JAMES | Journal of Advances in Modeling Earth Systems®

RESEARCH ARTICLE
10.1029/2021MS002717

Key Points:

- We present a pure Python ocean model that leverages the JAX accelerator library to achieve competitive performance on CPU and GPU clusters
- On CPU, performance is similar to Fortran. On GPU, each device can replace hundreds of CPU cores, with at least 3 times less energy usage
- To show how GPUs can be used in practice, we integrate an eddying 0.1° global ocean setup on a single cloud compute instance with 16 GPUs

Correspondence to:
D. Häfner,
dion.haefner@nbi.ku.dk

Citation:
Häfner, D., Nuterman, R., & Jochum, M. (2021). Fast, cheap, and turbulent—Global ocean modeling with GPU acceleration in Python. *Journal of Advances in Modeling Earth Systems*, *13*, e2021MS002717. https://doi.org/10.1029/2021MS002717

Received 23 JUL 2021
Accepted 29 NOV 2021

Author Contributions:
Conceptualization: Dion Häfner, Markus Jochum
Data curation: Dion Häfner
Formal analysis: Dion Häfner
Funding acquisition: Markus Jochum
Investigation: Dion Häfner, Roman Nuterman
Methodology: Dion Häfner
Project Administration: Markus Jochum
Resources: Roman Nuterman
Software: Dion Häfner, Roman Nuterman
Supervision: Markus Jochum
Validation: Dion Häfner
Visualization: Dion Häfner
Writing – original draft: Dion Häfner

Fast, Cheap, and Turbulent—Global Ocean Modeling With GPU Acceleration in Python

Dion Häfner[1], **Roman Nuterman**[1], **and Markus Jochum**[1]

[1]Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark

Abstract Even to this date, most earth system models are coded in Fortran, especially those used at the largest compute scales. Our ocean model Veros takes a different approach: it is implemented using the high-level programming language Python. Besides numerous usability advantages, this allows us to leverage modern high-performance frameworks that emerged in tandem with the machine learning boom. By interfacing with the JAX library, Veros is able to run high-performance simulations on both central processing units (CPU) and graphical processing unit (GPU) through the same code base, with full support for distributed architectures. On CPU, Veros is able to match the performance of a Fortran reference, both on a single process and on hundreds of CPU cores. On GPU, we find that each device can replace dozens to hundreds of CPU cores, at a fraction of the energy consumption. We demonstrate the viability of using GPUs for earth system modeling by integrating a global 0.1° eddy-resolving setup in single precision, where we achieve 1.3 model years per day on a single compute instance with 16 GPUs, comparable to over 2,000 Fortran processes.

Plain Language Summary Climate models are an invaluable tool to understand the earth system and inform policies to combat climate change. Climate simulations often run for thousands of model years, which consumes a substantial amount of resources—both in terms of electricity and time (months) spent by researchers waiting for results. On the other hand, climate models are highly complicated software projects that require countless man-hours from scientists and engineers to build. Ocean models are one of the main components of a climate model. Here, we present a new type of ocean model that combines strong performance and ease of use and development. We show that by using graphical processing units, we can perform realistic ocean simulations at a high speed, and with a fraction of the energy usage.

1. Introduction

Virtually all of the most commonly used earth system models, for example those used for the CMIP climate model intercomparison studies (Meehl et al., 2000), are implemented in the Fortran programming language. Their ocean components, such as POP2 (Danabasoglu et al., 2012), NEMO (Madec et al., 2017), or MICOM (Bleck et al., 1995), are no exception.

Our ocean model Veros follows a different route. Veros is implemented in the high-level programming language Python, which has several key usability advantages over Fortran (see Häfner et al., 2018, for a discussion), and is arguably the most popular programming language in science today. This allows even undergraduates to perform non-trivial numerical experiments.

However, model performance has been a long-standing issue. The lack of a built-in optimizing compiler makes it that vectorized Python code is typically about 3–5 times slower than equivalent Fortran code. This may be fine for idealized experiments, but is unacceptable for large setups that occupy thousands of central processing units (CPU) cores for months.

Recently, we succeeded to close most of this performance gap by exploiting the just-in-time compiler of the JAX library (Bradbury et al., 2018). This results in competitive CPU performance, as we will demonstrate in this article. But using JAX has another advantage: It allows us to use graphical processing units (GPUs) without any additional code.

With the advent of machine learning in general and deep learning in particular, GPUs have experienced a renaissance. Model training is vastly more efficient on massively parallel hardware, which has led to a feedback loop between supply and demand that has amplified their capabilities. Today, GPUs are the industry standard devices to train artificial neural networks. This trend has also impacted the design of modern compute facilities; for

Writing – review & editing: Dion Häfner, Roman Nuterman, Markus Jochum

example, out of the 8 upcoming supercomputers in the EuroHPC Joint Undertaking, 7 are going to provide GPU resources, typically making up around 10% of the total compute power (see EuroHPC, 2021). These resources would be unusable with traditional Fortran models without considerable additional effort, such as a complete re-implementation using CUDA Fortran or by using a framework like OpenACC (Wienke et al., 2012), which requires compiler directives for every loop (see also Norman et al., 2015).

Here, we confirm that GPUs are a viable alternative for earth system modeling due to their strong performance and high energy efficiency. Perhaps more importantly, we also show that operational simulations on GPU are possible *today*, since we now have wide access to mature hardware and a modern software stack that allows us to exploit this hardware without maintaining a separate GPU implementation.

This combination of competitive CPU performance and seamless transition to GPUs puts Python on the map as an attractive alternative to traditional, low-level programming languages, even from a performance perspective.

In this article, we outline the computational task behind ocean models through a brief introduction to their underlying equations (Section 2). We introduce Veros and JAX, and show some of the design decisions we made to obtain a well-performing implementation (Section 3). We present extensive benchmarks of Veros against a Fortran backend, and study scaling behavior and energy usage on CPU and GPU (Section 4). Finally, we describe a case study where we integrate a realistic 0.1° global resolution setup using a single cloud instance with 16 GPUs (Section 5, Figure 1).

2. Model Equations and Discretization

To illustrate the computational challenge behind ocean modeling, we briefly introduce the equations that are solved by most ocean general circulation models.

