Real-time computation of interactive waves using the GPU

Martijn de Jong, m.d.jong@marin.nl
Auke van der Ploeg, a.v.d.ploeg@marin.nl
Auke Ditzel, a.ditzel@marin.nl
Kees Vuik, c.vuik@tudelft.nl

1 Introduction

The Maritime Research Institute Netherlands (MARIN) supplies innovative products for the offshore industry and shipping companies. Among their products are highly realistic, real-time bridge simulators [2], see Figure 1.

Figure 1: Left: full-scale bridge simulator. Right: towing tank.

Currently, the waves are deterministic and are not affected by ships, moles, breakwaters, piers, or any other object. To bring the simulators to the next level, a new interactive wave model is being developed. This is the so-called Variational Boussinesq model (VBM) as proposed by Gert Klopman [3]. The main improvement will be that the waves and ships really interact, i.e., the movements of the ship are influenced by the waves and the waves in their turn are influenced by the ship. However, one pays for the higher realism: the new model is much more computational intensive and therefore a really fast solver is needed to fulfill the requirements of real-time simulation.

In this paper we present how a very efficient iterative solver can be combined with a very efficient implementation on the graphical processing unit (GPU). In this way speed up factors of more than 30 can be obtained compared to sequential code on the CPU for realistic problems. With the new solver interactive waves can be computed in real-time for large domains.

2 Model equations and discretization method

The governing linearized VBM equations are given by:

$$\frac{\partial \zeta}{\partial t} + \nabla \cdot (\zeta \mathbf{U} + h \nabla \varphi - h \mathcal{D} \nabla \psi) = 0, \tag{1a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U} \cdot \nabla \varphi + g \zeta = -P_s, \tag{1b}$$

$$\mathcal{M} \psi + \nabla \cdot (h \mathcal{D} \nabla \varphi - \mathcal{N} \nabla \psi) = 0, \tag{1c}$$

Equations (1a) and (1b) are the mass conservation equation and the Bernoulli equation. They describe the evolution in time of the free surface elevation $\zeta(x_1, x_2, t)$ and free-surface velocity potential $\varphi(x_1, x_2, t)$, respectively, where (x_1, x_2) are the Cartesian horizontal coordinates and t is the time. The third equation is an elliptic equation for the free-surface vertical velocity $\psi(x_1, x_2, t)$, and has to be solved at each time frame for given $\zeta(x_1, x_2, t)$ and $\varphi(x_1, x_2, t)$. The other symbols in (1a-c) are:

$\mathbf{U}$	horizontal flow-velocity
h	water depth
g	gravitational acceleration
P_s	"pressure pulse" ship
$\mathcal{D}, \mathcal{M}, \mathcal{N}$	model parameters

The VBM equations are discretized with the finite volume method (FVM) on a Cartesian grid. Discretization leads to:

$$\frac{\mathrm{d}\mathbf{q}}{\mathrm{d}t} = L\mathbf{q} + \mathbf{f}, \tag{2}$$

$$S\vec{\psi} = \mathbf{b}. \tag{3}$$

Equation (2) is solved using the Leapfrog integration scheme, a second order explicit integration method. System (3) is a linear system that has to be solved. In this system the matrix S is real-valued, sparse (5-point, pentadiagonal), diagonally dominant (not very strong for small mesh sizes), symmetric positive definite (SPD), and large (in the order of millions by millions).

3 The RRB-solver

For a system with a matrix as described above a Preconditioned Conjugated Gradient (PCG) type solver is most proficient. The PCG-algorithm is given by Algorithm 1 (cf. [4]; Algorithm 9.1).

Rather than solving system (3) we solve a preconditioned system

$$M^{-1}S\psi = M^{-1}\mathbf{b}, \tag{4}$$

Algorithm 1 The PCG algorithm.

