Ocean Modelling 41 (2012) 16–21

Contents lists available at SciVerse ScienceDirect

Ocean Modelling

journal homepage: www.elsevier.com/locate/ocemod

Accelerating a barotropic ocean model using a GPU

Folkert Bleichrodt [a,b], Rob H. Bisseling [b], Henk A. Dijkstra [a,*]

[a] Institute for Marine and Atmospheric Research Utrecht, Department of Physics and Astronomy, Utrecht University, Utrecht, The Netherlands
[b] Mathematical Institute, Utrecht University, Utrecht, The Netherlands

ARTICLE INFO

Article history:
Received 16 May 2011
Received in revised form 29 September 2011
Accepted 3 October 2011
Available online 17 October 2011

Keywords:
Barotropic ocean model
Acceleration
GPU

ABSTRACT

The two-dimensional barotropic vorticity equation is one of the basic equations of ocean dynamics. It is important to have efficient numerical solution techniques to solve this equation. In this paper, we present an implementation of a numerical solution using a Graphics Processing Unit (GPU). The speedup of the calculation on the GPU with respect to that on a CPU depends on the grid size, but reaches a factor of about 50 for resolutions from 2049×2049 up to 4097×4097. It may therefore be efficient to use green GPUs in future high-resolution ocean modelling studies.

1. Introduction

The computer and mobile devices industry today is one of the most flourishing markets in the world. It always has been since the introduction of the personal computer in the late eighties. According to Moore's Law, the number of transistors on a computer chip doubles every two years. For the past twenty years this has largely been correct. Of course, this effect can only be sustained with a driving force behind it: a thriving consumer electronics market.

While the scientific community embraces the use of High Performance Computing and has access to the fastest computers in the world, it is a relatively small user group. Therefore the decisions and the directions of hardware companies are still largely based on consumer revenues. Many scientists therefore rely on large clusters of CPUs connected by a fast network. Distributed-memory systems make out most of the TOP500 of supercomputers, see www.top500.org/lists/. Nevertheless, CPUs are not the only hardware components that are being heavily developed. The video gaming industry has driven the development of specialized graphical hardware, the Graphics Processing Unit (GPU). Its main task is to render realistic images for video games. To keep up with the increasing complexity of games, the GPU has turned into a highly parallel device. With a large number of processors, it allows fine-grained parallelism for parallel rendering of pixels and vertices.

While the GPU has a lot of potential as a parallel device, it was close to useless for scientists for a long time. Apart from accelerated visualizations, a GPU was not suitable for general-purpose computing. Some early attempts were made for the solution of dense linear systems (Galoppo et al., 2005), but the development efforts were huge, since all mathematical computations had to be translated into graphics operations. However, in 2006 NVIDIA provided a toolkit which extends the C programming language with a minimal set of primitives for parallelism which allows to use GPUs for general-purpose computing. This extension is called the *Compute Unified Device Architecture* or CUDA. With CUDA, it is now possible to create parallel programs in C for the GPU, similar to OpenMP or MPI programs. The introduction of GPUs in science has proven to be successful. The TOP500 (e.g., November 2010) features more and more clusters extended with several GPUs per node. On NVIDIA's website, applications of CUDA are being showcased and numerous scientific papers are listed showing moderate to impressive improvements in performance of codes on a GPU over a CPU.

The GPU has been employed in areas such as magnetohydrodynamics (Wang et al., 2010; Wong et al., 2011) and CFD codes (Thibault and Senocak, 2009). In ocean and/or atmospheric modelling, the use of the GPU has also been explored. Michalakes and Vachharajani (2008) port a computationally intensive module of the Weather Research and Forecast (WRF) model to CUDA. They achieved a performance of 7.4 Gflop/s (using an NVIDIA 8800 GTX), a more than $20 \times$ speedup compared to a Fortran implementation running on a 2.8 Ghz Pentium workstation. Using a GTX 280 (Fermi architecture) showed a tremendous increase of performance to almost 65 Gflop/s. Hill (2008) explored the use of GPUs for accelerating parts of MITgcm (Marshall et al., 1997). The code

* Corresponding author. Address: Institute for Marine and Atmospheric research Utrecht, Utrecht University, 3584 CC Utrecht, The Netherlands. Tel.: +31 30 2535441; fax: +31 30 2543163.
E-mail address: H.A.Dijkstra@uu.nl (H.A. Dijkstra).

1463-5003/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ocemod.2011.10.001

was run on a cluster of 30 PCs each hosting one NVIDIA GPU. He achieved a performance increase of nearly an order of magnitude compared to the same algorithm running on the cluster's CPUs.