Ocean dynamics are governed by the *primitive equations*, a combination of momentum balance, mass continuity, and equation of state. They are typically solved in form of the semi-compressible, hydrostatic Boussinesq

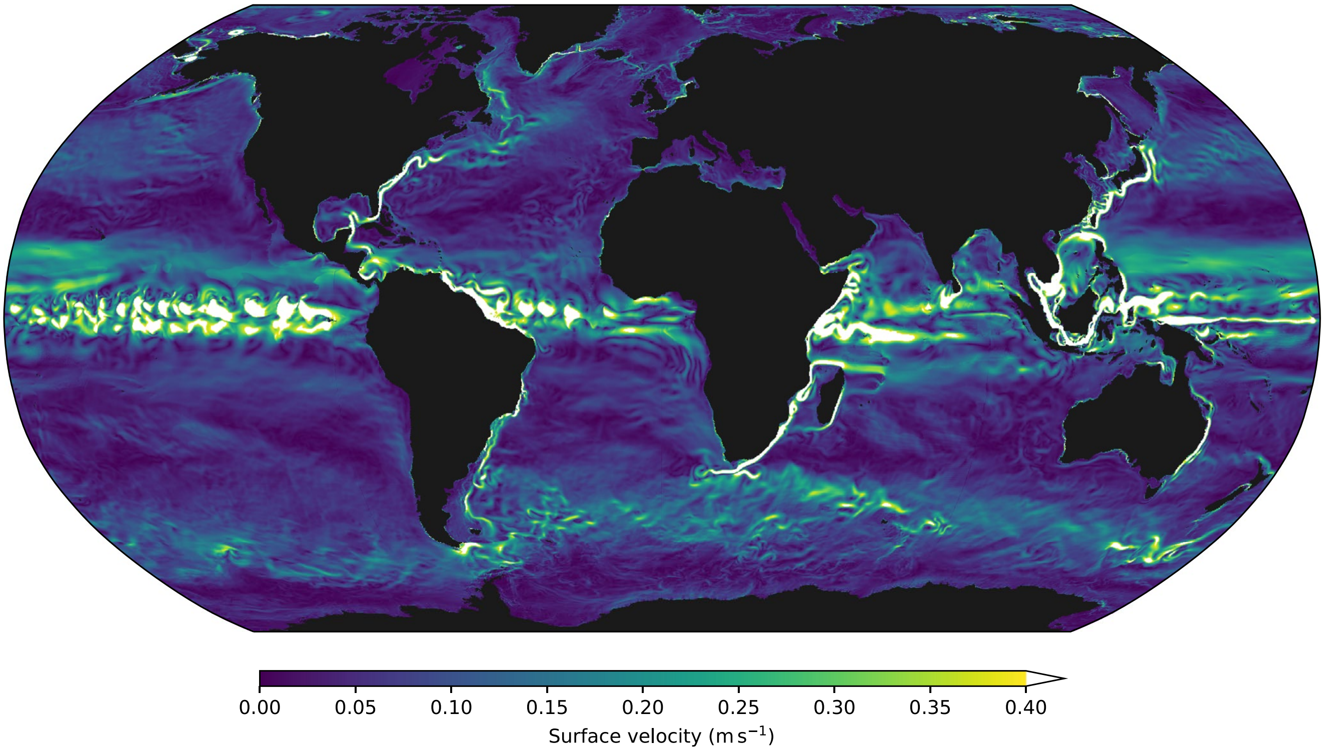

Figure 1. Snapshot of the ocean surface velocity after 1 year of integration of a global 0.1° Veros setup. Simulated in about 24 hr on a compute instance with 16 NVIDIA A100 GPUs in single precision. All model code is written in Python.

equations (see e.g., Adcroft et al., 2021), which suppress acoustic modes and neglect vertical momentum. In (x, y, z) coordinates and time t they read:

$$\frac{\mathrm{D}\vec{v}_h}{\mathrm{D}t} + f\hat{k} \times \vec{v}_h + \frac{1}{\rho_0}\nabla_h p' = \vec{\mathcal{F}} \tag{1}$$

$$\nabla_h \cdot \vec{v}_h + \frac{\partial w}{\partial z} = 0 \tag{2}$$

$$\frac{\partial p'}{\partial z} = -g\rho' \tag{3}$$

$$\rho' = \rho(\theta, S, p_0(z)) - \rho_0 \tag{4}$$

$$\frac{\mathrm{D}\theta}{\mathrm{D}t} = \mathcal{Q}_\theta \tag{5}$$

$$\frac{\mathrm{D}S}{\mathrm{D}t} = \mathcal{Q}_S \tag{6}$$

with horizontal velocity $\vec{v}_h$, vertical velocity w, vertical unit vector $\hat{k}$, in-situ density ρ, background density ρ_0, dynamic pressure p', hydrostatic pressure p_0, gravitational acceleration g, Coriolis frequency f, potential temperature θ, and salinity S. ∇_h is the horizontal Nabla operator, and $\mathrm{D}/\mathrm{D}t = \partial/\partial t + \vec{v} \cdot \nabla$ is the total derivative. On the right hand side, $\mathcal{F}$, $\mathcal{Q}_\theta$, and $\mathcal{Q}_S$ denote forcing and dissipation of momentum, potential temperature, and salinity, respectively.

These equations are discretized using finite differences, which leads to a set of discrete equations that can be stepped forward in time explicitly. Fast barotropic modes of the ocean surface—which would require unacceptably small time steps to integrate explicitly—are usually treated separately through an implicit formulation, which requires a linear system to be solved in each time step.

Implementing a full dynamical core requires at least several thousand lines of Fortran code, while mature ocean models like POP2 (Danabasoglu et al., 2012) can reach close to 100,000 lines of Fortran code.

3. High-Performance Computing in Python

Among scientists, Python has long held a reputation of being slow. This is true to some extent, as the common NumPy library (Harris et al., 2020) is often several times slower than equivalent low-level code, and there is no unique, established way to accelerate numerical Python code. The following sections introduce the steps we took to obtain the near-Fortran performance described in the benchmarks in Section 4, and give a brief introduction to Veros.

3.1. Veros

Here, we outline the core capabilities of Veros, and present some recent developments that have not been described previously (see also Häfner et al., 2018, for a more general description of Veros).

The dynamical core of Veros is a direct translation of the Fortran backend of the ocean model PyOM2 (Eden, 2016; Eden & Olbers, 2014). Veros features a regular, fully staggered computational grid (Arakawa C-grid, see Arakawa & Lamb, 1977); closures for mesoscale eddies (Eden & Greatbatch, 2008; Gent et al., 1995), turbulence (Gaspar et al., 1990), isoneutral mixing of tracers (S. M. Griffies, 1998), and internal waves (Olbers & Eden, 2013); and various parameterizations for friction, diffusion, advection, and equation of state (e.g., TEOS-10, McDougall & Barker, 2011).

In a nutshell, Veros is a full-fledged primitive-equation model, with the current limitations of a regular lat-lon grid (which implies that the Arctic cannot be modeled due to a vanishing grid size at the poles) and missing sea-ice components. Veros supports both highly idealized setups and realistic configurations at any resolution. Just as most Fortran models, Veros writes output in NetCDF4 format, which ensures that its output can be analyzed with all standard post-processing tools.

