A Fast GPU-accelerated Mixed-precision Strategy for Fully Nonlinear Water Wave Computations

Stefan L. Glimberg, Allan P. Engsig-Karup, and Morten G. Madsen

Abstract We present performance results of a mixed-precision strategy developed to improve a recently developed massively parallel GPU-accelerated tool for fast and scalable simulation of unsteady fully nonlinear free surface water waves over uneven depths (Engsig-Karup et.al. 2011). The underlying wave model is based on a potential flow formulation, which requires efficient solution of a Laplace problem at large-scales. We report recent results on a new mixed-precision strategy for efficient iterative high-order accurate and scalable solution of the Laplace problem using a multigrid-preconditioned defect correction method. The improved strategy improves the performance by exploiting architectural features of modern GPUs for mixed precision computations and is tested in a recently developed generic library for fast prototyping of PDE solvers. The new wave tool is applicable to solve and analyze large-scale wave problems in coastal and offshore engineering.

1 Introduction

Recent development significantly improves the strategy proposed by Li & Fleming in 1997 [10] to simulate fully nonlinear water waves. A flexible-order finite difference algorithm for solving the governing equations in two (Bingham & Zhang [4]) and three space dimensions (Engsig-Karup, Bingham & Lindberg [7]) enables ef-

Stefan L. Glimberg
Department of Informatics and Mathematical Modelling, Technical University of Denmark, 2800 Kgs Lyngby, Denmark, e-mail: slgl@imm.dtu.dk

Allan P. Engsig-Karup
Department of Informatics and Mathematical Modelling, Technical University of Denmark, 2800 Kgs Lyngby, Denmark, e-mail: apek@imm.dtu.dk

Morten G. Madsen
Department of Informatics and Mathematical Modelling, Technical University of Denmark, 2800 Kgs Lyngby, Denmark, e-mail: morten.gorm.madsen@gmail.com

ficient, scalable and low-storage solution of the equations. Recent developments in modern many-core hardware and programming tools for general-purpose scientific computing, suggest that a combination could further improve the overall performance.

In recent work [8], we have demonstrated that it is now possible to significantly reduce the barriers for practical use of full potential flow theory as the modeling basis for efficient solution of coastal and offshore engineering problems. Our strategy was to do proof-of-concept by utilizing modern Graphics Processing Units (GPUs) for massively parallel computations using a heterogeneous CPU-GPU hardware setup. Interestingly, such a hardware setup constitutes what can be considered an affordable standard consumer desktop environment.

To establish the model as an efficient massively parallel tool we have both redesigned and reimplemented the entire algorithm using a newly developed library for PDE solver proto-typing. The library enables efficient utilization of allocated hardware resources, targeting modern many-core GPUs. Algorithmic efficiency is achieved by solving the computational bottleneck problem iteratively with a defect correction method, preconditioned by a robust multigrid method. This strategy gives more than one order of magnitude in both problem size and practical speedup (relative to optimized single-threaded CPU code).

1.1 Governing Equations

We present recent progress on the development of the OceanWave3D model[4, 7]. In short, the flexible-order finite difference OceanWave3D model is based on a unified potential flow formulation. These model equations can account for fully nonlinear and dispersive waves within the breaking limit and under the assumption of irrotational inviscid flow. The temporal derivatives for the surface variables, i.e. the free surface elevation η and the velocity potential $\tilde{\phi}$ is given by

$$\partial_t \eta = -\nabla\eta \cdot \nabla\tilde{\phi} + \tilde{\omega}(1+\nabla\eta\cdot\nabla\eta) \tag{1}$$

$$\partial_t \tilde{\phi} = -g\eta - \frac{1}{2}(\nabla\tilde{\phi}\cdot\nabla\tilde{\phi} - \tilde{\omega}^2(1+\nabla\eta\cdot\nabla\eta)), \tag{2}$$

where $\nabla = [\partial_x\, \partial_y]^T$, $\tilde{\omega} = \partial_z\phi|_{z=\eta}$ and g is the gravitational acceleration. In order to integrate these equations in time, the vertical velocity on the surface $\tilde{\omega}$, must be determined from the full potential inside the domain. The following Laplace equation along with boundary conditions uniquely defines the full velocity potential

$$\begin{aligned} \phi &= \tilde{\phi}, \quad && z = \eta \\ \nabla^2\phi + \partial_{zz}\phi &= 0, \quad && -h \le z < \eta \\ \partial_z\phi + \nabla h\ \cdot \nabla\phi &= 0, \quad && z = -h, \end{aligned} \tag{3}$$