From these results, it is expected that ocean model simulations as well as atmosphere model simulations may enjoy performance boosts when employing a GPU. In this paper, we will explore the possibilities for accelerating a finite-difference method for solving the barotropic vorticity equation. This model describes the homogeneous quasi-geostrophic wind-driven ocean circulation and is one of the cornerstone models in modern physical oceanography. As the implementation is tightly coupled to the architecture of the GPU and the CUDA language, we provide a very brief overview to hardware and software in Section 2. In Section 3, the ocean model and the numerical methods used are briefly described. The main new elements follow in Section 4, where we describe the CUDA implementation details of the fast Poisson solver. Optimization steps are also discussed to show the coding difficulties that may arise when programming for the GPU. In Section 5, numerical results and achieved performance are presented and the results are summarized and discussed in the final Section 6.

2. Hardware and software

The CUDA programming language is tailor-made for the NVIDIA GPU and therefore has some restrictions based on the hardware layout. This leads to the best performance, but it undermines portability. As a result of this, it is important to understand the hardware layout and memory organization in detail.

The CUDA hardware is based on the single instruction, multiple data (SIMD) paradigm. This means that multiple threads run the same instructions, defined in C functions called *kernels*, on different data. A GPU consists of several multiprocessors (MPs, 30 for the Tesla M1060), each containing 8 scalar processors. With several hundreds of threads running concurrently on the GPU, this allows very fine-grained data parallelism.

The threads are organized into a (1D or 2D) grid, consisting of several (1D, 2D, or 3D) blocks comprised of up to hundreds of threads. The thread blocks are distributed over MPs and there they are scheduled for execution. The number of threads that run concurrently depends on the available register resources on the MP provided for the block. This favours small kernels (i.e., those using few registers). Block sizes should be as large as possible, since idle time of threads can be hidden by running other threads of the block. However, enough blocks should be available to keep all MPs busy.

Memory on the GPU is divided in global DRAM, with typical sizes of 1–4 GB and local (i.e., local per MP) and fast shared memory (16 or 32 KB). In addition, each MP has a register memory (of at most 16384 registers for 1.3 devices), that allows fast access to kernel parameters and variables. Memory operations from GPU to CPU and visa versa are very slow compared to global memory DRAM access. The throughput is limited by the bandwidth of the PCIe bus on the motherboard. This can be a limiting factor when GPUs are employed for accelerating only parts of existing codes. The best performance is achieved when the code is designed from scratch with GPUs in mind.

On the software side, CUDA provides a compiler and profiling/debugging tools. CUDA also comes with a number of libraries which allows higher level programming. The most important ones are CUBLAS, CUFFT, and CUSPARSE providing Basic Linear Algebra Subroutines, Fast Fourier Transforms, and sparse matrix operations respectively.

For a detailed introduction to CUDA, we suggest to read NVIDIA (2010b). For optimization details and performance issues, see NVIDIA (2010a).

3. The numerical ocean model

Consider a rectangular ocean basin of size $L \times L$ having a constant depth D. The basin is situated on a mid-latitude β-plane with a central latitude $\theta_0 = 45°\text{N}$ and Coriolis parameter $f_0 = 2\Omega\sin\theta_0$, where Ω is the angular velocity of the Earth. The meridional variation of the Coriolis parameter at the latitude θ_0 is indicated by β_0. The density ρ of the water is constant, the flow is forced at the surface through a wind-stress vector $\mathbf{T} = \tau_0(\tau^x(x,y), \tau^y(x,y))$, the bottom-friction coefficient is ϵ_0 and the lateral friction coefficient is A_H.

3.1. The barotropic vorticity equation

The governing equations are non-dimensionalized using a horizontal length scale L, a vertical length scale D, a horizontal velocity scale U, the advective time scale L/U and a characteristic amplitude of the wind stress, τ_0. The dimensionless barotropic quasi-geostrophic model of the flow can be written as (Pedlosky, 1987)

$$\begin{cases} \frac{\partial \zeta}{\partial t} = -\mu\zeta + \frac{1}{\mathrm{Re}}\nabla^2\zeta + \frac{\partial \psi}{\partial y}\frac{\partial \zeta}{\partial x} - \frac{\partial \psi}{\partial x}\left(\frac{\partial \zeta}{\partial y} + \beta\right) + \alpha\left(\frac{\partial \tau^y}{\partial x} - \frac{\partial \tau^x}{\partial y}\right) \\ \zeta = \nabla^2\psi, \end{cases} \tag{1}$$

where ζ is the vorticity. The dimensionless horizontal velocities are given by $u = -\partial\psi/\partial y$ and $v = \partial\psi/\partial x$. The parameters in (1) are the planetary vorticity gradient β, the wind-stress strength α, the bottom-friction coefficient μ, and the Reynolds number Re given by

$$\alpha = \frac{\tau_0 L}{\rho D U^2}; \quad \beta = \frac{\beta_0 L^2}{U}; \quad \mu = \frac{\epsilon_0 L}{U}; \quad \mathrm{Re} = \frac{UL}{A_H}. \tag{2}$$

The wind-stress forcing pattern is prescribed as

$$\tau^x(x,y) = -\frac{1}{2\pi}\cos 2\pi y; \quad \tau^y(x,y) = 0, \tag{3}$$

which is symmetric with respect to the mid-axis of the basin (the standard double-gyre case (Dijkstra and Katsman, 1997)). At the lateral boundary, slip conditions are prescribed and hence the boundary conditions are $\psi = \zeta = 0$. The initial conditions at $t = 0$ are $\psi = \zeta = 0$.

