Simplified Ocean Models on GPUs

André R. Brodtkorb*

Abstract

This paper describes the implementation of three different simplified ocean models on a GPU (graphics processing unit) using Python and PyOpenCL. The three models are all based on the solving the shallow water equations on Cartesian grids, and our work is motivated by the aim of running very large ensembles of forecast models for fully nonlinear data assimilation. The models are the linearized shallow water equations, the non-linear shallow water equations, and the two-layer non-linear shallow water equations, respectively, and they contain progressively more physical properties of the ocean dynamics. We show how these models are discretized to run efficiently on a GPU, discuss how to implement them, and show some simulation results. The implementation is available online under an open source license, and may serve as a starting point for others to implement similar oceanographic models.

1 Introduction

Operational oceanographic forecasts are today based on numerical models that capture many aspects of the complex physics of the ocean. Whilst these models can be accurate in a statistical sense, they often have large errors when it comes to short term ocean currents. This is due to large uncertainties in initial conditions and forcing, as we have very few observations of the ocean compared to the atmosphere.

By disregarding the long-term driving forces of the ocean currents (such as temperature and salinity gradients), we can construct models that are valid for short-term dynamics. Such simplified models were in fact used operationally in the early days of computational oceanography. Our motivation in this work is to construct, implement, and run a very large ensemble of simplified ocean models. With a simplified model, we will be able to run a much larger number of model evaluations in the same time frame. A second benefit of these models is that they are highly suited for socalled GPU computing [1], and GPUs have been shown to outperform the traditional CPU by 5-50 times for a variety of algorithms [2]. By designing a simplified model that fits the GPU well, we may benefit from its potentially large speed increase, which again may bring us closer to the goal of using fully non-linear data assimilation techniques such as particle filters that may maximize our use of observations. The end goal is a more accurate model initialization, and hence a more accurate short-term current forecast.

Mapping the shallow water equations to the GPU has been done by several authors previously(see for example [1] and the references therein). The novelty

*Department of Mathematics and Cybernetics, SINTEF Digital, Andre.Brodtkorb@sintef.no

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

of this work lies in the use of traditional oceanographic ocean models, and the availability of the source code as Jupyter notebooks under an open source license.

2 Simplified Ocean Models

To construct simplified ocean models we make the assumption that the ocean current is predominantly horizontal and that we can represent it with a vertical average. Whilst we know this assumption to be false, it nonetheless represents a useful simplification that can be used to model the ocean dynamics under certain circumstances. To increase the realism we can model the ocean as a stratified media with several vertically averaged layers.

In this work, we have implemented two one-layer shallow water models, and one two-layer model. For the one-layer model, we assume a uniform density throughout the water column, and for the two-layer we have a top layer with a density smaller than the bottom layer. The one-layer model is discretized first using a linearization around a mean depth, and then using a non-linear numerical scheme. For the two-layer model, we use a non-linear discretization with interlayer momentum transfer through a friction term. A detailed derivation of the numerical schemes can be found in Røed [3], and a Fortran reference of the linearlized numerical scheme together with several reference benchmark cases and computational results is also available [4].

The shallow water equations can be written in many ways, and it is customary to include bed shear stress and Coriolis source terms in oceanography [3]:

$$\frac{\partial \eta}{\partial t} = -\frac{\partial hu}{\partial x} - \frac{\partial hv}{\partial y} \tag{1}$$

$$\frac{\partial hu}{\partial t} = -\frac{\partial hu^2}{\partial x} - \frac{\partial huv}{\partial y} - gH\frac{\partial \eta}{\partial x} - \frac{R}{H}hu + A\left[\frac{\partial hu}{\partial x} + \frac{\partial hu}{\partial y}\right] + f \cdot hv \tag{2}$$

$$\frac{\partial hv}{\partial t} = -\frac{\partial huv}{\partial x} - \frac{\partial hv^2}{\partial y} - gH\frac{\partial \eta}{\partial y} - \frac{R}{H}hv + A\left[\frac{\partial hv}{\partial x} + \frac{\partial hv}{\partial y}\right] - f \cdot hu \tag{3}$$

Here, η is the water surface deviation (measured from the mean sea level), H is the depth at mean sea level so that $h = H + \eta$ is the water depth, hu and hv the mass transports along the abscissa and ordinate, g the gravitational constant, R represents linear bed shear stress, f represents the Coriolis force, and A represents eddy viscosity (see also Figure 1).

Linearized one-layer numerical scheme

By assuming that η is small and that we are in Geostophic balance (second order terms in velocity become negligible), we can neglect several of the terms in Equations (2) and (3), and arrive at

$$\frac{\partial \eta}{\partial t} = -\frac{\partial hu}{\partial x} - \frac{\partial hv}{\partial y} \tag{4}$$

$$\frac{\partial hu}{\partial t} = -gH\frac{\partial \eta}{\partial x} - \frac{R}{H}hu + f \cdot hv \tag{5}$$

$$\frac{\partial hv}{\partial t} = -gH\frac{\partial \eta}{\partial y} - \frac{R}{H}hv - f \cdot hu \tag{6}$$

These equations can be discretized using an explicit forward-backward linear (Dufort-Frankel leap-frog) scheme on an Arakawa C-type staggered grid. The

