Geosci. Model Dev., 6, 17–28, 2013
www.geosci-model-dev.net/6/17/2013/
doi:10.5194/gmd-6-17-2013

Geoscientific Model Development

Porting marine ecosystem model spin-up using transport matrices to GPUs

E. Siewertsen[1], **J. Piwonski**[2], **and T. Slawig**[2]

[1]Institute for Computer Science, Christian-Albrechts Universität zu Kiel, 24098 Kiel, Germany
[2]Institute for Computer Science and Kiel Marine Science – Centre for Interdisciplinary Marine Science, Cluster The Future Ocean, Christian-Albrechts Universität zu Kiel, 24098 Kiel, Germany

Correspondence to: T. Slawig (ts@informatik.uni-kiel.de)

Received: 19 July 2012 – Published in Geosci. Model Dev. Discuss.: 31 July 2012
Revised: 7 November 2012 – Accepted: 5 December 2012 – Published: 8 January 2013

Abstract. We have ported an implementation of the spin-up for marine ecosystem models based on transport matrices to graphics processing units (GPUs). The original implementation was designed for distributed-memory architectures and uses the Portable, Extensible Toolkit for Scientific Computation (PETSc) library that is based on the Message Passing Interface (MPI) standard. The spin-up computes a steady seasonal cycle of ecosystem tracers with climatological ocean circulation data as forcing. Since the transport is linear with respect to the tracers, the resulting operator is represented by matrices. Each iteration of the spin-up involves two matrix-vector multiplications and the evaluation of the used biogeochemical model. The original code was written in C and Fortran. On the GPU, we use the Compute Unified Device Architecture (CUDA) standard, a customized version of PETSc and a commercial CUDA Fortran compiler. We describe the extensions to PETSc and the modifications of the original C and Fortran codes that had to be done. Here we make use of freely available libraries for the GPU. We analyze the computational effort of the main parts of the spin-up for two exemplar ecosystem models and compare the overall computational time to those necessary on different CPUs. The results show that a consumer GPU can compete with a significant number of cluster CPUs without further code optimization.

1 Introduction

This work is motivated by the usually huge effort that is needed when computing steady annual cycles (or, mathematically speaking, periodic solutions) of spatially three-dimensional marine ecosystem models. In most cases this is done by "spinning up" the model, i.e. by using a time-stepping algorithm with climatological, periodic forcing data until the steady cycle is reached, at least up to a certain tolerance. This can take a huge number of iterations, in typical cases about 3000 to 5000 model years, each of which involves thousands of time steps (e.g. 2880 steps for a three-hour step-size). Thus the overall number of iterations may be in the range of 10^6 to 10^7. When aiming at parameter optimization or sensitivity studies, the spin-up process has to be repeated several times, and thus in these cases a reduction of the computational time of a single spin-up run is even more important.

There are several strategies to reduce this computational effort. The following ones are more or less independent from each other: one of them is of course parallelization, usually by domain decomposition methods. The second one is the usage of precomputed *transport matrices* (see Khatiwala, 2007) that represent the (possibly linearized) tracer transport scheme applied in an ocean model. Monthly averaged matrices for the explicit and the implicit parts of the ocean tracer transport operator are usually used. In the ecosystem spin-up, these "climatological" matrices are then interpolated accordingly in every time step. With this method, the transport part of the ecosystem model reduces to matrix-vector multiplications, whereas the biogeochemical source-minus-sink terms are evaluated separately. A third way to reduce computational effort is to replace the standard spin-up (which, in mathematically terms, is a fixed-point iteration) by variants of Newton's method, which have higher convergence rates.

In this work we start from an implementation of a spin-up that applies the first two strategies. In order to drive the biogeochemical tracers, the software handles transport matrices that are stored in a common sparse format. Moreover, it uses routines of the Portable, Extensible Toolkit for Scientific Computation (PETSc; Balay et al., 1997, 2012) library to perform matrix-vector multiplications in parallel. The main advantages of this toolkit is that all Message Passing Interface (MPI; Walker and Dongarra, 1996) calls are hidden in built-in functions, and that optimized functions for matrix-vector operations (and more) already exist. The resulting software can be coupled with a wide range of biogeochemical models, as long as they conform to a rather flexible and general interface.

The main focus of this work is to describe the necessary changes to the software to port it to GPU hardware and to determine the resulting speed-up. High-performance computing on GPU or other special, highly parallel hardware is becoming more and more attractive in climate and geophysical research as well (e.g. Hanappe et al., 2011; Horn, 2012). To our knowledge there is no publication about using GPUs for marine ecosystem simulations. Since sparse matrix-vector multiplication (SpMVM) is an integral part of our spin-up implementation, this work is clearly motivated by the performance gains (up to a speedup of 24) achieved by the algorithms presented by Bell and Garland (2008). Moreover, we are interested in the behavior of the incorporated biogeochemical models ported to the GPU. For this purpose, we take here two examples with two tracers each. One of them is a simple linear model, describing for example the radioactive decay of two compounds. The second one is a well-known biogeochemical model that serves as a basis for more complex descriptions of the interplay of ocean biota and its major nutrients. It was used for numerical experiments by Parekh et al. (2005) or Kriest et al. (2010) for example.

Since we want to explicitly show what steps were necessary for the mentioned CPU-to-GPU port, we start by describing the original software for the ecosystem spin-up and the used biogeochemical models in Sect. 2. Afterwards we describe the standards, tools and libraries used for GPU programming in Sect. 3. We then show which GPU-adapted software can be used and what kind of adaption we additionally had to make in Sect. 4. We then show numerical results in Sect. 5 for the two models, both on CPU and GPU hardware. Finally, we conclude our work and give an outlook in Sect. 6.

2 Coupled marine tracer transport simulation using transport matrices

A marine ecosystem is usually modeled as a system of equations for the ocean circulation and the transport of temperature, salinity and the incorporated biogeochemical tracers, including their interactions. A fully coupled simulation – reflecting the fact that tracers are advected by the ocean circulation, their diffusion is dominated by the turbulent mixing of marine water, and, vice versa, a tracer concentration may effect the ocean circulation – is computationally expensive. Even on high-performance hardware, such a coupled (also called "online") simulation in three spatial dimensions is restricted to single model evaluations only, especially if steady annual cycles, which require long term spin-ups, are under investigation.

In contrast, a so-called "offline" computation is a simplified approach for tracers that are (or are regarded as) "passive", i.e. they do not affect the ocean physics, or this influence is neglected. This results in a one-way coupling from the ocean circulation to the tracer dynamics only, where the pre-computed circulation data (advection velocity vector field $\boldsymbol{v}$, mixing coefficient κ, temperature, and optionally salinity) enter the tracer transport equations as forcing.

