High-level GPU code: a case study examining JAX and OpenMP

Nestor Demeure
ndemeure@lbl.gov
National Energy Research Scientific Computing Center
Berkeley, California, U.S.A.

Rollin Thomas
rcthomas@lbl.gov
National Energy Research Scientific Computing Center
Berkeley, California, U.S.A.

Wahid Bhimji
wbhimji@lbl.gov
National Energy Research Scientific Computing Center
Berkeley, California, U.S.A.

Theodore Kisner
tskisner@lbl.gov
Computational Cosmology Center
Berkeley, California, U.S.A.

Reijo Keskitalo
rtkeskitalo@lbl.gov
Computational Cosmology Center
Berkeley, California, U.S.A.

Julian Borrill
jdborrill@lbl.gov
Computational Cosmology Center
Berkeley, California, U.S.A.

ABSTRACT

In recent years, a new scientific software design pattern has emerged, pairing a Python interface with high-performance kernels in lower-level languages. The rise of general-purpose GPUs necessitates the rewriting of many such kernels, which poses challenges in GPU programming and ensures future portability and flexibility. This paper documents our experience and observations during the process of porting TOAST, a cosmology software framework designed to take full advantage of a supercomputer, to work with GPUs. This exploration led us to compare two different porting strategies: utilizing the JAX Python library and employing OpenMP Target Offload compiler directives. JAX allows kernel code to be written in pure Python, whereas OpenMP Target Offload is a directive-based strategy that integrates seamlessly with our existing OpenMP-accelerated C++ kernels. Both frameworks are high-level, abstracting system architecture details while aiming for straightforward, portable, yet performant GPU code. Through the porting of a dozen kernels, we delve into the analysis of development cost, performance, and the viability of employing either of these frameworks for complex numerical Python applications.

CCS CONCEPTS

• **General and reference** → General conference proceedings; • **Computing methodologies** → **Parallel computing methodologies**; • **Software and its engineering**;

KEYWORDS

GPU Programming, Numerical application, JAX, OpenMP, Performance Analysis

ACM Reference Format:
Nestor Demeure, Rollin Thomas, Wahid Bhimji, Theodore Kisner, Reijo Keskitalo, and Julian Borrill. 2023. High-level GPU code: a case study examining JAX and OpenMP. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3624062.3624186

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624186

1 INTRODUCTION

Scientific software frequently employs a high-level Python interface that interacts with optimized kernels written in lower-level languages like C or C++. While this allows domain scientists to engage with applications in a familiar environment, it often requires dependence on a niche group of HPC experts for high-performance modifications. This reliance can lead to bottlenecks, particularly as an application's user base expands.

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.

Given the growing influence of GPUs in scientific computing and the challenges they introduce, it is instructive to study real-world examples of software frameworks grappling with these dynamics. We centered our research on TOAST [22] (Time Ordered Astrophysics Scalable Tools)[1], a framework that simulates and processes timestream data from microwave telescopes for Cosmic Microwave Background (CMB) analysis. TOAST provides simulation and analysis capabilities for experiments like CMB-S4 [1] (a strategic DOE Office of High Energy Physics project), Simons Observatory [2], 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 [27]. 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 internally with

[1] Available, with all of our work, at https://github.com/hpc4cmb/toast/tree/toast3 .

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, meaning that we can observe a wide range of commonly used numerical building blocks and patterns found in Python scientific applications with a single benchmark. Furthermore, TOAST serves as a flagship application at the National Energy Research Scientific Computing Center (NERSC), being part of the NERSC10 benchmarking suite utilized to evaluate proposals for the next generation of supercomputers at NERSC. Therefore, its ability to run efficiently on diverse hardware has a far reaching impact.

We evaluated two high-level frameworks for porting the TOAST application to GPU architectures. First, we considered OpenMP Target Offload [3] – an extension of OpenMP, which TOAST already employs for parallelizing its C++ kernels. Introduced in OpenMP 4.0 [30], and supported by major C++ compilers like Clang [26] and GCC [35], this construct facilitates portable GPU parallelism across multiple architectures. It offers a high-level integration approach that dovetails with our existing C++ kernels, obviating the need for additional compiler tools. We also investigated JAX [6], 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 [34], computational fluid dynamics [5, 24] and ocean simulation [19]. Offering an array-oriented library reminiscent of NumPy [39], its just-in-time compiler optimizes code for a diverse range of hardware, from CPUs and GPUs (NVIDIA and AMD) to more specialized units like TPUs. 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, promising efficiency alongside flexibility. Our overarching goal was to attain commendable GPU performance without compromising on portability, readability, and maintainability by domain specialists as the software iterates.