$\mathbf{r} = \mathbf{b} - S\vec{\psi}$, solve $M\mathbf{z} = \mathbf{r}$ for $\mathbf{z}$,
$\rho_1 = \langle \mathbf{r}, \mathbf{z} \rangle$, set $\mathbf{p} = \mathbf{z}$.
While (not converged)

$\rho_0 = \rho_1$	
$\mathbf{q} = S\mathbf{p}$	Matrix-vector product
$\sigma = \langle \mathbf{p}, \mathbf{q} \rangle$	Inner product
$\alpha = \rho_0 / \sigma$	
$\vec{\psi} = \vec{\psi} + \alpha \mathbf{p}$	Vector update
$\mathbf{r} = \mathbf{r} - \alpha \mathbf{q}$	Vector update
Solve $M\mathbf{z} = \mathbf{r}$	Preconditioner step
$\rho_1 = \langle \mathbf{r}, \mathbf{z} \rangle$	Inner product
$\beta = \rho_1 / \rho_0$	
$\mathbf{p} = \mathbf{z} + \beta \mathbf{p}$	Vector update

End while

where the preconditioning matrix M^{-1} is chosen such that the location of the of eigenvalues of $M^{-1}S$ are more favorable than those of S leading to faster convergence, i.e., fewer CG-iterations.

The RRB-solver is such a PCG solver with the RRB-method [1] as preconditioner. RRB stands for "Repeated Red-Black" which refers to how nodes in a 2D grid are colored and numbered. The RRB-method makes an incomplete factorization

$$S = LDL^T + R, \tag{5}$$

where L is a lower triangular matrix, D a block diagonal matrix, and R a matrix of adjustments resulting from so-called lumping procedures. As preconditioner for CG the matrix

$$M = LDL^T \approx S$$

is taken.

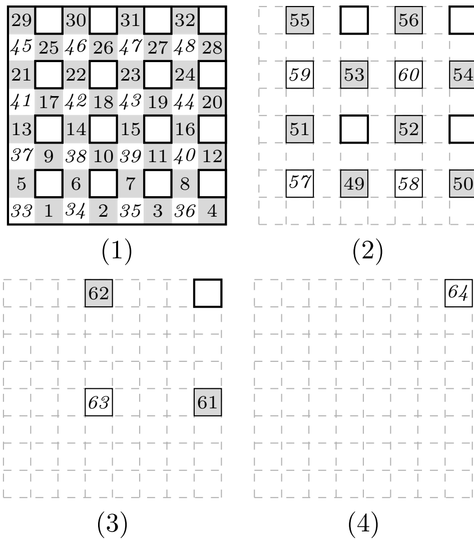

Figure 2: RRB-numbering for an 8×8-grid. The black nodes are represented by gray squares, red nodes by white squares.

Let us now explain the RRB-numbering using an 8×8-example, see Figure 2. The nodes in the first level (1) are divided into two groups: the nodes (i, j) with $\mathrm{mod}(i+j, 2) = 0$ are red nodes, the nodes with $\mathrm{mod}(i+j, 2) = 1$ are black nodes. Then all black nodes are numbered sequentially (1-32), and half of the red nodes are numbered (33-48) in the way as indicated. The remaining 16 nodes form the next level (2). For level (2) the numbering procedure is repeated. Ultimately this procedure leads to 4 levels for an 8×8-grid. For grids with "not so perfect" dimensions, i.e., N_x and N_y are not powers of 2, things are a little more complicated, but the same procedure can be applied. For general N_x and N_y the maximal number of levels is given by:

$$k_{\max} = 1 + \lfloor (\log_2(\max\{N_x, N_y\})) \rfloor. \tag{6}$$

We do not have to go all the way down; we can stop at any level k and make a complete Cholesky factorization on that level. The corresponding method is called RRB-k. In Figure 3 the sparsity pattern of S is shown for RRB-1 and RRB-4 for the 8×8-example.

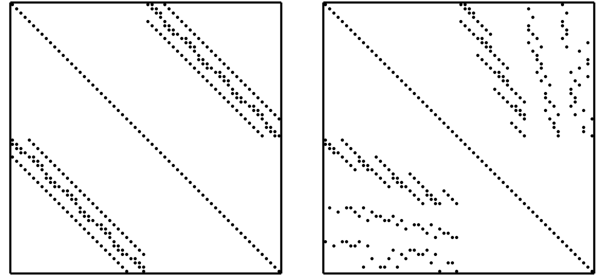

Figure 3: Left: Sparsity pattern of $S \in \mathbb{R}^{64 \times 64}$ when the basic Red-Black numbering is applied (RRB-1). Right: sparsity pattern after the RRB-4 numbering.