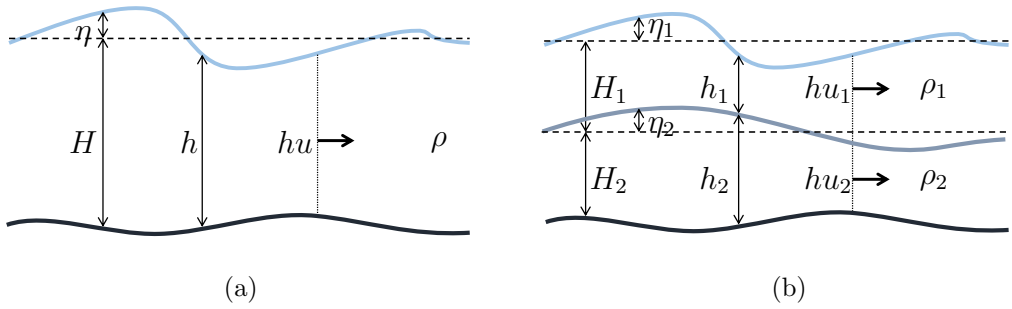


Figure 1: Sketch of the variables in the one-layer model (a) and the two-layer model (b). The two-layer model has a bottom layer with a slightly more dense fluid than the top layer ($\rho_2 > \rho_1$).

Figure 2: Reconstruction of physical quantities at other grid locations. The reconstruction of the horizontal velocity hu at $(i+\frac{1}{2}, j)$ internal nodes (a) and boundary nodes for closed boundaries (b). The reconstruction of hu is similar.

discrete conservation of mass formulation then becomes

$$\eta^{n+1}_{i+1/2,j+1/2} = \eta^{n}_{i+1/2,j+1/2} - \frac{\Delta t}{\Delta x}\left[hu^{n+1}_{i,j+1/2} - hu^{n+1}_{i+1,j+1/2}\right] - \frac{\Delta t}{\Delta y}\left[hv^{n+1}_{i+1/2,j} - hv^{n+1}_{i+1/2,j+1}\right]. \tag{7}$$

We need to reconstruct our conserved quantities at certain points because of the staggering. We use bilinear interpolation, and for cells on the domain boundary, this reduces to linear interpolation when using wall boundary conditions (see also Figure 2). For example, to reconstruct the momentum along the x-axis, we use

$$\bar{hu}^{n}_{i+1/2,j} = \begin{cases} \frac{1}{2}\left[hu^{n}_{i+1,j-1/2} + hu^{n}_{i+1,j+1/2}\right], & i = 0 \\ \frac{1}{4}\left[hu^{n}_{i,j-1/2} + hu^{n}_{i,j+1/2} + hu^{n}_{i+1,j-1/2} + hu^{n}_{i+1,j+1/2}\right], & i \in [1, nx-2] \\ \frac{1}{2}\left[hu^{n}_{i,j-1/2} + hu^{n}_{i,j+1/2}\right], & i = nx-1 \end{cases}$$

and equivalently for the momentum along the y-axis. The reconstructed mean sea level depth at $(i, j{+}1/2)$ is computed using $\bar{H}_{i,j+1/2} = 1/2(H_{i-1/2,j+1/2}+H_{i+1/2,j+1/2})$, and equivalently along the y-axis.

With this in mind, we can now write up the discretized conservation of

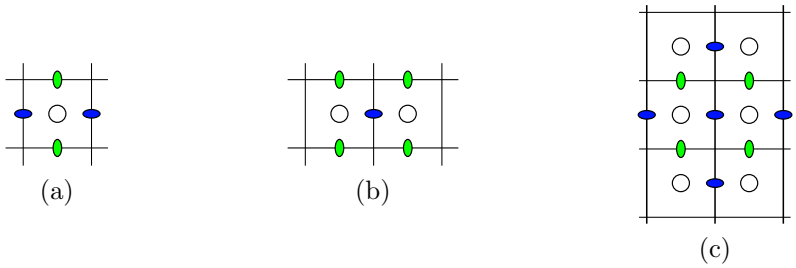


Figure 3: Computational stencils used to compute (a) $\eta^{n+1}_{i+1/2,j+1/2}$ and (b) $hu^{n+1}_{i,j+1/2}$ for the linearized shallow water equations. (c) shows the stencil for $hu^{n+1}_{i,j+1/2}$ using the nonlinear numerical scheme. The stencils for hv is similar.

momentum.

$$hu^{n+1}_{i,j+1/2} = \frac{1}{B_{i+1/2,j}}\Bigg[hu^{n}_{i,j+1/2} + \Delta t \cdot \mathrm{f}\,\bar{hv}^{n}_{i,j+1/2} - \Delta t \cdot g\bar{H}_{i+1/2,j}\frac{\eta^{n}_{i+1/2,j+1/2} - \eta^{n}_{i-1/2,j+1/2}}{\Delta x}\Bigg)\Bigg], \tag{8}$$

in which $B_{i+1/2,j} = 1 + \Delta t R/\bar{H}_{i,j+1/2}$ is our implicitly handled friction term. The observant reader will note that the discrete momentum terms are taken to be at time step $n+1$ in Equation (7). In a similar fashion, we compute hv^{n+1} using the updated hu^{n+1} values. Apart from that, our second momentum equation is discretized equivalently to the first. These stencils are visualized in Figure 3, showing that η is evolved in time using a five point stencil, and hu and hv using a seven point stencil for the linear scheme.