Where h is the still water depth. Notice that the Laplace problem is of three dimensions, whereas the surface time integration is only of two dimensions. Thus, the computational effort to solve the discretized Laplace problem (3) is the most time consuming part of a numerical solver for this problem. In the following we focus on the numerical approach to solve (3) efficiently on many-core GPUs. In practice we actually solve the so-called σ-transformed version of (3), in order to avoid time changing domains and variable finite difference coefficients from approximating the derivatives. See [10] or [7] for details on the transformed equations.

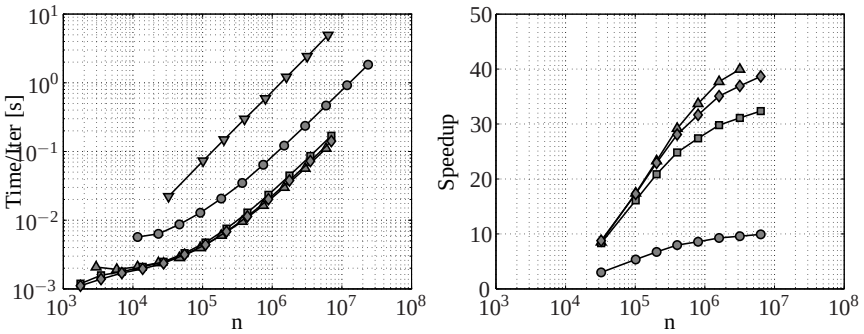
2 Development of a Massively Parallel Wave Analysis Tool

The flexible-order finite difference scheme presented by Engsig-Karup, Bingham, and Lindberg [7] was originally implemented as a stand-alone serial code. The tool was referred to as OceanWave3D. In a recent proof-of-concept study the algorithmic strategy for the OceanWave3D model was first improved and then a massively parallel implementation was carried out and tested on a single GPU [8] with significant performance improvements. The flexible-order finite difference operators was implemented as matrix-free compact stencil operators, in order to further minimize the memory overhead of storing identical entries and avoiding the extra index tables required by traditional sparse matrix formats. Fig. 1 is replicated from [8] and illustrates linear scalability of absolute timings as the problem size increases along with speedups relative to optimized single-threaded CPU code. Recently, a library for high-performance PDE solver proto-typing has been established and the OceanWave3D strategy was again transferred to this new library. The existing dedicated GPU implementation has been used as a reference, to ensure no significant performance loss using the new high-level library. A short outline of the library is presented next.

2.1 A Library for Fast PDE Solver Proto-typing

Our generic high-performance C++ library is subject to ongoing development and improvements within our research group. The purpose is to enable fast proto-typing of efficient massively parallel solvers, inspired by the PETSc toolkit library [2]. Our library facilitates massively parallelization through GPU computing and contains components for various iterative strategies for solution of large linear systems. The goal has been to create a portable and reusable framework without losing noticeable performance – a common tradeoff between generality and dedicated solvers.

The generic nature of our library enables the end users to easily change solver parts through type bindings. The backbone of the library is a generic vector class. It takes two template parameters to define the container type along with a memory space identifier, inspired by the Thrust and Cusp GPU libraries [9, 3]. The following



(a) Absolute timings. (b) Speedup relative to CPU (single thread).

