URL: https://github.com/pybind/pybind11*, 2017.

[18] K. M. Gorski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann, "Healpix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere," *The Astrophysical Journal*, vol. 622, no. 2, p. 759, 2005.

[19] Daniel Margala, Laurie Stephey, Rollin Thomas, and Stephen Bailey, "Accelerating Spectroscopic Data Processing Using Python and GPUs on NERSC Supercomputers," in *Proceedings of the 20th Python in Science Conference*, Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, Eds., 2021, pp. 33 – 39.

[20] F. D. Witherden, A. M. Farrington, and P. E. Vincent, "Pyfr: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, vol. 185, no. 11, pp. 3028–3040, 2014.

[21] R. Nishino and S. H. C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," *31st conference on neural information processing systems*, vol. 151, no. 7, 2017.

[22] T. Hricik, D. Bader, and O. Green, "Using rapids ai to accelerate graph data science workflows," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–4.

[23] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[24] W. McKinney *et al.*, "pandas: a foundational python library for data analysis and statistics," *Python for high performance and scientific computing*, vol. 14, no. 9, pp. 1–9, 2011.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[26] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[27] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[28] P. Bauman, N. Chalmers, N. Curtis, C. Freitag, J. Greathouse, N. Malaya, D. McDougall, S. Moe, R. van Oostrum, N. Wolfe *et al.*, "Introduction to amd gpu programming with hip," *Presentation at Oak Ridge National Laboratory. Online at: https://www. olcf. ornl. gov/calendar/introto-amd-gpu-programming-with-hip/. 17Specifically, the GPU_MAX_HW_QUEUES variable*, 2019.

[29] R. Reyes and V. Lomüller, "Sycl: Single-source c++ accelerator programming," in *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[30] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[31] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[32] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: a language for high-performance computation on spatially sparse data structures," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–16, 2019.

[33] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[35] T. Developers, "Tensorflow," *Zenodo*, 2022.

[36] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[37] C. Daley, H. Ahmed, S. Williams, and N. Wright, "A case study of porting hpgmg from cuda to openmp target offload," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, K. Milfeld, B. R. de Supinski, L. Koesterke, and J. Klinkenberg, Eds. Cham: Springer International Publishing, 2020, pp. 37–51.

[38] N. Corporation, "Multi-process service: Gpu deployment and management," 2023. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[39] A. Sabne, "Xla: Compiling machine learning for peak performance," 2020.

[40] D. Häfner and F. Vicentini, "mpi4jax: Zero-copy mpi communication of jax arrays," *Journal of Open Source Software*, vol. 6, no. 65, p. 3419, 2021. [Online]. Available: https://doi.org/10.21105/joss.03419

[41] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, vol. 2, 2017.

[42] A. Danial, "cloc: v1.92," Dec. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5760077

[43] K. Iyer and J. Kiel, "Gpu debugging and profiling with nvidia parallel nsight," *Game Development Tools*, pp. 303–324, 2016.

## APPENDIX

### A. *OpenMP target offload implementation of the* `stokes_weights_IQU` *operator.*

```
/**
 * Compute the Stokes weights for one (detector, interval, sample).
 *
 * @param cal A constant to apply to the pointing weights.
 * @param quat_index Pointer to the array of detector quaternion indices (size n_det).
 * @param weight_index Pointer to the array of weight indices (size n_det).
 * @param quats Pointer to the array of detector quaternions (size ???*n_samp*4).
 * @param hwp Pointer to the array of HWP angles (size n_samp).
 * @param epsilon Pointer to the array of cross polar responses (size n_det).
 * @param weights Pointer to the array of flat packed detector weights for the specified mode
 ↪  (size n_det*n_samp*3).
 * @param isamp The current sample index.
 * @param n_samp Total number of samples.
 * @param idet The current detector index.
 *
 * @return None (the result is put in weights).
 */
void stokes_weights_IQU_inner(
    double cal,
    int32_t const * quat_index,
    int32_t const * weight_index,
    double const * quats,
    double const * hwp,
    double const * epsilon,
    double * weights,
    int64_t isamp,
    int64_t n_samp,
    int64_t idet
) {
    const double xaxis[3] = {1.0, 0.0, 0.0};
    const double zaxis[3] = {0.0, 0.0, 1.0};
    double eta = (1.0 - epsilon[idet]) / (1.0 + epsilon[idet]);
    int32_t q_indx = quat_index[idet];
    int32_t w_indx = weight_index[idet];

    double dir[3];
    double orient[3];

    int64_t off = (q_indx * 4 * n_samp) + 4 * isamp;
    quaternion_rotate(&(quats[off]), zaxis, dir);
    quaternion_rotate(&(quats[off]), xaxis, orient);

    double y = orient[0] * dir[1] - orient[1] * dir[0];
    double x = orient[0] * (-dir[2] * dir[0]) +
               orient[1] * (-dir[2] * dir[1]) +
               orient[2] * (dir[0] * dir[0] + dir[1] * dir[1]);

    double ang = 2.0 * (atan2(y, x) + 2.0 * hwp[isamp]);
    double cang = cos(ang);
    double sang = sin(ang);

    off = (w_indx * 3 * n_samp) + 3 * isamp;
    weights[off] = cal;
    weights[off + 1] = cang * eta * cal;
    weights[off + 2] = sang * eta * cal;
}


/**
 * Compute the Stokes weights for the "IQU" mode.
 *
 * @param quat_index Pointer to the array of detector quaternion indices (size n_det).
```

```
 * @param quats Pointer to the array of detector quaternions (size ???*n_samp*4).
 * @param weight_index Pointer to the array of weight indices (size n_det).
 * @param weights Pointer to the array of flat packed detector weights for the specified mode
 ↪  (size n_det*n_samp*3).
 * @param hwp Pointer to the array of HWP angles (size n_samp).
 * @param epsilon Pointer to the array of cross polar responses (size n_det).
 * @param cal A constant to apply to the pointing weights.
 * @param intervals Pointer to the array of intervals to modify (size n_view).
 * @param n_det Total number of detectors.
 * @param n_samp Total number of samples.
 * @param n_view Total number of intervals/views.
 *
 * @return None (the result is put in weights).
 */
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
                {
                    # pragma omp 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);
                    }
                }
            }
        }
    }
}
```