3.2. Discretization

The choice for the discretization scheme has an important impact on the performance of the GPU program. First of all, memory access needs to be coalesced to achieve maximal bandwidth. Basically, this means that the stencil should have adjacent points in the spatial dimension. For the reason of coalesced memory access, we choose the central finite-difference scheme in the space direction; most other schemes are also suitable. For time integration, we choose the Adams–Bashforth linear multistep method (Butcher, 2005), which is a common choice for this model.

We consider a square domain $\Omega = [0,1] \times [0,1]$ which is divided in a grid with $N_x \times N_y$ grid points:

$$(x_i, y_j), \quad i = 0, 1, \ldots, N_x - 1, \quad j = 0, 1, \ldots, N_y - 1. \tag{4}$$

In the x-direction, the step size is $\Delta x = 1/(N_x - 1)$ and in the y-direction it is $\Delta y = 1/(N_y - 1)$. We choose an equal step size h in both directions, $h = \Delta x = \Delta y$, and an equal number of points $N_x = N_y$, and write $N = N_x - 2 = N_y - 2$ which simplifies the notation. This will allow us to employ a fast Poisson solver for the second equation in (1).

We use the following notation for the unknowns evaluated on the grid:

$$\psi_{ij}^n = \psi(t_n, x_i, y_j) = \psi(n\Delta t, i\Delta x, j\Delta y).$$

The right-hand side of the first equation of (1) is then written in matrix notation as:

$$G := -\mu Z + \frac{1}{\mathrm{Re}}(CZ + ZC) - (\Psi B)(BZ) + (B\Psi)(ZB - \beta I_N) - T, \tag{5}$$

where the unknowns of the interior of the grid are contained in

$$\Psi_{ij} = \psi_{i+1,j+1}, \quad Z_{ij} = \zeta_{i+1,j+1}, \quad i,j = 0,\ldots,N-1,$$

with I_N the $N \times N$ identity matrix. Here, $(\Psi B)(BZ)$ denotes the element-wise multiplication of the matrices (ΨB) and (BZ). The matrices B and C follow from the standard central finite-difference approximation with local truncation error $\mathcal{O}(h^2)$,

$$B = \frac{1}{2h}\begin{bmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \end{bmatrix}, \quad C = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}, \tag{6}$$

$$T = \alpha\begin{bmatrix} \sin 2\pi y_1 & \ldots & \sin 2\pi y_N \\ \vdots & & \vdots \\ \sin 2\pi y_1 & \ldots & \sin 2\pi y_N \end{bmatrix}, \tag{7}$$

where T is the matrix containing the discrete wind-stress forcing patterns. The boundary and initial conditions are formulated as $\psi_{0,j} = \psi_{N+1,j} = \psi_{i,0} = \psi_{i,N+1} = 0$, $\zeta_{0,j} = \zeta_{N+1,j} = \zeta_{i,0} = \zeta_{i,N+1} = 0$ and $\psi^0_{i,j} = \zeta^0_{i,j} = 0$ for $i, j = 0,\ldots,N+1$.

The Adams–Bashforth method for time integration is as follows. Given is the system of ordinary differential equations (ODEs) that arises from the discretization in space of (1):

$$\frac{d\zeta_{ij}(t)}{dt} = G_{ij}, \quad \zeta_{ij}(t_0) = \zeta^0_{ij},$$

using the two-step method yields:

$$\zeta^{n+2}_{ij} = \zeta^{n+1}_{ij} + \Delta t\left(\frac{3}{2}G^{n+1}_{ij} - \frac{1}{2}G^{n}_{ij}\right), \tag{8}$$

with $i, j = 0,\ldots,N-1$.