By applying a basic Red-Black numbering we can write system (3) as

$$\begin{bmatrix} D_b & S_{br} \\ S_{rb} & D_r \end{bmatrix} \begin{bmatrix} \vec{\psi}_b \\ \vec{\psi}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_b \\ \mathbf{b}_r \end{bmatrix}, \tag{7}$$

where the red nodes are indicated by "r" and the black nodes by "b". Herein D_r and D_b are diagonal matrices and $S_{rb} = S_{br}^T$ are matrices with 4 diagonals, see Figure 3. Applying Gaussian elimination yields

$$\begin{bmatrix} D_b & S_{br} \\ 0 & S_1 \end{bmatrix} \begin{bmatrix} \vec{\psi}_b \\ \vec{\psi}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_b \\ \mathbf{b}_1 \end{bmatrix}, \tag{8}$$

where $S_1 := D_r - S_{rb} D_b^{-1} S_{br}$ is called the *1st Schur complement* (given by a 9-point stencil) and $\mathbf{b}_1 := \mathbf{b}_r - S_{rb} D_b^{-1} \mathbf{b}_b$ is the corresponding right-hand side. Hence the original system (3) can be solved as follows:

1. Compute $\mathbf{b}_1$;

2. Apply CG to the system $S_1\vec{\psi}_r = \mathbf{b}_1$;
3. Compute $\vec{\psi}_b$ via $\vec{\psi}_b = D_b^{-1}(\mathbf{b}_b - S_{br}\vec{\psi}_r)$.

This is beneficial for the amount of computational work as for the vector updates and inner products in CG the work is reduced by a factor two. Note that the matrix-vector product in CG becomes $\mathbf{q} = S_1\mathbf{p}$, with S_1 given by a 9-point stencil and not by a 5-point stencil, and hence the work is not reduced by a factor two for this operation.

3.1 Construction of the preconditioner

The RRB-method makes an incomplete factorization (5) as follows. In each level Gaussian elimination is applied. Elimination of nodes leads to fill-in: a 5-point stencil becomes a 9-point stencil. By using graph representation, see Figure 4, the occurence of fill-in can be explained nicely. A *lumping* procedure is then used to simplify the 9-point stencil to a 5-point stencil: the four outermost coefficients are added to the center coefficient, which leads to a rotated 5-point stencil after elimination of black nodes, see Figure 4.

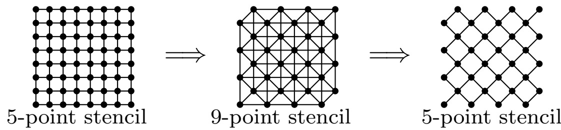

Figure 4: Elimination of the black nodes leads to fill-in and a 9-point stencil. A lumping procedure is used to obtain a (rotated) 5-point stencil again.

In Figure 5 the sparsity pattern of $L + D + L^T$ is shown when the RRB-method is applied to matrix $S \in \mathbb{R}^{64\times 64}$.

The RRB-solver offers good parallelization options. The basic operations in the CG-algorithm, see Algorithm 1, such as matrix-vector products, vector updates and inner products parallelize very well on shared memory machines. Further, from Figure 5 we see that within a block (the gray shaded areas) the nodes do not depend on each other. Therefore, within such a block the elimination described above can be performed fully in parallel. As we shall see this is the key in parallelizing the application of the preconditioner as well.

3.2 Application of the preconditioner

At each CG-iteration the preconditioning step $M\mathbf{z} = \mathbf{r}$ needs to be solved for $\mathbf{z}$. The preconditioner matrix M can be written as $M = LDL^T$ so that solving $M\mathbf{z} = \mathbf{r}$ can be done in three steps as follows. Set $\mathbf{y} := L^T\mathbf{z}$ and $\mathbf{x} := DL^T\mathbf{z} = D\mathbf{y}$, then:

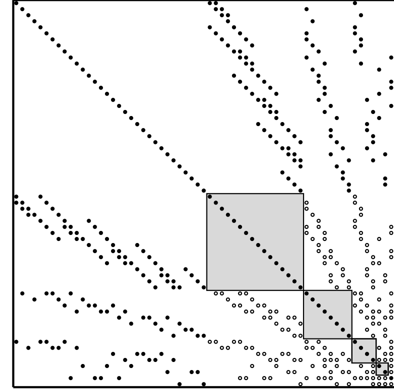

Figure 5: Sparsity pattern of $L + D + L^T$. The gray areas indicate where fill-in has been lumped.