*B. JAX implementation of the `stokes_weights_IQU` operator.*

```
def stokes_weights_IQU_inner(eps, cal, pin, hwpang):
    """
    Compute the Stokes weights for one (detector,interval,sample).

    Args:
        eps (float):  The cross polar response.
        cal (float):  A constant to apply to the pointing weights.
        pin (array, float64):  The array of detector quaternions (size 4).
        hwpang (float64):  The HWP angle.

    Returns:
        weights (array, float64):  The detector weights for the specified mode (size 3)
    """
    # applies quaternion rotations
    zaxis = jnp.array([0.0, 0.0, 1.0])
    dir = quaternion_rotate(pin, zaxis)
    xaxis = jnp.array([1.0, 0.0, 0.0])
    orient = quaternion_rotate(pin, xaxis)

    # computes by and bx
    by = orient[0] * dir[1] - orient[1] * dir[0]
    bx = orient[0] * (-dir[2] * dir[0]) + \
         orient[1] * (-dir[2] * dir[1]) + \
         orient[2] * (dir[0] * dir[0] + dir[1] * dir[1])

    # computes detang
    detang = 2.0 * (jnp.arctan2(by, bx) + 2.0 * hwpang)

    # creates weights
    eta = (1.0 - eps) / (1.0 + eps)
    weights = jnp.array([cal, jnp.cos(detang) * eta * cal, jnp.sin(detang) * eta * cal])
    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',...])


def stokes_weights_IQU_interval(quat_index, quats, weight_index, weights, hwp, epsilon, cal,
                                interval_starts, interval_ends, intervals_max_length):
    """
    Process all the intervals as a block.
```

```
48
49      Args:
50          quat_index (array, int): size n_det
51          quats (array, double): size ???*n_samp*4
52          weight_index (array, int): The indexes of the weights (size n_det)
53          weights (array, float64): The flat packed detectors weights for the specified mode
    ↪   (size n_det*n_samp*3)
54          hwp (array, float64):  The HWP angles (size n_samp).
55          epsilon (array, float):  The cross polar response (size n_det).
56          cal (float):  A constant to apply to the pointing weights.
57          interval_starts (array, int): size n_view
58          interval_ends (array, int): size n_view
59          intervals_max_length (int): maximum length of an interval
60
61      Returns:
62          weights (array, float64)
63      """
64      # extract interval slices
65      intervals = JaxIntervals(interval_starts, interval_ends+1, intervals_max_length) # end+1
        ↪  as the interval is inclusive
66      quats_interval = JaxIntervals.get(quats, (quat_index,intervals,ALL)) #
        ↪  quats[quat_index,intervals,:]
67      hwp_interval = JaxIntervals.get(hwp, intervals) # hwp[intervals]
68
69      # does the computation
70      new_weights_interval = stokes_weights_IQU_inner(epsilon, cal, quats_interval,
        ↪  hwp_interval)
71
72      # updates results and returns
73      # weights[weight_index,intervals,:] = new_weights_interval
74      weights = JaxIntervals.set(weights, (weight_index, intervals, ALL), new_weights_interval)
75      return weights
76
77
78  # jit compiling
79  stokes_weights_IQU_interval = jax.jit(stokes_weights_IQU_interval,
80                                        static_argnames=["cal", "intervals_max_length"],
81                                    donate_argnums=[3]) # donating weights
82
83
84  def stokes_weights_IQU_jax(quat_index, quats, weight_index, weights,
85                             hwp, intervals, epsilon, cal):
86      """
87      Compute the Stokes weights for the "IQU" mode.
88
89      Args:
90          quat_index (array, int): size n_det
91          quats (array, double): size ???*n_samp*4
92          weight_index (array, int): The indexes of the weights (size n_det)
93          weights (array, float64): The flat packed detectors weights for the specified mode
    ↪   (size n_det*n_samp*3)
94          hwp (array, float64):  The HWP angles (size n_samp).
95          intervals (array, Interval): The intervals to modify (size n_view)
96          epsilon (array, float):  The cross polar response (size n_det).
97          cal (float):  A constant to apply to the pointing weights.
98
99      Returns:
100         None (the result is put in weights).
101     """
102     # prepares inputs
103     intervals_max_length = INTERVALS_JAX.compute_max_intervals_length(intervals)
104     quat_index_input = MutableJaxArray.to_array(quat_index)
105     quats_input = MutableJaxArray.to_array(quats)
106     weight_index_input = MutableJaxArray.to_array(weight_index)
107     weights_input = MutableJaxArray.to_array(weights)
108     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)
```