Note that this method uses two time steps from the past, while the ODE has only one initial condition ζ^0. To solve this problem, we can use Euler Forward to compute ζ^1 first and then continue with Adams–Bashforth. For the model, we assume that $\psi^0 = 0$, i.e., we assume that the wind stress initiates at $t = 0$. Therefore, we simply define $\psi^{-1} \equiv 0$. In matrix notation, the full discretization of (1) is hence given by

$$Z^{n+2} = Z^{n+1} + \frac{\Delta t}{2}\left(3G^{n+1} - G^n\right). \tag{9}$$

3.3. Poisson solver

To extract the solution ψ^{n+1} from ζ^{n+1} we have to solve the discrete Poisson equation:

$$A\psi = \zeta, \tag{10}$$

where A is a square, negative definite, symmetric matrix resulting from the discretization of the Laplace operator with the usual 5-point approximation in 2D:

$$\zeta_{ij} \approx \frac{\psi_{i+1,j} - 2\psi_{ij} + \psi_{i-1,j}}{(\Delta x)^2} + \frac{\psi_{i,j+1} - 2\psi_{ij} + \psi_{i,j-1}}{(\Delta y)^2}. \tag{11}$$

The matrix A has size $J \times J$ with $J = N^2$, so the grid is stored in a vector of length J. The unknowns are stored row-wise in a vector, such that grid points are consecutive in the y-direction. For example, ψ is stored as:

$$\psi = [\psi_{11}, \psi_{12}, \ldots, \psi_{1N}, \psi_{21}, \ldots, \psi_{2N}, \ldots, \psi_{N1}, \ldots, \psi_{NN}].$$

The eigenvectors of A are given by

$$y_{kl}(i,j) = \sin\frac{ik\pi}{N+1}\sin\frac{jl\pi}{N+1}, \tag{12}$$

with corresponding eigenvalues,

$$\lambda_{kl} = -4 + 2\cos\frac{k\pi}{N+1} + 2\cos\frac{l\pi}{N+1}, \tag{13}$$

for $k, l, i, j = 1,\ldots,N$. It can be proven (Strang, 2007) that the eigenvectors form an orthogonal basis of $\mathbb{R}^{N^2}$, so we can write ζ in the basis of eigenvectors:

$$\zeta = \sum_{k=1}^{N}\sum_{l=1}^{N} c_{kl}y_{kl}, \tag{14}$$

with coefficients c_{kl}. Next, we scale the coefficients c_{kl} by the eigenvalues λ_{kl}. We can then reconstruct the solution ψ by

$$\psi = \sum_{k=1}^{N}\sum_{l=1}^{N} \frac{c_{kl}}{\lambda_{kl}}y_{kl}. \tag{15}$$

Note that the one-dimensional discrete sine transform (DST-I) $\{X_k\}$ of a sequence of numbers $\{x_k\}$ is given by

$$X_k = \sum_{j=1}^{N} x_j \sin\left(\frac{\pi}{N+1}jk\right), \quad k = 1,\ldots,N. \tag{16}$$

A closer look at the eigenvectors hence tells us that (14) and (15) are in fact two-dimensional DSTs.

Instead of actually computing the matrix–vector products of complexity $\mathcal{O}(N^2)$ from (16), we can use a fast algorithm for computing the sine transform. Ideally this would be an algorithm of $\mathcal{O}(N\log_2 N)$ complexity as proposed by Puschel and Moura (2008), but implementing these methods requires a substantial amount of work and there also exist FFT-based (Fast Fourier Transform) algorithms for the sine transform with the same asymptotic complexity, albeit with a higher constant (by a factor of 2). These algorithms can benefit from optimized implementations of the FFT.

Solving (10) consists of three steps:

1. Compute the coefficients c_{kl} using the inverse 2D DST-I applied to ζ,
2. Divide the coefficients by the eigenvalues: $c_{kl} \mapsto \frac{c_{kl}}{\lambda_{kl}}$,
3. Construct ψ by performing again a 2D DST-I on the coefficients c_{kl}.

The DST-I is its own inverse, except for a constant scaling factor $2/(N+1)$. Instead of applying the inverse DST, we apply (16) both in step 1 and 3. The scaling is then applied afterwards. It is clear that most of the work of the Poisson solver consists of computing the DST-I. Therefore we will focus on finding an efficient algorithm for the DST-I.

NVIDIA has a library that implements the Fast Fourier Transform in CUDA. It is based on the FFTW library, *The Fastest Fourier Transform in the West* (Frigo and Johnson, 1998). Unfortunately, it only implements the complex Discrete Fourier Transform (DFT) and no real transforms like the sine or cosine transforms. However, we can still use the DFT after preprocessing, since a sine transform contains the imaginary parts of a similar DFT.

The Discrete Fourier Transform (DFT) is given by:

$$Y_k = \sum_{j=0}^{N-1} y_j e^{-\frac{2\pi jki}{N}},$$

where i now denotes the imaginary unit. Given the input signal $\{x_k\}_{k=1}^{N}$, before the DFT is applied, we extend it to roughly twice its length to have odd symmetry:

$F = \mathrm{DFT}([0, x_1, x_2, \ldots, x_N, 0, -x_N, -x_{N-1}, \ldots, -x_2, -x_1])$. Here, $F = \{F_k\}_{k=0}^{2N+1}$.