Since the initial publication of Veros (Häfner et al., 2018), we have taken some additional steps to optimize model performance:

- **JAX Backend:** Up until recently, Veros relied on the accelerator framework Bohrium (Kristensen et al., 2013) as a high-performance backend. Switching to JAX yielded substantial speedups, especially for small problems, on many CPU cores, and on GPUs (see Section 3.2 for more information about JAX and Section 4 for benchmarks). Specifically, we find that JAX is typically between 2 and 10 times faster than Bohrium on GPU, and is up to 4 times faster on many CPU cores due to reduced overhead (see also Häfner, 2021).
- **Distributed Computation:** Perhaps the most impactful change is that Veros is now fully parallelized via Message Passing Interface (MPI; Gropp et al., 1999) primitives. This means that Veros can be used in distributed contexts on any number of processes, and across any number of independent computational nodes. This is possible through mpi4py (Dalcin et al., 2011) and mpi4jax (Häfner & Vicentini, 2021), which allow NumPy and JAX data to be communicated without intermediate copies (i.e., data can be read and written directly via network, even from/to GPU memory).
- **Distributed Barotropic Solvers:** To compute the barotropic external mode, a 2-dimensional Poisson equation has to be solved in every model step, which becomes a bottleneck at high processor counts. Veros now supports fully distributed linear solvers via PETSc and petsc4py (Balay et al., 1997). Specifically, PETSc's GAMG preconditioner applies an Algebraic Multigrid method (AMG, Ruge & Stüben, 1987) and works on both CPU and GPU, which makes it so the same solver can be used on any architecture.
- **Hand-Written Tridiagonal Solvers:** Veros uses some implicit parameterizations, for example, for vertical friction. These require a tridiagonal linear system to be solved for each horizontal grid cell, which cannot be expressed in NumPy/JAX array operations. To make this more efficient, Veros supplies a hand-written implementation of the Thomas algorithm (see e.g., Press et al., 1989) for each backend, written in Cython for CPU and CUDA for GPU (contributed as part of a student project; see Grenzdörffer, 2021). These special implementations are usable from JAX without overhead through its custom call mechanism. This is the only non-Python code in the Veros repository, and entirely optional. Using these extensions increases general model performance by about 10%.

Veros supports both NumPy and JAX as numerical backends. Additionally, since the dynamical core of Veros is a one-to-one translation of that of PyOM2, its original Fortran components can be used instead of the Veros core routines. In other words, we can use PyOM2 as a Fortran backend to Veros, which allows us to conduct a direct comparison between Python and Fortran performance (as done in Section 4).

Veros enforces consistent results between all computational backends through an automated testing suite that compares the output of each model subroutine and entire model setups to the Fortran reference. We find that both NumPy and JAX are consistent with the Fortran reference to a high degree (absolute error $<10^{-8}$, relative error $<10^{-6}$ for all variables after 1 model step).

3.2. JAX

Massively parallel devices—such as GPUs and tensor processing units (TPUs)—are notoriously difficult to program with low-level code. For example, writing CUDA kernels by hand requires in-depth awareness of different types of GPU memory, internal memory layouts, asynchronous execution, and the optimal number of threads per block for the problem at hand (see e.g., Sanders & Kandrot, 2010).

This calls for a more accessible way to run code on these devices. Therefore, the rising popularity of GPUs has also lead to the birth of accelerator frameworks such as TensorFlow (Abadi et al., 2016), Torch (Collobert et al., 2002), Theano (Bergstra et al., 2010), and JAX (Frostig et al., 2018), which perform native code generation for various hardware targets based on vectorized array operations.

Since all performance-heavy operations are executed within the accelerator framework, performance is virtually independent of the programming language used to express the numerical routines. This makes them an ideal fit for high-level languages like Python, and most numerical Python code can only achieve competitive performance by leveraging such an accelerator framework.

Fortran

```fortran
subroutine get_tke_surface_correction
  ! explicit loop in Fortran, modify state in-place
  use main_module
  tke_surf_corr = 0.0
  do j=js_pe,je_pe
    do i=is_pe,ie_pe
      if (tke(i,j,nz,taup1) < 0.0) then
        tke_surf_corr(i,j) = -tke(i,j,nz,taup1)*(0.5*dzw(nz)) / dt_tke
        tke(i,j,nz,taup1) = 0.0
      endif
    enddo
  enddo
```

NumPy

```python
import numpy as np

def get_tke_surface_correction(tke, dzw, taup1, dt_tke):
  # use a boolean mask instead of `if` statement
  # and return changed arrays
  tke_surf_corr = np.zeros(tke.shape[:2])
  mask = tke[2:-2, 2:-2, -1, taup1] < 0.0
  tke_surf_corr[2:-2, 2:-2] = (
    -tke[2:-2, 2:-2, -1, taup1] * 0.5 * dzw[-1] / dt_tke
    * mask
  )
  tke[2:-2, 2:-2, -1, taup1] *= mask
  return tke_surf_corr, tke
```

JAX

```python
import jax, jax.numpy as jnp

@jax.jit  # mark for JIT compilation
def get_tke_surface_correction(tke, dzw, taup1, dt_tke):
  # same approach for JAX
  tke_surf_corr = jnp.zeros(tke.shape[:2])
  mask = tke[2:-2, 2:-2, -1, taup1] < 0.0
  tke_surf_corr = tke_surf_corr.at[2:-2, 2:-2].set(
    -tke[2:-2, 2:-2, -1, taup1] * 0.5 * dzw[-1] / dt_tke
    * mask
  )
  tke = tke.at[2:-2, 2:-2, -1, taup1].multiply(mask)
  return tke_surf_corr, tke
```

Figure 2. From Fortran to NumPy to JAX. The same code snippet (from the turbulent kinetic energy closure) for all 3 backends. When going from Fortran to Python, explicit loops are replaced with array operations. Changes from NumPy to JAX are minimal (an additional JIT decorator and a different syntax for slicing). This results in tremendous speedups on CPU and GPU (see benchmarks).

In our case, we performed extensive benchmarks and comparisons between the most popular accelerator frameworks (available on GitHub, see Häfner, 2021). Given its ease of use and consistently strong performance, we decided to use JAX as the computational backend for Veros. JAX provides a just-in-time (JIT) compiler that traces the given Python code, extracts the performed array operations, and executes them on the target device. Array operations are specified using the NumPy API, which makes it easy to port existing Python code that uses NumPy to JAX.

Internally, JAX generates a so-called HLO IR (high level operations intermediate representation) that is then passed to the compiler XLA, which performs the actual translation to machine code. XLA originated as TensorFlow's TPU compiler and has since been extended to also target CPU and GPU devices. XLA applies a number of transformations to the original code, including loop fusion, elimination of redundant computations, and memory layout optimizations. Additionally, all JAX kernels are executed asynchronously, which further helps to hide Python overhead and host-device communication latencies.

JAX is gaining popularity in science, and has been successfully applied to molecular dynamics (Schoenholz & Cubuk, 2020), probabilistic programming (Phan et al., 2019), rigid body simulation (Freeman et al., 2021), and many-body quantum systems (Carleo et al., 2019).

For Veros, implementing a JAX backend has been a negligible effort compared to traditional approaches to GPU programming, and the result is much more concise and maintainable, since the same code runs on all hardware targets (see Figure 2 for an example). In fact, the biggest changes required to support JAX in Veros were related to its data model. Because compiled JAX functions must be pure (without side effects), we changed all core routines to explicitly return their results instead of modifying the model state in-place.

