

Julia for Geophysical Fluid Dynamics: Performance Comparisons between CPU, GPU, and Fortran-MPI

Siddhartha Bishnu^{1,2}, Robert R. Strauss^{1,3}, and Mark R. Petersen¹

¹Computational Physics and Methods Group, Los Alamos National Laboratory, NM, 87545, USA

²Department of Earth, Atmospheric and Planetary Sciences, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

³Center for Nonlinear Studies, Los Alamos National Laboratory, NM, 87545, USA

Correspondence: Mark R. Petersen (mpetersen@lanl.gov)

Abstract. Some programming languages are easy to develop at the cost of slow execution, while others are ~~fast-at-run-time-but~~ [fast at runtime but](#) much more difficult to write. Julia is a programming language that aims to be the best of both worlds—a development and production language at the same time. To test Julia’s utility in scientific high-performance computing (HPC), we built an unstructured-mesh shallow water model in Julia and compared it against an established Fortran-MPI ocean model, MPAS-Ocean, as well as a Python shallow water code. Three versions of the Julia shallow water code were created, for: single-core CPU; graphics processing unit (GPU); and Message Passing Interface (MPI) CPU clusters. Comparing identical simulations revealed that our first version of the Julia model was 13 times faster than Python [using NumPy, where both used an](#) [unthreaded single-core CPU](#). Further Julia optimizations, including static typing and removing implicit memory allocations, provided an additional 10–20x speed-up of the single-core CPU Julia model. The GPU-accelerated Julia code ~~a speed-up of 230-380x compared to the single-core CPU Julia code~~ [was almost identical in terms of performance to the MPI parallelized code on 64 processes, an unexpected result for such different architectures](#). Parallelized Julia-MPI performance was identical to Fortran-MPI MPAS-Ocean for low processor counts, and ranges from 2x faster to 2x slower for higher processor counts. Our experience is that Julia development is fast and convenient for prototyping, but that Julia requires further investment and expertise to be competitive with compiled codes. We provide advice on Julia code optimization for HPC systems.

1 Introduction

A major concern in computer modeling is the trade-off between execution speed and code development time. In general, programs in scripting languages like Python and Matlab are faster to develop due to their simpler syntax and more relaxed typing requirements, but are limited by slower execution time. On the other end of the spectrum, we have compiled languages like C/C++ and Fortran, which have been extensively used in scientific computing for many decades. Programs in such languages are blessed with faster execution time, but are cursed with stricter and more cumbersome syntax, leading to slower development time. The Julia language strikes a balance between these two categories (Perkel, 2019). It is a compiled language

with execution speed similar to C/C++ or Fortran, if carefully written with strict syntax (Lin and McIntosh-Smith, 2021; 25 Gevorkyan et al., 2019). It is also equipped with a more convenient syntax and features, such as dynamic typing, to accelerate code development in prototyping. To this day, the majority of scientific computing models are programmed in compiled languages, which execute fast but can take can take years to develop—for example, the first version of MPAS-Ocean required three years (Ringler et al., 2013). In this paper, we investigate the feasibility of writing Julia codes for computational physics simulations, since a Julia program ~~can not~~ cannot only ensure high performance but also less development time in the initial 30 stages. We develop a shallow water solver in Julia and compare its performance to an equivalent Fortran code.

An additional complication in choosing the best language is that layers of libraries have been added to C/C++ and Fortran to accommodate evolving computer architectures. For the past 25 years, computational physics codes have largely used the Message Passing Interface (MPI) to communicate between CPUs on separate nodes that do not share memory, and OpenMP to parallelize within a node using shared-memory threads. With the advent of heterogeneous nodes containing both CPUs and 35 GPUs, scientific programmers have several new choices: writing kernels directly for GPUs in CUDA (Bleichrodt et al., 2012; Zhao et al., 2017; Xu et al., 2015); adding OpenACC pragmas for the GPUs (Jiang et al., 2019); or calling libraries such as Kokkos (Trott et al., 2022) and YAKL Norman et al. (2022) that execute code optimized for specialized architectures on the back-end, while providing a simpler front-end interface for the domain scientist. All of these require additional expertise, and add to the length and complexity of the code base. Julia also provides an MPI library for parallelization across nodes in a 40 cluster, and a CUDA library to parallelize over GPUs within a node. We have written shallow water codes in Julia that adopt each of these parallelization strategies.

In recent years, shallow water solvers ~~such as~~ have been developed in Julia by Oceananigans.jl (Ramadhan et al., 2020) and ShallowWaters.jl (Klöwer et al., 2022) ~~have been developed in Julia~~. These codes employ structured rectilinear meshes to discretize their ~~domain, and are~~ spatial domain. Oceananigans.jl is equipped with capabilities for running on GPUs to achieve 45 high performance, and can also solve the nonhydrostatic and hydrostatic Boussinesq equations. Here we conduct a comparison on unstructured-mesh models, using the Fortran code MPAS-Ocean (Ringler et al., 2013) as a point of reference. MPAS-Ocean employs unstructured near-hexagonal meshes with variable resolution capability and is parallelized with MPI for running on supercomputer clusters. We developed a Julia model employing the same spatial discretization of MPAS-Ocean, ~~and which~~ is capable of running in serial mode on a single core, or in parallel mode on a supercomputer cluster or a graphics card. We 50 discuss the subtle details of our implementations, compare the speed-ups attained, and describe the strategies employed to enhance performance.

The structure of this paper is arranged as follows. Section 2 presents a comprehensive introduction to the Julia programming language, the primary subject of our experiments in this paper. This section elucidates how Julia’s innovative compiler, employing its Just-in-Time (JIT) compilation approach and dynamic type inference, equips the language with the capability to rival the performance of statically-typed languages such as Fortran and C++/C. We highlight some key features of Julia, which are not only fundamental to our research, but also provide valuable insights for researchers new to Julia, aiding their understanding of Julia’s unique concepts and terminologies. In Section 3, we delineate the process of creating three versions of the Julia model: Julia-CPU, Julia-GPU and Julia-MPI. We also provide details on an equivalent Python-CPU code and

55

the Fortran-based MPAS-Ocean, both of which serve as comparative yardsticks for assessing Julia's performance in the ensuing section. Additionally, we offer an explicit account of the hardware configurations and toolchain specifications used in the process. Section 4 furnishes the findings from our performance comparison tests, including an explanation of how we fine-tuned our preliminary model to generate the reported results. This is accompanied by an in-depth discussion and analysis of our experimental results. This section serves as a benchmark for contrasting the proficiency of Julia and Fortran in High Performance Computing (HPC) applications. In Section 5, we share insights and provide guidance to HPC developers on how to effectively utilize Julia, with an emphasis on the necessary steps to attain performance on par with that of Fortran and C++/C. This is informed by our experiences and the lessons learned throughout this experiment. Finally, Section 6 concludes the paper, encapsulating our findings and providing instructions on how to replicate our results.

2 Julia in a Nutshell

Julia is a high-level, just-in-time (JIT) compiled dynamic programming language, which was developed with the intention of marrying the speed of compiled languages like Fortran or C++/C and the usability of interpreted languages like Python or MATLAB. Conceived in 2012 by Dr. Viral Shah, Prof. Alan Edelman, Dr. Jeff Bezanson, and Stefan Karpinski at MIT (Bezanson et al., 2017), Julia has rapidly grown in popularity thanks to its innovative features and design.

As a relatively new addition to the world of programming languages, Julia benefits from the ability to incorporate the best aspects of established languages while avoiding their less convenient attributes. It provides the speed of a compiled language (owing to Just-in-Time compiler and superior memory management) and the simplicity of an interpreted language, making it highly appealing for developers. The language also includes a REPL (Read-Eval-Print loop) environment that interprets code lines as they are written, enhancing programmers' convenience.

Specifically designed for technical and scientific users, Julia is versatile and boasts an extensive library of mathematical functions and numerical accuracy. It also facilitates seamless interoperability with other programming languages, enabling direct calls to Fortran, C++/C, Java, or Python.

Compatible with a range of operating systems, including Windows, MacOS, and Unix, Julia has found widespread application in domains requiring algebraic and numerical computing, data science and machine learning, artificial intelligence, distributed and parallel computing, and even web application development, due to its math-friendly syntax and impressive speed.

In the upcoming subsections, we succinctly delve into three crucial facets of Julia: Just-in-Time (JIT) compilation, multiple dispatch, and type hierarchy. These salient characteristics are not only pertinent to our study but will also be instrumental in elucidating our model and interpreting the results. We wrap up this section with a discussion on scenarios where type inference may fail, potentially leading to slower code execution.

2.1 Just-in-Time Compilation

90 Julia's high performance can be attributed to one of its key features: low level virtual machine (LLVM) based on-the-fly or just-in-time (JIT) compilation, which is a combination of Ahead Of Time (AOT) compilation and interpretation. Here is a breakdown of how it works:

- (a) When a function is first run in Julia, the interpreter converts the high-level code into an intermediate representation;
- (b) The compiler then uses this intermediate representation to generate optimized machine code tailored to the specific types in use;
- 95 (c) This machine code is executed, and the results are returned;
- (d) Most importantly, the machine code for the specific function and type combination is cached. So, if the same function is called with the same types later on, Julia can bypass the compilation step and directly execute the pre-optimized machine code.

100 This JIT compilation enables Julia to match the performance of statically-typed compiled languages such as Fortran, C++/C, while preserving the flexibility of dynamic languages like Python. However, it also introduces a delay referred to as 'time-to-first-plot' or compilation latency on the initial run of a function. Subsequent calls are significantly faster due to the cached machine code.

2.2 Function, Method, and Multiple Dispatch

105 In Julia, a function is a named sequence of statements that performs a computation. A method is a specific implementation of a function for particular types of arguments. A function definition starts out with a single method. But when additional definitions are provided with different combinations of argument types, the function accrues more methods. This concept is intimately tied to Julia's support for multiple dispatch, which means that the version of the function (i.e. the method) that gets called is determined by the types of all arguments. This can provide a flexible and powerful way to express program behavior.

110 The traditional form of multiple dispatch in Julia is dynamic or runtime dispatch. When a function is called, Julia examines the types of all arguments and chooses the most specific method that can apply to these types. The benefit of this approach is that it allows for polymorphism and code that adapts based on the types it encounters during execution.

115 Although not a separate dispatch mechanism, static or compile-time dispatch occurs when Julia's compiler knows the types of all arguments to a function call at compile time. In such cases, the compiler can select the appropriate method to call right away, instead of deferring the decision to runtime. This is an optimization that can result in more efficient code, but it requires the compiler to have enough information about types, which is not always possible in dynamically-typed languages like Julia.

2.3 Type Hierarchy

The type hierarchy in Julia is a system that organizes all possible types into a tree-like structure, allowing for the categorization of types and subtypes.

120 At the top of the type hierarchy tree is the Any type, which is a supertype of all other types. When a variable is defined as Any, it can hold a value of any type. This is useful in certain scenarios where maximum flexibility is required, but it can also potentially slow down the code. This is because the compiler does not know the specific type of the values, and cannot make as many optimizations.