Nonlinear one-layer numerical scheme

To capture the inherent nonlinearities of the equations, we need to keep the nonlinear terms and discretize them accordingly. The discretization of these equations is somewhat lengthy, and the reader is referred to Røed [3] for a detailed derivation. The discretized mass conservation is similar to the linearized scheme,

$$\eta^{n+1}_{i,j} = \left(\frac{2\Delta t}{\Delta x}(hu^{n}_{i,j} - hu^{n}_{i-1,j}) - \frac{2\Delta t}{\Delta y}(hv^{n}_{i,j} - hv^{n}_{i,j-1})\right) \tag{9}$$

whilst the discretized conservation of momentum is more complex

$$hu^{n+1}_{i,j+1/2} = \frac{1}{C_{i,j+1/2}}\Bigg[hu^{n}_{i,j+1/2} + 2\Delta t \cdot \mathrm{f}\,\bar{hv}^{n}_{i,j+1/2} + 2\Delta t \cdot \left(\frac{N_{i,j+1/2}}{\Delta x} + \frac{P_{i,j+1/2}}{\Delta x} + AE_{i,j+1/2}\right)\Bigg] \tag{10}$$

Here, C is the friction term, N are the nonlinear terms, P are the pressure terms, and E is the eddy viscosity:

$$C_{i,j+1/2} = 1 + \frac{2A\Delta t}{\bar{H}_{i,j+1/2}} + \frac{2A\Delta t(\Delta x^2 + \Delta y^2)}{(\Delta x^2 + \Delta y^2)} \tag{11}$$

$$\begin{aligned} N_{i,j+1/2} = \frac{1}{4}\left[\frac{(hu^n_{i+1,j+1/2} + hu^n_{i,j+1/2})^2}{h^n_{i+1/2,j+1/2}} - \frac{(hu^n_{i,j+1/2} + hu^n_{i-1,j+1/2})^2}{h^n_{i-1/2,j+1/2}}\right] \\ + \frac{1}{4}\frac{\Delta x}{\Delta y}\left[\frac{(hu^n_{i,j+3/2} + hu^n_{i,j+1/2})(hv^n_{i+1/2,j+1} + hv^n_{i-1/2,j+1})}{\bar{h}^n_{i,j}}\right] \\ + \frac{1}{4}\frac{\Delta x}{\Delta y}\left[\frac{(hu^n_{i,j+1/2} + hu^n_{i,j-1/2})(hv^n_{i+1/2,j} + hv^n_{i-1/2,j})}{\bar{h}^n_{i-1,j}}\right] \end{aligned} \tag{12}$$

$$\begin{aligned} P_{i,j+1/2} = g\bar{H}_{i,j+1/2}\left(\eta^n_{i+1/2,j+1/2} - \eta^n_{i-1/2,j+1/2}\right) \\ + \frac{1}{2}\left((\eta^n_{i+1/2,j+1/2})^2 - (\eta^n_{i-1/2,j+1/2})^2\right) \end{aligned} \tag{13}$$

$$\begin{aligned} E_{i,j+1/2} = \frac{1}{\Delta x^2}\left(hu^n_{i+1,j+1/2} - hu^{n-1}_{i,j+1/2} + hu^n_{i-1,j+1/2}\right) \\ + \frac{1}{\Delta y^2}\left(hu^n_{i,j+3/2} - hu^{n-1}_{i,j+1/2} + hu^n_{i,j-1/2}\right) \end{aligned} \tag{14}$$

The discretization for hv follows that of hu, and ends up with very similar expressions. The stencil of this numerical scheme can be seen in Figure 3.

3 Two-layer Shallow Water model

Our two-layer shallow water model consists of two shallow water systems (with a lighter ocean layer on top of a more dense deep ocean layer) which are coupled through inter-layer friction terms on the momentum equations. We also have a slightly altered pressure term for the bottom layer to account for the fluid on top:

$$\frac{\partial \eta_1}{\partial t} = -\frac{\partial hu_1}{\partial x} - \frac{\partial hv_1}{\partial y} + \frac{\partial \eta_2}{\partial t} \tag{15}$$

$$\frac{\partial hu_1}{\partial t} = -\frac{\partial hu_1^2}{\partial x} - \frac{\partial huv_1}{\partial y} - gH_1\frac{\partial \eta_1}{\partial x} - \frac{\tau_I}{\rho_2}hu_1 + A_1\left[\frac{\partial hu_1}{\partial x} + \frac{\partial hu_1}{\partial y}\right] + f \cdot hv_1 \tag{16}$$

$$\frac{\partial hv_1}{\partial t} = -\frac{\partial huv_1}{\partial x} - \frac{\partial hv_1^2}{\partial y} - gH_1\frac{\partial \eta_1}{\partial y} - \frac{\tau_I}{\rho_2}hv_1 + A_1\left[\frac{\partial hv_1}{\partial x} + \frac{\partial hv_1}{\partial y}\right] - f \cdot hu_1 \tag{17}$$