The DST-I $\{X_k\}_{k=1}^{N}$ can then be retrieved by (Press et al., 1992)

$$X_k = \frac{i}{2} F_k, \quad k = 1, \ldots, N.$$

For the inverse DST-I we need to scale the result by $2/(N+1)$.

4. Implementation details

4.1. Poisson solver

The most computationally intensive part of the Poisson solver is the two-dimensional discrete sine transform. As we have seen, it can be computed using the Fast Fourier Transform. The advantage of this approach is the fact that a lot of work has been put in creating a parallel FFT in CUDA for the GPU (CUFFT). In general, an implementation of the FFT is one of the first things to be ported to new parallel platforms. This adds a certain degree of portability to the program.

By using an FFT-based sine transform we do lose some performance. First of all, the FFT needs an input signal of length $2(N+1)$, which approximately doubles the complexity of the algorithm compared to a DST algorithm of complexity $\mathcal{O}(N \log_2 N)$. Since the grid is stored in linear memory, it needs to be copied out-of-place and should be extended as described previously. Below is an example for a 5×5 grid:

$$\begin{pmatrix} \zeta_{11} & \zeta_{12} & \zeta_{13} \\ \zeta_{21} & \zeta_{22} & \zeta_{23} \\ \zeta_{31} & \zeta_{32} & \zeta_{33} \end{pmatrix} \mapsto \begin{pmatrix} 0 & \zeta_{11} & \zeta_{12} & \zeta_{13} & 0 & -\zeta_{13} & -\zeta_{12} & -\zeta_{11} \\ 0 & \zeta_{21} & \zeta_{22} & \zeta_{23} & 0 & -\zeta_{23} & -\zeta_{22} & -\zeta_{21} \\ 0 & \zeta_{31} & \zeta_{32} & \zeta_{33} & 0 & -\zeta_{33} & -\zeta_{32} & -\zeta_{31} \end{pmatrix}.$$

Moreover, each row should be converted to a complex vector. The CUFFT library provides a cufftComplex type, which is simply a structure with real and complex parts interleaved. So a real vector in complex form is the real vector interleaved with zeros. A copy operation therefore writes the matrix row-wise to a matrix of type cufftComplex with a stride of two. This introduces a performance penalty of approximately 50% of the theoretical bandwidth for devices with compute capability 1.2 or higher (CUDA, C Best Practices Guide, version 3.2, NVIDIA). Devices with lower compute capability have an even lower performance. This is a disadvantage when working with complex types. Fortunately, there is a way to avoid copying complex data types altogether by using a real DFT algorithm (Press et al., 1992). This DFT of a real vector $\{x_k\}_{k=0}^{N-1}$ takes as input a complex vector of half the length that is built up as

$$y_k = x_{2k} + x_{2k+1} i, \quad k = 0, 1, \ldots, \frac{N}{2} - 1. \tag{17}$$

Note that this allows us to copy the real matrix to a complex matrix of half the row-size without strided memory writes.

After this packing phase, an unscaled inverse FFT is applied to the sequence $\{y_k\}$, giving as output a sequence $\{Y_k\}$. The final stage is an unpacking step that gives us the DFT of the original real signal. Rather than using the one as given in (Press et al., 1992) we use the numerically more robust extract phase from (Inda, 2000, Eq. 3.7):

$$F_k = Y_k - \frac{1}{2}\left(1 + i e^{2\pi i k/N}\right)\left(Y_k - Y^*_{N/2-k}\right), \quad k = 0, \ldots, N/2, \tag{18}$$

where * denotes the complex conjugate.

This algorithm has several advantages. First of all, the packing phase is trivial since it is a cast from one data type to another. Secondly, the algorithm only provides the first $N/2+1$ coefficients of the Fourier transform. Remember that when computing the DST using the FFT, the input signal needs to be extended to approximately twice the length. The output holds the DST in coefficients with index 1 to N; therefore, the first $N/2+1$ coefficients are indeed all we need for the DST. This reduces the amount of work.

4.2. Grid update

Before the results are presented, we will briefly discuss the implementation of (5); again the grid is divided over thread blocks. To improve memory bandwidth, each block loads part of the grid to shared memory, but threads on the boundary need grid points from an adjacent thread block. Therefore, each thread block should also load the halo points, see Fig. 1. This is done by threads on the boundary of the block. This is similar to how distributed-memory systems handle finite differences. The updated grid points are directly written to global memory, row-wise and fully coalesced.