125 Abstract types serve a similar role as interfaces in other languages. They are nodes in the tree that can have subtypes but cannot be instantiated themselves. In other words, they define a kind of protocol or set of behaviors, but one cannot create objects of these types. They are only used for organizing other types into a hierarchy.

Finally, concrete types form the leaves of the tree and represent types that can actually be instantiated, but they cannot have subtypes themselves. Examples of concrete types are Int, Float64, String, Array etc. Each concrete type is a subtype of one or more abstract types.

130 The type hierarchy is crucial in Julia because it enables the powerful feature of multiple dispatch, allowing functions to behave differently depending on the types of all their arguments.

2.4 Type Inference

135 In statically typed languages, such as Fortran, C++/C or Java, the programmer needs to declare the type of a variable when it is defined. This allows the compiler to generate efficient code because it knows exactly what types it is dealing with. However, Julia is designed to be easy to use like a dynamically typed language (such as Python or MATLAB), where it is not necessary to declare the types of variables. But unlike most dynamically typed languages, Julia can still produce very efficient code, thanks to its JIT compilation and aggressive type inference system, which allows the compiler to determine the type of a variable or expression without the programmer explicitly mentioning it. The compiler infers the type based on the values assigned or the operations used on the variable. However, type inference in Julia can fail or be suboptimal in a few different scenarios. Here are some of the most common ones:

- 140 (a) Functions with insufficient information about arguments: In some cases, a function might not have enough information about what arguments it will receive. This can make it hard for the compiler to infer the types.
- (b) Global variables: Using global variables can lead to performance issues, because the global scope can change, which in turn prevents the compiler from inferring a stable type.
- 145 (c) Type instability: In Julia, type instability refers to a situation where the type of a variable cannot be inferred consistently by the compiler at compile time. This usually happens when the type of a variable changes within a function, or when a function's return type depends on the values (not the types) of its arguments.
- (d) Containers with multiple data types: In Julia, containers are data structures used for storing collections of data. These can include arrays (an ordered collection of elements, indexed by integers), tuples (an ordered collection of elements, similar to an array, but immutable), dictionaries (an unordered collection of key-value pairs), and sets (an unordered collection

150 of unique elements), among others. If containers are used to store different types of data, the compiler may not be able to precisely infer their types.

Even when type inference fails or is suboptimal, the Julia code should still run correctly (assuming it does not have other types of errors), but it may run slower due to the additional overhead of runtime type checking. To improve performance, it is generally a good practice to try to write type-stable code and provide the compiler with as much type information as possible.

155 **2.5 Struct**

In Julia, a struct is a composite data type, similar to a class in object-oriented languages. However, Julia's struct itself is not object-oriented and has no methods directly attached to them. A struct is used to encapsulate a few related values together into a single entity, and those values are stored in fields.

160 Structs with abstract types or containers as fields can slow down Julia code, since they prevent the compiler from producing highly optimized, type-specific machine code. In other words, the lack of concrete type information at compile time forces the compiler to generate less efficient, generic code that can accommodate any potential subtype, resulting in performance penalties from dynamic dispatch and missed optimization opportunities.

165 While one approach to improving the performance of Julia structs with abstract types or containers is to specify concrete types for all fields, this could reduce the flexibility of Julia's powerful type abstraction and potentially lead to repetitive code. An alternative technique involves the use of function barriers, where abstract fields are unpacked within a function that then calls an inner function, effectively passing concrete types. This strategy leverages Julia's ability to generate efficient machine code based on the specific types of function arguments. However, a more elegant solution could be the use of parametric structs. With parametric structs, the type of the field is determined at the time of struct instance creation rather than at the struct definition. This approach allows the Julia compiler to generate highly optimized machine code tailored to the precise, concrete type used for each instance, significantly reducing the performance overhead typically associated with handling abstract types.

170

3 Methods

With an exploration of the relevant key features of Julia in the preceding section, we are now set to shift our focus to the details of the shallow water equations that we aim to solve, including its numerical discretization and implementation across various architectures.

175

3.1 Equation Set & TRiSK-Based Spatial Discretization

Our Julia model solves the shallow water equations (Cushman-Roisin and Beckers, 2011) in vector-invariant form. This is sufficiently close to the governing equations for ocean and atmospheric models to be used as a proxy to test performance with

new codes and architectures. The equation set is

$$180 \quad \mathbf{u}_t + qh\mathbf{u}^\perp = -g\nabla\left(g\eta - \frac{1}{2}\nabla K\right), \quad (1a)$$

$$\eta_t + \nabla \cdot (h\mathbf{u}) = 0, \quad (1b)$$

where \mathbf{u} is the horizontal velocity vector, $\mathbf{u}^\perp = \mathbf{k} \times \mathbf{u}$, h is the layer thickness, η is the surface elevation or sea surface height (SSH), $K = |\mathbf{u}|^2/2$ is the kinetic energy, and g is the acceleration due to gravity. If b represents the topographic height and H the mean depth, then $\eta = h + b - H$. Moreover, if f denotes the Coriolis parameter, and $\zeta = \mathbf{k} \cdot \nabla \times \mathbf{u}$ the relative vorticity, then
 185 the absolute vorticity, $\omega_a = \zeta + f$, and the potential vorticity, $q = \omega_a/h$. The term $qh\mathbf{u}^\perp$ is the thickness flux of the PV in the direction perpendicular to the velocity, rotated counterclockwise on the horizontal plane. Ringler et al. (2010) refer to it as the non-linear Coriolis force since it consists of the quasi-linear Coriolis force $f\mathbf{u}^\perp$ and the rotational part $\zeta\mathbf{u}^\perp$ of the non-linear advection term $\mathbf{u} \cdot \nabla \mathbf{u}$. We spatially discretize the prognostic equations in (1) using a mimetic finite volume method based on the TRiSK scheme, which was first proposed by (Thuburn et al., 2009), and then generalized by (Ringler et al., 2010). This method
 190 was chosen to horizontally discretize the primitive equations of MPAS-Ocean while invoking the hydrostatic, incompressible, and Boussinesq approximations on a staggered C-grid. Since this horizontal discretization guarantees conservation of mass, potential vorticity, and energy, it makes MPAS-Ocean a suitable candidate to simulate mesoscale eddies.

Our spatial domain is tessellated by two meshes, a regular planar hexagonal primal mesh and a regular triangular dual mesh. Each corner of the primal mesh cell coincides with a vertex of the dual mesh cell and vice versa. A line segment connecting
 195 two primal mesh cell centers is the perpendicular bisector of a line segment connecting two dual mesh cell centers and vice versa. Regarding our prognostic variables, the scalar SSH η is defined at the primal cell centers, and the normal velocity vector \mathbf{u}_e is defined at the primal cell edges. The divergence of a two-dimensional vector quantity is defined at the positions of η , while the two-dimensional gradient of a scalar quantity is defined at the positions of \mathbf{u}_e and oriented along its direction. The curl of a vector quantity is defined at the vertices of the primal cells. Finally, the tangential velocity \mathbf{u}_e^\perp along a primal cell
 200 edge is computed diagnostically using a flux mapping operator from the primal to the dual mesh, which essentially takes a weighted average of the normal velocities on the edges of the cells sharing that edge. Interested readers may refer to Thuburn et al. (2009) and Ringler et al. (2010) for further details on the mesh specifications.

At each edge location \mathbf{x}_e , two unit vectors \mathbf{n}_e and \mathbf{t}_e are defined parallel to the line connecting the primal mesh cells, and in the perpendicular direction rotated counterclockwise on the horizontal plane, such that $\mathbf{t}_e = \mathbf{k} \times \mathbf{n}_e$. The discrete equivalent of
 205 the set of equations (1) is

$$(u_e)_t = -F_e^\perp \widehat{q}_e = -[\nabla(g\eta_i + K_i)]_e, \quad (2a)$$

$$\left(\frac{\eta h_i}{t}\right) = -[\nabla \cdot F_e]_{i,t}, \quad (2b)$$

where $F_e = \widehat{h}_e u_e$ and F_e^\perp represent the thickness fluxes per unit length in the \mathbf{n}_e and \mathbf{t}_e directions respectively. The layer thickness h_i , the SSH η_i , the topographic height b_i , and the kinetic energy K_i are defined at the centers \mathbf{x}_i of the primary
 210 mesh cells, while the velocity u_e are defined at the edge points \mathbf{x}_e . The symbol $\widehat{(\cdot)}_e$ represents an averaging of a field from its

native location to \mathbf{x}_e . The discrete momentum equation ~~(2b)~~-(2a) is obtained by taking the dot product of (1b) with \mathbf{n}_e , which modifies the non-linear Coriolis term to

$$\begin{aligned}\mathbf{n}_e \cdot \widehat{q}_e \widehat{h}_e \mathbf{u}^\perp &= \widehat{q}_e \widehat{h}_e \mathbf{n}_e \cdot (\mathbf{k} \times \mathbf{u}) = \widehat{q}_e \widehat{h}_e \mathbf{u} \cdot (\mathbf{n}_e \times \mathbf{k}) \\ &= -\widehat{q}_e \widehat{h}_e \mathbf{u} \cdot \mathbf{t}_e = -\widehat{q}_e \widehat{h}_e u_e^\perp = -F_e^\perp \widehat{q}_e.\end{aligned}\tag{3}$$

215 Given the numerical solution at time level $t^n = n\Delta t$, with Δt representing the time step and $n \in \mathbb{Z}_{\geq 0}$, the Julia model first computes the time derivative or tendency terms of (2) as functions of the discrete spatial and flux-mapping operators of the TRiSK scheme. Then it advances the numerical solution to time level t^{n+1} using the forward-backward method

$$u^{n+1} = u^n + \Delta t \mathcal{F}(u^n, h^n),\tag{4}$$

$$h^{n+1} = h^n + \Delta t \mathcal{G}(u^{n+1}, h^n),\tag{5}$$

220 where \mathcal{F} and \mathcal{G} represent the discrete tendencies of the normal velocity and the layer thickness in functional form, and the subscripts representing the positions of these variables have been dropped for notational simplicity.

The following sections introduce the new codes that were created for this study. Three versions of the Julia code were written (Strauss, 2023): the base single-core CPU version, an altered version for GPUs with CUDA, and a multi-node CPU implementation with Julia-MPI. These were compared against existing Fortran-MPI and [single-core](#) Python versions of shallow-water
225 TRiSK models. All use a standard MPAS unstructured-mesh file format that specifies the geometry and topology of the mesh, and includes index variables that relate neighboring cells, edges, and vertices. All models have an inner (fastest-moving) index for the vertical coordinate and were tested with 100 vertical layers to mimic performance in a realistic ocean model.

3.2 Single-Core CPU Julia Implementation

The serial-mode implementation on a single core involves looping over every cell and edge of the mesh to (a) compute the
230 tendencies, i.e. the right-hand side terms of the prognostic equations (2), and (b) advance their values to the next time step. The tendencies can be functions of the dependent and independent variables as well as spatial derivatives of the dependent variable. The serial version of our model is the simplest one from the perspective of transforming the numerical algorithms into code.