With this data given, a marine ecosystem model considered in an offline computation consists of the following system of parabolic partial differential equations (here for n tracers y_i summarized in the vector $\boldsymbol{y} = (y_i)_{i=1,\dots,n}$):

$$\frac{\partial y_i}{\partial t} = \nabla \cdot (\kappa \nabla y_i) - \nabla \cdot (\boldsymbol{v}\, y_i) + q_i(\boldsymbol{y}), \quad i = 1, \dots, n, \qquad (1)$$

in the space–time cylinder $\Omega \times [0, T]$ with $\Omega \in \mathbb{R}^3$ being the spatial domain (i.e. the ocean) and $[0, T], T > 0$, the time interval. Here, we neglect the additional dependency on the space and time coordinates $(\boldsymbol{x}, t)$ in the notation for brevity. Additionally, homogeneous Neumann boundary conditions on $\Gamma = \partial\Omega$ for all tracers y_i are imposed. The source-minus-sink or coupling terms q_i in general are nonlinear and represent growth, dying, and tracer interaction. Each of them need not necessarily depend on *all* tracers in $\boldsymbol{y}$, but usually on more than the y_i itself. The q_i also include model parameters (as growth and dying rates, sinking velocities etc.) that are often subject to identification or estimation. They are usually spatially and temporally constant and not mentioned explicitly here.

2.1 Transport matrices

Since in an offline simulation the ocean circulation data is only used as pre-computed input for the tracer transport equations (Eq. 1), the spatial differential operators therein can be represented as a linear operator and the equations can be formally written as

$$\frac{\partial y_i}{\partial t} = L(\kappa, \boldsymbol{v}, t)\, y_i + q_i(\boldsymbol{y}), \quad i = 1, \dots, n. \qquad (2)$$

Here, $L(\kappa, \boldsymbol{v}, t)$ is a linear operator comprising the whole transport, i.e. diffusion and advection, for the given ocean circulation data κ and $\boldsymbol{v}$. It is time-dependent since the circulation data also depend on time, both in case of a transient simulation, and where a steady annual cycle driven by climatological data is sought. The operator L is identical for all tracers if the molecular diffusion of the tracers is small

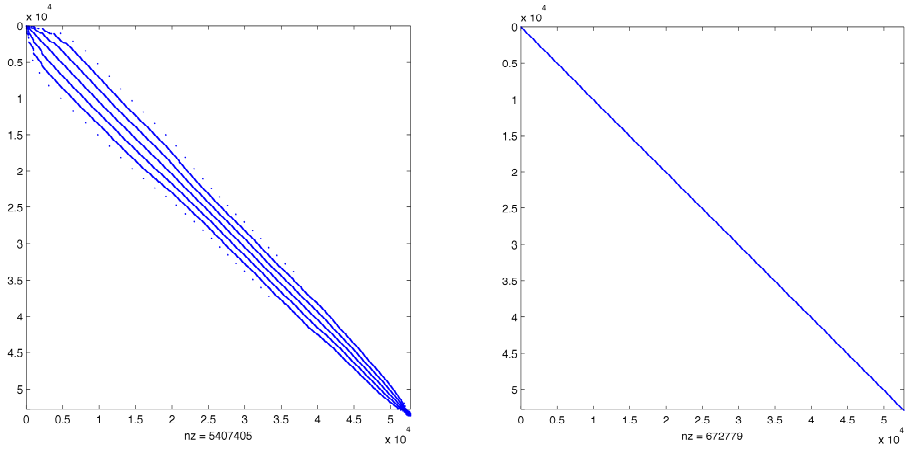

Fig. 1. One block of the explicit (left) and implicit (right) transport matrices $\mathbf{A}_{\text{exp}}$, $\mathbf{A}_{\text{imp}}$ computed using the MITgcm for a 2.8125° resolution (output of MATLAB®'s `spy` command).

compared to the turbulent mixing, which is a reasonable simplification.

The idea of the Transport Matrix Method (TMM) introduced in Khatiwala et al. (2005) is to compute or approximate the matrices that represent an appropriate discretization of L. This is done by running time steps of the ocean model that has produced the circulation data $\boldsymbol{v}, \kappa$ etc., with special, only locally non-zero initial distributions for one tracer. By varying the support of the initial distributions over the whole spatial domain, an approximation for one or several time steps can be obtained, which can be then used to build up a matrix representation of L. A comprehensive discussion of the temporal and spatial discretization as well as the process of evaluating transport matrices, especially in combination with operator splitting schemes can be found in Khatiwala et al. (2005). For our results we used twelve implicit and twelve explicit transport matrices, which represent monthly averaged diffusion and advection. The matrices are interpolated linearly to the corresponding discrete time step during simulation.

As a result, we obtain the following fully (temporal and spatial) discrete scheme where we now denote by $\boldsymbol{y}_j$ the appropriately arranged vector of the values of all n tracers on all spatial grid points at time step j. In the same way, we denote by $\boldsymbol{q}_j$ the vector of the discretized source-minus-sink terms at all spatial grid points in time step j. Using the TMM with a fixed time step-size τ, the time integration scheme for (Eq. 2) reads

$$\boldsymbol{y}_{j+1} = \mathbf{A}_{\text{imp},j}(\mathbf{A}_{\text{exp},j}\,\boldsymbol{y}_j + \tau\,\boldsymbol{q}_j(\boldsymbol{y}_j)) =: \varphi_j(\boldsymbol{y}_j). \tag{3}$$

Here n_τ is the total number of time steps and $\mathbf{A}_{\text{imp},j}, \mathbf{A}_{\text{exp},j}$ are the implicit and explicit transport matrices at time step $j = 0, \ldots, n_\tau - 1$. The matrices are block-diagonal and sparse and depend on the used time-stepping scheme: if – as a simple and unrealistic example – the whole system were solved explicitly by an Euler step, $\mathbf{A}_{\text{imp},j}$ would be the identity and $\mathbf{A}_{\text{exp},j}$ would be the discrete counterpart of $I + \tau L(\kappa, \boldsymbol{v}, t_j)$. Summarizing, starting from a vector $\boldsymbol{y}_0$ of initial values, each step in the time integration scheme (Eq. 3) to solve the tracer transport equations (Eq. 1) consists of the evaluation of the source-minus-sink term and two matrix-vector multiplications per tracer.

Table 1 shows typical values for the sizes and sparsity of transport matrices generated by the MIT General Circulation Model (MITgcm; Marshall et al., 1997) for two spatial resolutions, see Khatiwala et al. (2005); Piwonski and Slawig (2012). Since we deal with quadratic matrices and the sparsity patterns remain the same throughout the whole spin-up process a characterization of the used matrices by the number of rows (`nrows`) and the number of non-zero elements (`nnz`) is sufficient for our purpose. Figure 1 shows the sparsity patterns. The matrix entries are stored as *double precision* values.