Fig. 1 Scalability tests and performance comparisons in double precision arithmetic for Quadro FX 5800 (−●−), GeForce GTX 480 (−▲−), C2050 with ECC (−■−) and C2050 without ECC (−♦−) versus CPU (single thread) code (−▼−). Sixth order spatial discretization employed. The iterative defect correction method has been left-preconditioned with a Zebra Line Gauss-Seidel V-cycle multigrid strategy on each architecture.

simple example illustrates how to set a vector type definition such that the program uses the GPU for memory storage and computations.

```
// Make a type definition to determine the vector type of the coming program
typedef vector<float,device_memory> vector_type;
vector_type x(100);     // Create vector x in GPU memory
vector_type y(100);     // Create vector y in GPU memory
y.axpy(2.f,x);          // Calc. y = a*x + y on the GPU
```

The above example might seem trivial, but the use of type definitions can be taken further, using so called type binders. Setting up our free surface solver looks similar to the following code example, using the predefined type binder class potential_flow_solver_types.

```
// Potential flow setup
typedef free_surface::potential_flow_solver_types<
        vector_type                                 // Vector object
      , solvers::multigrid<multigrid_types>         // Laplace solver
      , integration::ERK4                           // Time integrator
> potential_flow_types;
```

Afterwards, the solver object is instantiated with this type binder definition given as template argument. The solver hereby implicitly knows all necessary types to use within its own implementation. Consequently, parts can be treated as building blocks to make up the entire solver. If for example the user wants to use another time integrator or Laplace solver, the corresponding line is exchanged with an alternative implementation, either user specific or from the library itself. Notice that the multigrid solver is a template class itself that depends on another type binder, also specified by the user. Concepts of template based programming is well presented in the book by Vandevoorde and Josuttis [12].

The Laplace equation (3) is solved with an iterative multigrid-preconditioned defect correction method. In practice, the defect correction method turns out to be as

effective as a reference GMRES solver. Furthermore the defect correction method has two important properties: i) Constant minimal memory footprint. ii) Few synchronization barriers. These two properties make it very attractive from a parallel point of view. A textbook recipe of the defect correction method is given in Algorithm 1. This algorithm is implemented into our library in the same generic way as

Algorithm 1: DC method for approximate solution of $Ax = b$		
1	Choose $x^{[0]}$	`/* initial guess */`
2	$k = 0$	
3	**Repeat**	
4	$r^{[k]} = b - Ax^{[k]}$	`/* high order defect */`
5	Solve $M\delta^{[k]} = r^{[k]}$	`/* preconditioner */`
6	$x^{[k+1]} = x^{[k]} + \delta^{[k]}$	`/* defect correction */`
7	$k = k + 1$	
8	**Until** convergence or $k > k_{max}$	

previously described. Building the solver using a predefined type binder class could look as follows, assuming that proper types for the vector, matrix, and preconditioner are set beforehand.

```
1  typedef solvers::defect_correction_types<
2        vector_type
3      , matrix_type
4      , preconditioner_type> dc_types;   // DC type binder
5  typedef solvers::defect_correction<dc_types> dc_solver_type;
6
7  // Create solver, assume vectors (x,b) and matrices (A,P) are already created
8  dc_solver_type solver( A );            // Create solver
9  solver.set_preconditioner( P );        // Set preconditioner
10 solver.solve(x, b);                    // Solve Ax = b
```

From building blocks in the library, we have set up a 2D time integration solver for the fully nonlinear free surface waves. The library has tools for most of the needed components, such as the time integration scheme, solver for the linear system, printing functionality and so on. The main functionality that the user has to deliver, is an implementation of the matrix-vector product from the discretization of (3), required to calculate the residual in line 4 of Algorithm 1. Algorithmic efficiency is achieved with a multigrid preconditioning strategy based on a low-order discretization of the linearized system matrix (see [7]) and red-black Gauss-Seidel smoothening. This smoother must also be made available to the multigrid solver by the user.

2.2 Improving Defect Correction with Mixed Precision

In order to further improve the nonlinear free surface solver, a mixed precision strategy has recently been added to the defect correction scheme. The purpose of the

mixed precision algorithm is to reduce the overall computational and storage requirements by introducing low (single) precision arithmetics.