1. Solve $L\mathbf{x} = \mathbf{r}$ using forward substitution;
2. Compute $\mathbf{y} = D^{-1}\mathbf{x}$;
3. Solve $L^T\mathbf{z} = \mathbf{y}$ using backward substitution.

If we have a closer look at the structure of matrix L, see Figure 6, we see that Step 1 can be done level-wise in parallel as follows.

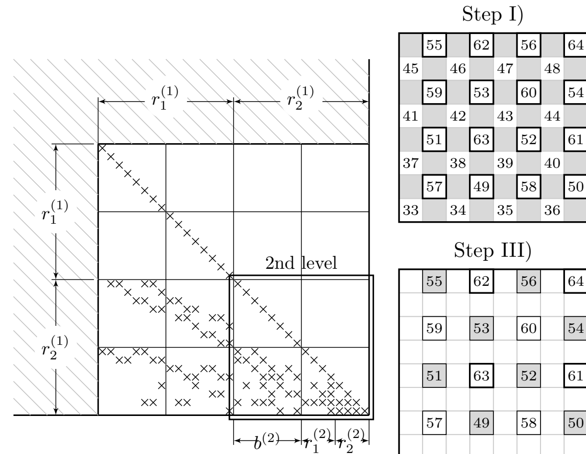

Figure 6: Forward and backward substitution can be done level-wise in parallel.

Solving $L\mathbf{x} = \mathbf{r}$ using forward substitution:

I) *Do in parallel*: Update $\mathbf{x}$-values corresponding to r_2-nodes using the $\mathbf{x}$-values of the r_1-nodes from the same level (according to a rotated 5-point stencil);

II) Go to the next level if there is any; otherwise: *stop*;

III) *Do in parallel*: Update $\mathbf{x}$-values corresponding to r_1- and r_2-nodes using the $\mathbf{x}$-values of the b_1- and b_2-nodes from the same level (according to a straight 5-point stencil);

IV) Repeat I).

Step 3, solving $L^T \mathbf{z} = \mathbf{y}$, can be done level-wise in parallel by a similar procedure (but in the reverse order). Step 2 is trivially parallelized as it comes down to elementwise division.

3.3 Convergence behaviour

Because of the multiple levels the RRB-solver shows "Multigrid-like" behaviour: the required number of CG-iterations grows only very slowly with increasing problem size N. In Figure 7 the required number of CG-iterations is shown when Poisson's equation on the unit square with Dirichlet boundary conditions, i.e.,

$$\begin{aligned} -\Delta u &= f(x,y) \quad \text{on} \quad \Omega = (0,1) \times (0,1), \\ u(x,y) &= 0 \quad\quad\;\; \text{on} \quad \partial\Omega, \end{aligned} \tag{9}$$

is discretized on an $N \times N$ (internal) nodes grid, and solved with the RRB-solver. The right-hand side f is taken such that $u(x,y) = x(x-1)y(y-1)\exp(xy)$ (cf. [1]; model problem (2)). As termination criterium we have taken: $\|r_i\|_{M^{-1}}/\|r_0\|_{M^{-1}} \leq 10^{-5}$ and as initial guess the zero-vector.

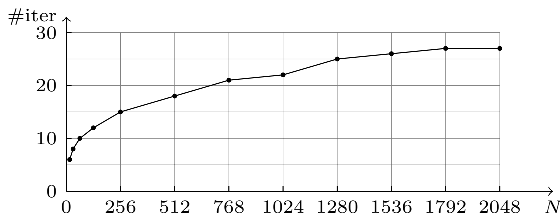

Figure 7: Number of CG-iterations versus problem size for model problem (9).

4 A parallel implementation on the GPU using CUDA

We have implemented the RRB-solver on the GPU using CUDA C. CUDA stands for "Compute Unified Device Architecture". It is NVIDIA's parallel programming environment to program the GPU.