$$\frac{\partial \eta_2}{\partial t} = -\frac{\partial hu_2}{\partial x} - \frac{\partial hv_2}{\partial y} \tag{18}$$

$$\begin{aligned} \frac{\partial hu_2}{\partial t} = -\frac{\partial hu_2^2}{\partial x} - \frac{\partial huv_2}{\partial y} - gH_2\left[\frac{\rho_1}{\rho_2}\frac{\partial \eta_1}{\partial x} + \frac{\rho_2 - \rho_1}{\rho_2}\frac{\partial \eta_2}{\partial x}\right] \\ - \left[\frac{\tau_I}{\rho_2} + \frac{R}{H_2}\right]hu_2 + A_2\left[\frac{\partial hu_2}{\partial x} + \frac{\partial hu_2}{\partial y}\right] + f \cdot hv_2 \end{aligned} \tag{19}$$

$$\begin{aligned} \frac{\partial hv_2}{\partial t} = -\frac{\partial huv_2}{\partial x} - \frac{\partial hv_2^2}{\partial y} - gH_2\left[\frac{\rho_1}{\rho_2}\frac{\partial \eta_1}{\partial y} + \frac{\rho_2 - \rho_1}{\rho_2}\frac{\partial \eta_2}{\partial y}\right] \\ - \left[\frac{\tau_I}{\rho_2} + \frac{R}{H_2}\right]hv_2 A_2\left[\frac{\partial hv_2}{\partial x} + \frac{\partial hv_2}{\partial y}\right] - f \cdot hu_2 \end{aligned} \tag{20}$$

Here τ_I is the interlayer friction coefficient, and ρ_2 is the density of the bottom layer. The bottom layer can equivalently be written as in which the term $g(\rho_2 - \rho_1)/\rho_2$ typically is referred to as the reduced gravity. The discretization of the two-layer system follows that of the nonlinear one-layer equations presented above, except that the momentum transfer through the friction coefficient τ_I is taken explicitly.

4 GPU Implementation

The numerical schemes outlined in the previous sections can be written as stencil computations, which means that the value of cell (i, j) at the next timestep can be computed independently of all other cells using only local information. This means that we can evolve all cells in parallel, and these algorithms typically map very well to architectures such as the GPU.

We have used OpenCL [5] to develop our GPU code, which is an open standard for massively parallel computations. OpenCL is supported on platforms including CPUs from Intel and AMD, and GPUs from NVIDIA and AMD. This contrasts the very similar NVIDIA CUDA programming language, which only works on NVIDIA hardware. In the following, we discuss how the numerical schemes outlined above can be implemented efficiently on GPU hardware, and the implementation is available online on Github[1]. In this text, however, we do not dive into the low-level optimizations which are often required to achieve peak performance. Readers interested in these optimization strategies are referred to [6, 1, 7, 8].

Numerical schemes on the GPU

The linear numerical scheme is the simplest of the above presented ocean models, and is relatively simple to implement on the GPU. For this scheme, we first evolve hu in time and then use the updated values to evolve hv. Finally, we use the updated values of hu and hv to evolve η, and we have completed one single timestep. Because this numerical method imposes a strict order for how to update our conserved variables, we have to create three different *kernels* that compute hu, hv, and η, respectively. A kernel is a function that executes on the GPU, meaning that each of these kernels run after each other sequentially. Within each kernel, however, we are able to exploit the inherent parallelism of the different stencil computations.

One of these kernels is outlined in Listing 1, and shows some of the steps required to compute Equation (8). This kernel is essentially just like a regular CPU function, only that it is executed in parallel, operates on data that is located on the GPU, and stores the results also on the GPU. We start by specifying how many parallel threads we want to execute for this function. These threads are organized as a grid of blocks. Each block consists of $m \times n$ threads, and the grid will consist of $o \times p$ blocks so that we in total have a computational grid of $mo \times np$ threads. In our case, we have (somewhat arbitrarily) chosen our block size to be 8×8 threads. Finding a good block size will often be a key ingredient to achieve high performance, and a power of two is often a good choice. In general, the optimal block size will vary between different GPUs and different kernels, and the size should typically neither be too small nor too large.

The only way to distinguish between two threads in the computational grid is through the variables `ti` and `tj`. They specify the logical position of the thread, and

[1] `https://github.com/babrodtk/gpu-ocean/tree/master/doc/reference_python`

Listing 1: GPU kernel that computes hu at the next timestep. __kernel means that this is a GPU function, and __global means that this memory is located in GPU main memory. The source code has been edited and simplified for readability.

```
1  //GPU Kernel that evolves U in time
2  __kernel void computeUKernel(
3          int nx_, int ny_,
4          __global float* H_ptr_, int H_pitch_,
5          __global float* U_ptr_, int U_pitch_,
6          __global float* V_ptr_, int V_pitch_,
7          __global float* eta_ptr_, int eta_pitch_) {
8
9      //Index of cell within domain
10     const int ti = get_global_id(0);
11     const int tj = get_global_id(1);
12
13     //Compute address of row tj
14     __global float* const U_row =
15         (__global float*) ((__global char*) U_ptr_ + U_pitch_*tj);
16
17     //Read input data, test out of bounds, etc.
18     ...
19
20     //Compute result and store to main GPU memory
21     U_row[ti] = (1.0/B)*(U_current + .dt*(f*V_m + g*H*dEtadx));
22 }
```

can then be used to look up the correct data values for that location. An example of this is shown in lines 14, 15, and 21 of Listing 1. Here, we first find the memory location of row tj in hu (using the width of each line in bytes, U_pitch_), and then finally write to element ti.