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 [42]. Equipped with Slingshot interconnect, it is a heterogeneous system comprised of 3,072 CPU-only and 1,792 GPU-accelerated nodes. Each of those GPU node is equipped with one, 64 cores, AMD Milan CPU and 4 NVIDIA A100 GPUs. Together, they can deliver about 70 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 options available to port a Python code base to GPU and in particular OpenMP Target Offload and JAX. In Section Two, we cover the porting of the codebase, discussing the advantages and drawbacks of each framework as well as the efforts made to overcome their limitations. Finally, in Section Three, we examine the performance of the resulting implementations before concluding on the pros and cons of each approach.

2 GPU COMPUTING

Previous studies on hybrid language codes have demonstrated the feasibility and benefits of porting Python applications to GPUs. For example, the DESI project [11] 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 [41], a high-performance CFD framework that uses code-generation for solving problems on unstructured grids, targeting clusters of CPUs and NVIDIA GPUs.

2.1 GPU computing in Python

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

Using off-the-shelf kernels: Leveraging readily available kernels, such as those from CuPy [28] and RAPIDS [17], offers a simple solution, especially if the requisite functionalities are already in place. CuPy resembles NumPy [39], supporting both NVIDIA and AMD GPUs, while RAPIDS enhances data science GPU workflows. Despite their convenience, these libraries can be limiting, especially when custom kernels are required, potentially leading to performance bottlenecks due to increased data allocations and transfers.

Writing a kernel in a low-level language: Direct kernel creation using tools like CUDA [29], OpenCL [36], Kokkos [38], RAJA [4], OpenACC [15], OpenMP Target Offload [3], or even C++ Standard Parallelism gives us good control on the performance. These kernels can then be linked to Python using utilities like PyOpenCL [23]. While this method allows for peak optimization, it necessitates expertise, often complicating the integration process, and might lack portability. We found OpenMP Target Offload notable as it ensures portability while maintaining an abstraction layer, and lets us build on the preexisting OpenMP expertise within the team and the wider numerical computing community.

Writing a kernel in Python: Python-centric solutions, such as Numba [25], Taichi [18], and Triton [37], offer an all-Python ecosystem, reducing the development curve. However, while ensuring easier code management, those options have in common the fact that they drop down to a very low-level syntax and provide limited (or non-existent) support for libraries and common numerical operations.

Using a Deep-Learning library: Deep-learning libraries like PyTorch [31] and TensorFlow [13] are persuasive and well-documented. They provide comprehensive numerical operations but are often disappointing when used for straight numerical computations as they optimize for training speed and come with a large overhead, mostly due to gradient computation and intermediate values. JAX [6] stands out, as it builds on code transformation and JIT compilation – sidestepping sources of overheads – making it a viable choice for GPU kernel development.

After careful evaluation, we narrowed our focus to JAX and OpenMP Target Offload. OpenMP seamlessly integrates with our pre-existing C++ kernels, requiring no additional toolchain. Conversely, JAX offers the advantage of a unified Python codebase, targeting both CPU and GPU.

2.2 OpenMP Target Offload

OpenMP, since its 4.0 version, has integrated a mechanism to offload data or computations to “target" computing devices, which includes accelerators and GPUs [8]. Given TOAST’s existing reliance on OpenMP for CPU parallelism, transitioning towards OpenMP Target Offload for GPU compatibility was a logical progression.

2.2.1 Design: OpenMP Target Offload functions by marking compute-intensive code kernels with directives, typically represented as `#pragma omp target` in C++. These labeled instructions are subsequently mapped and executed on the target device during runtime. OpenMP directives come with “clauses" which grant developers nuanced control over facets like data privacy, execution, and synchronization. One of the salient features of OpenMP Target Offload, as highlighted by various case studies [9], is its ability to obscure hardware specifics. This abstraction means that the same codebase can be executed on multiple target devices without necessitating alterations. Moreover, developers can bolster performance by integrating vendor-optimized GPU libraries, such as those for BLAS and FFT, and directly passing device memory pointers to them.