In order to highlight differences in formulation, we provide a Julia code example for the single tendency term from (2) for the SSH gradient $-g\nabla\eta$, which is discretized as $-[g\nabla\eta_i]_e$. We then add a vertical index k to mimic the performance of
235 a multi-layer ocean model, but each layer is trivially redundant. In a full ocean model this [SSH gradient](#) term would be the pressure gradient, and would involve the computation of pressure as a function of depth and density. For the single-core CPU version, the Julia function [computing to compute](#) the SSH gradient is

`cpu.jl`

Listing 1. Julia example for serial CPU

```
1: function velocity\_tendencies!\(sshGradient, ssh, ...\)
240 2:     for iEdge in 1:nEdges
```



```

3: cell1 = cellsOnEdge[1,iEdge]
4: cell2 = cellsOnEdge[2,iEdge]
5: for k in 1:nVertLevels
6: sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
245 7: * (ssh[k,cell2] - ssh[k,cell1])
8: end
9: end
10: end

```

Here `cellsOnEdge` is an array of index variables describing the mesh that points to the cells neighboring an edge, and
250 `dcEdge` represents the distance between the centers of adjacent cells sharing the edge on which the normal velocity tendency
is computed. In the actual code all the tendency terms are computed within this function, but here we only show the `ssh` SSH
gradient as a brief sample.

3.3 SIMD GPU Julia Implementation

GPUs are ~~very powerful tools~~ exceedingly efficient for SIMD (Same Instruction Multiple Data) computations: ~~they have,~~
255 leveraging their thousands of independent threads, ~~which can to~~ execute the same operation ~~at the same time with~~ simultaneously
on different input values. ~~Since we numerically solve~~ As we are solving the same prognostic equation for (a) the SSH at every
cell center x_i , and (b) the normal velocity at every edge x_e of the mesh, a GPU ~~is a logical tool to employ~~ naturally emerges
as a powerful asset for our computations. By ~~placing~~ assigning subsets of cells and edges ~~on different threads of the GPU to~~
distinct GPU threads, we can ~~perform~~ conduct the tendency computations, ~~and advance and update~~ the prognostic variables ~~at~~
260 ~~once in parallel rather than~~ concurrently in parallel. This approach stands in stark contrast to looping over every cell and edge,
~~which would scale in wall-clock time according to~~ an operation that would scale linearly with the size of the mesh, ~~thereby~~
significantly impacting the wall-clock time in large scale computations.

~~We wrote CUDA kernels for~~ To harness the power of an Nvidia GPU, ~~we crafted CUDA kernels~~ using the CUDA.jl library ~~for~~
~~computing,~~ specifically for calculating the tendencies and ~~advancing~~ updating the prognostic variables to the ~~next~~ subsequent
265 time step. ~~The code for the~~ One of the remarkable features of working with GPUs and CUDA.jl is the relative ease of code
transition from a single-core ~~implementation can be converted to CUDA with surprising ease by removing~~ to a multi-threaded
GPU context. ~~Primarily, it involved replacing~~ the `for` loop ~~over the,~~ which iterated over cells and edges, ~~and instead performing~~
~~the underlying computation on a single~~ with a more GPU-friendly design where the computation is carried out independently
for each cell or edge, as shown below:

Listing 2. Julia example for GPU with CUDA

```

270 1: CUDA.@cuda blocks=cld(nEdges, 1024) threads=1024 maxregs=64
2:   velocity_tendencies_cuda!(sshGradient, ssh, ...)
3:

```

```

4: function velocity_tendencies_cuda!(sshGradient, ssh, ...)
5:     iEdge = (CUDA.blockIdx().x - 1) * CUDA.blockDim().x
275 6:         + CUDA.threadIdx().x
7:     cell1 = cellsOnEdge[1,iEdge]
8:     cell2 = cellsOnEdge[2,iEdge]
9:     for k in 1:nVertLevels
10:         sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
280 11:             * ( ssh[k,cell2] - ssh[k,cell1] )
12:     end
13: end

```

~~Each~~ In our implementation, each cell and edge of the mesh ~~will be designated to a different~~ is assigned to a distinct thread on the GPU. ~~The computation~~ Thus, computations for a single cell or edge ~~will run on a single thread, and a CUDA method~~ will be used to map the index of the thread to the index ~~are carried out on a solitary thread, with a dedicated CUDA method~~ enabling the mapping of thread indices to the corresponding indices of the cell (i) or edge (e) ~~, at which~~ where the prognostic variable is being updated. To ~~execute this method over~~ ensure the method's execution across all threads on the GPU, we ~~use~~ employ a CUDA macro to ~~call our kernel and designate~~ invoke our kernel, specifying the number of threads to ~~use, which should~~ be equal to the number of cells or edges ~~in within~~ the mesh. ~~Note that the inner computation of~~ pressureGradient ~~is identical for the~~ It is important to underscore that the core computation of `sshGradient` ~~remains identical in both~~ CPU and CUDA ~~kernel codes~~ kernel codes, demonstrating the ease of porting computational logic from CPU to GPU context.

3.4 CPU/MPI Julia Implementation

~~Rather than iterating through every~~ Instead of cycling through each cell or edge of the mesh, we ~~may parallelize the simulation~~ with multiple processors by assigning to each processor a portion ~~can optimize the simulation by employing multiple processors~~ and apportioning a segment of the mesh ~~to each one~~, a process ~~called~~ referred to as domain decomposition. However, ~~the computations of some spatial operators may require information to compute certain spatial operators, we need data from the outermost cells of the adjacent processors. So, we need the neighboring processors to communicate these neighboring processors. This necessitates inter-processor communication to exchange these critical pieces of information with each other. To ensure an efficient~~ . To streamline this communication, we ~~include an extra ring or "halo" of cells around~~ introduce an ~~additional 'halo' layer, consisting of rings of cells encircling the boundary of the region assigned to each processor each processor's assigned region, which overlaps with the region assigned to adjacent processors. We do not compute the adjacent processors' regions. Since computation is typically much cheaper than communication, it is standard practice to perform this 'halo exchange' after at least a few time steps.~~

~~As previously mentioned, we are applying a mimetic finite volume method based on the TRiSK scheme in our calculations.~~ Consequently, the computation of spatial operators such as the gradient, divergence, curl, and flux mapping operators (used

for diagnostically computing the tangential velocities) only requires the values of the prognostic variables at the cell centers and edges of adjacent cells. Thus, to compute the spatial operators that constitute the tendencies of the prognostic variables in the halo region of a processor. In fact, we cannot perform this operation without information in an additional ring of halo cells, which is not assigned to the processor under consideration. So, we obtain the updated defined at the center and edges of a specific cell, we need to consider just one small ring of cells around the cell and the values of the prognostic variables at the center and edges of each cell within this ring. The intersection of these small rings around each boundary cell of the assigned region of a processor forms the innermost ring of its halo layer. When using a k^{th} order time-stepping method involving k tendency computations within a time step, and executing the halo exchange after m time steps, the number of rings in the halo layer is set to $n = km$. For the q^{th} stage of the p^{th} time step, with $1 \leq p \leq m$ and $1 \leq q \leq k$, we compute the tendencies on the assigned region of the processor, as well as on $(m - p + 1)k - q$ rings of the halo layer, starting from the innermost one. This process is repeated, progressively ‘peeling off’ the outermost ring after each tendency computation, until after m time steps, we update the values of the prognostic variables in the halo region by within the mk rings of the halo layer via communication with adjacent processors, which contain these halo cells in their interior, and update the prognostic variables in them. In our work, we are using a forward-backward method with $k = 1$, and performing the halo exchange after every time step, resulting in $m = p = 1$. While we acknowledge this may not be the most efficient choice, our primary concern here is ensuring equivalent computational and communicative workload between the Fortran and Julia MPI models. So, as long as this parity is maintained, we consider our methodology satisfactory.

A number of crucial modifications are necessary to implement this parallelization scheme. Finally, implementing this parallelization approach necessitates a few significant modifications. For instance, we adjust the simulation methods are amended so that each process (rank) only performs computations for the set of or rank) performs computations only for its assigned cells or edges assigned to it. We use. We utilize the MPI communication channel (comm) to receive the updated values of the prognostic variables in the halo region of a processor from adjacent processors which a processor’s halo region from the adjacent processors that advance these variables. Similarly, we send. Conversely, we transmit the updated values of the prognostic variables along from the outermost region of the above-mentioned processor to adjacent processors, for which these variables belong processor under consideration to the neighboring processors, where these variables reside in the halo regions. For the TRISK-based spatial discretization and the forward-backward time-stepping method, the halo region consists of only one layer (one halo ring) of cells.

Listing 3. Julia example for CPU with MPI

```

1: # each process executes the following, receiving a different value
2: # on each rank:
335 3: comm = MPI.COMM_WORLD
4: rank = MPI.Comm_rank(comm)
5:
6: myCells = cells_for_rank(mesh_file, rank, partition_file)
7: myEdges, myHaloEdges = edges_on_cells(myCells)

```

```

340 8:
9: velocity_tendencies!(myEdges, sshGradient, ssh, ...)
10: update_halo_edges!(sshGradient, myHaloEdges, rank, comm)
11:
12: function velocity_tendencies!(myEdges, sshGradient, ssh, ...)
345 13:   for iEdge in myEdges
14:     cell1 = cellsOnEdge[1,iEdge]
15:     cell2 = cellsOnEdge[2,iEdge]
16:     for k in 1:nVertLevels
17:       sshGradient[k,iEdge] = - gravity / dcEdge[iEdge]
350 18:         * ( ssh[k,cell2] - ssh[k,cell1] )
19:     end
20:   end
21: end
22:
355 23: function update_halo_edges!(data, edgesInMyHalo, rank, comm)
24:   for neighborRank in find_neighbors(rank, comm)
25:     MPI.Irecv!(data[edgesInMyHalo,:], neighborRank, 0, comm)
26:     edgesToNeighbor = find_halo_overlap(rank, neighbor, comm)
27:     MPI.Isend(data[edgesToNeighbor,:], neighborRank, 0, comm)
360 28:   end
29: end

```

Here myCells and myEdges are the lists of cells and edges in the local domain, owned by the rank running this code, plus its halo.

3.5 CPU/MPI Fortran Implementation

```

365 †

```

The baseline comparison code for this study is the Model for Prediction Across Scales (MPAS-Ocean) (Ringler et al., 2013; Petersen et al., 2015), which is written in Fortran with MPI communication commands. It is the ocean component of the Energy Exascale Earth System Model (E3SM) (Golaz et al., 2019; Petersen et al., 2019), the climate model developed by the US Department of Energy. In this study, the code is reduced from a full ocean model solving the primitive equations to simply solving for velocity and thickness (1). Thus the majority of the code is disabled, including the tracer equation, vertical advection and diffusion, the equation of state, and all parameterizations. In order to match the Julia simulations, we employ a

forward-backward time-stepping scheme, exchange one-cell-wide halos after each time step, compute 100 layers in the vertical array dimension, and use the identical Cartesian hexagon-mesh domains (Petersen et al., 2022).

