Using the GPU to Predict Drift in the Ocean

André Rigland Brodtkorb¹, Kai Håkon Christensen², Lars Petter Reed²,³, and Martin Lilleeng Sætra²,⁴

¹ SINTEF, Dept. Appl. Math., P.O. Box 124 Blindern, 0314 Oslo, Norway ² Norwegian Meteorological Institute, P.O. Box 43 Blindern, 0313 Oslo, Norway ³ University of Oslo P.O. Box 1072 Blindern 0316 Oslo Norway ⁴ Westerdals Oslo ACT, P.O. Box 9215 Granland, 0134 Oslo, Norway

 Emails: André.Brodtkorb@sintef.no, Kai.H.Christensen@sintef.no, LarsPetter.Reed@sintef.no, Martin.L.Sætra@sintef.no

1. Abstract: We describe the implementation of a simple numerical scheme for solving the shallow water equations on a graphics processing unit (GPU), that will be used in the further development of a massive ensemble prediction system running on the GPU. The numerical scheme has previously been used in operational forecasting, and benchmarks comparing the FORTRAN CPU version with the new GPU version have been performed. The results show that the GPU implementation gives a speed-up over the CPU of slightly more than 200 times. This is highly promising regarding the possibilities of running a large number of ensembles cost effectively on a computer and thereby increasing the usefulness of short-term ocean current forecasts and drift trajectory predictions.

2. Motivation

Although the dynamics is similar, there is a huge difference in scale between an atmospheric low pressure system (bottom left) and an oceanic low pressure system (top left), or ocean eddy, as these features are usually called. The modelled position of an atmospheric eddy is shown on the right, an error of similar size in the placement of an ocean eddy has a large negative impact on the ocean forecast. By utilizing the GPU’s parallel processing power, we can run very large ensembles, and get more accurate results.

3. Target Application Areas

Icebergs and Sea Ice
Oil Spills
Floating Structures
Search and Rescue

4. Governing Equations

The shallow water equations written in flux form:

\[
\begin{align*}
\partial_t U &= -f V + g H \nabla \cdot U + R_U \\
\partial_t V &= f U - g H \nabla \cdot V + R_V \\
\partial_t H &= -\nabla \cdot (U H) + P
\end{align*}
\]

where \( U = (u, v) \) is the horizontal velocity at the surface, \( V = (v, -u) \) is the bottom stress, \( H \) is the water depth, \( f \) is the Coriolis parameter, \( g \) is the gravitational acceleration, \( P \) is the forcing, and \( R_U \) and \( R_V \) are the bottom friction and wind stress.

5. Simplified Ocean Model on the GPU

The GPU is a massively parallel processor that requires different algorithms than traditional CPUs. For explicit schemes with compact stencils, we essentially have an embarrassingly parallel problem, which suits the GPU perfectly. To map the computation to the parallel architecture of the GPU, we perform domain decomposition, in which each block is computed independently. Within each block, however, we use a kernel with 16x16 threads that collectively computes its subdomain. A staggered grid is used, where \( \eta \) and \( R \) are represented in cell centres, \( \Gamma \) at vertical cell interfaces, and \( \Gamma_\ell \) at horizontal cell interfaces.

6. CUDA and OpenCL

The simplified ocean model has been implemented in both CUDA C/C++ and OpenCL C. This enables us to compare the implementations and run the model on many different hardware architectures. In both versions, the CPU-side code is written in C++. All benchmarking and validation results presented here are for the CUDA version. The reference CPU-implementation is written in FORTRAN.

OpenCL code example, computeEta-kernel:

```c
__kernel void computeEta(
    __global float *Ux,
    __global float *Uy,
    __global float *Uz,
    __global float *H,
    __global float *eta,
    __global float *f,
    __global float *R_h,
    __global float *R_v,
    __global float *Ux_z,
    __global float *Uy_z,
    __global float *Uz_z,
    __global float *f_z,
    __global float *R_h_z,
    __global float *R_v_z,
    __constant float *Ux_first, Uy_first, Uz_first,
    __constant float *H_first,
    __constant float *f_first,
    __constant float *R_h_first,
    __constant float *R_v_first,
    __constant float *Ux_z_first, Uy_z_first, Uz_z_first,
    __constant float *f_z_first,
    __constant float *R_h_z_first,
    __constant float *R_v_z_first,
    __constant float *Ux_last, Uy_last, Uz_last,
    __constant float *H_last,
    __constant float *f_last,
    __constant float *R_h_last,
    __constant float *R_v_last,
    __constant float *Ux_z_last, Uy_z_last, Uz_z_last,
    __constant float *f_z_last,
    __constant float *R_h_z_last,
    __constant float *R_v_z_last,
    __constant float *Ux_first_2, Uy_first_2, Uz_first_2,
    __constant float *H_first_2,
    __constant float *f_first_2,
    __constant float *R_h_first_2,
    __constant float *R_v_first_2,
    __constant float *Ux_z_first_2, Uy_z_first_2, Uz_z_first_2,
    __constant float *f_z_first_2,
    __constant float *R_h_z_first_2,
    __constant float *R_v_z_first_2,
    __constant float *Ux_last_2, Uy_last_2, Uz_last_2,
    __constant float *H_last_2,
    __constant float *f_last_2,
    __constant float *R_h_last_2,
    __constant float *R_v_last_2,
    __constant float *Ux_z_last_2, Uy_z_last_2, Uz_z_last_2,
    __constant float *f_z_last_2,
    __constant float *R_h_z_last_2,
    __constant float *R_v_z_last_2)
{
    // 1. Compute U and V
    // 2. Reconstruct U and V at horiz cell interfaces
    // 3. Compute eta and reconstruct eta at horiz cell interfaces
    // 4. Apply boundary conditions
}
```

Validation Against FORTRAN CPU Reference Code

The figure shows performance comparison of the CPU and GPU runtimes for deterministic simulation runs for different problem sizes.

The GPU version is approximately 200 times faster than the CPU reference version. This is highly promising for creating large model ensembles on the GPU, permitting forcing and initial conditions, since the GPU provides the possibility of running 200 ensemble members within the same time frame as it takes to run one ensemble member on the CPU. Whilst this comparison is clearly unfair (single core untuned CPU code vs. tuned GPU), it demonstrates that the problem fits very well to the GPU architecture, and shows the potential of GPU computing.