2.2 Computation of steady annual cycles

Computing a periodic solution of the discretized system (Eq. 3) means looking for a fixed point of the mapping $\Phi = \varphi_{n_\tau - 1} \circ \cdots \circ \varphi_0$, i.e. for a trajectory $(\boldsymbol{y}_j)_{j=0,\ldots,n_\tau}$ with

$$\boldsymbol{y}_{n_\tau} = \Phi(\boldsymbol{y}_0) = \boldsymbol{y}_0. \tag{4}$$

Thus one application of the mapping Φ corresponds to the computation of one year model time (or model year). The time step used in our computations was 3 h, which corresponds (taking 360 days a year) to $n_\tau = 2880$. The discretization of the biogeochemical terms q_i may include shorter time steps (typically 8 per outer 3-h step).

The whole iteration to compute a steady cycle (or fixed point) now consists of a repeated application of the mapping Φ:

$$\boldsymbol{y}^{l+1} = \Phi(\boldsymbol{y}^l), \quad l = 0, \ldots, n_l - 1, \tag{5}$$

Table 1. Resolution, sizes and sparsity of one block of the explicit and implicit transport matrices for two resolutions computed with the MITgcm.

Horizontal resolution	Vertical layers	Matrix size (nrows)	Number of non-zeros, total (nnz) and percent	
			$\mathbf{A}_{\text{exp}}$	$\mathbf{A}_{\text{imp}}$
2.8125°	15	52 749	5 407 405 (0.1943 %)	672 779 (0.0024 %)
1°	23	682 604	76 567 216 (0.0164 %)	13 339 210 (0.0029 %)

where $\boldsymbol{y}^l$ is the vector of discretized tracer after l model years, i.e. $\boldsymbol{y}^l = \boldsymbol{y}_{l \cdot n_\tau}$, and n_l the total number of model years necessary to reach a steady annual cycle. The resulting structure of the spin-up is sketched in Algorithm 1.

From several computations it can be observed that after about $n_l = 3000$ iterations, a numerical steady solution (up to an accuracy of about 10^{-2} in discrete $L^2(\Omega)^n$ norm) is obtained. Thus we refer to this as a "converged steady annual cycle". This value of n_l was also used in (Kriest et al., 2010). The residual can be further decreased by using a higher number n_l of model years.

2.3 Applying parallel algorithms using the PETSc library

Obviously, a parallelization of the matrix-vector multiplication occurring every time step can significantly speed up the process of computing the steady annual cycle by the pseudo-time stepping (or fixed point iteration) described above. In the CPU setting (e.g. Piwonski and Slawig, 2012) the parallelization is carried out on a multi-processor, distributed-memory architecture. In order to avoid the direct implementation of MPI directives, we make use of the PETSc library. It is a collection of data structures and algorithms for the parallel solution of numerical problems and provides interfaces (APIs) to programming languages as Fortran, C, C++, Python, and MATLAB®. Main advantages of PETSc for our application are the parallelized matrix-vector-multiplication routines and the usage of an efficient sparse matrix storage format, in our case the default PETSc format, namely the "AIJ" or "Yale sparse" or "CSR" (compressed sparse row) format.

In our original implementation, the biogeochemical part (Algorithm 1, line 4) is implemented in Fortran, whereas the remainder of the code is realized in C. There is a difference with respect to the access of the tracer data that becomes important later on the GPU: for the biogeochemical computations (line 4), the values of the separate tracers and also on different spatial grid points (compare Eq. 6) are needed simultaneously. In contrast, the matrix-vector products (lines 6, 7) are executed separately for each tracer, thus allowing us to store and work with one block of the transport matrices only. Each matrix-vector product is computed by one call to the PETSc routine `MatMult()`.

For the interpolation step in line 5, three other PETSc routines are used (for explicit and implicit matrix separately) to compute the appropriately weighted matrices:

```
MatCopy(A[i_alpha], A_work, ...);
MatScale(A_work, alpha);
MatAXPY(A_work, beta, A[i_beta], ...);
```

These three routines together compute a linear interpolant or convex combination of two succeeding monthly averaged matrices, which are stored in the array `A` starting at index `i_alpha` and `i_beta`, respectively. Thus the above lines compute `A_work = alpha * A[i_alpha] + beta * A[i_beta]`, which gives the desired interpolated matrix in `A_work`, if `alpha, i_alpha` and `beta, i_beta` are chosen correctly with respect to the time step j.

2.4 Ecosystem and biogeochemical model examples

We use two simple models to test the computational gain possible with the GPU hardware. Each of them has two tracers (i.e. $n = 2$ in Eq. 1 and thereafter). Source codes for both models are available at Piwonski and Slawig (2012).

The first one is a simple radioactive decay model which is uncoupled and has the autonomous source-minus-sink term

$$\boldsymbol{q}(\boldsymbol{y}) = \begin{pmatrix} -\lambda_1 \, y_1 \\ -\lambda_2 \, y_2 \end{pmatrix}.$$

The parameters $\lambda_1, \lambda_2 > 0$ are the decay rates of the two radioactive elements. We chose Iodine I^{131} with $\lambda_1 \approx 44.88$ and Caesium Cs^{137} with $\lambda_2 \approx 0.0331$. This uncoupled model is used in order to test the gain in CPU time for the pure matrix-vector multiplication and interpolation in the TMM.

The second model is a typical biogeochemical model, including both coupling and nonlinearities. It is based on the N-DOP model described in Parekh et al. (2005), which was also used in Kriest et al. (2010), from which we basically take the notation. The model incorporates phosphate (nutrients, N, y_1) and dissolved organic phosphorus (DOP, y_2). The source-minus-sink term is split up into the upper, sun-lit or productive euphotic zone Ω_1 with depth z', and the lower, aphotic zone Ω_2:

Algorithm 1: Marine ecosystem spin-up using TMM

Require: Set of monthly averaged transport matrices $\mathbf{A}_{\text{imp}}, \mathbf{A}_{\text{exp}}$, initial tracer distribution $\boldsymbol{y}_0$, time step τ
Ensure : At the end $\boldsymbol{y}$ is a tracer distribution (at one point in time) of a steady annual cycle