MPAS-Ocean is an excellent comparison case for Julia because it is a well-developed code base that uses Fortran and MPI, which have been standard for computational physics codes since the late 1990s. The highest resolution simulations in past studies used over three million horizontal mesh cells and 80 vertical layers, scale well to tens of thousands of processors (Ringler et al., 2013) and have been used for detailed climate simulations (Caldwell et al., 2019). MPAS-Ocean includes OpenMP for within-node memory access, and is currently adding OpenACC for GPU computations, but these were not used for this comparison to Julia-MPI on a CPU cluster.

3.6 Single-Core CPU Python Implementation

~~In addition to~~ Apart from MPAS-Ocean, we ~~compare~~ examine the performance of the Julia shallow water code ~~against an~~ relative to a single-core, object-oriented Python code (Bishnu, 2022) ~~. The Python code solves that utilizes NumPy. This Python code employs two types of spatial discretizations to solve~~ the rotating shallow water system of equations ~~using two types of spatial discretizations~~: the TRiSK-based mimetic finite volume method used in MPAS-Ocean, and a ~~discontinuous~~ Discontinuous Galerkin Spectral Element Method (DGSEM). ~~The code offers a number of standard~~ Moreover, it supports various predictor-corrector and multistep time-stepping methods, including those ~~analyzed previously scrutinized~~ for ocean modeling in Shchepetkin and McWilliams (2005).

The Julia shallow water code was first written by translating this Python code into Julia syntax. While the Julia code was ~~expanded subsequently optimized~~ for parallelization and enhanced performance, the Python code ~~was further developed to~~ serve continued to evolve as a platform ~~for conducting a verification to conduct a~~ suite of shallow water test cases for the barotropic solver of ocean models. Each ~~of these test cases~~ test case in the Python code verifies ~~the implementation of a a specific~~ subset of terms in the prognostic momentum and continuity equations, ~~e.g. the linear pressure such as the linear SSH~~ gradient term, ~~the~~ linear constant or variable-coefficient Coriolis and bathymetry terms, and ~~the~~ non-linear advection terms. Bishnu et al. (2022) and Bishnu (2021) ~~provide detailed discussions on~~ offer an in-depth exploration of these test cases ~~along with specifics of,~~ discussing the numerical implementation, ~~the~~ time evolution of the numerical error for both spatial discretizations and a subset of the time-stepping methods, and the results of convergence studies with refinement in both space and time, only in space, and only in time. ~~Out of all of these test cases~~ However, for the purposes of the present study, only the linear coastal Kelvin wave and inertia-gravity wave test cases were implemented in the Julia code ~~for the current study~~.

While ~~not used in this study,~~ a number of libraries exist to accelerate Python for various architectures. These include this study did not leverage them, several libraries exist for enhancing Python performance across various architectures, including Numba and PyCuda for GPUs, mpi4py for CPU clusters, and Cython for single-CPU acceleration. Numba (Lam et al., 2015) is an open-sourced, an open-source Anaconda-sponsored project, (Lam et al., 2015) serves as a NumPy-aware optimizing compiler, which JIT compiler. It translates Python functions to fast into swift machine code at runtime ~~using the remarkable industry-standard,~~ employing the robust LLVM compiler library. PyCUDA (Klöckner et al., 2012), ~~written which is structured~~

in C++ (~~the base at its foundational~~ layer) and Python, ~~provides~~ ~~facilitates~~ access to Nvidia’s CUDA parallel computation API from Python. ~~Mpi4py (Dalcín et al., 2005, 2008), provides~~ ~~within Python.~~ Lastly, ~~mpi4py (Dalcín et al., 2005, 2008) offers~~ Python bindings for the ~~universally recognized~~ Message Passing Interface (MPI) standard.

~~As an alternative, one can ‘cythonize~~

410 ~~Another option involves ‘cythonizing’ an existing Python code by providing~~ ~~introducing~~ static type declarations and class attributes, ~~that can then be translated~~ ~~which can subsequently be converted~~ to C++/C code and to C-Extensions for Python. Cython is an ~~optimising static compiler for both the Python programming language and the extended Cython programming language.~~ ~~It is designed to offer~~ ~~optimizing static compiler designed to yield~~ C-like performance ~~with code mostly written in Python with additional~~ ~~from Python code with supplemental~~ C-inspired syntax. The rotating shallow water Python code
415 ~~Bishnu (2022) (Bishnu, 2022)~~ is currently undergoing cythonization. ~~Cythonized~~ ~~Once cythonized, these~~ codes can further be accelerated on GPUs using Nvidia’s HPC C++ compiler ~~–~~ and the C++ Standard Parallelism (stdpar) for GPUs (Srinath, 2022). However, the ~~extent of additional effort required for such extensive~~ modifications and enhancements ~~required~~ to bring GPU-accelerated C++ algorithms to ~~the Python ecosystem~~ ~~Python~~ may not always ~~be a reasonable investment of time~~ ~~justify the time investment.~~ As we will ~~see in later~~ ~~illustrate in subsequent~~ sections, a serial Julia code, ~~which already achieves~~
420 ~~—already rivaling~~ the performance of ~~a fast compiled language, does not require extensive modifications to be parallelized on GPUs or multiple cores, and is therefore more convenient than python~~ ~~fast compiled languages—~~ ~~requires fewer modifications for GPU or multi-core parallelization.~~ This makes Julia a more convenient choice for high-performance scientific computing applications ~~compared to Python.~~

3.7 ~~Hardware and Compiler Specifications~~

425 ~~Multi-core CPU and GPU simulations were conducted on Perlmutter at the National Energy Research Scientific Computing Center (NERSC). In June of 2022 Perlmutter achieved 70.9 Pflop/s using 1,520 compute nodes, and was ranked 7th in the Top500 list. Erich Strohmaier (2022) Perlmutter is based on the HPE Cray Shasta platform. It is a heterogeneous system comprised of both CPU-only AMD ‘Milan’ nodes and GPU-accelerated ‘Ampere’ nodes, as detailed in Table 1. The Ampere’s Nvidia A100 GPU is appropriate for this study because it is designed for HPC workloads and double precision calculations.~~
430 ~~The Julia-MPI and Fortran-MPI tests were both run with up to 64 ranks per node.~~

~~The software toolchain is as follows. Both Fortran and Julia use the MPICH implementation of the Message Passing Interface (MPI). The Fortran compiler was gnu version 11.2.0, with MPICH 3.4, which is packaged on Perlmutter with the modules PrgEnv-gnu/8.3.3 and cray-mpich/8.1.24. Multi-threading was disabled (no OpenMP).~~

~~When running on a single node (up to 64 processes), we experimented with both block and cyclic distributions (run~~
435 ~~command~~ ~~srun --distribution=block:block~~ ~~versus~~ ~~srun --distribution=cyclic:cyclic.~~ The block distribution would be expected to reduce communication time because it restricts processes to a single socket for 1 to 32 processes. The cyclic distribution could speed computations because processes are distributed equally across the two sockets. In practice, there was little difference between the two distributions. The figures show the block distribution. On multi-node tests, we use 64 processes per node.

	<u>CPU Nodes</u>	<u>GPU nodes</u>
<u>overview</u>	<u>2x AMD EPYC 7763 (Milan) CPUs</u> <u>64 cores per CPU</u> <u>AVX2 instruction set</u>	<u>Single AMD EPYC 7763 (Milan) CPU</u> <u>64 cores per CPU</u> <u>Four NVIDIA A100 (Ampere) GPUs</u>
<u>memory</u>	<u>512 GB of DDR4 memory total</u>	<u>256 GB of DDR4 DRAM</u>
<u>communication</u>	<u>204.8 GB/s memory bandwidth per CPU</u> <u>PCIe 4.0 NIC-CPU connection</u> <u>1x HPE Slingshot 11 NIC</u>	<u>204.8 GB/s CPU memory bandwidth</u> <u>40 GB of HBM per GPU with:</u> <u>1555.2 GB/s GPU memory bandwidth</u> <u>12 3rd gen NVLink links between pairs of GPUs</u> <u>25 GB/s/direction for each link</u> <u>PCIe 4.0 NIC-CPU connection</u> <u>PCIe 4.0 GPU-CPU connection</u> <u>4 HPE Slingshot 11 NICs</u>
<u>performance</u>	<u>39.2 GFlops per core</u> <u>2.51 TFlops per socket</u> <u>4 NUMA domains per socket (NPS=4)</u>	<u>19.5 GPU TFlops (FP32)</u> <u>9.7 GPU TFlops (FP64)</u> <u>155.9 GPU TFlops (TF32, tensor)</u> <u>311.9 GPU TFlops (FP16, tensor)</u> <u>19.5 GPU TFlops (FP64, tensor)</u>
<u>power</u>	<u>280W thermal design power</u>	<u>300W thermal design power</u>

Table 1. Technical specifications for NERSC Perlmutter CPU and GPU nodes. (NERSC, 2023)

440 The Julia version is 1.8.3 with MPICH 4.0.2. The necessary julia packages are listed in Manifest.toml and Project.toml. These packages can be installed by executing the following lines in the julia console (opened by running the julia binary with no arguments) in the root directory of the MPAS_Ocean_Julia repository:

```
activate .  
instantiate
```

445 Subsequently, when running julia with the flag --project=. in the root directory of the MPAS_Ocean_Julia repository, all the necessary packages for the environment will be loaded in their appropriate versions.

4 Results and Discussion

We now present the results of the verification and performance tests for our Julia model. The verification tests encompass convergence plots for the spatial operators and the numerical solution. Performance tests, on the other hand, reveal the speed-up attained by initially transforming the benchmark Python-CPU code into Julia-CPU code, performing additional optimization

450

on the Julia-CPU code, and ultimately transitioning to the Julia-GPU code. We evaluate and compare the performance metrics of the Julia-GPU code, the Julia-MPI code, and the Fortran MPAS-Ocean code run on a single node. Lastly, we provide scaling plots comparing the Julia-MPI code with Fortran MPAS-Ocean, examining the variation of the wall-clock times with the processor count for two scenarios: strong scaling (maintaining the overall problem size constant) and weak scaling (preserving a constant problem size per processor).

For Julia-GPU and Julia-MPI computations, we sequentially measure wall-clock times for six samples, each comprising ten time steps. Although not a pragmatic approach, we execute the halo exchange for Julia-MPI (as detailed in Section 3.4) and the GPU to CPU transfer for Julia-GPU after every time step. For ten time steps, this leads to ten alternating computations and ten MPI exchanges or GPU to CPU data transfers per sample. Given the compilation latency attributable to Julia's just-in-time (JIT) compilation and caching of machine code (for subsequent use) during the initial function call (as elaborated in Section 2.1), the wall-clock time for the first sample is significantly larger, as anticipated. Consequently, we disregard the first sample as an outlier, utilizing only the succeeding five samples to compute the average wall-clock time. It is worth reiterating that in a realistic ocean model, an adequately large halo layer is designed to reduce the frequency of halo exchanges and minimize communication overhead. Similarly, when running an ocean model on a GPU, the GPU to CPU data transfers are required only when solution outputs are written to disk files.