To the user, the resulting code behaves for the most part like any other Python code. This includes the correct propagation of exceptions and seamless interplay with other Python libraries (such as Matplotlib for plotting) outside of JIT compiled functions. Debugging within compiled functions is more difficult, but since Veros also

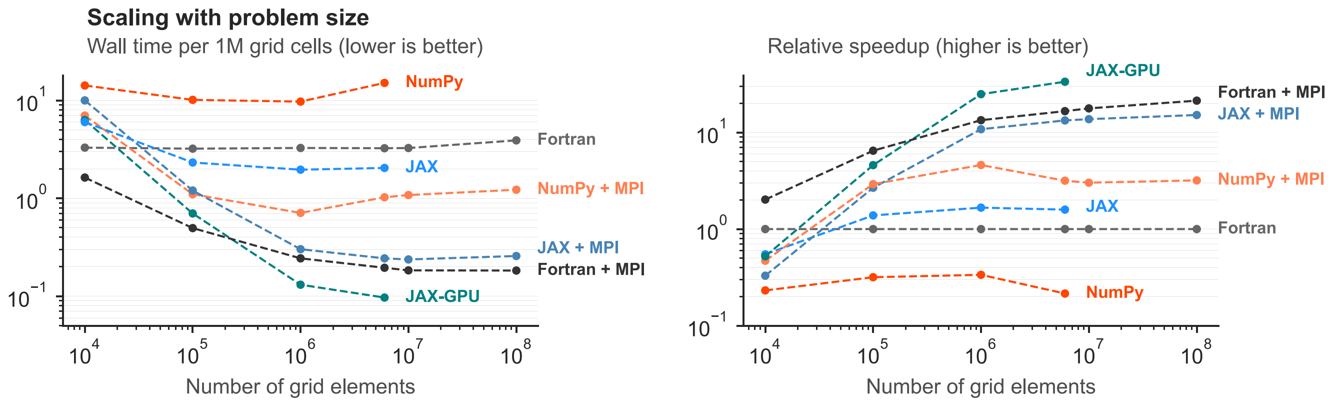

Figure 3. With JAX, Veros performance is close to native Fortran code throughout a wide range of problem sizes (number of 3D grid cells), and on both on 1 and 24 processes. Left: wall time per iteration for each backend. Right: speedup relative to single-process Fortran. Benchmarks executed on a single compute node (Table A1, architecture I).

supports NumPy as a computational backend, all initial debugging can be done in NumPy before switching to JAX for performance.

4. Benchmarks

Here, we present full model benchmarks using different hardware and software stacks. One goal is to compare the performance of Veros to PyOM2's Fortran backend; this allows us to quantify the impact of switching from Fortran to Python, since both implement the same numerical routines (see Section 3.1). Another goal is to benchmark the scaling behavior of CPUs and GPUs when increasing domain sizes and number of devices.

The setup we use here is part of the Veros repository (ACC benchmark) and consists of an idealized rectangular ocean basin with an open channel representing the Southern Ocean. All optional components (isoneutral mixing, turbulence, mesoscale eddy, and internal wave closures) are activated. We measure the average wall time per model iteration, excluding I/O (which incurs the same cost for all backends). Since this is an end-to-end model benchmark its results are representative for real-world scenarios, with the possible exception of the barotropic solver (the underlying linear system is easier to solve for an idealized geometry).

Because JAX relies on just-in-time compilation, the first model iteration has a constant additional overhead of about 40 s. Therefore, we discard the first 2 iterations, and report the mean wall time of the following 10 iterations.

The following sections present the results for varying problem size, varying number of CPUs, and varying number of GPUs. Finally, we quantify the energy savings when using GPUs instead of CPUs.

4.1. Scaling With Problem Size

For this benchmark, we keep the number of processes fixed at 24 for MPI-enabled backends; all other backends run on a single process. It is executed on a compute node with 24 CPU cores and a NVIDIA Tesla P100 GPU (architecture I in Table A1). Here, and in all following benchmarks, “number of grid elements” refers to the total (3-dimensional) number of grid points.

The results are shown in Figure 3. We find that, when using JAX with MPI, Veros is at most a factor of 1.4 slower than Fortran with MPI for intermediate to large setups (more than 10^6 grid cells). Notably, JAX is about 1.6 times *faster* than Fortran on a single process, which is because JAX uses some thread-based parallelism internally. NumPy is consistently 3 to 5 times slower than Fortran, which illustrates the substantial payoff from using JAX for acceleration.

On a single NVIDIA P100 GPU (JAX-GPU), Veros is about 2 times faster than Fortran on 24 CPU cores when the GPU is fully saturated (i.e., for large problems). In this configuration, this seems to be the case for domains exceeding 10^6 elements (corresponding to about 2° global resolution). In contrast, JAX achieves its full performance on CPU for domains exceeding 10^5 elements (i.e., all but the smallest idealized setups).

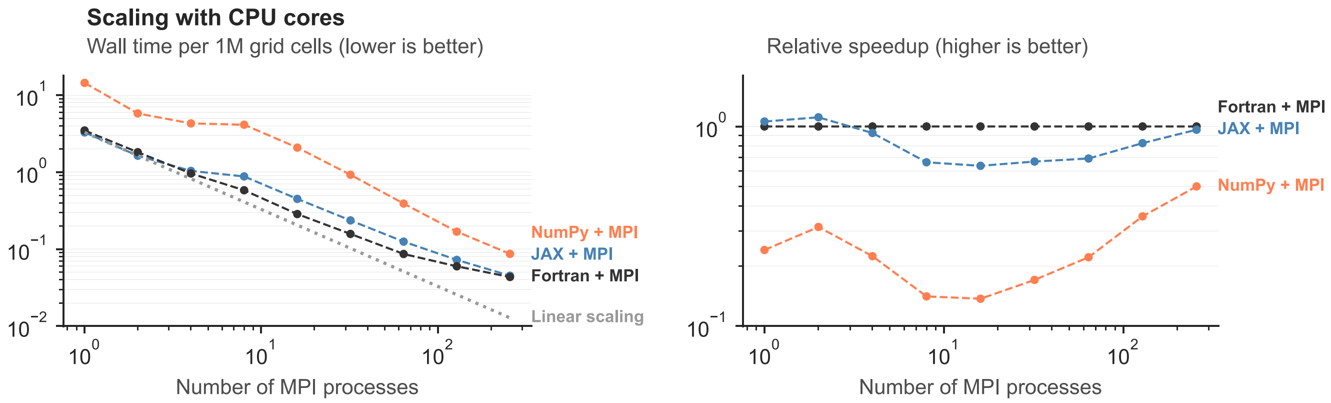

Figure 4. With JAX, Veros reaches Fortran speed for both few and many CPU cores, with a 40% performance gap for an intermediate number of cores. Left: wall time per iteration for each backend. Right: speedup relative to multi-process Fortran. Benchmarks executed on a CPU cluster with 32 cores per node (Table A1, architecture II).