5. Results

The numerical simulations were conducted on an NVIDIA Tesla M1060 GPU and an Intel Xeon L5420 2.50 GHz CPU. The full specifications are listed in SARA (2011). At the time when the simulations were performed this device did not support double precision and therefore simulations on the GPU use single precision. For the numerical experiments, parameter values are the same as used by Berloff and McWilliams (1999) and are shown in Table 1. A choice of $U = \tau_0/(\rho D \beta_0 L) = 2.17 \times 10^{-2}\ \mathrm{ms}^{-1}$ leads to $\alpha = \beta = 1.36 \cdot 10^4$, $\mathrm{Re} = 20.8$ and $\mu = 0$. As the time scale is L/U, a simulation time of 10 years, which is enough to obtain an equilibrium solution, corresponds to a dimensionless time of $t = 1.78$. Very small time steps have to be taken for fine grids, but the time step is still significantly larger than the machine epsilon (i.e., accuracy) for the single-precision floating-point format on the GPU. Note that for a wiggle-free solution, the spatial step size has to satisfy

$$5h < \sqrt[3]{\frac{A_H}{\beta_0}},$$

to resolve the Munk boundary layer.

As the block size we have chosen 16×16, such that half-warps access separate rows of the grid. This is the minimal block size to enable fully coalesced memory access. First of all we have compared the time per iteration of the sequential program and the CUDA implementation. This gives an overview of the performance and it enables us to estimate the time needed to perform a full simulation of 10 years. Since the time per iteration does not depend on the time step we have chosen $\Delta t = 10^{-7}$ and a final time of 0.001 for each grid size, which is a total of 10, 000 iterations. The time per iteration is taken as the average over these 10, 000 iterations. The results are shown in Fig. 2a. For a grid size of 33 the sequential

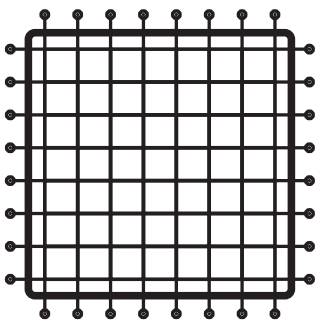

Fig. 1. One thread block writes one grid block to shared memory together with its halo.

Table 1
Standard values of parameters of the barotropic ocean model, as used in the computations.

Parameter	Value	Parameter	Value
L	3.84×10^6 m	τ_0	0.5×10^{-1} Pa
D	3.0×10^2 m	β_0	2.0×10^{-11} (ms)$^{-1}$
f_0	8.3×10^{-5} s^{-1}	A_H	4.0×10^2 m^2 s^{-1}
ρ_0	10^3 kg m^{-3}	ϵ_0	0.0 s^{-1}

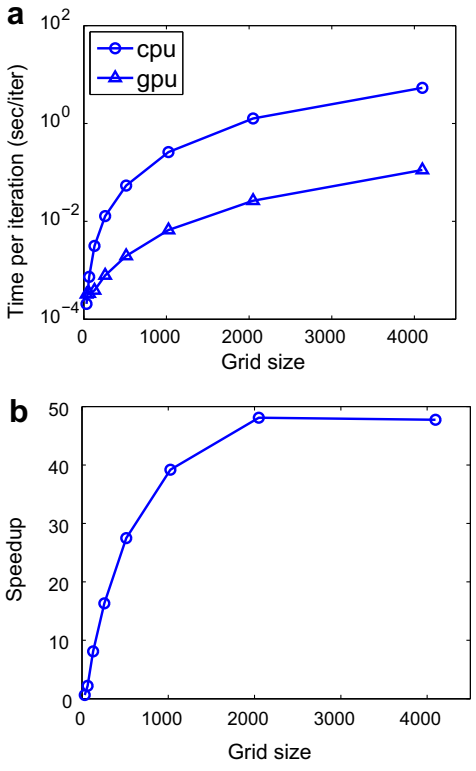

Fig. 2. Performance comparison of CPU and GPU, (a) time per iteration of CPU and GPU algorithms, (b) speedup of GPU compared to CPU.

program is faster, but already for a grid size of 65 the GPU wins. From this figure we see that the GPU performs very well.

Whenever we compare the speed of the GPU with a CPU,[1] we implicitly mean a single core of the Intel Xeon. In Fig. 2b, we have plotted the speedup comparing the GPU with one core of the CPU. A speedup of 48 is reached for a grid size of 2049. For small grid sizes, the speedup is less impressive. If we look closely, the speedup decreases slightly if the grid size is increased to 4097. This is probably due to the overhead caused by CUDA. At some point the speedup should stagnate, when the total number of active threads is reached. At that point, further increasing the number of threads (larger grid sizes) does not result in a larger speedup, but it will only add more overhead.