4.1 Model Verification

Both serial and parallel implementations of the shallow water model described in the previous section were verified for accuracy with-through convergence tests against exact solutions. We obtained the expected second-order convergence of the various TRiSK-based spatial operators on a uniform planar hexagonal MPAS-Ocean mesh. The operators included the gradient, the divergence, the curl, and the flux-mapping operator used to interpolate the tangential velocities from the normal velocities (Figure ??). These operators are formulated as shown in Figure -3 of Ringler et al. (2010). Once the operator tests were complete, the linearized shallow water equations were verified against exact solutions for the coastal Kelvin wave and inertia-gravity wave cases, as described in Bishnu et al. (2022) and Bishnu (2021). With refinement in both space and time, we observe the expected first-order convergence of the numerical solution (Figure ??), spatially discretized with the second-order TRiSK scheme, and advanced with the first-order forward-backward time-stepping method (Bishnu, 2021).

4.2 Acceleration of Julia with GPU Hardware Typing Optimizations

The first comparisons were made between the Julia serial CPU version of the shallow water model was compared against the Julia-CUDA library GPU version and the reference Python CPU code (Table 2 and Figure ??). Tests, as outlined in Tables 2 and 3. The initial serial development and testing were conducted on the Darwin cluster at Los Alamos National Laboratory, using a single node equipped with an Intel Cascade Lake CPUs (Gold 6248 with a clock rate of 2.5 GHz and

27.5M-Cache) and the Nvidia Quadro RTX 8000 “Turing” GPU architecture (4608 CUDA cores, 16.3 TFLOPS peak single
485 precision performance, 48 GB GPU memory, and GPU memory bandwidth of 672 GB/s). All performance tests described in
this and the following sections used the coastal Kelvin wave test case platform equipped with an Intel Xeon processor. The
performance tests detailed in this section and subsequent ones involve advancing the linear shallow water equations on a planar
hexagon mesh with the linear shallow water equations and hexagonal mesh with 100 vertical layers. Samples are averaged over
ten trials These equations incorporate the coastal Kelvin wave’s exact solution to specify the initial and boundary conditions. All
490 codes use double-precision (8 byte) real numbers, and performance tests do not include the time for initialization, input/output,
or generating plots.

~~In our first version of the Julia~~

In its primary state, the single-core CPU code, we did not take any special steps for code optimization, and it was already 13
times faster than Python. Julia code, even without specific optimizations, outperformed Python by a factor of 13. Despite both
495 Julia and Python both have dynamic typing, but Julia has the ability to go much faster since it also supports concrete typing.
Julia is compiled, but hides it cleverly by compiling on the fly based on what datatypes are provided at run time. It supports a
hierarchical abstract typing system, allowing for semi-specified types, such as “Any”, which all types extend and is the default
if no type is specified (thus acting like python), or “AbstractArray”, which can be occupied at run time with any Array-like
data. being dynamically typed, Julia gains a notable computational edge through its ability to infer types at runtime, perform
500 JIT (Just-In-Time) compilation, cache and directly manipulate machine code (Section 2.1).

~~After the initial Julia development~~ Following the initial development phase in Julia, further effort was put into optimization,
which led to a 10–20 times speed-up for the CPU-serial code. ~~The changes included optimizing~~ These enhancements involved
optimization for memory management by ~~tracking down and reducing~~ identifying and curtailing unnecessary allocations that
~~contributed significantly to the run time, as well as making all types and subtypes concrete rather than abstract, to minimize~~
505 ~~on-the-fly compilation. These improvements are explained in more detail in section~~ substantially increased runtime, along with
replacing the generic ‘Any’ type with concrete types in function definitions, and specifying the types of components within
structs. Detailed explanations of some of these improvements can be found in Section 5.

~~We found the CUDA GPU implementation to be significantly faster than the single-core implementation. Because the~~
~~memory transfer between the~~ In order to effectively compare CPU and GPU takes many orders of magnitude longer than
510 the actual on-GPU computations, we split them out in Table 2 and Figure ???. The memory transfers require between 0.015s
and 0.68s and scale with the array size, while the GPU computations alone are, at 0.00027s for the 512x512 resolution
case, and do not scale with resolution. This shows the power of GPUs, where computations alone can run over 40,000 times
faster on the GPU than the CPU, but this speed-up is substantially diminished by the memory transfer time. Still, codes
that are designed with a small memory footprint and limited memory transfer can greatly benefit from GPU computations.
515 Strategically reducing array precision to 4-byte or even 2-byte reals for certain variables allows higher-resolution domains to
fit on GPUs (Ye et al., 2022; Klöwer et al., 2022). In addition, single-precision floating point numbers (CUDA Float32 data
type) calculations may execute significantly faster than Float64 (Julia Development Team). We did not leverage Float32
in this work, but it shows that GPU simulations could run even faster than the results shown here.

Summing the GPU memory transfer and compute for the 10 timestep performance test, the GPUs were 229 to 386 times faster than the single CPU (Table 3). This compares to published studies of ocean models that show a speed-up from CPU to GPU ranging from 5–50 (Bleichrodt et al., 2012; Zhao et al., 2017; Xu et al., 2014), and a speed-up of up to 1556x for a GPU/CUDA-Based Parallel Weather and Research Forecast Model (WRF) (Mielikainen et al., 2012). Note that our speed-up factor could be increased substantially by transferring data from the GPU to CPU less frequently. For a low-resolution ocean model with 30-minute time steps, the speed-ups in Table 3 correspond to collecting data every 10 time steps, which is 5 hours of model time. One could instead collect data for analysis every 100 time steps (~times, one must first decide which architectures can provide a fair comparison. We chose to conduct tests on Perlmutter, with single-node CPU performance using 64 cores compared to the associated single-node GPU performance on the same machine. The results depicted in Figure ?? indicate similar wall-clock times, with Julia-GPU times being 2 days), and that would result in a ~3 times slower than Julia-MPI times. Fortran-MPI speeds were comparable to Julia-GPU for larger problem sizes, but faster for smaller domains. The similarity of the full-node CPU and GPU speed-up of 2290 to 3860, because the compute time is negligible compared to the memory transfer. On the other hand, if model communication is required frequently for surface data forcing or coupling with atmospheric and sea ice components, the speed-up is drastically reduced. For example, if memory must be transferred between the CPU and GPU every time step, the speed-ups range from 23–39. The point is that GPU performance is wholly dependant on the GPU communication frequency. Timings is rather surprising, given the architectural differences. The listed performance for the A100 is 9.7 TFlops for 64-bit floats, while the AMD EPYC 7763 delivers 39.2 TFlops per core, resulting in a total of 2.5 TFlops for 64 cores. Based on these manufacturer specifications, we would expect the A100 to perform faster.

data type) calculations may execute significantly faster than `Float64` (Julia Development Team). We did not leverage `Float32` in this work, but it shows that GPU simulations could run even faster than the results shown here.

Python, CPU

Julia, CPU-serial (unoptimized)

Julia, CPU-serial (optimized)

Julia, GPU, total $4.90E-02$ $2.03E-01$ $8.64E-01$ ——— transfer to GPU $2.98E-02$ $1.16E-01$ $4.58E-01$ ——— compute on GPU $2.51E-04$ $2.67E-$

Table 2. Wall clock duration (seconds) of performing ten timesteps with 100 layers on an Intel Cascade Lake CPU or an NVidia Turing GPU.

The same kernel was executed with the same data but different block sizes and the average execution time over 1000 runs was recorded. Fewer threads per block results in faster execution times on the GPUs.

GPU threads are grouped into threadblocks (or just “blocks”) for efficiency. While calling the kernel function, we must specify the number of blocks and number of threads per block (the “block size”), as shown in listing 2. Within the kernel, we obtain the index of the block and thread, multiply the block index by the block size, and add the thread index to compute a global index. There is a maximum possible block size, but we can choose any smaller value to execute the kernel with. The block size does have an effect on how quickly the kernel runs, so we benchmarked the evaluation time of the same kernel

	128x128	256x256	512x512
Python, CPU	274	177	149
Julia, CPU-serial (unoptimized)	20	12	12
Julia, CPU-serial (optimized)	1	1	1

Julia, GPU **229 366 386**

Table 3. Speed-up (bold) or slow-down (non-bold) factor Increase in run time compared to the optimized CPU-serial Julia version at the same resolution. GPU speed-ups are based on transferring arrays between GPU and CPU every ten time steps.

~~run with different block sizes, as shown in Figure ?? . Smaller block sizes run faster on the GPUs by 15%. This is interesting to note, but GPU compute time is so small compared to the memory transfer time that thread tuning has little impact on the overall simulation time.~~

4.3 Julia-MPI versus Fortran-MPI

550 Julia and Fortran codes were compared on multi-node CPU clusters, where both used MPI for communication between processors. Comparisons were made with domains of 128, 256, and 512-squared grid cells solving the shallow water equations. All timing tests were conducted for 10 time steps and repeated 12 times on each processor count, spanning 2 to 2048 processors by powers of two. The vertical dimension included 100 layers to mimic ocean model arrays and provide sufficient computational work on each processor. Separate timers report on computational work versus MPI communication within the

555 time-stepping routine. The i/o, initialization, and finalization time is excluded.

~~Simulations were conducted on Cori-Haswell at the National Energy Research Scientific Computing Center (NERSC). Cori-Haswell consists of 2,388 nodes in 14 cabinets, using Intel Xeon Processor E5-2698 v3 with a clock rate of 2.3 GHz. Each processor has 32 physical threads per node and two hyper-threads per core, with~~ We compared the Julia and Fortran models with both strong and weak scaling. In strong scaling, the same problem size (the mesh size and number of timesteps to simulate) is run with a varying degree of parallelization. In weak scaling, the problem size scales with the degree of parallelization, such that the problem size allotted to each process is constant. In both, the duration of time it takes to complete the simulation is the dependent variable and the number of processors used for the simulation is the independent variable. This is additionally separated out into three columns: the total time to simulate the problem, the time spent on just the computation (the mathematical implementation of the equation set), and the time spent on just communicating the necessary data between

560 processes.

The strong scaling for hexagonal meshes of 128x128, 256x256, and 512x512 cells is shown in Figure ?? . In strong scaling, we expect a downwards trend of computation time with the number of processors, often giving way to a flatter behavior at high enough processor counts where communication time dominates. We indeed observe this trend with both the Julia and Fortran implementations. In the total time column, we see Julia and Fortran match very closely at lower processor counts,

570 taking almost identical time to run. In the middle range of processor counts (16-128), Fortran takes more time than Julia. Then

at high processor counts, Julia tends to become slower than Fortran, no longer scaling as well. In the computation-only column, we see that Julia is actually faster than Fortran across the board. But due to the necessary communication time (which is the dominant effect at higher process counts where insufficient work is being done by each processor) the Julia implementation is not as efficient at high processor counts as the communication time does not decrease with greater parallelization like it does for Fortran (right column.)

The weak scaling with 64, 128GB of memory per node. The interconnect is a Cray Aries with Dragonfly topology and > 45 TB/s global peak bisection bandwidth. The Julia-MPI and Fortran-MPI tests were both run with up to 32 ranks per node, and 256 cells per process is shown in Figure ???. The rows here do not represent distinct mesh resolutions like in Figure ??. Mesh size instead varies with the number of processors (resolution changes along the x-axis). In weak scaling, we expect an initially increasing trend of computation time increasing with the number of processors due to the more communication required with more processors, giving way to a flat behavior as the communication time reaches its maximum and computation is constant. Indeed, we observe a very flat behavior in the computation only column. Julia again is better across the board, while Fortran is slower in the middle range of process counts. In communication, we see Julia and Fortran increase as expected, although Fortran communication time levels out sooner while Julia is slower at communication with higher process counts, like we observed in strong scaling.