4.2. Scaling With Number of CPUs

After the initial benchmark with varying problem size, we now investigate the scaling behavior in fully distributed, multi-node contexts. For this, we hold the problem size constant at 6×10^6 grid cells (approximately the size of a global 1° × 1° setup), but vary the number of tasks committed to a CPU cluster (architecture II in Table A1) using a maximum of 8 nodes with 32 CPU cores each (256 CPU cores total).

We find that, when using JAX, Veros performance lies again within a factor of 1.4 of the Fortran implementation (Figure 4). For a low number of processes (≤ 4) *and* a high number of processes (256) we even find the same performance as Fortran, with a gap for an intermediate number of cores. For NumPy, this gap is even more pronounced, with a plateau between 2 and 8 processes, before scaling continues.

We interpret this as evidence that Veros performance is memory bound, which would manifest only for an intermediate number of processes. For a low process count, the node's memory bandwidth is large enough to keep up with demand. For a high node count, memory bandwidth matters less as communication overhead and the barotropic solver start to dominate the overall runtime. Interestingly, this also leads to the fact that NumPy is only a factor of 2 slower than Fortran on 256 processes (from a factor of 7 at 16 processes).

This memory bandwidth bottleneck is more pronounced for NumPy because NumPy manifests every temporary result in memory. But even JAX does not manage to fuse *all* array operations, which contributes to the observed gap.

4.3. Scaling With Number of GPUs

To investigate the scaling with number of GPUs, we conduct experiments using both strong scaling (constant problem size) and weak scaling (constant problem size *per device*). All multi-GPU benchmarks are executed on a a2-megagpu-16g instance on Google Cloud with 16 NVIDIA A100 GPUs (architecture III in Table A1).

The results show that saturating each GPU is critical for a good scaling behavior (Figure 5). During strong scaling, with a constant global problem size of 2×10^7 (filling a single GPU's 40 GB of memory), 16 GPUs are only about 3.4 times faster than 1 GPU. However, when each GPU is fully saturated (weak scaling), Veros/JAX achieves almost perfect scaling, with 16 GPUs being only about 20% slower than 1 GPU in terms of total solution time (for a 16 times larger domain).

In an additional CPU benchmark with the same problem size of 2×10^7 (included in raw benchmark results, see data availability section), we find that 256 CPU cores running Fortran reach a performance of about 3.2×10^{-2} s per 1M grid cells (as opposed to 2.5×10^{-2} s for a single GPU). This shows that each high-end GPU can replace well over 200 CPU cores running a traditional ocean model, both due to a higher computational efficiency and reduced communication overhead (since fewer devices participate in the solution).

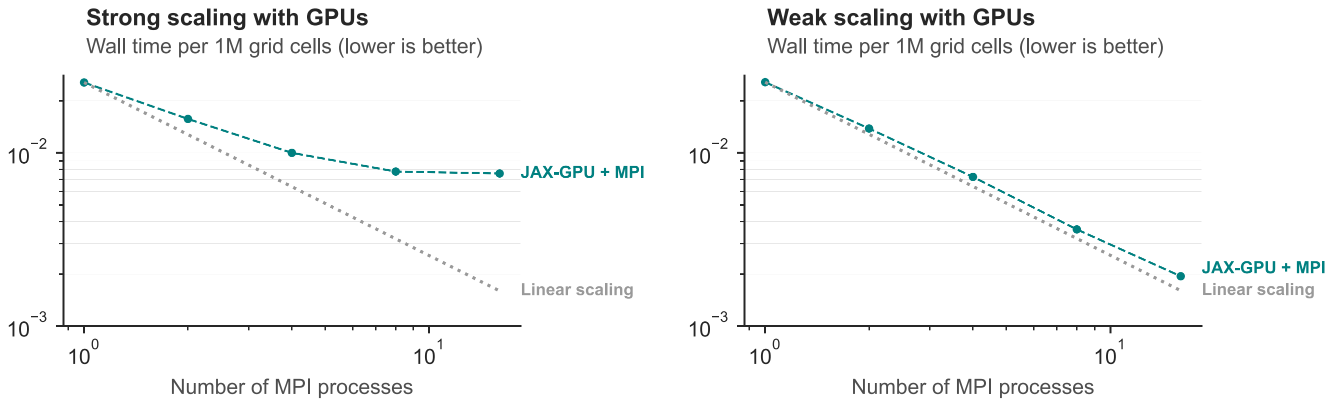

Figure 5. Veros scales well in multi-GPU contexts, as long as the problems are big enough to saturate each GPU. Left: Strong scaling with constant problem size of 2×10^7 elements. Right: Weak scaling with problem size of 2×10^7 elements *per GPU*. Benchmarks executed on a Google Cloud a2-megagpu-16g instance (Table A1, architecture III).

4.4. Energy Consumption

Raw performance is not the only important metric in high-performance computing. A low energy consumption is key to reduce computing costs and minimize carbon footprints. Because every GPU can ideally replace several computational nodes they provide a promising way to achieve this. This is especially true in multi-GPU contexts, since several GPUs (in practice up to 16) can be attached to the same node, reducing overhead.

However, total energy consumption is difficult to estimate and depends on many factors. Here, we provide one data point by measuring the total energy consumption of a CPU and GPU simulation, respectively. For this we integrate a global 1° × 1° Veros setup (containing about 6×10^6 grid cells) for 2,000 iterations and record the power usage (as reported by the node's remote access controller) every 3 s. This only measures the energy usage by the node itself; in a real-world setting, there are external systems like network and storage that contribute to the total energy footprint of a simulation that we cannot estimate here.

The results indicate that, in this case, using a GPU is 2.6 times more efficient than the node's CPU cores (Table 1). This factor is increased even further to 3.2 when using 2 GPUs. This is despite the poor strong scaling behavior on GPU (see Section 4.3) which causes each device to be underutilized (this is also why we only see a 50 W increase when adding a second GPU).

We are optimistic that a setup with 16 next-generation GPUs attached to 1 node (such as the architecture in Section 4.3) reaches even higher energy efficiencies, especially if all devices are saturated, as in this configuration a single GPU node is able to replace several CPU nodes.

In terms of purchasing cost, we find that CPU and GPU systems of the same performance are comparable. Based on our benchmarks in Section 4.3, we estimate that 1 NVIDIA A100 GPU (retail price of 24,959 $) has the same performance as 3 to 4 high-end CPUs, such as the AMD EPYC 7742 with 64 cores each (as used e.g., in the Cambridge 1 supercomputer; total retail price of 20,850$ to 27,800 $). This implies that these energy savings could also translate into tangible cost savings. However, further studies on real-world energy consumption and performance are necessary to determine how large this difference turns out to be in practice.

Table 1
Simulations Can be Substantially More Energy Efficient on GPU

	Time to solution	Mean power usage	Total energy usage
24 CPU cores	3297.17 s	354.47 W	0.32 kWh
1 P100 GPU	1386.74 s	316.34 W	0.12 kWh
2 P100 GPUs	945.55 s	364.88 W	0.10 kWh

Note. Shown is the power consumption of a single GPU node (architecture I, see Table A1) when running a global 1° Veros setup for 2,000 iterations, depending on the used hardware.