Both the nonlinear model, and the two-layer ocean model follow the same implementation strategy, but are increasingly complex. For example, since our discretized nonlinear model requires both hu^n and hu^{n-1} (see Equation (14)), we need to have both these values in our kernel. For the two-layer model, we evolve η_1 and η_2 in the same kernel, and similarly for the momentum equations, resulting in three kernels here as well. In total, the kernels which evolve hu_1 and hu_2 each become around 300 lines long.

Launching kernels and memory management

We use Python and PyOpenCL[2] to execute the different OpenCL kernels on the GPU. Python is a high-level language which enables rapid experimentation, prototyping, and development of code. Using PyOpenCL, we can easily transfer data back and forth between the GPU and the CPU, and coupled with Matplotlib, we are able to easily plot our solution and check if it is reasonable.

Listing 2 shows how we can launch the kernel which computes hu at the next time step from Python. The first thing to note in this example is that we actually have

[2] https://mathema.tician.de/software/pyopencl/

Listing 2: Python code required to compile and launch a kernel on the GPU, and to transfer data between the CPU and the GPU.

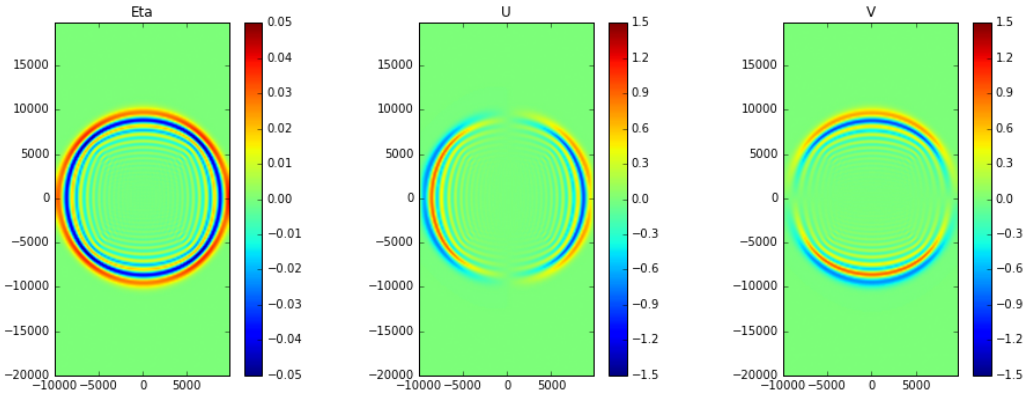
```
1  import pyopencl
2
3  #Access the GPU OpenCL driver and compile the kernel
4  cl_ctx = pyopencl.create_some_context()
5  cl_queue = pyopencl.CommandQueue(cl_ctx)
6  eta_kernel_string = ... # Read from file here
7  U_kernel = pyopencl.Program(cl_ctx, eta_kernel_string).build()
8
9  #Create CPU version of data
10 host_data = np.ones((ny, nx), dtype=np.float32, order='C');
11
12 #Allocate data on the GPU, and upload
13 H.data = pyopencl.Buffer(cl_ctx, \
14      mf.READ_WRITE | mf.COPY_HOST_PTR, \
15      hostbuf=host_data)
16 H.pitch = host_data.itemsize*nx
17 ... #Similar for hu, hv, and eta.
18
19 #Launch the kernel
20 local_size = (8, 8)
21 global_size = (ceil(nx/8.0), ceil(ny/8.0))
22 U_kernel.computeUKernel(cl_queue, global_size, local_size, \
23                         nx, ny, \
24                         H.data, H.pitch, \
25                         hu.data, hu.pitch, \
26                         hv.data, hv.pitch, \
27                         eta.data, eta.pitch)
28
29 #Download data from the GPU
30 result = np.empty((ny, nx), dtype=np.float32, order='C')
31 pyopencl.enqueue_copy(cl_queue, result, eta)
```

two different memory spaces: variables that live on the CPU (such as host_data) cannot be accessed from the GPU, and vice versa. Therefore, we need to allocate and upload data to the GPU to be able to use it. Both uploading and downloading data is a slow process, and should only be performed when required. We will therefore achieve the highest performance if we perform many timesteps on the GPU before downloading the result back to the CPU again. In lines 21–26, we see that launching a kernel is much like calling a regular Python function, except that arrays passed as arguments must first be copied to the GPU.