The scaling plots in Figure ?? show that the As another way of measuring scaling we keep the computational resources constant, using one node (64 processes) to compare simulation time of various mesh sizes (shown in Figure ??). Here it is appropriate to also compare the GPU implementation with Julia since this represents a fixed-size computational resource. We observe an increasing trend for the Julia-MPI, Fortran-MPI and Julia-GPU implementations, which we expect for an increasing problem size with constant computational power. In the middle plot while Julia-MPI and Fortran-MPI models have identical performance at two cores; Julia-MPI is faster by up to a factor of two for mid-range core counts; and Fortran-MPI is 2x faster than Julia-MPI at higher ranges, closely match, the Julia-GPU takes more time at lower mesh sizes before coming to match the MPI models. It costs some time to launch a GPU kernel. The GPU does not have to communicate data between threads like MPI, instead using shared memory; but the data produced by the computation must be moved back to main memory at some point to make use of the results (to write them to disk or further process them), so we time this memory movement and compare it to MPI time. It is important to note, however, that depending on the resolution, application this memory movement could be done very infrequently unlike MPI communication which must be done every few time steps to exchange the information in the halo layers. If only the final state of the model is important and intermediate steps don't need to be recorded this communication time could be foregone, making the GPU more efficient for such a use case.

It is notable that such a mass-parallel shared-memory based architecture as a GPU is so similar to communication-based CPU-parallelization over a full node, as shown in Figure ??. Based on the technical specifications in Table 1, the GPU would be expected to run nearly four times faster than a CPU node. The GPU has a reported performance of 9.7 TFlops for 64-bit floating point operations, while the CPU is 2.51 per CPU-core (39.2 GFlops per core, with 64 cores). It is not clear why the GPU runs at the same speed as the CPU node.

605 For both languages ~~, computation scales well with~~ computation time scales well, decreasing at close to perfect scaling with
the processor count, while communication ~~does not, and communication time does not and so~~ progressively requires a much
larger fraction of time at higher processor counts (Figure ??). Once computations are optimized, communication, which is
fixed by the interconnect speed, will remain a bottleneck regardless of the language (see, e.g. Koldunov et al. (2019)). ~~At the~~
~~lowest resolution of 128x128, there is insufficient work beginning at 512 processors (which corresponds to 32 grid-cells per~~
610 ~~processor), and timing is dominated by communication, resulting in poor scaling above 512 processors. Communication times~~
~~in Julia are much more variable than in Fortran across samples and processor counts, as shown in the right column of Figure~~
~~??.~~ ~~When measuring computation time without communication (Figure ??, right column), Julia-MPI scales nearly perfectly,~~
~~while Fortran-MPI computational time drops off from perfect scaling at 8 and 16 cores. This produces the Julia times that are~~
~~2x faster for the total times for mid-range processor counts of 16 and higher. Overall, Julia performance on CPU clusters is~~
615 ~~competitive with Fortran. Once the high-level codes have been optimized, the “winner” between Julia and Fortran will likely~~
~~depend on the details of the MPI libraries and hardware.~~

5 Optimization Tips for Julia Developers

Julia serves the dual purpose of a prototyping language ~~as well as and~~ a production language. ~~Not only can we construct~~
~~quick-to-write but slow-performing code (although still significantly~~ It allows for the creation of code that is quick to write but
620 slower in performance, although it is still considerably faster than other ~~development interpreted~~ languages, as ~~we saw with~~
~~comparison to python) to demonstrate an idea, we can also spend~~ evidenced by our comparison with Python, for conceptual
demonstration. Additionally, with a bit more time ~~to carefully construct an optimized code to achieve performance on par~~
~~with Fortran. Julia’s ability to act as a prototyping language can be attributed to one of its key features: dynamic typing.~~
~~Just like Python, variables may be initialized without defining their types. However, Julia is also endowed with a static typing~~
625 ~~feature, even though it is optional. If the variable types are statically defined in a concrete fashion, performance is greatly~~
~~improved. Julia activates its dynamic typing feature with an “Any” type which could be any type at run time. So, Julia must~~
~~compile parts of the code on the fly (Julia Development Team, 2016). A method involving an “Any” type is compiled at run~~
~~time for whatever type is actually provided during execution (called just-in-time compiling). The implication is that without~~
~~static typing, performance will greatly suffer from compilation at run time. Additionally, with concrete types~~ investment and
630 thoughtful construction, it is possible to develop highly optimized code that achieves performance comparable to compiled
languages like Fortran.

In Julia, the Julia compiler may optimize the code much further than if it is compiled for an unknown type it is the types of
objects, not their values, that the compiler leverages to construct efficient machine code. This means, barring a few specific
scenarios (as outlined in Section 2.4), Julia can carry out extensive type inference and generate highly optimized code without
635 requiring explicit type declarations for variables.

~~When first creating the MPAS shallow water core in Julia, we did not specify the array types, and let Julia assign them the~~
~~“Any” type.~~ Nevertheless, there are situations where type declarations can significantly enhance performance. A prime example

is a struct containing fields with abstract types or containers. A more efficient approach in such cases, however, would be to transform these structs into parametric ones, a process detailed in Section 2.5. Let us consider an example in the context of our Julia-CPU and Julia-GPU codes.

In the first (unoptimized) iteration of the CPU code, we omitted specific array type declarations, allowing Julia to assign the default 'Any' type:

```
struct MPAS_Ocean
    layerThickness
    normalVelocity
    ...
end
```

However, by concretely defining these variables to be floating-point arrays, we gain a substantial performance boost:

explicitly typed as arrays of floating-point numbers, we witnessed a substantial boost in performance:

```
struct MPAS_Ocean
    layerThickness::Array{Float64}
    normalVelocity::Array{Float64}
    ...
end
```

When parallelizing for the graphics card,

In the process of parallelizing our code for GPU execution, we employed a different array type is used that is suited for GPUs. We tried defining `CUDA.CuArray`, specifically designed for GPU workloads. Our first approach was to create an abstract array type that encompasses both the could encapsulate both CPU and GPU data types, so that. This allowed `CUDA.CuArrays` and regular `Arrays` could to be used interchangeably, allowing enabling the model to be run on operate on either the GPU or CPU at will. We also used an abstract type specification on the contents of these arrays `F <: Float`, meaning any type extending as required. Additionally, we imposed a parametric constraint on the array contents (`F <: AbstractFloat`), signifying that any subtype of the abstract floating point type can be used could be passed at runtime.

```
struct MPAS_Ocean{F <: AbstractFloat}
    layerThickness::AbstractArray{F}
    normalVelocity::AbstractArray{F}
end
```

This approach seems like it should be performant, since the types are defined before run time. However, While this strategy may appear efficient because types are declared before runtime, the use of abstract types, like an Any type, slow down execution since at run time they may actually be a different type that extends akin to the 'Any' type, can actually hinder execution

~~speed. At runtime, these types could be different subtypes of the abstract type (like `CUDA.CuArray` or `Array`), meaning the compiler is doing just-in-time compiling. Similarly, specifying an inexact. This means that the specific methods to be used for these types cannot be determined at compile-time, leading to dynamic dispatch and negatively affecting performance. Furthermore, specifying a broad element type (`F <: AbstractFloat`) rather than a concrete type (`Float64`) is very inefficient.~~

675

~~Instead, two separate structures should be defined concretely when running on GPUs versus CPUs: `F <: AbstractFloat` instead of a concrete one (`Float64`) may lead to additional performance issues because the compiler cannot optimize operations for a specific type. Instead, it must prepare for all possible types that fall under `AbstractFloat`, which can introduce additional overhead.~~

680

~~An alternative would be to define two distinct structures with concrete types depending on whether we are targeting GPUs or CPUs:~~

```
struct MPAS_Ocean_CUDA
    layerThickness::CUDA.CuArray{Float64,2}
    normalVelocity::CUDA.CuArray{Float64,2}
```

685

end

```
struct MPAS_Ocean
    layerThickness::Array{Float64,2}
    normalVelocity::Array{Float64,2}
```

690

end

~~Now In this modification, both the array types are concrete, and element types are concrete (`Float64`), and the number of dimensions is specified (`Float64, 2`). This code no longer has the advantageous feature of being able to switch between running on the `Float64, 2`, resulting in a considerable performance boost. However, this code loses the versatility of dynamically switching between CPU and GPU on the fly. However, the execution speed is massively improved. We found that making this change from abstract to concrete array types sped up computation execution, and also necessitates additional lines of code.~~

695

~~A superior solution involves the use of parametric structs, where abstract types are included in the type argument of the struct. This empowers the compiler to infer the type of fields at compile time and optimize the code accordingly for the appropriate array type:~~

700

```
struct MPAS_Ocean{A<:AbstractArray{<:AbstractFloat, 2}}
    layerThickness::A
    normalVelocity::A
end
```

~~We discovered that either of these modifications resulted in a computation speedup by a factor of 34x.~~

705 ~~The key in optimizing~~ A pivotal strategy in enhancing the performance of Julia code, we ~~found, was reducing allocations.~~
~~Memory allocation significantly slows down execution. And it is not always obvious what seemingly innocent actions may~~
~~allocate memory.~~ For example, simply reading discovered, lies in minimizing memory allocations. Excessive memory allocation
can drastically impede code execution, and it is often not readily apparent when seemingly trivial operations are culprits of
unnecessary memory allocation. To illustrate, consider the act of extracting a pair of values from ~~an array with two columns:-~~
710 a two-column array:

```
cell1Index, cell2Index = cellsOnEdge[:,iEdge]
```

~~can allocate significant memory~~ Surprisingly, this operation can lead to considerable memory allocation. In one particular test,
this ~~one line (executed repeatedly single line—recurrent throughout the simulation) allocated~~ —was found to allocate as much
as 408 KiB. This is ~~because the line is really creating due to the creation of~~ a tuple, ~~not directly reading rather than a direct~~
715 extraction of each column into the two-respective scalar variables. ~~If we separate this into two lines to enforce only using~~
~~scalars and not allocating tuples or arrays:-~~

By dividing the operation into two distinct lines, thus bypassing tuple or array creation:

```
cell1Index = cellsOnEdge[1,iEdge]
```

```
cell2Index = cellsOnEdge[2,iEdge]
```

720 ~~then this cuts allocations to zero—making this line almost instantaneous, and dropping the~~ we successfully reduce memory
allocations to zero. This modification causes the operation to be nearly instantaneous, reducing the total time spent on the ~~whole~~
~~tendency calculation entire tendency calculation by 50%,~~ from 198 μs to 99 μs . ~~That means this line alone was responsible for~~
~~about 50% of the computation time, when it could be rewritten to take no time at all.~~

~~There are likely many inconspicuous lines like this lurking in one~~ One's Julia code ~~, slowing it down substantially~~ can
725 potentially harbor many such covert operations, contributing significantly to slower performance. Additionally, even ~~one~~
~~overlooked field which is not concretely typed may significantly slow execution.~~ Luckily, Julia is equipped with a tool to
quickly locate such memory-hoarding lines. This tool is called `@code_warntype`. Prefixing a function call with it will print
out a color-coded list breaking each line down to individual memory operations:-

a single struct with abstract types or containers as fields can notably hamper execution speed. Fortunately, Julia provides the
730 @code_warntype tool for quickly identifying such memory-intensive lines:

```
@code_warntype calculate_normal_velocity_tendency!(mpas)
```

It ~~helpfully highlights inexact~~ color-codes non-concrete types and memory allocations ~~with red, pointing a user right in red,~~
thereby directing users precisely to the lines and fields that ~~need to be optimized.~~ This feature alone makes Julia very powerful
require optimization. This singular feature elevates Julia's utility for high-performance applications, ~~significantly speeding up~~
735 development time substantially accelerating the development time needed to optimize a model's performance.