All grid sizes are such that the lengths of the Fourier transforms are a power of two. The performance of radix-2 FFTs is generally much better than that of mixed-radix transforms.[2] In Table 2, more details are listed on the performance. With a grid of 1025×1025 and a dimensionless time step of $5 \cdot 10^{-7}$ a 10 year simulation takes a little less than 7 h. The sequential program would take over 11 days for such a simulation. On the GPU, full simulations on grids of size 2049 and 4097 are also feasible with run times of 13 h to several days, respectively. While the speedup for small grids is moderate, simulations on larger resolutions are possible. We have measured the run times of the Poisson solver and the grid update. On larger grids, the Poisson solver makes up approximately 75% of the total run time and the grid update 25%.

Table 2
Time per iteration on the CPU and GPU, and the corresponding speedup.

N	GPU ms/iter	CPU ms/iter	Speedup
33	0.3235	0.2056	0.64
65	0.3338	0.7342	2.2
129	0.3883	3.147	8.1
257	0.7845	12.81	16.3
513	1.963	53.94	27.5
1025	6.646	260.5	39.2
2049	26.33	1266	48.1
4097	111.8	5337	47.7

Fig. 3 shows a breakdown in parts of the performance. A logarithmic scale is used to compare the total runtime on a large scale. The proportions of the bar segments have been preserved. This shows that the FFT is responsible for only a small part of the runtime of the Poisson solver (less than 50%, the Poisson solver is the FFT and (un) packing phase combined). Most of the work is in the packing and unpacking phase of the DST-I. On the GPU the corresponding kernels are mostly memory bound, although part of the unpacking phase is computationally more intensive. These results suggest that the packing phases might not be fully optimized and a better performance might be achieved in these steps.

In Table 3, the register use per thread of each kernel as well as its shared memory usage is listed. The first five kernels combined form the Poisson solver. As seen, the register use of each kernel is low, which allows allocation of more blocks on an MP, resulting in better performance scaling.

6. Summary, discussion, conclusions and prospects

We have presented an efficient solution for the time-dependent two-dimensional barotropic vorticity equation on a GPU using a CUDA implementation. Most of the computing time in the code is spent in solving the Poisson equation, which relates the stream function with the vorticity, and we have presented an efficient CUDA implementation for the FFT-based solution method. The speedup for this two-dimensional single-layer model is impressive (up to a factor 50 for the highest resolutions used here).

An interesting subject for future work is three-dimensional models, since they demand the employment of multiple GPUs. However, a problem in using multiple GPUs is that communication between them is needed and this involves moving data from one GPU to another via the CPU's memory. These memory operations are very slow compared to the bandwidth of global memory. The main idea is to partition the domain, possibly using all dimensions, and let each GPU do the computations of one sub-domain. After each time step, a halo update is required. Although the results for two-dimensional models are impressive, it might not scale as well to three dimensions. Besides the additional communication overhead, the available memory per GPU is limited (up to 6 GB currently).

GPUs can also be added to distributed-memory systems to form a hybrid system. Implementations in, e.g., MPI of ocean models can be accelerated using the GPU. The local computations then have to

[1] The sequential program is written in C and performs exactly the same tasks as the CUDA implementation. The difference is that the sequential program uses lookup tables for the Poisson solver. Also, it uses double precision, since the FFT library (FFTW) was compiled with double precision on the test system.

[2] Recall that the Fourier transform has size $2(N+1)$ with N the number of points in the interior of the grid. Therefore, a radix-2 FFT is possible when $N+1$ is a power of two.

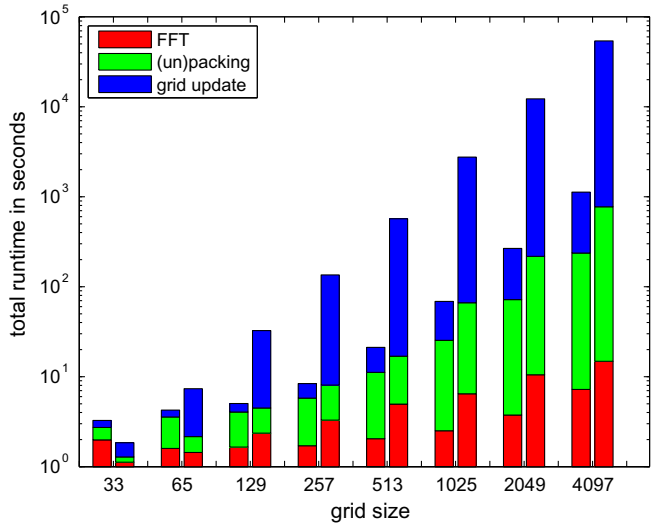

Fig. 3. Performance of individual stages of the computation with (for each grid size) the left bar indicating the GPU and the right bar the CPU. The bar plot is on a logarithmic scale, but proportions of the bar segments have been preserved. The runtimes are for 10,000 iterations.