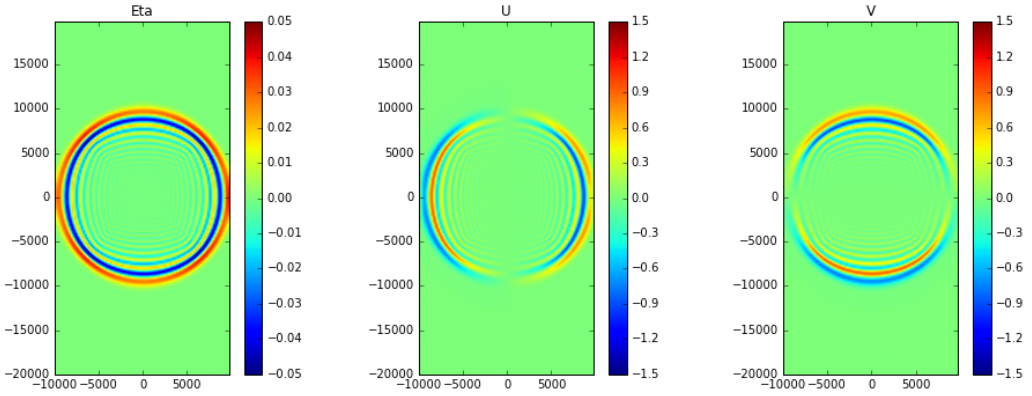
5 Simulation results

We have previously shown that the linear ocean model discussed here can reproduce the results of a reference Fortran implementation [9], and we have checked our current OpenCL implementation against this reference as well.

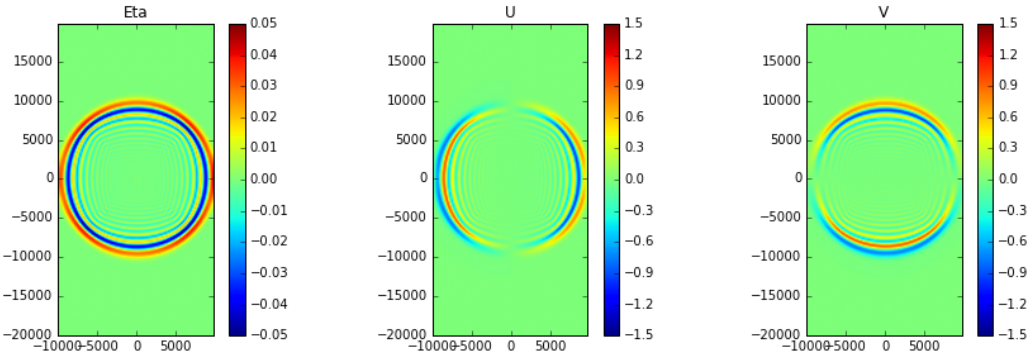
We have used a synthetic circular dam break as our benchmark case. The