$\boldsymbol{y} = \boldsymbol{y}_0$
repeat
  for $j = 0, \ldots, n_\tau - 1$ **do**
    compute biogeochemical source-minus-sink terms: $\tilde{\boldsymbol{y}} = \boldsymbol{q}_j(\boldsymbol{y})$
    interpolate the monthly averaged transport matrices to the current time step j
    perform explicit step: $\hat{\boldsymbol{y}} = \mathbf{A}_{\text{exp},j}\,\boldsymbol{y}$
    perform implicit step: $\boldsymbol{y} = \mathbf{A}_{\text{imp},j}(\hat{\boldsymbol{y}} + \tau\,\tilde{\boldsymbol{y}})$
  end
until *steady annual cycle is reached*

$$q_1(\boldsymbol{y}) = \begin{cases} -f(y_1) + \lambda\ y_2 & \text{in } \Omega_1 \\ (1-\sigma)\frac{\partial}{\partial z}F(y_1) + \lambda\ y_2 & \text{in } \Omega_2 \end{cases}$$

$$q_2(\boldsymbol{y}) = \begin{cases} \sigma f(y_1) - \lambda\ y_2 & \text{in } \Omega_1 \\ -\lambda\ y_2 & \text{in } \Omega_2 \end{cases}$$

z being the vertical coordinate. The biological production is calculated as a function

$$f(y_1) = \alpha\ \frac{y_1}{y_1 + K_{\text{N}}}\ \frac{I}{I + K_{\text{I}}}$$

of nutrients y_1 and light I. The dependence on the latter is omitted here in the notation for brevity. The production is limited by a half saturation function, also known as Michaelis-Menten kinetics, and a maximum production rate parameter α. Light is modeled as a portion of shortwave radiation I_{SWR}, which is computed as a function of latitude and season following the astronomical formula of Paltridge and Platt (1976). The portion depends on the photo-synthetically available radiation $\sigma_{\text{PAR}} = 0.4$, the ice cover σ_{ice}, and the exponential attenuation of water, i.e.

$$I = I_{\text{SWR}}\ \sigma_{\text{PAR}}\ (1 - \sigma_{\text{ice}})\ \exp(-z\,K_{\text{H}_2\text{O}}).$$

A fraction σ of the biological production remains suspended in the water column as dissolved organic phosphorus, which remineralizes with a rate λ. The remainder of the production sinks as particulate to depth where it is remineralized according to the empirical power–law relationship determined by Martin et al. (1987),

$$F(y_1) = \left(\frac{z}{z'}\right)^{-b} \int_0^{z'} f(y_1)\,dz. \tag{6}$$

Similar modeling of biological production can be found for example in Dutkiewicz et al. (2005). Algorithm 2 sketches the implementation of the N-DOP model, whereas the model parameters are given in Table 2.

Table 2. Parameters in the N-DOP model.

Name	Description	Unit
λ	remineralization rate of DOP	d^{-1}
α	maximum community production rate	d^{-1}
σ	fraction of DOP	1
K_{N}	half saturation constant of N	mmol P m^{-3}
K_{I}	half saturation constant of light	W m^{-2}
$K_{\text{H}_2\text{O}}$	attenuation of water	m^{-1}
b	sinking velocity exponent	1

3 GPU computing with CUDA

In this section we describe the basic architecture of GPUs and give an overview of some useful libraries. We concentrate on NVIDIA's Compute Unified Device Architecture (CUDA; NVIDIA Corporation, 2012). One alternative is, for example, OpenCL (The Khronos Group, 2012).

NVIDIA, as one of the leading producers of graphic cards, has developed its own parallel architecture for executing computationally expensive code on GPUs. By exploiting the architecture of graphic cards as well as the increased memory bandwidth, it is possible to perform a far greater number of floating point operations per second (FLOPS) than on CPUs. While CPUs have about one to eight cores each with up to 4 GHz clock rate, GPUs nowadays do have a lower clock rate, but hundreds of cores which can run multiple threads simultaneously.

The basic unit of the CUDA programming model is called *kernel*. A kernel is a piece of program code invoked on the CPU *host* and executed on the GPU *device* by threads. These threads are organized in a "grid" of thread "blocks". A call to

```
kernel<<<gridSize, blockSize>>>();
```

creates `gridSize` blocks of `blockSize` threads ready for execution, whereas the order of processing the blocks depends on the hardware.

Algorithm 2: Computation of $\tilde{\boldsymbol{y}} = \boldsymbol{q}_j(\boldsymbol{y})$ for the N-DOP model.

Require: Tracer vectors $\boldsymbol{y}$, latitude ϕ, ice cover σ_{ice}, depths z, layer heights dz and parameters: $\lambda, \alpha, \sigma, K_{\text{N}}, K_{\text{H}_2\text{O}}, K_{\text{I}}, b$
Ensure : $\tilde{\boldsymbol{y}}$ consists of the computed sinks and sources

for *every water column i with* n_i *layers* **do**
  $I = 0.4 * (1 - \sigma_{\text{ice},i}) * I_{\text{SWR}}(\phi_i)$ // compute insolation
  $\tilde{\boldsymbol{y}} = 0$ // zero all bio steps
  for *8 biosteps* **do**
    $\boldsymbol{y}' = \boldsymbol{y} + \tilde{\boldsymbol{y}}$ // take previous steps into account
    $\tilde{\boldsymbol{y}}' = 0$ // zero one bio step
    for *layer* $j = 1$ *to* $\min(n_i, 2)$ **do** // production layers
      $I_j = I * \exp(-z_j \; K_{\text{H}_2\text{O}})$
      $f_j = \alpha * y'_{1,j} * I_j / (y'_{1,j} + K_{\text{N}}) / (I_j + K_{\text{I}})$
      $\tilde{y}'_{1,j} = \tilde{y}'_{1,j} - f_j$
      $\tilde{y}'_{2,j} = \tilde{y}'_{2,j} + \sigma * f_j$
      if *last layer* **then**
        $\tilde{y}'_{2,j} = \tilde{y}'_{2,j} + (1 - \sigma) * f_j$
      else
        for *every layer k beneath* **do** // approximation of dF/dz
          if *last layer* **then**
            $\tilde{y}'_{2,j} = \tilde{y}'_{2,j} + (1 - \sigma) * f_j * dz_j * (z_{k-1}/z_j)^{-b} / dz_k$
          else
            $\tilde{y}'_{2,j} = \tilde{y}'_{2,j} + (1 - \sigma) * f_j * dz_j * ((z_{k-1}/z_j)^{-b} - (z_k/z_j)^{-b}) / dz_k$
          end
        end
      end
    end
    for *layer* $j = 1$ *to* n_i **do** // all layers
      $\tilde{y}'_{1,j} = \tilde{y}'_{1,j} + \lambda * y'_{2,j}$
      $\tilde{y}'_{2,j} = \tilde{y}'_{2,j} - \lambda * y'_{2,j}$
    end
    $\tilde{\boldsymbol{y}} = \tilde{\boldsymbol{y}} + 1/8 * \tilde{\boldsymbol{y}}'$ // scale and add to all bio steps
  end
end

The GPU hardware consists of several Streaming Multiprocessors (SMs). Each SM has its own buffer memory, registers, and a number of cores. The cores have their own units for integer and floating-point calculation. For example, the GeForce GTX 480 used here has 15 SMs with 32 cores each, i.e. a total of 480 cores. On a core, the smallest executable unit is a "warp", which consists of 32 threads. The total number of threads that can run simultaneously on a multiprocessor is dependent on the Compute Capability (CC) of the graphics chips. For the GTX 480 the limit is 1536 threads, which results in a maximum number of concurrent threads for the entire GPU of $15 \times 1536 = 23\,040$ (p. 159, NVIDIA Corporation, 2011).