Another ~~very helpful tool when optimizing Julia code~~ valuable tool in the Julia optimization arsenal is `--track-allocations`,
a command line option that can be ~~added~~ appended to any Julia execution ~~as follows:-~~

⋮

```
$ julia --track-allocations=user ./anyJuliascript.jl
```

740 ~~A new file is created~~This generates a new file at ./anyJuliascript.jl.XXX.mem (where XXX ~~is some unique number~~represents a unique identifier). This file ~~contains~~presents each line of the script, prefixed by the ~~number~~amount of memory allocations ~~created~~generated by that line, ~~giving a~~providing a comprehensive line-by-line ~~breakdown~~overview of where allocations occur.

6 Conclusions

As new programming languages and libraries become available, it is important for model developers to learn new techniques and evaluate them against their current methods. This is particularly true as computing architectures continue to evolve, and long-standing languages such as C++ and Fortran require additional libraries to remain competitive on new supercomputers.

In this work, we created three implementations of a shallow water model in Julia in order to compare ease of development and performance to standard Fortran and Python implementations. The three Julia codes were designed for single-CPU, ~~GPU-enhanced single-CPU~~GPU, and parallelized multi-core CPU architectures. Julia-MPI speeds were identical to Fortran-MPI at low core counts, 2x faster for mid-range, and 2x slower at higher core counts. Julia-MPI exhibited better scaling than Fortran-MPI for computation-only times, and more variability for communication times.

~~The most surprising result of this study was the speed of computations on the GPUs—a speed-up of 40,000 to over 100,000 times compared to the CPU. Of course, this comes with the caveat that memory transfer between CPU and GPU can take thousands of times longer than the computation, up to 0.5s at our highest resolution. So the key is to transfer memory to and from the GPU as little as possible, which is a well-known practice~~

Julia-GPU scaled very similarly to Julia-MPI, despite these implementations being based on not only vastly different architectures but entirely different parallelization paradigms (shared memory versus communication). However, this result assumes the data from the GPU is communicated back to main CPU memory every time step. For many applications this is not necessary. If one can fit the full resolution of a computational physics domain within the memory of a single graphics card and sample results rarely, GPUs ~~offer~~can offer significant speed-ups. For climate models, a single low-resolution component may well fit into GPU memory if the developers are careful with their memory footprint. The difficulty is that including ocean, atmosphere, land, and sea ice components requires the use of multiple nodes, and inter-node communication will keep the model slow, regardless of the GPU speed. Higher-resolution domains will need many nodes for each component and present the same problem.

765 The shallow water equations are simple enough for rapid development and verification, yet contain the salient features of any ocean model: intensive computation of the tendency terms, a time-stepping routine, and for the parallel version, interleaved halo communication of the partition boundary. Indeed, this layout, and the lessons learned here, apply to almost all computational physics codes.

This work specifically tests unstructured horizontal meshes, as opposed to structured quadrilateral grids. Unstructured meshes refer to a neighbor's index using additional pointer arrays, so require an extra memory access for horizontal stencils.

In structured grids, the physical neighbors are also neighbors in array space ($i + 1, j + 1$, etc), which leads to more contiguous memory access patterns that are easier for compilers to optimize. Our results show that unstructured meshes do not present any significant challenge in either Fortran or Julia. The use of a structured vertical index in the innermost position and testing with 100 layers provides sufficient contiguous memory access for cache locality.

775 In the end, we were impressed by our experience with Julia. It did fulfill the promise of fast and convenient prototyping, with the ability to eventually run at high speeds on multiple high performance architectures—after some effort and lessons learned by the developers. The Julia libraries for MPI and CUDA were powerful and convenient. E3SM does not have plans to develop model components with Julia, but this study provides a useful comparison to our C++ and Fortran codes as we move towards heterogeneous, exascale computers.

780 ~~GitHub:~~

~~Zenodo:~~

~~Zenodo:~~

~~Zenodo:~~

Code and data availability. Three code repositories were used for the performance comparisons in this study. These are publicly available
785 on both GitHub and Zenodo:

1. Julia Shallow Water code for serial CPU, CUDA-GPU, and MPI-parallelized CPU

[GitHub:](https://github.com/robertstrauss/MPAS_Ocean_Julia) https://github.com/robertstrauss/MPAS_Ocean_Julia (license: GNU General Public License v3.0)

[Zenodo:](https://doi.org/10.5281/zenodo.7493064) <https://doi.org/10.5281/zenodo.7493064> (license: Creative Commons Attribution 4.0 International)

2. Python Rotating Shallow Water Verification Suite

790 [GitHub:](https://github.com/siddharthabishnu/Rotating_Shallow_Water_Verification_Suite.git) https://github.com/siddharthabishnu/Rotating_Shallow_Water_Verification_Suite.git. (license: LANL/UCAR*)

This study used the specific code version https://github.com/siddharthabishnu/Rotating_Shallow_Water_Verification_Suite/tree/v1.0.1 (license: LANL/UCAR, <https://github.com/MPAS-Dev/MPAS-Model/blob/master/LICENSE>.)

[Zenodo:](https://doi.org/10.5281/zenodo.7421135) <https://doi.org/10.5281/zenodo.7421135> (license: BSD 3-Clause “New” or “Revised”)

3. Fortran-MPI MPAS Shallow Water code with Coastal Kelvin wave initial condition (Petersen et al., 2022)

795 [GitHub:](https://github.com/MPAS-Dev/MPAS-Model) <https://github.com/MPAS-Dev/MPAS-Model>. (license: LANL/UCAR, <https://github.com/MPAS-Dev/MPAS-Model/blob/master/LICENSE>.) This study used the specific code version https://github.com/mark-petersen/MPAS-Model/releases/tag/SW_julia_comparison_V1.0.

[Zenodo:](https://doi.org/10.5281/zenodo.7439133) <https://doi.org/10.5281/zenodo.7439133> (license: Creative Commons Attribution 4.0 International)

800 The planar hexagonal MPAS-Ocean meshes used in this study for the numerical simulations and convergence tests of the coastal Kelvin wave and the inertia-gravity wave can be obtained from the Zenodo release of the Python Rotating Shallow Water Verification Suite Meshes at <https://doi.org/10.5281/zenodo.7421135>.

In order to reproduce the figures in this paper, follow the instructions below:

- Download the code for this project from (1) above. Acquire the necessary mesh files from <https://doi.org/10.5281/zenodo.7421135>, extract the zip file, and copy the:
 - 805 ● ‘MPAS_Ocean_Shallow_Water_Meshes/MPAS_Ocean_Shallow_Water_Meshes_Julia_Paper/InertiaGravityWaveMesh/’ directory into the MPAS_Ocean_Julia repository at path ‘MPAS_Ocean_Julia/’;
 - ‘MPAS_Ocean_Shallow_Water_Meshes/MPAS_Ocean_Shallow_Water_Meshes_Julia_Paper/CoastalKelvinWaveMesh/ConvergenceStudyMeshes/’ directory into the MPAS_Ocean_Julia repository at path ‘MPAS_Ocean_Julia/MPAS_O_Shallow_Water/ConvergenceStudyMeshes/’.
- 810 ■ Reproduce the figures in this paper as follows:
 - Figure ??: Run the Jupyter notebooks ‘/Operator_testing.ipynb’ to generate the data for the convergence tests of the spatial operators, and ‘/operator_convergence_plotting.ipynb’ to create plots from this data at ‘/output/operator_convergence/<operator>/Periodic/<figure>.pdf’. Run the notebook ‘./InertiaGravityWaveConvergenceTest.ipynb’ to generate the numerical solution and convergence plot of the inertia-gravity wave test case at ‘./output/simulation_convergence/inertiagravitywave/Periodic/CPU/’.
 - 815 ● Figures ??, ??, ??, and ??: On a cluster with at least 128 nodes and 64 processes per node, use the script ‘./run_scaling_16x_to_512x.sh’ to run the performance scaling tests on each mesh resolution starting from 16x16 all the way up to 512x512. The results will be saved in ‘./output/kelvinwave/resolution<mesh size>/procs<maximum number of processors>/steps10/nvlevels100/’. Run the notebook ‘/GPU_performance.ipynb’ on a node with an NVIDIA graphics card to initiate the performance tests on the GPU. Run the notebook ‘./scalingplots.ipynb’ or the Julia script ‘./scalingplots.jl’ to generate the plots in the paper at ‘/plots/<type>/<figure>.pdf’.
 - 820 ● Tables 2 and 3: Run ‘./serial_julia_performance.jl’ with Julia to generate the timing data of the optimized Julia-CPU code. Download the unoptimized version of the code from https://github.com/robertstrauss/MPAS_Ocean_Julia/tree/unoptimized or MPAS_Ocean_Julia-unopt.zip from <https://doi.org/10.5281/zenodo.7493064>. Run the Julia script ‘./serial_julia_performance.jl’ in the directory of the unoptimized code. The results will be saved in text files at ‘./output/serialCPU_timing/coastal_kelvinwave/unoptimized/steps_10/resolution_<mesh size>/’ in the unoptimized directory and ‘./output/serialCPU_timing/coastal_kelvinwave/
 - 825 unoptimized/steps_10/resolution_<mesh size>/’ in the main/optimized directory.

Author contributions. Code development, testing, and timing were conducted by all authors. SB led the test case design and verification. RRS led the data analysis and Julia optimization. The manuscript was written cooperatively by all authors. MRP conceptualized the project and conducted funding acquisition.

830 *Competing interests.* The authors declare no competing interests.

Acknowledgements. ~~RRS gratefully acknowledges the support of the U.S. Department of Energy (DOE) through the Los Alamos National Laboratory (LANL) LDRD Program and the Center for Nonlinear Studies for this work.~~ SB was supported by Scientific Discovery through Advanced Computing (SciDAC) projects LEAP (Launching an Exascale ACME Prototype) and CANGA (Coupling Approaches for Next

835 Generation Architectures) under the DOE Office of Science, Office of Biological and Environmental Research (BER). [RRS gratefully acknowledges the support of the U.S. Department of Energy \(DOE\) through the Los Alamos National Laboratory \(LANL\) LDRD Program and the Center for Nonlinear Studies for this work.](#) MRP was supported by the Energy Exascale Earth System Model (E3SM) project, also funded by the DOE BER.