5. Eddying 0.1° Setup

To show that GPUs are already viable for research applications, we use a a2-megagpu-g16 Google Cloud instance (the same as in Section 4.3) with 16 A100 GPUs to integrate a global eddy-resolving setup.

For this, we use a model grid with constant 0.1° spacing and 3600 × 1600 × 60 grid cells (3.50×10^8 total elements). This results in a nominal resolution of 10 km (as defined in the CMIP6 specification), which is sufficient to resolve mesoscale eddies.

Because eddies are resolved explicitly, we disable the mesoscale eddy closure and use a reduced, constant thickness diffusivity of 100 $\mathrm{m}^2\,\mathrm{s}^{-1}$. We also disable the internal wave parameterization, and leave the turbulence closure active. To enable a more realistic representation of mesoscale eddies, we use biharmonic friction (Griffies & Hallberg, 2000) instead of the default Laplacian lateral friction, with a biharmonic viscosity of 10^{11} $\mathrm{m}^4/\mathrm{s}^2$ that decays with latitude ϕ as $\cos^3\phi$. Initially, we use a time step of 120 s, which is increased to 150 s after 90 model days and to 180 s after 180 model days (when initialization shocks have settled down).

One important limitation when running large-scale models on GPU is available memory. GPUs come with a fixed, non-configurable amount of memory attached, which makes them less flexible to use. Fortunately, GPU memory has now grown to the sizes that we need to store even large-scale setups. Each A100 card used here has 40 GB of memory, which combined (640 GB) is just enough to hold the entire model state and diagnostics in double precision.

In this experiment, we also showcase Veros' capability to use single precision. Thanks to a fully dynamic type system in Python and JAX, we can change all data types with a simple runtime switch without code changes. This results in an about 70% higher overall model performance compared to double precision for this setup.

In fact, the use of single precision for earth system modeling has gained popularity in recent years. For example, after a series of preliminary studies (e.g., Nakano et al., 2018; Váňa et al., 2017), and a substantial effort to rewrite model components, ECMWF (the European Centre for Medium-Range Weather Forecasts) recently announced a switch to single precision for high-resolution and ensemble forecasts (Lentze, 2021), with convincing results. While the use of single precision is not yet mainstream in climate modeling (also due to issues like spurious wave growth, Nakano et al., 2018), we see Veros also as a tool to investigate the feasibility of this promising way to increase computational performance with acceptable loss of accuracy.

After about 24 hr of real time, we obtained the output shown in Figure 1 at the beginning of this article. Even though the overturning circulation is far from spun up after 1 model year, we can clearly identify most real-world ocean currents, and the integration is stable. With the final time step of 180 s we achieve a throughput of about 1.3 model years per day in single precision, and 0.77 model years per day in double precision.

To put this into perspective, the Geophysical Fluid Dynamics Laboratory ocean model OM4 includes a nominal 0.25° setup that achieves a performance of 12 model years per day on 4671 CPU cores (Adcroft et al., 2019). Given that this setup has 1440 × 1080 × 75 grid elements and a time step of 900 s, this gives an equivalent 0.1° throughput of 0.81 model years per day. Taking into account that OM4 is a more complicated model and includes a sea ice component, we estimate that our setup on 16 GPUs is able to replace at least 2000 (Fortran) CPU cores. This is in line with our benchmark comparisons with PyOM2 (Section 4.3), where we find that 1 GPU can replace at least 200 CPU cores on the same setup, leading to a conservative equivalent performance estimate of 3200 CPU cores for 16 fully saturated GPUs.

6. Discussion and Outlook

In the previous sections, we show that a pure Python code base coupled with an accelerator library like JAX can achieve comparable performance to a traditional Fortran model across a wide range of architectures. Additionally, we demonstrate that earth system modeling on GPUs is a viable alternative that performs well and uses significantly less energy than CPUs, even when applied to real applications like an eddying model setup.

While Moore's law still seems to apply in the strict sense (in terms of transistor density), CPU clock speeds have increased sub-exponentially since the early 2000s. On GPUs however, exponential scaling seems very much alive—an observation that has been coined as Huang's law (Perry, 2018). While it is too early to determine whether this trend will be as persistent as Moore's law, it seems that GPUs offer huge potential for high-performance computing in general and earth system modeling in particular.

We are not the only ones to recognize this trend. Fuhrer et al. (2018) present COSMO, an atmospheric model written in a domain-specific language that can be compiled to different hardware architectures. They estimate their GPU-based integrations to be at least 5 times more energy efficient than similar experiments on CPU, in line with what we find here. Gridtools (Afanasyev et al., 2021), along with its Python bindings gt4py, is a project that specifically aims to accelerate stencil operations, the main operation throughout the Veros core. This is similar to what we achieve via JAX. Oceananigans.jl (Ramadhan et al., 2020) is an ocean model written in the high-level language Julia, which has a built-in just-in-time compiler that can also generate GPU code, and which has excellent performance.

Recently, cloud TPUs have been gaining traction in some large-scale machine learning applications (e.g., Dosovitskiy et al., 2021) as an alternative to GPUs. Since JAX has native TPU support, almost all routines of Veros can already run on TPU. However, TPUs are primarily optimized for matrix multiplication operations, which are not used in Veros outside of the linear solver. While TPUs *could* prove to be a way to even faster and more energy efficient models, it remains to be seen whether they can be used in practice. But with JAX on our hands we already have the tools to use them should they prove valuable.

We see projects like Veros as a way for academia to profit from the enormous amount of resources that has been invested into machine learning, both in terms of hardware and software developments. This investment is ongoing, and JAX is still evolving rapidly, which is no surprise considering that the project is less than 3 years old.

For Veros, we see two important developments in the near future. First of all, we plan to address its current shortcomings by implementing a tripolar grid structure (S. Griffies et al., 2005) and adding a sea ice model to achieve a realistic representation of the Arctic. While this is a challenging undertaking, we expect these new features to be well compatible with our current software stack (that is, JAX and PETSc), and be well suited for GPUs. Second, we anticipate Veros to become even faster on GPU, especially in multi-GPU contexts. With increasing number of devices, the linear (barotropic) solver eventually becomes a bottleneck. We expect to be able to remedy this by using NVIDIA's AmgX library (Naumov et al., 2015), which is optimized from the ground up for multi-GPU, distributed use cases like ours.

While the flexibility and rich library ecosystem of Python is a strong asset, there are also some notable obstacles when choosing Python over Fortran. Decades of real-world usage and the relative simplicity of the Fortran language have led to an established community standard of model development. As a consequence, most Fortran models read similarly to each other. This is currently not the case in Python development, where the chosen abstraction and library stack have a huge influence on the structure of the model code. This calls for a collective effort to formalize a common interface for the development of high-performance models in Python. We are confident that this can and will happen should this approach gain the required momentum.