2.2.2 Limitations: Despite the allure of facile porting that OpenMP Target Offload promises, this seamless transition is primarily confined to code that is both simply structured and compute-intensive. Challenges arise when handling device memory allocations and transferring extensive buffers, as these operations can be time-consuming. When porting modular (or unfused) lightweight kernels, meticulous management of data movement is essential, particularly beyond the boundaries of individual kernel blocks. Some compilers, such as LLVM, attempt to employ internal asynchronous data movement and kernel submission. However, to achieve a satisfactory overlap between kernel submission and execution, manual specification of data dependencies is often indispensable. Advanced OpenMP strategies can extend the capabilities of existing compiler support, yet concrete instances employing these intricate patterns remain rare.

2.3 The JAX Library

JAX [6] is a Python library that provides a high level interface designed to be as similar as possible to NumPy [39] and SciPy [40] (similarly to CuPy [28]) coupled with a JIT compiler to fuse kernels and elide intermediate results. It is purposefully high-level, restricting the freedom of the developer (see section 2.3.2) but, in exchange, separating the semantics of the code (left to the developer) from its optimization (left to the compiler).

2.3.1 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[2] 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[3] to be compiled by the XLA[4] [32] compiler which will then run it on the target architecture (as seen on figure 1) which could be CPU, GPU (both NVIDIA and an experimental support for AMD) or some deep-learning focused alternative architectures (TPU and specialized hardware). Subsequent runs will reuse the compiled function.

Figure 1: JAX workflow, from tracing to hardware execution.

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 [20]) all of which are used within TOAST.

2.3.2 Limitations: JAX enforces certain constraints due to its commitment to pure and statically composed functions, which is particularly noticeable when porting pre-existing codes to align with its paradigm:

- *Purity:* JAX prohibits in-place array modifications, emphasizing pure operations. For example, updating an array element in-place is not allowed, but JAX offers alternatives like `x.at[idx].set(y)`.
- *Loops and Conditionals:* JAX’s tracing mechanism regards values as unknown during the tracing, disallowing conditional dependence on these values. This makes some patterns, such as loops depending on traced values, harder to represent. While JAX introduces primitives to work around this limitation, it hasn’t posed a significant concern in our application.
- *Static Array Shapes:* All array shapes must be known during tracing, meaning that dynamically sized arrays whose shape is a function of the data are not feasible[5]. This was particularly challenging for us, as many of our kernels iterate over varying interval sizes. We had to implemented workarounds, padding to a static interval size, to fit JAX’s model.

[2]Technically, a lot of GPU code is JIT-compiled when lowering the assembly to the hardware. For example, the GPU driver will often have to JIT compile PTX instructions into SASS assembly on NVIDIA hardware. However, we specifically employ this term for JAX as it does *all* of its optimizations (such as loop unrolling) just-in-time while traditional approaches, such as compiling CUDA code, perform most of those at a compile time.

[3]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.

[4]The XLA compiler is open source and is now part of the OpenXLA Project, meaning that development moved from Google to a collaboration which includes several software and hardware providers (including Amazon, AMD, Apple, Arm, Cerebras, Google, Graphcore, Hugging Face, Intel, Meta, and NVIDIA) which is promising for the sustainability, and in particular portability to newer hardware, of JAX.

[5]This might be temporary, the XLA team has been working on relaxing those limitations.

Furthermore, as seen in section 4.2, JAX's compiler is principally optimized for GPU and TPU execution. While it does support CPU execution, its performance on CPUs is roughly comparable to single-core C++, which may render it suboptimal for applications solely reliant on CPU processing.

3 PORTING THE TOAST CODEBASE

3.1 Port of the Kernels

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. Maintaining modularity was a goal when moving these kernels to GPU.

3.1.1 Target Kernels. Trying to cover common patterns in our code base as well as ensure that several common pipelines would be able to run fully on GPU, we ported the following 10 kernels – 8 of which will be used in our benchmark[6] – carefully preserving the API of the original code:

- `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 [16] 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).

Most kernels consists of a triple loop over detectors, intervals and samples within intervals (the intervals having varying lengths). The loop bodies are often independent, extracting a piece of data using some indexing information, computing a result, then storing it in an output variable.

3.1.2 OpenMP Target Offload. The OpenMP Target Offload parallelization was fairly straightforward: the triple loop is collapsed and parallelized using a `#pragma omp target teams distribute parallel for collapse(3)`. The only real subtlety is in the way we dealt with the intervals, whose variable size made unsuitable for loop collapsing and straightforward parallelization. We precompute the maximum interval size then iterate on that interval size, using a test to cut work when an iteration falls out of the current interval size. Conditionals are famously bad for GPU performance, especially within loop bodies, but we believe that here the impact is minimal since one branch of the conditional is a no-op.