The device memory on the GPU is divided into three types of physical and virtual portions. At first, a thread has access to its own private memory which is, depending on the CC, between 16 kB and 512 kB. Secondly, threads within one block have access to a shared memory of between 16 kB and 48 kB. Finally, all threads have access to a shared global memory whose size is limited by the total amount of memory of the GPU. In order to run kernel code on the GPU, all data must be transferred from the host memory of the CPU to the device memory on the GPU.

NVIDIA provides a compiler (`nvcc`) that translates C code into the CUDA Instruction Set (called PTX) and behaves similarly to the C compiler (`gcc`) included in the GNU Compiler Collection (GCC). A port of the GNU debugger `gdb` is also included in the CUDA toolkit.

3.1 Libraries

We make use of libraries that provide basic algorithms while working with GPUs. The first one is: Thrust (Bell and Hoberock, 2011), a C++ collection of generic algorithms, similar to the C++ Standard Template Library (STL), that exploit the parallelism of the GPU in a transparent way. Using Thrust, many problems can be solved without even writing code for

the GPU. For documentation and sample code we refer to (Hoberock and Bell, 2012).

The second library is Cusp (Bell and Garland, 2010), which provides data types for sparse matrices and algorithms for basic linear algebra operations on them. All data structures in Cusp have a parameter that determines whether it is stored in CPU or GPU memory. Operations on the data will then take place in the respective storage area. For our application, in particular the structure `cusp::csr_matrix` for the CSR format and the matrix-vector multiplication routine `cusp::multiply` that uses the algorithm described in Bell and Garland (2008, 2009), which was specially developed for GPUs, are important. Documentation and sample code can be found at (Bell and Garland, 2010).

The third library we used was the preliminary implementation of PETSc for the CUDA architecture presented in Minden et al. (2010). With the help of the Thrust and Cusp libraries, a large part of the PETSc `Vector` and some parts of the `Matrix` class have been implemented. The fundamental problems of interaction of PETSc with the GPU have been resolved, but only the routines that were necessary for the example treated in Minden et al. (2010) have been implemented. Basically this "PETSc GPU" extends the built-in structures by a value that indicates in which memory the most recent data are stored. This guarantees that the correct data is available (and if necessary copied to) the memory that is currently used. Here, we employed the developer PETSc library version 3.2-p5.

4 Port of the marine ecosystem simulation onto the GPU

We now describe the necessary modifications and extensions of the original program that was running on a multi-processor CPU cluster in order to perform the simulation on a GPU. Basically these modifications are extensions of PETSc GPU, modifications necessary to use the CUDA Fortran compiler for the biogeochemical model code and some routines for conversions between different data alignments.

4.1 Necessary extensions of PETSc GPU

The preliminary PETSc GPU implementation was designed to solve systems of equations, and thus not all functions necessary for our applications were included. To avoid any copying of data between CPU and GPU storage that would have destroyed the speed-up, we had to extend the library. In our case, the three PETSc routines `MatCopy`, `MatScale`, and `MatAXPY` mentioned in Sect. 2.2 had to be modified.

If using sparse matrices with PETSc and working with GPUs the PETSc wrapper function

```
MatCopy(Ain, Aout, ...);
```

accesses `MatCopy_SeqAIJCUSP()` to copy the values from matrix `Ain` to `Aout`. Here, it is theoretically possible that both matrices are either currently in the GPU memory, the CPU memory or in both. For a complete and correct implementation, it would have been necessary to cover all these cases, and accordingly select the memory the matrices are actually copied to. For our application it was sufficient to cover only the case where the matrices are both in the GPU memory, thus only this case was implemented. Therefore, an additional call to `MatCUSPCopyToGPU()` in `MatCopy_SeqAIJCUSP()` ensures that both matrices are in the GPU memory.

The PETSc routines `MatScale()` and `MatAXPY()` implement typical linear algebra subproblems, which are only performed on the non-zero matrix elements. Consequently, they could be completely realized using the Cusp BLAS library for the GPU.

4.2 PGI CUDA-Fortran

Many biogeochemical models are implemented in Fortran. Mostly, they are part of a software that has evolved over decades (e.g. MITgcm). Since we want to use them with GPUs without any modification to original source code, we need a Fortran compiler and the appropriate libraries. At the time of this work there was only one candidate, namely the PGI CUDA Fortran compiler (The Portland Group, 2012). It extends the language by constructs for calling kernel as well as the CUDA API functions. The syntax of a kernel call in Fortran is

```
call kernel<<<gridSize, blockSize>>>()
```

and thus similar to CUDA C++ . There are some extensions compared to CUDA C++, but also some restrictions. For details we refer to the manual (The Portland Group, 2011a, p. 14).

4.3 Other extensions to the implementation on the CPU

As mentioned in Sect. 2.3, there are two different data alignments useful for the spin-up using the TMM: one for the biogeochemical source-minus sink terms, where all tracers of a water column are kept in a contiguous piece of memory, and another one for the multiplication with the transport matrices, where every water column of a tracer is kept together to reduce the storage requirements for the matrices. Thus a copying between these two data alignments is necessary in every step of the algorithm. For the use on the GPU, three copying functions in the original code were additionally modified using the Thrust library.

4.4 The compilation process for the GPU

Here we briefly sketch the overall compilation and linking process of the resulting code for the use on the GPU. The process is visualized in Fig. 2.

In a first step (top right in Fig. 2) the biogeochemical model file `model.F` is included into `driver.CUF`

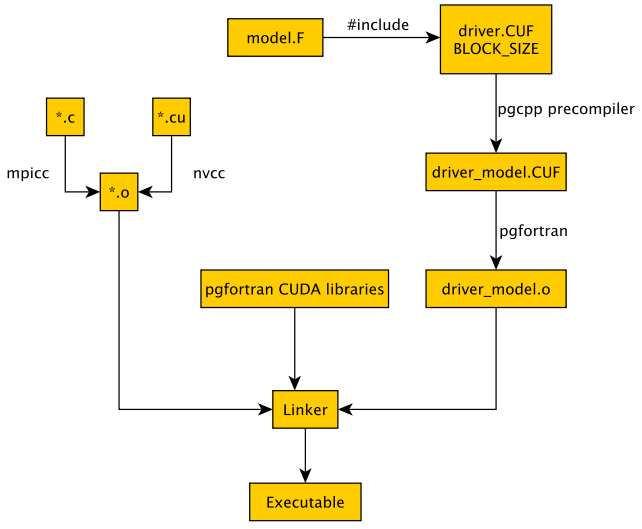

Fig. 2. Compilation and linking process of the spin-up for usage on the GPU.

and processed to `driver_model.CUF` by the preprocessor of the C++ compiler `pgcpp`. The Fortran compiler `pgfortran` then generates the object file `driver_model.o`.