The advantage from a memory perspective is obvious: single precision numbers take up only half the storage of a double precision number (32 bits vs 64 bits). Thus, storage and bandwidth requirements are halved. The computational demands are also reduced. However, this is somewhat more hardware dependent. Most modern CPU architectures obtain twice the performance for single precision execution compared to double precision, see [5]. On GPU architectures this relation might be more distinct. On a TESLA S1070 computing system, single precision operations are up to twelve times faster.

As noted in [1], any refinement process is a candidate to benefit from mixed precision computations, since often only the refinement itself needs to be in double precision arithmetic. Rewriting the defect correction scheme from Algorithm 1 into a single expression for iterative refinement of x at iteration $k+1$ gives

$$x^{[k+1]} = x^{[k]} + M^{-1}(b - Ax^{[k]}). \tag{4}$$

Assuming that the iterative scheme converges towards the exact solution, the correction term $M^{-1}(b - Ax^{[k]})$ reduces in magnitude for each iteration until an acceptable accuracy threshold can be met. If both $x^{[k]}$ and the correction term are in single precision, round off errors naturally occur earlier than they would in double precision. The trick is to calculate only the correction term in single precision and do the update in double precision. Since the correction term is approaching zero, the values are well represented in single precision and the double precision update only suffer from rounding errors when the correction approaches values near $\sim 10^{-16}$. Thus, we get a double precision accurate solution, while being able to do parts of the calculations in single precision. Applying this technique to the defect correction scheme, the preconditioning step in line 5 of Algorithm 1, is simply executed in pure single precision arithmetics.

With this strategy we have been able to further improve the OceanWave3D model. Performance results for the mixed precision strategy on a Tesla C2050 are given in Fig. 2. The C2050 has a $2:1$ ratio on the peak performance for double precision vs. single precision. However, the algorithm is memory bound, so we expect the observed behavior to be caused by the $2:1$ restriction on the memory bandwidth. As expected, a pure single precision iteration takes approximately half the time (x1.9 faster) for larger systems. The mixed precision strategy is however the only one that would give a high precision solution and therefore the only fair comparison to the double precision strategy. Roughly a speedup of x1.6 is achieved for large enough systems. Absolute timings and relative speedups of the Laplace solver are depicted in Fig. 3. The double precision timings are slightly better than the ones previously presented in Fig. 1 from [8]. This is not surprisingly since the 3D finite difference operations in [8] are more expensive than the 2D operations in the present work. Still, we would expect an extension to 3D of the present solver to give results in the same range as the dedicated 3D solver. Taking also the mixed precision ex-

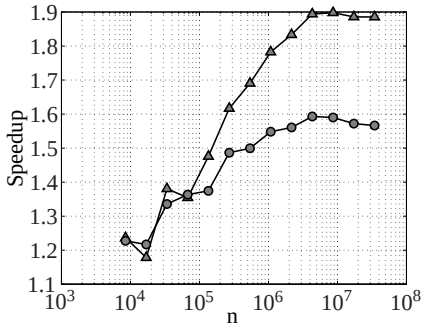
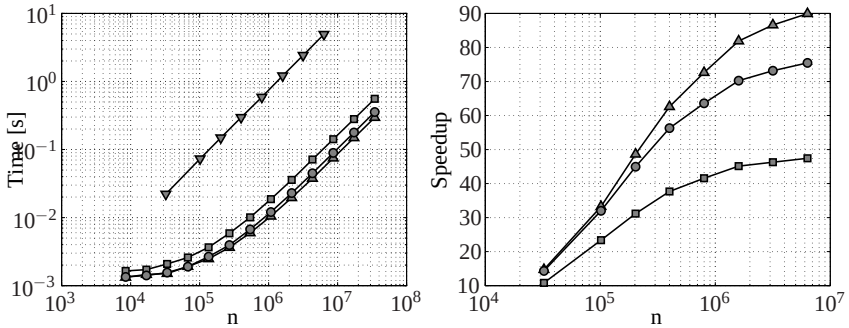


Fig. 2 Speedups for a defect correction iteration using single precision (−▼−) and mixed precision (−●−) relative to double precision. All timings are on a Tesla C2050 GPU.

tension into consideration, we expect a 3D solver to gain about the same x1.6 extra speedup as well.