We urge the community to explore this exciting possibility of high-level models with low-level performance, towards a more resource efficient generation of earth system models.

Appendix A: Benchmark Platforms

Benchmark platforms are shown in Table A1.

Table A1
Benchmark Platforms

	Compute node (I)	CPU cluster (II)	Google cloud instance a2-megagpu-16g (III)
CPU	2 × Intel Xeon E5-2650 v4 @ 2.20 GHz (24 cores total)	2 × Intel Xeon E5-2683 v4 @ 2.1 GHz (32 cores total) per node	96 virtual CPU cores
GPU	2 × NVIDIA Tesla P100 (16 GB HBM2 memory)	–	16 × NVIDIA A100 (40 GB HBM2 memory)
Main memory	512 GB	128 GB per node	1360 GB
File system	LUSTRE filesystem @ 128 MB s^{-1} read/write performance	LUSTRE filesystem @ 128 MB s^{-1} read/write performance	1 TB persistent SSD storage
Operating system	CentOS 7.9	CentOS 7.9	Ubuntu 20.04
Software stack	CUDA 11.2, OpenMPI 4.0.5, PETSc 3.15, jaxlib 0.1.67	MPICH 3.3.2, PETSc 3.13, jaxlib 0.1.67	CUDA 11.3, OpenMPI 4.0.5, HDF5 1.12.0, PETSc 3.15, jaxlib 0.1.68
Interconnectivity	–	40 GBps Mellanox QDR Infiniband inter-node	9.6 TBps NVIDIA NVlink GPU-to-GPU

Note. Specifications of architecture III according to Parthasarathy and Kleban (2021).

Data Availability Statement

Veros is available under MIT license at https://github.com/team-ocean/veros. The described version is v1.3.4. All scripts used to perform the benchmarks in this article, raw benchmark results, and plotting scripts are available at https://doi.org/10.17894/ucph.abb1726e-3e99-434d-92f9-2e112ee4c778. The 0.1° setup described in Section 5 is available at https://github.com/dionhaefner/veros-01deg/tree/v1.0.

Acknowledgments

D. Häfner and R. Nuterman acknowledge funding from the Danish Hydrocarbon Research and Technology Centre (DHRTC). The authors acknowledge Google LLC for providing free research credits to conduct the experiments described in Sections 4.3 and 5 on Google Cloud instances. Other computing resources were provided by DC^3, the Danish Center for Climate Computing. We graciously thank the JAX developers for their support and various bug fixes. We also thank Till Grenzdörffer for his contribution of a tridiagonal solver in CUDA to Veros. We are grateful to James Avery and Brian Vinter for encouragement and support throughout the project. We thank 4 anonymous reviewers for their valuable remarks.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)* (pp. 265–283).

Adcroft, A., Anderson, W., Balaji, V., Blanton, C., Bushuk, M., Dufour, C. O., et al. (2019). The GFDL global ocean and sea ice model OM4.0: Model description and simulation features. *Journal of Advances in Modeling Earth Systems*, *11*(10), 3167–3211. https://doi.org/10.1029/2019MS001726

Adcroft, A., Campin, J., Dutkiewicz, S., Evangelinos, C., Ferreira, D., Forget, G., et al. (2021). *MITgcm user manual*. Retrieved from https://mitgcm.readthedocs.io/en/latest/index.html

Afanasyev, A., Bianco, M., Mosimann, L., Osuna, C., Thaler, F., Vogt, H., et al. (2021). GridTools: A framework for portable weather and climate applications. *SoftwareX*, *15*, 100707. https://doi.org/10.1016/j.softx.2021.100707

Arakawa, A., & Lamb, V. R. (1977). Computational design of the basic dynamical processes of the UCLA general circulation model. *Methods in Computational Physics*, *17*, 173–265. https://doi.org/10.1016/b978-0-12-460817-7.50009-4

Balay, S., Gropp, W. D., McInnes, L. C., & Smith, B. F. (1997). Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, & H. P. Langtangen (Eds.), *Modern software tools in scientific computing* (pp. 163–202). Birkhäuser Press. https://doi.org/10.1007/978-1-4612-1986-6_8

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., & Bengio, Y. (2010). Theano: A CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)* (Vol. 4, pp. 1–7).

Bleck, R., Dean, S., O'Keefe, M., & Sawdey, A. (1995). A comparison of data-parallel and message-passing versions of the Miami Isopycnic Coordinate Ocean Model (MICOM). *Parallel Computing*, *21*(10), 1695–1720. https://doi.org/10.1016/0167-8191(95)00043-3

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs*. Retrieved from http://github.com/google/jax

Carleo, G., Choo, K., Hofmann, D., Smith, J. E., Westerhout, T., Alet, F., et al. (2019). NetKet: A machine learning toolkit for many-body quantum systems. *SoftwareX*, *10*, 100311. https://doi.org/10.1016/j.softx.2019.100311

Collobert, R., Bengio, S., & Mariéthoz, J. (2002). *Torch: A modular machine learning software library*. Tech. Rep. Idiap.

Dalcin, L. D., Paz, R. R., Kler, P. A., & Cosimo, A. (2011). Parallel distributed computing using Python. *Advances in Water Resources*, *34*(9), 1124–1139. https://doi.org/10.1016/j.advwatres.2011.04.013

Danabasoglu, G., Bates, S. C., Briegleb, B. P., Jayne, S. R., Jochum, M., Large, W. G., & Yeager, S. G. (2012). The CCSM4 ocean component. *Journal of Climate*, *25*(5), 1361–1389. https://doi.org/10.1175/jcli-d-11-00091.1

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., & Houlsby, N. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.*arXiv:2010.11929 [cs]*. Retrieved from http://arxiv.org/abs/2010.11929

Eden, C. (2016). Closing the energy cycle in an ocean model. *Ocean Modelling*, *101*, 30–42. https://doi.org/10.1016/j.ocemod.2016.02.005

Eden, C., & Greatbatch, R. J. (2008). Towards a mesoscale eddy closure. *Ocean Modelling*, *20*(3), 223–239. https://doi.org/10.1016/j.ocemod.2007.09.002

Eden, C., & Olbers, D. (2014). An energy compartment model for propagation, nonlinear interaction, and dissipation of internal gravity waves. *Journal of Physical Oceanography*, *44*(8), 2093–2106. https://doi.org/10.1175/jpo-d-13-0224.1

EuroHPC. (2021). *Discover EuroHPC JU |European high performance computer Joint undertaking*. Retrieved from https://eurohpc-ju.europa.eu/discover-eurohpc-ju

Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., & Bachem, O. (2021). *Brax - A differentiable physics engine for large scale rigid body simulation*. Retrieved from http://github.com/google/brax

Frostig, R., Johnson, M. J., & Leary, C. (2018). *Compiling machine learning programs via high-level tracing*. Systems for Machine Learning.