840 This research used computational resources provided by: the Darwin testbed at LANL, which is funded by the Computational Systems and Software Environments subprogram of LANL's Advanced Simulation and Computing program (NNSA/DOE); the LANL Institutional Computing Program, which is supported by the DOE National Nuclear Security Administration under Contract No. 89233218CNA000001; and the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

845 [The authors extend their gratitude to the anonymous reviewers, whose valuable insights and constructive feedback were instrumental in elevating the quality and clarity of this paper. Additionally, the authors recognize the beneficial interactions with the CliMA team at MIT and Caltech, alongside the E3SM team spread across multiple national laboratories within the United States. Their significant input has enriched the manner in which the paper has been articulated.](#)

References

- Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B.: Julia: A fresh approach to numerical computing, *SIAM review*, 59, 65–98, 2017.
- 850 Bishnu, S.: Time-Stepping Methods for Partial Differential Equations and Ocean Models, Ph.D. thesis, Florida State University, <https://doi.org/10.5281/zenodo.7439539>, 2021.
- Bishnu, S.: Rotating Shallow Water Verification Suite, <https://doi.org/10.5281/zenodo.7421135>, 2022.
- Bishnu, S., Petersen, M., Quaife, B., and Schoonover, J.: Verification Suite of Test Cases for the Barotropic Solver of Ocean Models, <https://doi.org/10.22541/essoar.167100170.03833124/v1>, 2022.
- Bleichrodt, F., Bisseling, R. H., and Dijkstra, H. A.: Accelerating a barotropic ocean model using a GPU, *OCEAN MODEL*, 41, 16–21, 855 <https://doi.org/10.1016/j.ocemod.2011.10.001>, 2012.
- Caldwell, P. M., Mametjanov, A., Tang, Q., Van Roekel, L. P., Golaz, J. C., et al.: The DOE E3SM Coupled Model Version 1: Description and Results at High Resolution, *J ADV MODEL EARTH SY*, 11, 4095–4146, <https://doi.org/10.1029/2019MS001870>, 2019.
- Cushman-Roisin, B. and Beckers, J.-M.: Introduction to geophysical fluid dynamics: physical and numerical aspects, Academic press, 2011.
- Dalcín, L., Paz, R., and Storti, M.: MPI for Python, *J PARALLEL DISTR COM*, 65, 1108–1115, 2005.
- 860 Dalcín, L., Paz, R., Storti, M., and D’Elía, J.: MPI for Python: Performance improvements and MPI-2 extensions, *J PARALLEL DISTR COM*, 68, 655–662, 2008.
- Erich Strohmaier: Top 500 the list, <https://www.top500.org/lists/top500/2022/06>, visited on 2023-06-15, 2022.
- Gevorkyan, M. N., Demidova, A. V., Korolkova, A. V., and Kulyabov, D. S.: Statistically significant performance testing of Julia scientific programming language, *J PHYS CONF SER*, 1205, 012 017, <https://doi.org/10.1088/1742-6596/1205/1/012017>, 2019.
- 865 Golaz, J.-C., Caldwell, P. M., Van Roekel, L. P., Petersen, M. R., et al.: The DOE E3SM Coupled Model Version 1: Overview and Evaluation at Standard Resolution, *J ADV MODEL EARTH SY*, 11, 2089–2129, <https://doi.org/10.1029/2018MS001603>, 2019.
- Jiang, J., Lin, P., Wang, J., Liu, H., Chi, X., Hao, H., Wang, Y., Wang, W., and Zhang, L.: Porting LASG/ IAP Climate System Ocean Model to Gpus Using OpenAcc, *IEEE ACCESS*, 7, 154 490–154 501, <https://doi.org/10.1109/ACCESS.2019.2932443>, 2019.
- Julia Development Team: Introduction to CUDA, <https://cuda.juliagpu.org/stable/tutorials/introduction/#A-simple-example-on-the-CPU>, 870 2022, visited on 2022-12-13.
- Julia Development Team: Eval of Julia code, <https://docs.julialang.org/en/v1/devdocs/eval/#>, 2016, visited on 2022-12-13, 2016.
- Klößner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A.: PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation, *PARALLEL COMPUT*, 38, 157–174, <https://doi.org/10.1016/j.parco.2011.09.001>, 2012.
- Klöwer, M., Hatfield, S., Croci, M., Düben, P. D., and Palmer, T. N.: Fluid Simulations Accelerated With 16 Bits: Approaching 4x Speedup 875 on A64FX by Squeezing ShallowWaters.jl Into Float16, *J ADV MODEL EARTH SY*, 14, <https://doi.org/10.1029/2021MS002684>, 2022.
- Koldunov, N. V., Aizinger, V., Rakowsky, N., Scholz, P., Sidorenko, D., Danilov, S., and Jung, T.: Scalability and some optimization of the Finite-volumE Sea ice–Ocean Model, Version 2.0 (FESOM2), *GEOSCI MODEL DEV*, 12, 3991–4012, <https://doi.org/10.5194/gmd-12-3991-2019>, 2019.
- Lam, S. K., Pitrou, A., and Seibert, S.: Numba: A llvm-based python jit compiler, in: Proceedings of the Second Workshop on the LLVM 880 Compiler Infrastructure in HPC, pp. 1–6, 2015.
- Lin, W.-C. and McIntosh-Smith, S.: Comparing Julia to Performance Portable Parallel Programming Models for HPC, in: 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 94–105, IEEE, St. Louis, MO, USA, <https://doi.org/10.1109/PMBS54543.2021.00016>, 2021.

- Mielikainen, J., Huang, B., Huang, H.-L. A., and Goldberg, M. D.: Improved GPU/CUDA Based Parallel Weather and
885 Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics, *IEEE J SEL TOP APPL*, 5, 1256–1265,
<https://doi.org/10.1109/JSTARS.2012.2188780>, 2012.
- NERSC: Perlmutter architecture specification, <https://docs.nersc.gov/systems/perlmutter/architecture/>, 2023, visited on 2023-06-15, 2023.
- Norman, M., Lyngaas, I., Bagusetty, A., and Berrill, M.: Portable C++ Code that can Look and Feel Like Fortran Code with Yet Another
Kernel Launcher (YAKL), *International Journal of Parallel Programming*, <https://doi.org/10.1007/s10766-022-00739-0>, 2022.
- 890 Perkel, J. M.: Julia: come for the syntax, stay for the speed, *NATURE*, 572, 141–142, <https://doi.org/10.1038/d41586-019-02310-3>, 2019.
- Petersen, M. R., Jacobsen, D. W., Ringler, T. D., Hecht, M. W., and Maltrud, M. E.: Evaluation of the Arbitrary Lagrangian–Eulerian Vertical
Coordinate Method in the MPAS-Ocean Model, *OCEAN MODEL*, 86, 93–113, <https://doi.org/10.1016/j.ocemod.2014.12.004>, 2015.
- Petersen, M. R., Asay-Davis, X. S., Berres, A. S., Chen, Q., Feige, N., Hoffman, M. J., Jacobsen, D. W., Jones, P. W., Maltrud, M. E., Price,
S. F., Ringler, T. D., Streletz, G. J., Turner, A. K., Van Roekel, L. P., Veneziani, M., Wolfe, J. D., Wolfram, P. J., and Woodring, J. L.: An
895 Evaluation of the Ocean and Sea Ice Climate of E3SM Using MPAS and Interannual CORE-II Forcing, *J ADV MODEL EARTH SY*, 11,
1438–1458, <https://doi.org/10.1029/2018MS001373>, 2019.
- Petersen, M. R., Bishnu, S., and Strauss, R. R.: MPAS-Ocean Shallow Water Performance Test Case,
<https://doi.org/10.5281/zenodo.7439134>, 2022.
- Ramadhan, A., Wagner, G. L., Hill, C., Campin, J.-M., Churavy, V., Besard, T., Souza, A., Edelman, A., Ferrari, R., and
900 Marshall, J.: Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs, *J. OPEN SOURCE SOFTW.*, 5, 2018,
<https://doi.org/10.21105/joss.02018>, 2020.
- Ringler, T. D., Thuburn, J., Klemp, J. B., and Skamarock, W. C.: A unified approach to energy conservation and potential vorticity dynamics
for arbitrarily-structured C-grids, *J COMPUT PHYS*, 229, 3065–3090, 2010.
- Ringler, T. D., Petersen, M. R., Higdon, R. L., Jacobsen, D., Jones, P. W., and Maltrud, M.: A multi-resolution approach to global ocean
905 modeling, *OCEAN MODEL*, 69, 211–232, 2013.
- Shchepetkin, A. F. and McWilliams, J. C.: The regional oceanic modeling system (ROMS): a split-explicit, free-surface,
topography-following-coordinate oceanic model, *OCEAN MODEL*, 9, 347–404, 2005.
- Srinath, A.: Accelerating Python on GPUs with nvc++ and Cython, [https://developer.nvidia.com/blog/
accelerating-python-on-gpus-with-nvc-and-cython/](https://developer.nvidia.com/blog/accelerating-python-on-gpus-with-nvc-and-cython/), 2020, visited on 2022-12-13, 2022.
- 910 Strauss, R. R.: Julia Layered Shallow Water Model on Various Hardwares, <https://doi.org/10.5281/zenodo.7493064>, 2023.
- Thuburn, J., Ringler, T. D., Skamarock, W. C., and Klemp, J. B.: Numerical representation of geostrophic modes on arbitrarily structured
C-grids, *J COMPUT PHYS*, 228, 8321–8335, 2009.
- Trott, C. R., Lebrun-Grandié, D., et al.: Kokkos 3: Programming Model Extensions for the Exascale Era, *IEEE T PARALL DISTR*, 33,
805–817, <https://doi.org/10.1109/TPDS.2021.3097283>, 2022.
- 915 Xu, S., Huang, X., Zhang, Y., Hu, Y., and Yang, G.: A customized GPU acceleration of the princeton ocean model,
in: 2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, pp. 192–193,
<https://doi.org/10.1109/ASAP.2014.6868661>, 2014.
- Xu, S., Huang, X., Oey, L.-Y., Xu, F., Fu, H., Zhang, Y., and Yang, G.: POM.gpu-v1.0: a GPU-based Princeton Ocean Model, *GEOSCI
MODEL DEV*, 8, 2815–2827, <https://doi.org/10.5194/gmd-8-2815-2015>, 2015.
- 920 Ye, Y., Song, Z., Zhou, S., Liu, Y., Shu, Q., Wang, B., Liu, W., Qiao, F., and Wang, L.: swNEMO_v4.0: an ocean model based on NEMO4 for
the new-generation Sunway supercomputer, *GEOSCI MODEL DEV*, 15, 5739–5756, <https://doi.org/10.5194/gmd-15-5739-2022>, 2022.

Zhao, X.-d., Liang, S.-x., Sun, Z.-c., Zhao, X.-z., Sun, J.-w., and Liu, Z.-b.: A GPU accelerated finite volume coastal ocean model, J HYDRODYN, Ser. B, 29, 679–690, [https://doi.org/10.1016/S1001-6058\(16\)60780-1](https://doi.org/10.1016/S1001-6058(16)60780-1), 2017.