Table 3
Memory use per kernel. The shared memory is in bytes per thread block. The first five kernels constitute the Poisson solver.

Kernel name	Registers per thread	Shared memory
`extract_scale`	16	1136
`transpose_scale`	8	1136
`scale_copy_flip`	14	1064
`copy_flip`	7	1128
`extract_rfft`	16	1128
`grid_update`	12	2640

be ported to CUDA. In this case, however, the communication time is increased by the latency of transferring data to the device memory. The new Fermi architecture of NVIDIA's GPUs has a 64 kB L1 cache per multiprocessor and a shared 768 kB L2 cache for global memory (Patterson, 2009) which improves bandwidth and reduces latency. Algorithms with low data parallelism, e.g. sparse matrix computations, can especially benefit from the L2 cache. Furthermore, the performance penalty of double precision is now a factor of 2 compared to a factor of 8 for devices with compute capability 1.x such as the NVIDIA Tesla used here. Another major improvement is the ability for concurrent kernel execution.

With this work, we hope to have demonstrated that the speed-up obtained is worth the (programming) effort to develop ocean models which make use of GPUs. Also in other scientific fields, use of GPUs has proven a viable alternative for, or an addition to, the current HPC systems (see, for example the AMUSE framework which is used in astronomy, http://www.amusecode.org). The future of GPU computing, where the boundaries between CPUs and GPUs will likely disappear, is likely to be an exciting one.

Acknowledgements

Computations were done on the GPU cluster at SARA Amsterdam. Use of these computing facilities was sponsored by the National Computing Facilities Foundation (N.C.F.) under the project MP-189-10 with financial support from the Netherlands Organization for Scientific Research (N.W.O.).

References

Berloff, P.S., McWilliams, J.C., 1999. Large-scale, low-frequency variability in wind-driven ocean gyres. J. Phys. Oceanogr. 29, 1925–1949.
Butcher, J.C., 2005. Numerical Methods for Ordinary Differential Equations. John Wiley & Sons Ltd., pp. 98–99.
Dijkstra, H.A., Katsman, C.A., 1997. Temporal variability of the wind-driven quasi-geostrophic double gyre ocean circulation: basic bifurcation diagrams. Geophys. Astrophys. Fluid Dyn. 85, 195–232.
Frigo, M., Johnson, S.G., 1998. FFTW: An adaptive software architecture for the FFT. In: Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing. IEEE Press, Los Alamitos, CA, pp. 1381–1384.
Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D., 2005. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware, in: Proceedings of the ACM/IEEE Supercomputing 2005 Conference.
Hill, C., (2008). Experiences modeling ocean circulation problems on a 30 node commodity cluster with 3840 GPU processor cores, Eos Trans. AGU, Fall Meet. Suppl., Abstract IN23C-1100.
Inda, M.A., (2000). Constructing Parallel Algorithms for Discrete Transforms. Ph.D. Thesis. Dept. Mathematics, Utrecht University.
Marshall, J., Adcroft, A., Hill, C., Perelman, L., Heisey, C., 1997. A finite-volume, incompressible Navier–Stokes model for studies of the ocean on parallel computers. J. Geophys. Res. 102, 5753–5766.
Michalakes, J., Vachharajani, M., 2008. GPU acceleration of numerical weather prediction. Parallel Process. Lett. 18, 531–548.
NVIDIA, 2010a. CUDA C Best Practices Guide. 3.2 edition.
NVIDIA, 2010b. NVIDIA CUDA C Programming Guide. 3.2 edition.
Patterson, D., 2009. The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges. URL http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf.
Pedlosky, J., 1987. Geophysical Fluid Dynamics, second ed. Springer-Verlag, New York.
Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., 1992. Numerical Recipes in C: The Art of Scientific Computing, second ed. Cambridge University Press, Cambridge, UK, pp. 510–514.
Puschel, M., Moura, J.M.F., 2008. Algebraic signal processing theory: Cooley–Tukey type algorithms for DCTs and DSTs. IEEE Trans. Signal Process. 56, 1502–1521.
SARA, 2011. Hardware specifications. URL <http://www.sara.nl/systems/gpu-cluster/description>.
Strang, G., 2007. Computational Science and Engineering. Wellesley-Cambridge Press, pp. 283–288.
Thibault, J.C., Senocak, I. , 2009. CUDA Implementation of a Navier–Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows, in: 47th AIAA Aerospace Sciences Meeting.
Wang, P., Abel, T., Kaehler, R., 2010. Adaptive mesh fluid simulations on GPU. New Astron. 15, 581–589.
Wong, H.C., Wong, U.H., Feng, X., Tang, Z., 2011. Efficient magnetohydrodynamic simulations on graphics processing units with CUDA. Comput. Phys. Commun. 182, 2132–2160.