All data movement was handled manually using a C++ singleton class managing device memory buffers allocated with `omp_target_alloc()`, which uses a manually implemented memory pool, and their host memory counterparts. This let higher-level TOAST code (the `Pipeline` class) use operator dependency information to move data to/from the device efficiently after a sequence of lightweight kernels (as detailled in section 3.2).

The TOAST package already contained a compiled Python extension using OpenMP threading, 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, and GCC), we settled on NVIDIA NVC as it has good support on Perlmutter and covers all of the functionalities we needed for our kernels.

It is worth noting that the code *needs* to be run with NVIDIA Multi-Process Service (MPS) [7] for optimal performance on Perlmutter. MPS lets you run several kernels on a single GPU concurently, letting us oversubscribe the GPU. Previous attempt without MPS saw the CUDA driver context-switches between processes, effectively capping our performance to one process per device[7].

3.1.3 JAX. The port from C++ to Python and JAX 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 over the loop axes, a common pattern in JAX code) and removing side effects. The resulting functions were 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.

The intervals (being of various sizes, something that clashes directly with JAX static sizes requirement) are padded to the largest interval size (known at tracing time, making it an acceptable static size). Note that, while our OpenMP Target Offload implementation does nothing on out-of-interval values, the JAX implementation does some dummy work[8].

Our only two modifications to JAX default settings were the use of 64-bit floating-point arithmetic and the deactivation of device memory preallocation (by default JAX uses a memory pool). Deactivating memory preallocation is a recommended practice when oversubscribing a device with JAX, as it is simpler than carefully tweaking the memory allocation size per process and device and, as we observed (likely because we batch most of our allocations at the beginning and end of the pipelines), is unlikely to degrade performance.

Interestingly, MPS was not needed to use several processes per device efficiently with JAX (contrary to OpenMP Target Offload, as seen in section 3.1.2), likely because JAX uses the NVIDIA Collective Communication Library (NCCL) [21] to deal with multi-device connections.

[6] `stokes_weights_I` and `template_offset_apply_diag_precond` are not exercised by our benchmark but they are used for some key CMB experiments and were thus included.

[7] While we have not tested it on other systems, it appears to be a limitation of the CUDA driver that might not impact non-NVIDIA architectures.

[8] We could use the same pattern for *both* implementations. The padding idea was first tested on our JAX implementation then, when it proved beneficial, retrofitted with an improved design to the OpenMP Target Offload kernels. Later tests showed no significant performance difference between both patterns.

3.2 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.

3.2.1 Abstraction Layers. We designed a runtime dispatch system over kernels, enabling the selection of specific implementations for the entire code, individual pipelines, or 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 pipelines in a framework-agnostic manner.

3.2.2 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 pipelines. 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, any data left 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).

3.2.3 Profiling. Bundled with the TOAST code, was 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.

3.3 Discussion

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*[9] as illustrated in figure 3.

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 the 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).

[9]Measured with `cloc v1.82` [10], not counting empty lines and comments.

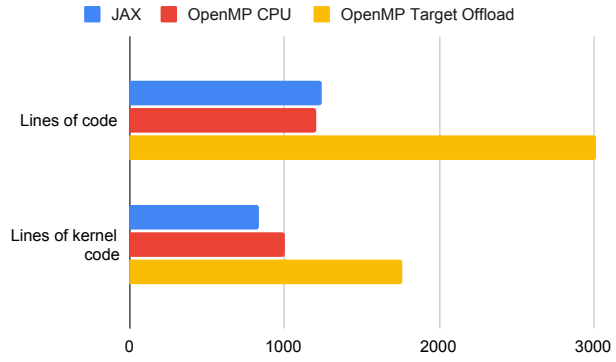

Figure 2: Number of lines of code per implementation. *Lines of kernel code* is the number of lines used strictly in the kernel implementations while *Lines of code* includes the dependencies of the kernels and the accelerator-related code (data movement, GPU related types, etc.).