(a) Linear scheme

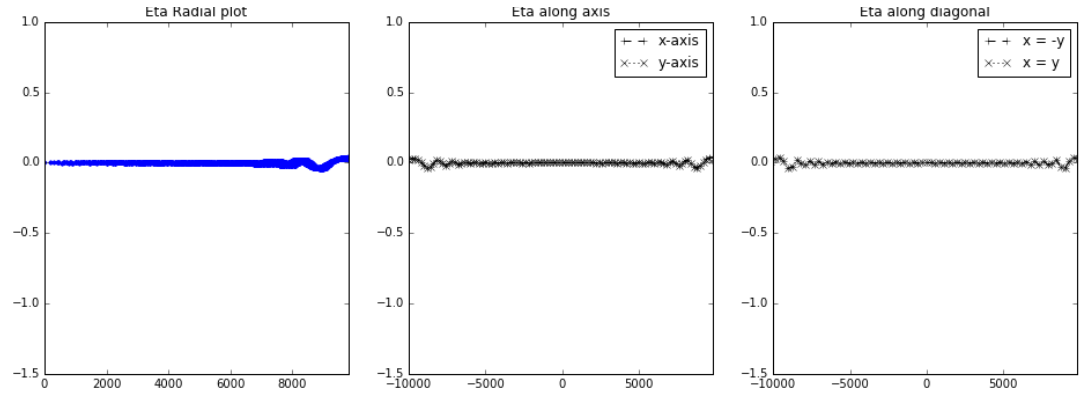


(b) Nonlinear scheme

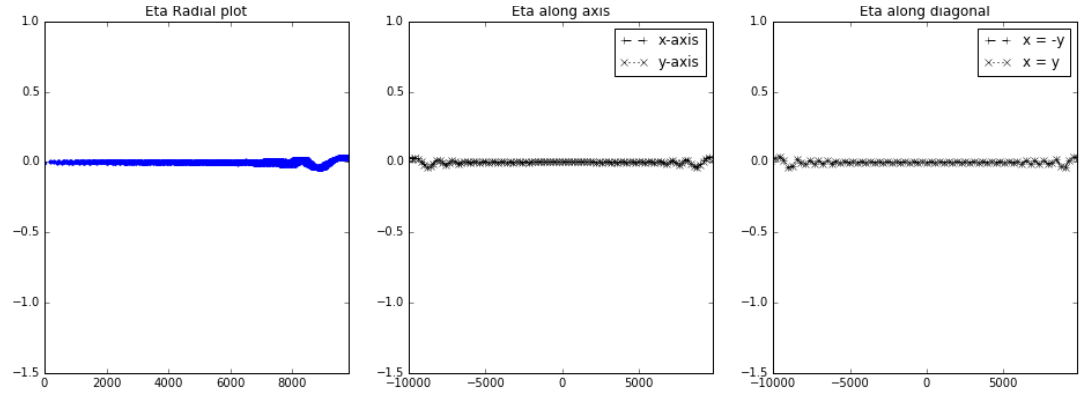


(c) Two-layer scheme

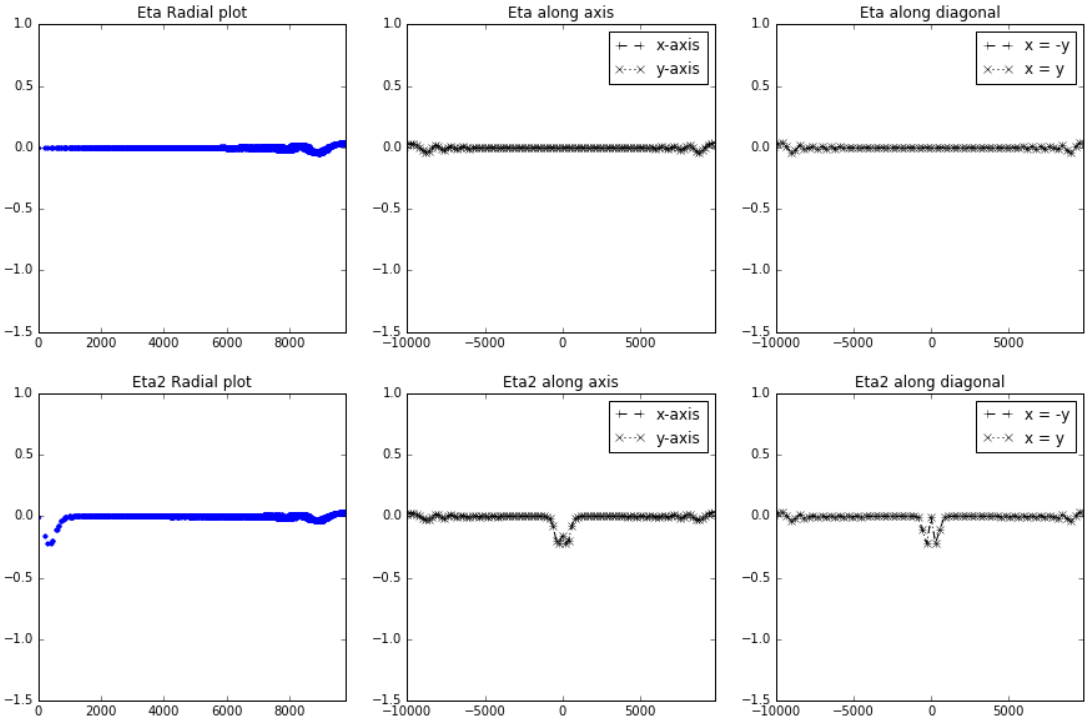
Figure 4: Simulation of a circular dam break after 400 seconds. The top row shows η, hu, and hv, respectively, for the linear scheme. The middle row shows results computed with the nonlinear formulation, whilst the bottom row shows the solution using the two-layer scheme (top layer). Qualitatively, all the schemes give similar solutions.



(a) Linear



(b) Nonlinear



(c) Two-layer

Figure 5: Simulation of a circular dam break after 400 seconds. The top row shows a radial plot illustrating the radial symmetry of η (left), cross sections along the x and y axis (center), and finally diagonal cross sections with a positive and negative slope (right). The second row shows the equivalent results for the nonlinear scheme, and the two bottom rows show results of the different layers of the the two-layer scheme.