The driver routine `driver.CUF` has two tasks: at first the Fortran compiler requires that all functions which shall run on the GPU are marked with the `device` attribute, see The Portland Group (2011b). Since the compiler has no ability to set default attributes for all functions, it is necessary to integrate them through a preprocessor macro. Therein the Fortran keyword `subroutine` is replaced by `attributes(device) subroutine`. Secondly, the driver provides support functions for the three entry points into the biogeochemical model, namely (i) the evaluation of the source-minus-sink term, (ii) the initialization and (iii) deinitialization of the model. These three functions need corresponding kernels for the GPU. This approach ensures the original Fortran interface of the biogeochemical model remains unaltered.

In a second step (top left of Fig. 2) the original, unmodified C code is compiled with the MPI wrapper of the GNU C compiler `mpicc`, while CUDA extensions are translated with `nvcc`. Finally all object code files are linked against PGI Fortran libraries, which results in the final executable.

5 Numerical results

In this section we compare the performance of the spin-up on our CPU/GPU test hardware. We use the two models described in Sect. 2.4. A special emphasis lies on the time needed for the individual parts, namely the evaluation of the biogeochemical source minus-sink term, the matrix interpolation and the matrix-vector multiplication. Moreover, we

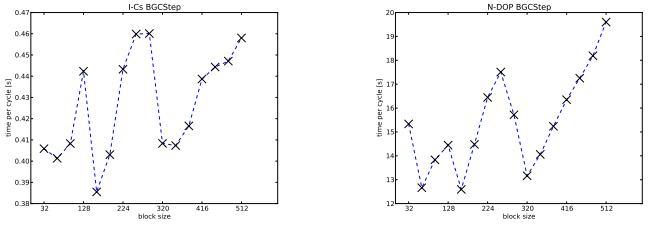

Fig. 3. Computational time needed for the I-Cs (left) and N-DOP (right) model within one model year depending on the block size.

contrast the best GPU result for the N-DOP model with results from three different distributed-memory architectures.

5.1 Setup

The CPU/GPU test hardware consists of two GeForce GTX 480 graphic cards and two Intel® Xeon® E5520 CPUs running at 2.27 GHz. However, the following tests were performed only on *one* GPU and only on *one core* of the CPU. No display was connected to the graphic card and computations on the GPU were performed with *double precision*, which is natively supported by the GTX 480. The theoretical peak performance of the GPU is at 168 GFlop s^{-1} and the internal bandwidth at 177 GB s^{-1}. The performance of one core of the CPU system is at 9.08 GFlop s^{-1}, its bandwidth at 21.2 GB s^{-1}.

To test a specific biogeochemical model, the software is compiled with the according source code and run for 100 model years. In detail, when the executable starts the data (matrices, initial vectors, etc.) is copied into the CPU or GPU memory and 100 iterations, 2880 time steps each, are performed consecutively. In the case of a GPU run, the results are copied back to CPU memory at the end.

Thus, the whole data has to fit into the memory of the device (or host). This is the case if the 2.8125° horizontal resolution is used. Here, the 1.5 GB RAM of a GTX 480 (or 40 GB of the CPU system) are enough for about 1 GB of data. However, a monthly averaged set of transport matrices based on a 1° resolution (approximately 13 GB) is too large for the used GPU system. Such an amount of data requires a different approach (see Sect. 6). Hence, we focus on the 2.8125° resolution and omit profiling of data transfers between CPU and GPU memory.

When processing source codes, the `mpicc`, `mpif90` and `nvcc` compilers are switched to `-O` (i.e. optimize). For `pgfortran` no optimization flags are used. To perform time measurements, the profiling system of PETSc is applied. No further source code optimization is performed regarding the GPU.

Table 3. Minimum, maximum, average and standard deviation of computational time for one model year spent on the CPU and GPU. Shown are results of 100 model years, each year timed separately. `BGCStep` block size: 160.

	CPU				GPU				
Model	Min	Max	Avg	StdDev	Min	Max	Avg	StdDev	CPU Min : GPU Min
I-Cs	159.58 s	161.44 s	160.19 s	0.47	15.49 s	15.52 s	15.50 s	0.002	10.30
N-DOP	621.43 s	626.79 s	622.14 s	0.54	28.17 s	28.20 s	28.18 s	0.003	22.06

Table 4. The three main portions in every time step of the spin-up.

Lines in Alg. 1	Routine	Description
4	`BGCStep`	Evaluation of source-minus-sink terms
5	`MatCopy`, `MatScale`, `MatAXPY`	Interpolation of transport matrices
6, 7	`MatMult`	Multiplication of transport matrices with tracer vectors

5.2 Results

We start by examining the block size parameter for the Fortran kernel calls of the biogeochemical model. The block size describes the number of vertical profiles (or water columns) that are processed within a block. While the grid and block dimensions are calculated automatically, if using Thrust or Cusp for example, a suitable value for the Fortran kernel must be determined experimentally for the time being. For all tests we use just 100 model years (instead of 3000 or more needed in practice, see Sect. 2.2) to render the numerical experiments feasible, especially when simulating the N-DOP model on the CPU, which still takes about 17 h.

Figure 3 depicts the mean of 100 model years' computational time spent on the GPU for biogeochemical model steps depending on the block size. In both models, strong fluctuations up to 100 % occur. However, both graphs show similar occurrence of minima and maxima. We suppose this is due to the unbalanced distribution of water columns (see Sect. 6). However, the absolute minimum (I-Cs: 0.38 s, N-DOP: 12.6 s) is obtained for a block size of 160.

This value is used for the subsequent test, in which every year is timed separately. Table 3 shows the minimum, maximum and mean of computational time for one model year spent on the CPU and GPU. The standard deviation is small on the CPU (I-Cs: 0.47, N-DOP: 0.54) and marginal on the GPU (I-Cs: 0.002, N-DOP: 0.003). However, the overall reduction is about 10 for the simpler I-Cs model and about 22 for the more complex N-DOP model, a difference we investigate further.

Thus, the next tests focus on the individual steps within the repeat-until loop of Algorithm 1, corresponding to one annual cycle. The invoked routines are listed in Table 4. Their individual performance gain is depicted in Table 5. Regarding `MatCopy`, `MatScale`, `MatAXPY` and `MatMult`, we see a similar relative performance gain for both models from about 9 to 13. In contrast, `BGCStep` shows a speed-up of about 10 for the I-Cs model, whereas for N-DOP a ratio between the CPU and GPU of 36 can be observed. In addition, in Fig. 4 we recognize that 75 % of the overall time on the CPU, which is spent for the evaluation of the N-DOP model, is sped up by this factor on the GPU. This explains the overall ratio of 22. Note that the slightly higher average computational times in Fig. 4 (compared to those in Table 3) are due to the higher granularity of profiling. Moreover, we see that the computational effort for the I-Cs model, which is just a scaling of the tracer vector, is smaller than 3 % on both architectures. Here, the overall speed up is dominated by the matrix operations.