A program on the GPU is divided over kernels which are invoked by the CPU. A kernel basically is a C function that is executed as many times in parallel as there are different CUDA threads. Threads are organized by the programmer by defining a grid and making a division of the grid in thread blocks. The GPU follows the SIMD (single instruction multiple data) programming model. The thread blocks are divided among the physical processors of the GPU. The physical processors are divided among several streaming multiprocessors (SMs), each having many cores; e.g., the GeForce GTX 580 has 16 SMs each having 32 cores, a total of 512 cores, hence a massively parallel architecture.

The GPU has different layers of memory including (cached) global memory, texture memory, shared memory and registers. The global memory is the largest in size (up to 6 GB) but it is also the slowest (400-800 cycles latency). Shared memory is very fast but also very limited in amount. Threads within the same SM communicate through this shared memory.

Programming on the GPU comes with a rich set of rules. One of the most important rules is related to the notion of coalesced memory. The global memory bandwidth is highest when the global memory accesses can be coalesced within a half-warp, e.g., for 16 threads in a half-warp the consecutive 4-byte words must fall within 64-byte memory boundaries, and the 16 threads must access the words in sequence: the kth thread in the half-warp must access the kth word. The penalty for non-coalesced memory transactions varies according to the actual size of the data type and architecture of the device. However, in any case performance is degraded when memory transfers are non-coalesced.

For modern architectures (Fermi, Kepler) the penalty for reading or writing data with a shift is small. However, when reading or writing data with a stride, the effective bandwidth is strongly reduced, see Figure 8 and Figure 9.

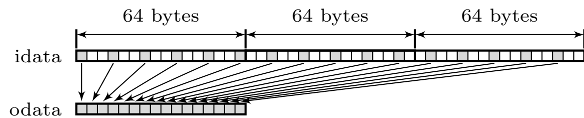

Figure 8: Copying data with a stride. Each thread handles 1 output. On the GPU data is read and written using 32-, 64-, and 128-byte memory transactions only (related to half-warps).

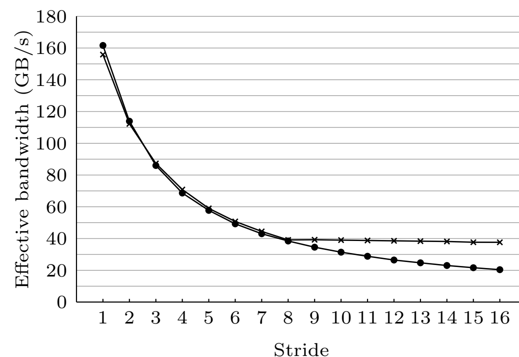

Figure 9: Throughput versus stride for GeForce GTX 580 with (×) and without (•) textures.

This small example already nicely illustrates the main problem we had to deal with when implementing the RRB-solver with CUDA on the GPU: with a naive storage format the required data for each of the operations in CG would not be located next

to each other in the global memory. As we mentioned earlier, the CG-algorithm operates on the first level red nodes only. This means that if we were to perform, say, a vector update, we would have to read data with stride 2 (red/black), and hence we would loose 1/3th bandwidth according to Figure 9. Even worse, when solving the preconditioner step $Mz = r$ for z we would access data with stride $2, 4, 8, 16, 32, \ldots$.

To overcome this problem we have introduced a new storage format: the so-called $r_1/r_2/b_1/b_2$ storage format, see Figure 10.

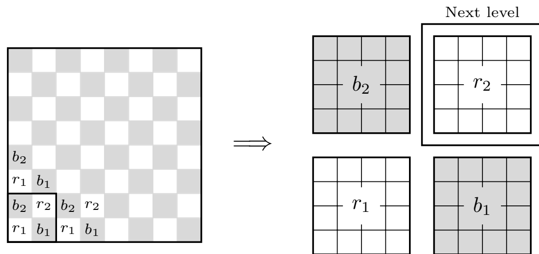

Figure 10: The nodes are divided into four groups: r_1-, r_2-, b_1- and b_2-nodes. Note that all the r_2-nodes together form the next level. On the next level the grouping procedure can be reapplied.

Every vector (and the matrix S) occuring in the CG-algorithm is only stored in this new format, except from $\mathbf{b}$ and $\vec{\psi}$ which are stored in both formats. This is necessary as in a real-time simulator each time frame we have to communicate $\mathbf{b}$ and $\vec{\psi}$. The overhead that comes with $r_1/r_2/b_1/b_2$ is thus restoring 2 vectors. The overhead is very little and worth it by realizing how much throughput we gain every CG-iteration: all operations can be performed fully coalesced. To ensure coalesced memory transactions throughout the preconditioner step, the $r_1/r_2/b_1/b_2$ storage format is recursively applied for all levels. Notice how using the new format comes for free for the preconditioner step.