Fuhrer, O., Chadha, T., Hoefler, T., Kwasniewski, G., Lapillonne, X., Leutwyler, D., et al. (2018). Near-global climate simulation at 1 km resolution: Establishing a performance baseline on 4888 GPUs with COSMO 5.0. *Geoscientific Model Development*, *11*(4), 1665–1681. https://doi.org/10.5194/gmd-11-1665-2018

Gaspar, P., Grégoris, Y., & Lefevre, J.-M. (1990). A simple eddy kinetic energy model for simulations of the oceanic vertical mixing: Tests at station Papa and long-term upper ocean study site. *Journal of Geophysical Research*, *95*(C9), 16179–16193. https://doi.org/10.1029/jc095ic09p16179

Gent, P. R., Willebrand, J., McDougall, T. J., & McWilliams, J. C. (1995). Parameterizing eddy-induced tracer transports in ocean circulation models. *Journal of Physical Oceanography*, *25*(4), 463–474. https://doi.org/10.1175/1520-0485(1995)025<0463:peitti>2.0.co;2

Grenzdörffer, T. (2021). *Accelerating ocean modelling: Adressing performance bottlenecks of the ocean modelling framework Veros*. (Student project outside of course scope). Retrieved from https://sid.erda.dk/share_redirect/CVvcrowL22/Thesis/Till_Grenzdorffer_MSc_thesis.pdf

Griffies, S., Gnanadesikan, A., Dixon, K. W., Dunne, J., Gerdes, R., Harrison, M. J., et al. (2005). Formulation of an ocean model for global climate simulations. *Ocean Science*, *1*(1), 45–79. https://doi.org/10.5194/os-1-45-2005

Griffies, S. M. (1998). The Gent–McWilliams skew flux. *Journal of Physical Oceanography*, *28*(5), 831–841. https://doi.org/10.1175/1520-0485(1998)028<0831:tgmsf>2.0.co;2

Griffies, S. M., & Hallberg, R. W. (2000). Biharmonic friction with a smagorinsky-like viscosity for use in large-scale eddy-permitting ocean models. *Monthly Weather Review*, *128*(8), 2935–2946. https://doi.org/10.1175/1520-0493(2000)128<2935:BFWASL>2.0.CO;2

Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (Vol. 1). MIT press.

Häfner, D. (2021). *HPC benchmarks for Python*. Retrieved from https://github.com/dionhaefner/pyhpc-benchmarks

Häfner, D., Jacobsen, R. L., Eden, C., Kristensen, M. R. B., Jochum, M., Nuterman, R., & Vinter, B. (2018). Veros v0.1 – A fast and versatile ocean simulator in pure Python. *Geoscientific Model Development*, *11*(8), 3299–3312. https://doi.org/10.5194/gmd-11-3299-2018

Häfner, D., & Vicentini, F. (2021). mpi4jax: Zero-copy MPI communication of JAX arrays. *Journal of Open Source Software*, *6*(65), 3419. https://doi.org/10.21105/joss.03419

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., et al. (2020). Array programming with numpy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Kristensen, M. R., Lund, S. A., Blum, T., Skovhede, K., & Vinter, B. (2013). Bohrium: Unmodified NumPy code on CPU, GPU, and cluster. In *Python for high performance and scientific computing (PyHPC 2013)*.

Lentze, G. (2021). Forecast upgrade innovates on single precision and ensemble resolution. Retrieved from https://www.ecmwf.int/en/about/media-centre/news/2021/forecast-upgrade-innovates-single-precision-and-ensemble-resolution

Madec, G., Bourdallé-Badie, R., Bouttier, P.-A., Bricaud, C., Bruciaferri, D., Calvert, D., & Vancoppenolle, M. (2017). *NEMO ocean engine*. report. https://doi.org/10.5281/zenodo.3248739

McDougall, T. J., & Barker, P. M. (2011). Getting started with TEOS-10 and the Gibbs Seawater (GSW) oceanographic toolbox. *SCOR/IAPSO WG*, *127*, 1–28.

Meehl, G. A., Boer, G. J., Covey, C., Latif, M., & Stouffer, R. J. (2000). The coupled model intercomparison project (CMIP). *Bulletin of the American Meteorological Society*, *81*(2), 313–318. https://doi.org/10.1175/1520-0477(2000)081<0313:tcmipc>2.3.co;2

Nakano, M., Yashiro, H., Kodama, C., & Tomita, H. (2018). Single precision in the dynamical core of a nonhydrostatic global atmospheric model: Evaluation using a baroclinic wave test case. *Monthly Weather Review*, *146*(2), 409–416. https://doi.org/10.1175/MWR-D-17-0257.1

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., et al. (2015). AmgX: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, *37*(5), S602–S626. https://doi.org/10.1137/140980260

Norman, M., Larkin, J., Vose, A., & Evans, K. (2015). A case study of cuda fortran and openacc for an atmospheric climate kernel. *Journal of Computational Science*, *9*, 1–6. https://doi.org/10.1016/j.jocs.2015.04.022

Olbers, D., & Eden, C. (2013). A global model for the diapycnal diffusivity induced by internal gravity waves. *Journal of Physical Oceanography*, *43*(8), 1759–1779. https://doi.org/10.1175/jpo-d-12-0207.1

Parthasarathy, B., & Kleban, C. (2021). *A2 VMs with NVIDIA A100 GPUs are GA*. Retrieved from https://cloud.google.com/blog/products/compute/a2-vms-with-nvidia-a100-gpus-are-ga/

Perry, T. S. (2018). Move over, Moore's law. Make way for Huang's law [Spectral Lines]. *IEEE Spectrum*, *55*(5), 7–7. https://doi.org/10.1109/mspec.2018.8352557

Phan, D., Pradhan, N., & Jankowiak, M. (2019). Composable effects for flexible and accelerated probabilistic programming in NumPyro. *ArXiv preprint*. *arXiv:1912.11554*.

Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T. (1989). *Numerical recipes*. Cambridge university press.

Ramadhan, A., Wagner, G. L., Hill, C., Campin, J.-M., Churavy, V., Besard, T., & Marshall, J. (2020). Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs. *Journal of Open Source Software*, *5*(53), 2018. https://doi.org/10.21105/joss.02018

Ruge, J. W., & Stüben, K. (1987). Algebraic multigrid. In *Multigrid methods* (pp. 73–130). SIAM. https://doi.org/10.1137/1.9781611971057.ch4

Sanders, J., & Kandrot, E. (2010). *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley Professional.

Schoenholz, S., & Cubuk, E. D. (2020). Jax md: A framework for differentiable physics. *Advances in Neural Information Processing Systems*, *33*.

Váňa, F., Düben, P., Lang, S., Palmer, T., Leutbecher, M., Salmond, D., & Carver, G. (2017). Single precision in weather forecasting models: An evaluation with the IFS. *Monthly Weather Review*, *145*(2), 495–502. https://doi.org/10.1175/MWR-D-16-0228.1

Wienke, S., Springer, P., Terboven, C., & an Mey, D. (2012). OpenACC—First experiences with real-world applications. In *European Conference on Parallel Processing* (pp. 859–870). https://doi.org/10.1007/978-3-642-32820-6_85