Concerning the latter, we pick `MatMult` for a detailed view on performance and bandwidth and compare our results with those reported by Bell and Garland (2008). We calculate the number of floating point operations for one model year as follows:

$$n_{\text{ops}} = n_\tau * 2 * (2 * \mathtt{nnz}_{\text{exp}} + 2 * \mathtt{nnz}_{\text{imp}}) \approx 70\ \text{GFlop}\,,$$

which is the number of time steps per year *times* number of tracers *times* (explicit plus implicit) sparse matrix-vector multiplication, which is exactly twice the number of nonzeros. We consider the results from the N-DOP model and divide n_{ops} by 58.19 s (CPU) and 5.87 s (GPU), respectively. We obtain a performance of approximately $1.2\,\text{GFlop}\,\text{s}^{-1}$ for the CPU and $11.9\,\text{GFlop}\,\text{s}^{-1}$ for the GPU. This is about 13 % of the theoretical peak performance of one CPU core ($9.08\,\text{GFlop}\,\text{s}^{-1}$) and about 7 % of $168\,\text{GFlop}\,\text{s}^{-1}$, regarding the GPU. The poor performance is due to the bandwidth limitation, which is typical for sparse matrix-vector multiplications. Following Bell and Garland (2008), we multiply n_{ops} by $10\,\text{Byte}\,\text{Flop}^{-1}$ (CSR vector kernel) and relate the result to the computational time spent on the CPU and GPU, respectively. We obtain 56.8 % ($12\,\text{GB}\,\text{s}^{-1}$) of the theoretical bandwidth for the CPU and 67.4 % ($119.4\,\text{GB}\,\text{s}^{-1}$) for the GPU.

Table 5. Mean computational time within one model year and performance gains of the individual routines depicted in Table 4.

	I-Cs			N-DOP		
Routine	CPU	GPU	CPU : GPU	CPU	GPU	CPU : GPU
`BGCStep`	3.79 s	0.38 s	9.93	469.76 s	13.05 s	36.00
`MatCopy`	34.52 s	3.91 s	8.83	34.04 s	3.91 s	8.70
`MatScale`	23.83 s	1.99 s	11.96	23.33 s	1.99 s	11.70
`MatAXPY`	38.71 s	2.89 s	13.39	37.49 s	2.89 s	12.96
`MatMult`	60.04 s	5.87 s	10.22	58.19 s	5.87 s	9.92

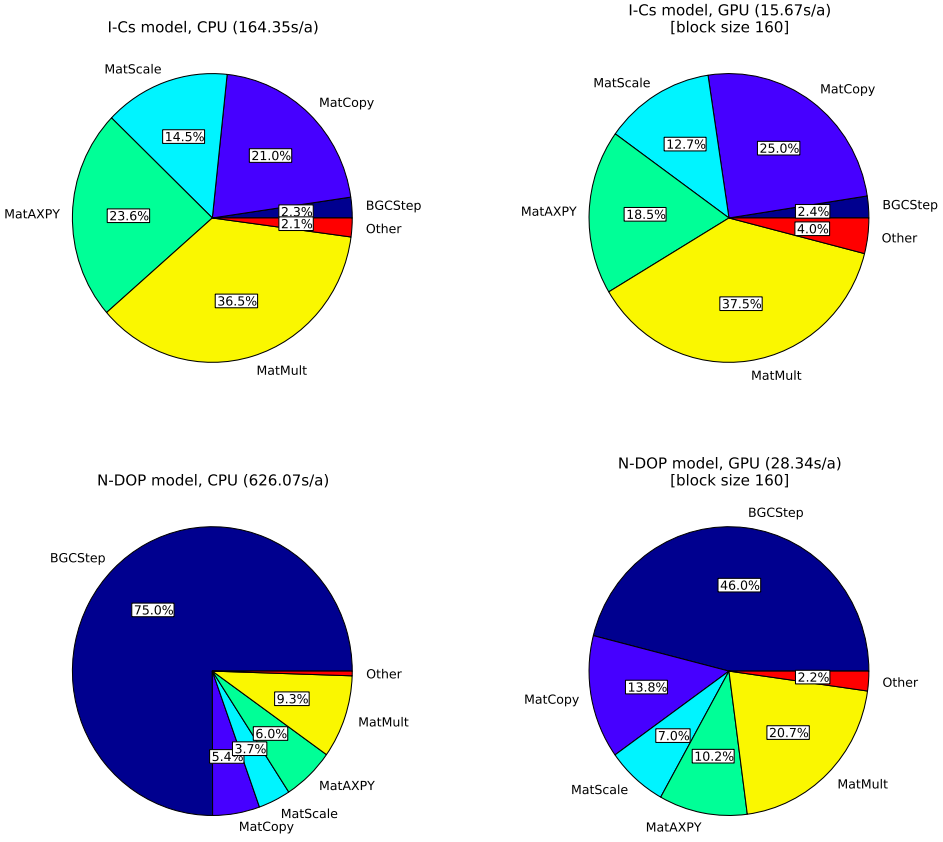

Fig. 4. Fraction of computational time needed for the individual parts in one year of the spin-up (Algorithm 1 and Table 4) for the I-Cs (top) and the N-DOP (bottom) model on the CPU (left) and GPU (right).

These figures in turn are satisfying and confirm a good performance of the CSR vector kernel used by `MatMult`. However, they also show that a sparse matrix-vector multiplication on a GTX 480, which is two generations ahead of the GTX 280 used by Bell and Garland (2008), is only slightly faster. Here, we refer to the 10 GFlop s^{-1}, achieved by the GTX 280 for "unstructured" matrices, compared to the 11.9 GFlop s^{-1} achieved by the GTX 480 for the transport matrices. This is obviously due to the only slightly increased memory bandwidth from 141.7 GB s^{-1} (GTX 280) to 177 GB s^{-1} (GTX 480).

Nevertheless, motivated by the overall speed up, we perform simulations of the N-DOP model on three different CPU clusters and put them in relation to the best performance on the GPU as a last comparison. Figure 5 shows that a GTX 480 can compete with approximately 56 Barcelona, 28 Westmere, and 17 Gainestown processors.