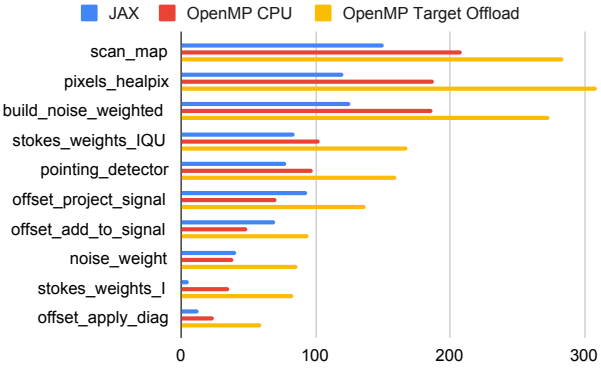

Figure 3: Number of lines of code per kernel.

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 using 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 arrays manually. Furthermore, looking at the code that is in the kernels (seen in the difference between the two bars of figure 2) 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 fewer dependencies (being able to reuse Python libraries) and are implementing the logic in a higher level language.

Overall, writing the JAX code "felt" very productive: 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. Its use of interfaces designed to be as close as possible to NumPy made it extremely easy to find help implementing functionality 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.

Finally, despite OpenMP being a more established technology, the OpenMP Target Offload port suffered from the fact that compiler support for this part of the standard is still in its infancy. Although GCC is the most prevalent choice on Linux for compiling Python extensions, it does not support the full set of needed OpenMP target features. The LLVM and NVHPC compilers do support target offload features well but, NVHPC has only recently begun supporting the compilation of dynamically loaded Python extensions embedding OpenMP Target Offload code. We found existing comparative studies [12, 14] to be good reviews of feature availability and runtime overhead for the various compilers implementing the standard, but they fall short on addressing user experience and compiler maturity: throughout the compilation process, we encountered critical compiler failures, as well as minimalist, often seemingly unrelated, error messages. Logic errors while running the code would, at best, result in segmentation faults, sometimes in unrelated parts of the code, and using a python package (e.g. Numpy) linking to another OpenMP implementation can produce undefined runtime behavior such as silent memory corruption. This made the debugging experience a far cry from the one we had dealing with JAX error messages.

4 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 cores.

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. We ran on two different problem sizes:

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

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.

Note that, as this is a realistic benchmark, it includes serial computation as well as kernels written in pure Python. Furthermore, more than 30 kernels have yet to be ported to GPU. All of this means that our overall speed-up is strictly bounded by Amdahl's law to about 3x times faster. However, the kernels that *have* been ported to GPU are already a strong baseline, optimized by performance engineers for previous CPU architecture.

4.1 Number of process

Figure 4 gives us the evolution of the runtime when increasing the number of processes for the medium problem size (1 node, 64 cores, 4 GPUs). Note that the computing power available is kept fixed, 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.

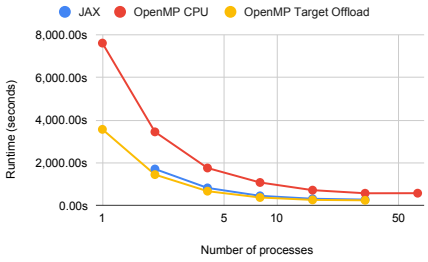

Figure 4: Runtime as a function of the number of processes, running the medium size problem on 1 node. The number of processes are on a logarithmic scale.

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 processes 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 reaches its peak speedup compared to CPU (2.4x) at 8 processes (two process per GPU), demonstrating a benefit to oversubscribing the GPUs. The speedup then slowly decreases to 2.3x (16 processes) then 2x (32 processes) as we progressively lose the oversubscription benefit.

OpenMP Target Offload follows the same trajectory as JAX but is *consistently* 20% faster. Going from a peak speedup of 2.9x at 8 processes then 2.7x (16 processes) and 2.3x (32) processes. This is likely a sign that OpenMP Target Offload has a lower overhead than JAX[10]. Note that, while the problem does not fit in memory when using 64 processes, it *does* fit on 1 process, hinting at a lower memory usage compared to JAX. JAX's memory management is based on a memory pool which, while it leads to code simplifications and significant performance benefits out of the box (we ended up implementing a memory pool manually for use with OpenMP Target Offload), does reduce developer control on memory use, something that is still a very limited resource on GPU.

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

[10]JAX overhead includes JIT compiling and checking if a given function has already been compiled for a given problem size; both of which would be expected to have a constant impact on the runtime while we observe a proportional impact, pointing towards performance differences at the runtime level

4.2 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, and 4 threads per process). We see a similar behavior as when running a smaller problem on one node and 16 processes: compared to the CPU reference, JAX is 2.28x times faster and OpenMP Target Offload is 2.58x times faster.