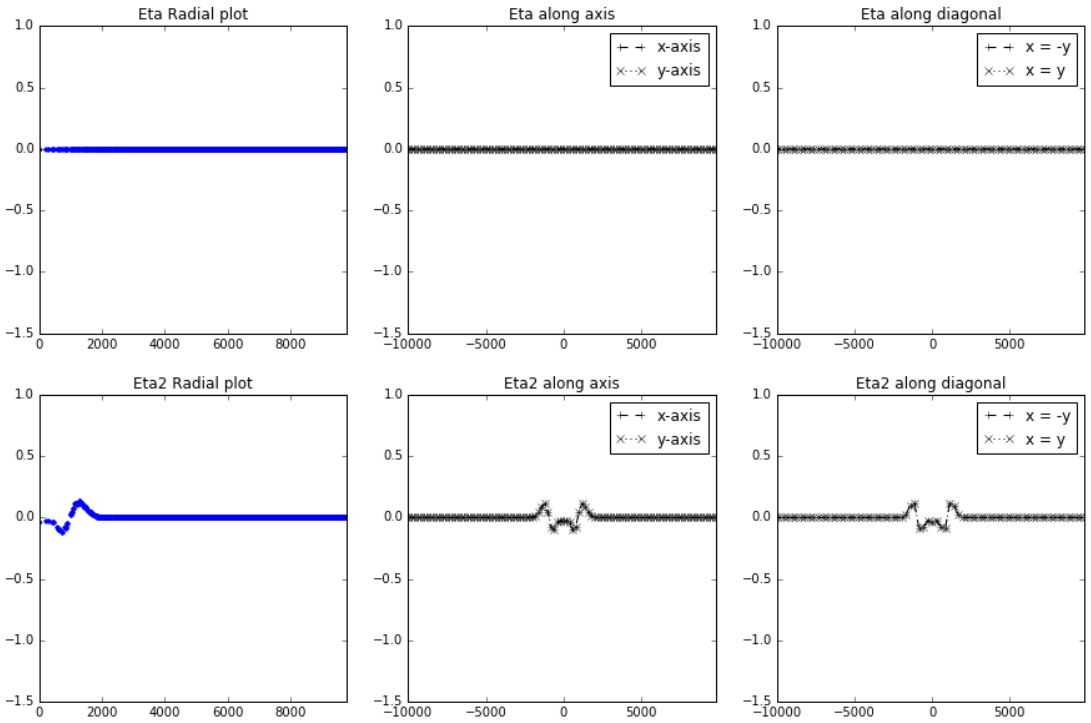


Figure 6: Simulation of an internal (bottom layer) circular dam break after 1900 seconds using the two-layer nonlinear model. The top row shows a radial plot illustrating the radial symmetry of the upper layer η_1 (left), cross sections along the x and y axis (center), and finally diagonal cross sections with a positive and negative slope (right). The lower row similarly shows the symmetry of the bottom layer, η_2.

computational domain consists of 100×200 cells, with $\Delta x = \Delta y = 200$. Our timestep is $\Delta t = 1$, the gravitational coefficient is set to $g = 9.81$, the Coriolis coefficient is $\mathrm{f} = 0$, and the bottom friction coefficient is set to $R = 0.001$. For our nonlinear models, we have used an eddy viscosity parameter of $A = 1.0$, and for our two-layer model, we have used an upper layer density of $\rho_1 = 1025$ and a lower layer density of $\rho_2 = 1030$. We start with an initial Gaussian bell in the middle of our domain, (x_0, y_0), for the water surface:

$$\eta = \begin{cases} e^{(x^2+y^2)/c_0}, & \text{if} \quad \sqrt{x^2+y^2} < c_0 \\ 0.0 & \text{else} \end{cases}$$

$$x = i\Delta x - x_0, \qquad y = j\Delta y - y_0, \qquad c_0 = 100000.$$

For our two-layer model, this disturbance is applied to the top layer. Our bathymetry depth is set to $H = 60$ (for the two layer model we use $H_1 = 10$ and $H_2 = 50$), and we simulate for 400 seconds. At this time the disturbance has just reached the global boundaries (which in our case are set to be reflective).

Figure 5a shows the result with the linear model. This model has previously been been verified against an existing reference Fortran implementation [9], and shown to give equal results (to within floating point precision). We see that the solution is quite symmetric, but with very slight grid effects since the computed wave is not completely circular. This shows up in the radial plot as a widening of the wave just around 8500 meters from the center. If we look at the axis symmetry, we see

that the solutions along the x axis and y axis are equal, and equivalently, that the diagonals also are equal.

If we now move to the nonlinear model in Figure 5b, we see that the solution is visually identical to that of the linear scheme, and displays the same solution characteristics. For the two-layer model, shown in Figure 5c, we see that the upper layer disturbance and total momenta are again visually identical to the linear scheme. If we then look at the upper and lower layer separately, we see that the fast upper layer wave travels as expected in the linear model, but that we additionally get a slow moving internal wave in the bottom layer.

Figure 6 shows a variation of this benchmark case, in which we disturb the lower layer instead of the top layer. This leads to an internal dam break with a slow moving wave in the lower layer. This case also gives rise to some very slight top layer ripples which are no longer visible in the solution after 1900 seconds.

6 Summary

We have implemented a GPU version of a linear one-layer shallow water model, a non-linear one-layer model, and a two-layer model. The implementations have been made using Python and PyOpenCL for rapid prototyping and development, and is available on Github under an open source license. The results are promising for running a large number of scenarios on the GPU to increase the accuracy of ocean current predictions through the use of nonlinear data assimilation.

References

[1] A.R. Brodtkorb, M.L. Sætra, M. Altinakar, Computers & Fuids **55**, 1 (2011). DOI 10.1016/j.compfluid.2011.10.012

[2] A.R. Brodtkorb, C. Dyken, T.R. Hagen, J.M. Hjelmervik, O. Storaasli, Scientific Programming **18**(1), 1 (2010)

[3] L.P. Røed, Documentation of simple ocean models for use in ensemble predictions. Part I: Theory. Tech. Rep. 3/2012, Norwegian Meteorological Institute (2012)

[4] L.P. Røed, Documentation of simple ocean models for use in ensemble predictions. Part II: Benchmark Cases. Tech. Rep. 5/2012, Norwegian Meteorological Institute (2012)

[5] Khronos Group. OpenCL. `http://www.khronos.org/opencl/`

[6] A.R. Brodtkorb, T.R. Hagen, M.L. Sætra, Journal of Parallel and Distributed Computing **73**(1), 4 (2013). DOI 10.1016/j.jpdc.2012.04.003

[7] P. Micikevicius. Fundamental performance optimizations for GPUs. [Conference presentation], 2010 GPU Technology Conference, session 2011 (2010)

[8] P. Micikevicius. Analysis-driven performance optimization. [Conference presentation], 2010 GPU Technology Conference, session 2012 (2010)

[9] A.R. Brodtkorb, T.R. Hagen, L.P. Røed, One-layer shallow water models on the GPU. Tech. Rep. 27/2013, Norwegian Meteorological Institute (2013)