(a) Absolute timings. (b) Speedup relative to CPU (single thread).

Fig. 3 Scalability tests and performance comparisons on Tesla C2050 in single precision (−▲−), double precision (−■−), mixed precision (−●−), and CPU (single thread) code (−▼−). Sixth order spatial discretization employed. The iterative defect correction method has been left-preconditioned with a Gauss-Seidel V-cycle multigrid strategy on each architecture.

3 Concluding Remarks

The potential flow equations describing fully nonlinear water waves have been efficiently solved and improved from previous work [8]. A highly generic GPU-based library has been developed, not only to solve the present equations, but also a broader range of PDEs that can be well discretized in a finite difference manner. The library is still at an early state and under continuously development. We expect

that the library will ease future development of PDE solvers for a variety of physical problems and simulations. Results indicate that the library does not suffer from serious overhead, as performance results are comparable to an existing dedicated solver for the same model problem. Future work is to confirm that this indication is valid, by assembling a full 3D solver using library components.

Furthermore we illustrated how to easily extend the defect correction method in order to utilize a fast mixed precision strategy, by computing the preconditioning step in pure single precision arithmetics. This approach gives an additional x1.6 speedup on the Tesla C2050 GPU architecture. Combining these results we are approaching almost two orders of magnitude in relative speedup compared to the optimized single threaded CPU reference code from previous work [7].

Ongoing work is also concerned with large-scale modelling, in which the discretized equations does not fit into the memory of one GPU. A domain decomposition strategy is thus necessary to decompose memory across multiple GPUs. In this case MPI is used for the communication between nodes. The impact on performance of transferring artificial boundary information between nodes is to be investigated in future work.

References

1. Baboulin, M. and Buttarib, A. and Dongarra, J. and Kurzak, J. and Langouc, J. and Langou, J. and Luszczek, P. and Tomov, S.: Accelerating scientific computations with mixed precision algorithms. Comp. Phys. Comm. **180**, 25262533, (2009)
2. Balay, S. and Brown, J. and Buschelman, K. and Gropp, W. D. and Kaushik, D. and Knepley, M. G. and McInnes, L. C. and Smith, B. F. and Zhang, H.: PETSc, version 3.2. (2011) http://www.mcs.anl.gov/petsc
3. Bell, N. and Garland, M.: Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, version 0.1.0. (2010) http://cusp-library.googlecode.com
4. Bingham, H. B. and Zhang, H.: On the accuracy of finite-difference solutions for nonlinear water waves. J. Engng. Math. **58**, 211–228, (2007)
5. Buttari A. and Dongarra, J. and Langou, J. and Langou, J. and Luszczek, P. and Kurzak J.: Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. Int. J. Hi. Perf. Comp. App. **21**, 457–466 (2007)
6. Engsig-Karup, A.P.: Efficient low-storage solution of unsteady fully nonlinear water waves using a defect correction method. Submitted for SIAM J. Sci. Comp. (2011)
7. Engsig-Karup, A.P. and Bingham, H.B. and Lindberg, O.: An efficient flexible-order model for 3D nonlinear water waves. J. Comp. Phys. **228**, 2100–2118, (2009)
8. Engsig-Karup, A. P. and Madsen, M. G. and Glimberg, S. L.: A massively parallel GPU-accelerated model for analysis of fully nonlinear free surface waves. Int. J. Num. Meth. Fluids. (2011)
9. Hoberock, J. and Bell, N.: Thrust: A Parallel Template Library, version 1.3.0. (2010) http://www.meganewtons.com/
10. Li, B. and Fleming, C. A.: A three dimensional multigrid model for fully nonlinear water waves. Coast. Engng. **30**, 235–258, (1997)
11. Martin, R. S. and Peters, G. and Wilkinson J. H.: Handbook Series Linear Algebra Iterative Refinement of the Solution of a Positive Definite System of Equations. Num. Math. **8**, 203–216 (1966)
12. Vandevoorde, D. and Josuttis, N. M.: C++ Templates: The Complete Guide. Addison-Wesley Professional, (2002)