We also ran JAX, using the GPU infrastructure but forcing it to use its *CPU backend*. Interestingly it was 7.4x times *slower* than our parallelized CPU baseline[11]. Part of it can be explained by overhead, some of which coming from the GPU infrastructure and the data copies it performs which are not required on CPU, but it is mostly down to the compiler's current CPU backend which limits parallelism to some individual operations, such as matrix multiplication[12]. Both of those factors mean that JAX's CPU backend, while faster than using Numpy, is slower that single core C++ and significantly slower than parallel C++ which makes this backend currently unsuitable for high performance use-cases.

Examining per-kernel performance (figure 6), we see that JAX goes from a slight speed up (as low as 1.5x times faster for `offset_add_to_signal`, a kernel doing very little computation) to 18x and 45x speedups on some of the kernels that the 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 11x speedup), this was however expected as this kernel has many branches, with dozens of variables declared per branch, something which is known to be expensive on GPU.

OpenMP Target Offload follows a similar pattern, going from 5x (`offset_add_to_signal`) to 61x (`stokes_weights_IQU`) times faster than the CPU baseline, and being on average 2.4x faster than JAX. Two noticeable kernels are `pixels_healpix`, where it does much better than JAX (a 41x speedup), maybe due to a better handling of the conditionals that characterize this kernel, and `offset_project_signal` where it does significantly worse (a 19x speedup) which might be due to the XLA compiler finding a way to express this particular kernel in terms of linear algebra while the OpenMP Target Offload implementation uses a straight loop.

Finally, while data movement to and from the device is expensive, our pipelining infrastructure seems to be doing a good job of minimizing this cost as most of the data operations barely register on the plot. It is, however, noticeable that JAX spends significantly less time updating device data (`accel_data_update_device`) and resetting device buffers (`accel_data_reset`) making it more efficient at data movement overall. These differences boil down to the way we implemented data transfers for each framework and could be erased.

[11]This result is absent from the figure as the runtime would dwarf other numbers.
[12]XLA's CPU backend appears to have received significantly less attention than the GPU and TPU backends (for example, while buffer donation is part of JAX's JIT API, it is not supported by the CPU backend). However, there seem to be work in progress, such as [33], to improve the situation.

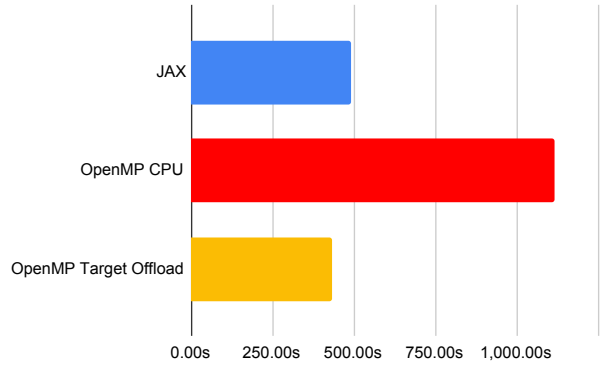

Figure 5: Runtime as a function of the kernel implementation, running the large problem size on 8 node, with 16 processes per node, and 4 treads per process.

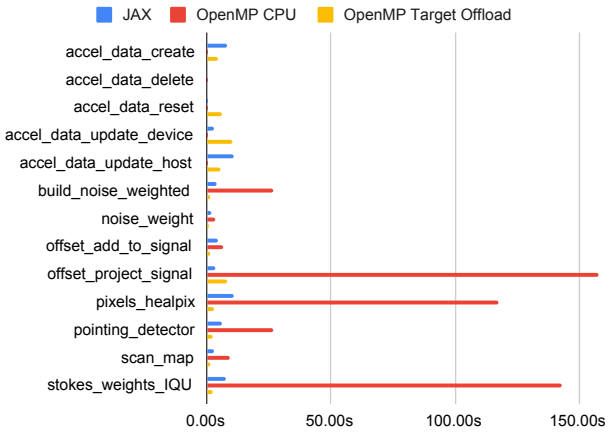

Figure 6: Total runtime for each kernel, running the medium problem size on 1 node, with 16 processes, and 4 treads per process. `accel_data` functions represent data movement operations happening out of the kernels.

5 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.