6 Conclusions

In order to port our existing implementation of the spin-up of marine ecosystem models using transport matrices from CPU to GPU hardware, modifications of our own code and

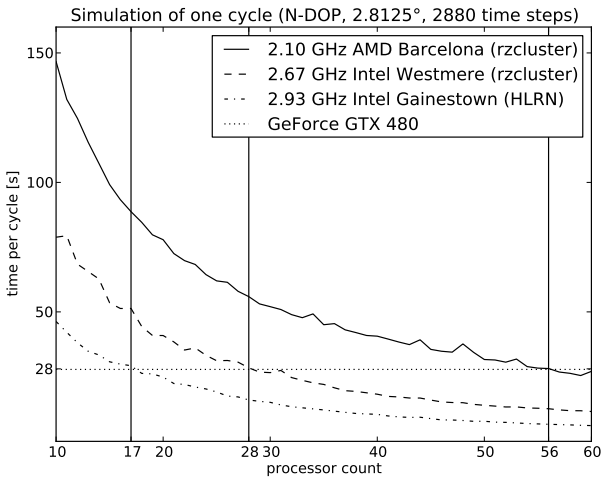

Fig. 5. Comparison between CPU cluster and the used GPU for one model year for the N-DOP model, ("rzcluster" refers to the Kiel University cluster, "HLRN" to the cluster of the *North-German Supercomputing Alliance*).

extensions to the used libraries were necessary. This work required knowledge in the computing architecture of the used CUDA programming framework and the PETSc, Thrust and Cusp libraries. In order to compile Fortran code for the GPU, a commercial compiler was necessary.

Concerning the computational gain of the used biogeochemical models, we were surprised by the good performance of the N-DOP implementation. Here, we can only speculate about the reasons and see a need for a more detailed investigation. Considering the complexity of Algorithm 2, however, such an effort was out of the scope of this work. We thus reported only results here.

Regarding `MatMult`, we observed a similar good utilization of memory bandwidth by the CSR vector kernel for transport matrices as reported by Bell and Garland (2008) for "unstructured" matrices. Moreover, all matrix operations showed a satisfactory performance gain.

Our results motivate us to investigate other biogeochemical models and to get to the bottom of the significantly higher speed-up of the N-DOP model compared to other operations. Additionally, we are eager to prepare the code for usage with multiple GPUs and/or techniques of simultaneous copying and computing.

Acknowledgements. The GPU computations were performed with kind permission and support of the Research Group for Communication Systems at Christian-Albrechts-Universität of Kiel, Institute for Computer Science. The access to the HLRN computing facility was kindly provided by the group Marine Biogeochemical Modelling at IfM GEOMAR, Kiel, Germany. Moreover, the authors would like to thank two anonymous reviewers for their very helpful comments.

Edited by: D. Ham

References

Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F.: Efficient Management of Parallelism in Object Oriented Numerical Software Libraries, in: Modern Software Tools in Scientific Computing, edited by: Arge, E., Bruaset, A. M., and Langtangen, H. P., Birkhäuser Press, 163–202, 1997.

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H.: PETSc Web page, available at: http://www.mcs.anl.gov/petsc (last access: 7 January 2013), 2012.

Bell, N. and Garland, M.: Efficient sparse matrix-vector multiplication on CUDA, Technical Report NVR-2008-04, NVIDIA Corporation, 2008.

Bell, N. and Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, New York 2009, ACM, 1–11, 2009.

Bell, N. and Garland, M.: Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, available at: http://cusp-library.googlecode.com (last access: 7 January 2013), 2010.

Bell, N. and Hoberock, J.: Thrust: A Producivity-Oriented Library for CUDA, GPU Computing Gems Jade Edition, 2011.

Dutkiewicz, S., Follows, M., and Parekh, P.: Interactions of the iron and phosphorus cycles: A three-dimensional model study, Global Biogeochem. Cy., 19, 1–22, 2005.

Hanappe, P., Beurivé, A., Laguzet, F., Steels, L., Bellouin, N., Boucher, O., Yamazaki, Y. H., Aina, T., and Allen, M.: FAMOUS, faster: using parallel computing techniques to accelerate the FAMOUS/HadCM3 climate model with a focus on the radiative transfer algorithm, Geosci. Model Dev., 4, 835–844, doi:10.5194/gmd-4-835-2011, 2011.

Hoberock, J. and Bell, N.: Thrust, available at: http://thrust.github.com (last access: 7 January 2013), 2012.

Horn, S.: ASAMgpu V1.0 – a moist fully compressible atmospheric model using graphics processing units (GPUs), Geosci. Model Dev., 5, 345–353, doi:10.5194/gmd-5-345-2012, 2012.

Khatiwala, S.: A computational framework for simulation of biogeochemical tracers in the ocean, Global Biogeochem. Cy., 21, GB3001, doi:10.1029/2007GB002923, 2007.

Khatiwala, S., Visbeck, M., and Cane, M.: Accelerated simulation of passive tracers in ocean circulation models, Ocean Modell., 9, 51–69, 2005.

Kriest, I., Khatiwala, S., and Oschlies, A.: Towards an assessment of simple global marine biogeochemical models of different complexity, Prog. Oceanogr., 86, 337–360, doi:10.1016/j.pocean.2010.05.002, 2010.

Marshall, J., Adcroft, A., Hill, C., Perelman, L., and Heisey, C.: A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers, J. Geophys. Res., 102, 5753–5766, 1997.

Martin, J. H., Knauer, G. A., Karl, D. M., and Broenkow, W. W.: VERTEX: carbon cycling in the northeast Pacific, Deep Sea Res. Part A, 34, 267–285, 1987.

Minden, V., Smith, B., and Knepley, M.: Preliminary implementation of PETSc using GPUs, in: Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering, Harbin Nov. 2010, Springer, 2010.

NVIDIA Corporation: CUDA C Programming Guide, available at: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (last access: 7 January 2013), 2011.

NVIDIA Corporation: CUDA, available at: http://www.nvidia.com/object/cuda_home_new.html (last access: 7 January 2013), 2012.

Paltridge, G. W. and Platt, C. M. R.: Radiative Processes in Meteorology and Climatology, Elsevier, New York, 1976.

Parekh, P., Follows, M. J., and Boyle, E. A.: Decoupling iron and phosphate in the global ocean, Global Biogeochem. Cy., 19, GB2020, doi:10.1029/2004GB002280, 2005.

Piwonski, J. and Slawig, T.: Metos3D: A Marine Ecosystem Toolkit for Optimization and Simulation, CAU Kiel, Institut für Informatik, available at: https://github.com/metos3d (last access: 7 January 2013), 2012.

The Khronos Group: OpenCL – The open standard for parallel programming of heterogeneous systems, available at: http://www.khronos.org/opencl/ (last access: 7 January 2013), 2012.

The Portland Group: PGI Fortran Compiler Reference Manual, available at: http://www.pgroup.com/doc/pgiref.pdf (last access: 7 January 2013), 2011a.

The Portland Group: CUDA Fortran Programming Guide and Reference, available at: http://www.pgroup.com/doc/pgicudafortug.pdf (last access: 7 January 2013), 2011b.

The Portland Group: PGI CUDA-Fortran Compiler, available at: http://www.pgroup.com/resources/cudafortran.htm (last access: 7 January 2013), 2012.

Walker, D. W. and Dongarra, J. J.: MPI: A Standard Message Passing Interface, Supercomputer, 12, 56–68, 1996.