OpenMP Target Offload showcased its potential by offering a 2.58x times speedup on a realistic benchmark. Individual kernels, particularly the most compute-intensive ones, witnessed up to a 61x acceleration. The derived kernel code resembles the pre-existing logic, resulting in a 1.8 times longer codebase that remains quite intuitive for those familiar with OpenMP. However, it is worth noting that the current state of compiler support and documentation for OpenMP Target Offload appears inadequate, making its full potential challenging to achieve despite its deep-rooted presence in shared memory parallelism. We believe it is best used on *codebases with large preexisting OpenMP kernels*, as it allows for progressive updates to existing code and can serve as a gateway to GPU computing.

By contrast, JAX, a burgeoning tool, offered a commendable 2.28x times speedup, propelling individual kernels to perform up to 45x times faster. The Python and NumPy API facilitate a more succinct codebase, about 1.2 times shorter than the reference. The key advantage of JAX lies in allowing developers to focus on semantics within JIT compiled sections, while delegating performance aspects to the interface and data movements. While JAX may not be the most efficient method for executing code on GPU – it is within 20% of OpenMP Target Offload's efficiency – it has proven to be productive to write and exceptionally accessible to domain experts with minimal high-performance computing experience. This accessibility enables a broader utilization of currently often underused GPU resources, allowing for both the creation and maintenance of projects by a wider range of developers. JAX seems particularly well suited for *new Python projects* which can be architected around it: focusing on immutability and static sizes, opening the door to code simplifications and JIT compiling as many things as possible.

Both solutions required code duplication, keeping the OpenMP CPU version of the code, for optimal performance when running the code on CPU. JAX due to its currently insufficient CPU performance and OpenMP Target Offload due to the need to deal with device pointers manually and to use different patterns for CPU and GPU parallelism. However, the GPU port has already proven its value to CMB experiments as it it now used to cut down on the time needed to run some simulations[13] that would previously have taken days to complete.

Our exploration into abstraction layers emerged as a response to the current maturity level of GPU computing solutions, and was a necessity to ensure the long-term viability of TOAST amid the evolving landscape of GPU hardware. It facilitated a more structured approach towards GPU porting and data movement. However, we posit that it may be best omitted in most projects, as it introduces a non-negligible layer of complexity to the code base. This added complexity might not be warranted for projects with less stringent development constraints. Moreover, the usefulness of such abstraction layers is likely to diminish as GPU technologies mature.

In the short term, we want to port more kernels, to run as much computation as possible on GPU, now that we have proven that it can be done in a way that meshes with the TOAST code base. Exploring different C++ compilers for building OpenMP Target Offload code could also be a fruitful object of study. We have anecdotal proofs that the code runs on other hardware targets but, in the longer term, it would be interesting to do a systematic study quantifying the performance on various targets. It would also be interesting to apply similar patterns to other code bases to see how reproducible our findings are.

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.

[13]While keeping the costs of running the code on Perlmutter roughly constant.

ACKNOWLEDGMENTS

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] Kevork Abazajian, Graeme Addison, Peter Adshead, Zeeshan Ahmed, Steven W Allen, David Alonso, Marcelo Alvarez, Adam Anderson, Kam S Arnold, Carlo Baccigalupi, et al. 2019. CMB-S4 science case, reference design, and project plan. *arXiv preprint arXiv:1907.04473* (2019).

[2] Peter Ade, James Aguirre, Zeeshan Ahmed, Simone Aiola, Aamir Ali, David Alonso, Marcelo A Alvarez, Kam Arnold, Peter Ashton, Jason Austermann, et al. 2019. The Simons Observatory: science goals and forecasts. *Journal of Cosmology and Astroparticle Physics* 2019, 02 (2019), 056.

[3] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. 2016. Offloading Support for OpenMP in Clang and LLVM. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 1–11. https://doi.org/10.1109/LLVM-HPC.2016.006

[4] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 71–81.

[5] Deniz A Bezgin, Aaron B Buhendwa, and Nikolaus A Adams. 2022. JAX-FLUIDS: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *arXiv preprint arXiv:2203.13760* (2022).

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[7] NVIDIA Corporation. 2023. *Multi-Process Service: GPU Deployment and Management.* https://docs.nvidia.com/deploy/mps/index.html

[8] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[9] Christopher Daley, Hadia Ahmed, Samuel Williams, and Nicholas Wright. 2020. A Case Study of Porting HPGMG from CUDA to OpenMP Target Offload. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 37–51.

[10] Albert Danial. 2021. *cloc: v1.92.* https://doi.org/10.5281/zenodo.5760077

[11] Daniel Margala, Laurie Stephey, Rollin Thomas, and Stephen Bailey. 2021. 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.). 33 – 39. https://doi.org/10.25080/majora-1b6fd038-004

[12] Joshua Hoke Davis, Christopher Daley, Swaroop Pophale, Thomas Huber, Sunita Chandrasekaran, and Nicholas J Wright. 2021. Performance assessment of OpenMP compilers targeting NVIDIA V100 GPUs. In *Accelerator Programming Using Directives: 7th International Workshop, WACCPD 2020, Virtual Event, November 20, 2020, Proceedings 7*. Springer, 25–44.

[13] TensorFlow Developers. 2022. TensorFlow. *Zenodo* (2022).

[14] Jose Monsalve Diaz, Kyle Friedline, Swaroop Pophale, Oscar Hernandez, David E Bernholdt, and Sunita Chandrasekaran. 2019. Analysis of OpenMP 4.5 offloading in implementations: correctness and overhead. *Parallel Comput.* 89 (2019), 102546.

[15] Rob Farber. 2016. *Parallel programming with OpenACC.* Newnes.

[16] Krzysztof M Gorski, Eric Hivon, Anthony J Banday, Benjamin D Wandelt, Frode K Hansen, Mstvos Reinecke, and Matthia Bartelmann. 2005. HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal* 622, 2 (2005), 759.

[17] Todd Hricik, David Bader, and Oded Green. 2020. Using RAPIDS AI to accelerate graph data science workflows. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–4.

[18] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.

[19] Dion Häfner, René Løwe Jacobsen, Carsten Eden, Mads R. B. Kristensen, Markus Jochum, Roman Nuterman, and Brian Vinter. 2018. Veros v0.1 – a fast and versatile ocean simulator in pure Python. *Geoscientific Model Development* 11, 8 (Aug. 2018), 3299–3312. https://doi.org/10.5194/gmd-11-3299-2018

[20] Dion Häfner and Filippo Vicentini. 2021. mpi4jax: Zero-copy MPI communication of JAX arrays. *Journal of Open Source Software* 6, 65 (2021), 3419. https://doi.org/10.21105/joss.03419

[21] Sylvain Jeaugey. 2017. Nccl 2.0. In *GPU Technology Conference (GTC)*, Vol. 2.

[22] Theodore Kisner, Reijo Keskitalo, Andrea Zonca, Jonathan R. Madsen, Jean Savarit, Maurizio Tomasi, Kolen Cheung, Giuseppe Puglisi, David Liu, and Matthew Hasselfield. 2021. *hpc4cmb/toast: Update Pybind11.* https://doi.org/10.5281/zenodo.5559597

[23] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Comput.* 38, 3 (2012), 157–174. https://doi.org/10.1016/j.parco.2011.09.001

[24] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. 2021. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences* 118, 21 (2021). https://doi.org/10.1073/pnas.2101784118 arXiv:https://www.pnas.org/content/118/21/e2101784118.full.pdf

[25] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.

[26] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

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

[28] ROYUD Nishino and Shohei Hido Crissman Loomis. 2017. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st conference on neural information processing systems* 151, 7 (2017).

[29] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[30] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface Version 4.0. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[32] Amit Sabne. 2020. XLA: Compiling Machine Learning for Peak Performance.

[33] Tao B Schardl and Siddharth Samsi. 2019. Tapirxla: Embedding fork-join parallelism into the xla compiler in tensorflow using tapir. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[34] Samuel S. Schoenholz and Ekin D. Cubuk. 2020. JAX M.D. A Framework for Differentiable Physics. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc. https://papers.nips.cc/paper/2020/file/83d3d4b6c9579515e1679aca8cbc8033-Paper.pdf

[35] Richard M Stallman et al. 1999. *Using and porting the GNU compiler collection*. Vol. 86. Free Software Foundation.

[36] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66.

[37] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. 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*. 10–19.

[38] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[39] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering* 13, 2 (2011), 22–30.

[40] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.

[41] Freddie D Witherden, Antony M Farrington, and Peter E Vincent. 2014. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications* 185, 11 (2014), 3028–3040.

[42] Charlene Yang and Jack Deslippe. 2020. Accelerate Science on Perlmutter with NERSC. *Bulletin of the American Physical Society* 65 (2020).