5 Test problems and testing method

To test our solver we have used a collection of various test problems, including Poisson's problem (9) (for throughput analysis) and several realistic domains from MARIN's extensive database such as the Gelderse IJssel, a small river in the Netherlands, Plymouth Sound, a bay located in the South Shore region of England, see Figure 11, and Port Presto, a fictional region that shows great similarities with Barcelona. Port Presto is used frequently as a reference harbour in real-time simulator studies and assesments of mariners. The realistic domains were discretized on Cartesian grids varying from 100k to 1.5M nodes.

Figure 11: Left: the Gelderse IJssel. Right: Plymouth Sound.

All experiments were performed using single-precision numbers (floats). The experiments were performed on a Dell T3500 workstation equipped with a Xeon W3520 processor (@2.67 GHz), 6GB RAM and a GeForce GTX 580 graphics card (CUDA dedicated). The operating system is Ubuntu 10.04.3 LTS (2.6.32-34-generic x86_64) with CUDA version 4.0 (driver 270.41.19).

We have compared an optimized C++ RRB-solver (without the $r_1/r_2/b_1/b_2$ format) on 1 core of the Xeon W3520 with the CUDA RRB-solver on all cores of the GTX 580. Speed ups were computed at the hand of wall-clock timings on the host. For throughput analysis of the CUDA RRB-solver NVIDIA's profiler was used.

6 Results

In Figure 12 we have plotted the speed up that we obtain when we use the CUDA RRB-solver instead of the C++ RRB-solver.

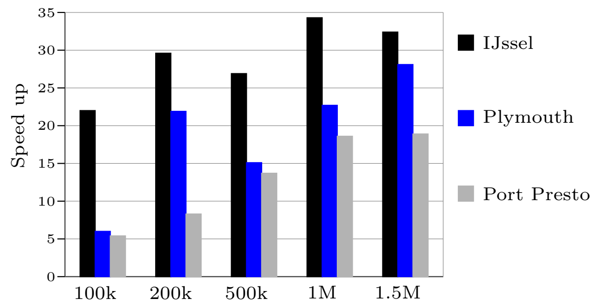

Figure 12: Speed up numbers for the realistic test problems.

For the largest test problems we find a speed up around a factor 25. The 1.5M IJssel, Plymouth and Port Presto problems were respectively solved within 10.7, 10.6 and 11.6 milliseconds, hence the 1.5M test problems can be solved in real-time. Depending on the specific problem it is to be expected that realistic domains consisting of up to 4 million nodes can be solved in real-time.

The RRB-solver is very efficient in itself as we can see from the required number of CG-iterations, see Table 1. All test problems were solved within 7 CG-iterations.

Problem	IJssel	Plymouth	Port Presto
100k	5.814	5.804	5.924
200k	5.832	5.892	5.962
500k	5.836	5.964	5.986
1M	5.859	5.976	6.362
1.5M	5.766	5.984	6.921

Table 1: Average number of CG-iterations over 1000 time frames.

To see how well the CUDA RRB-solver has been parallelized we have listed in Table 2 the performance for each of the routines that are part of the CG-algorithm, recall Algorithm 1. All routines are expected to be bandwidth bound as they are level 1 and level 2 BLAS routines. From the table this is clear as the achieved effective throughput is close to the device's peak bandwidth: the GTX 580 has a peak bandwidth of 193 GB/s. We have to remark that the listed throughput number for the preconditioner step is an average over the throughput number of the kernels on the 1st level; on coarser levels the throughput is lower due to overhead. The table shows that the time is well divided amongst the routines as there are 3 vector updates and 2 inner products in the CG-algorithm, hence there is no bottleneck.

Operation	Time (μs)	Gflop/s	Throughput (GB/s)
Matrix-vector	732	48.7	184
Vector update	160	13.1	164
Inner product	139	15.1	148
RRB solve step	1092	32.9	188

Table 2: Performance of the separate routines of the CUDA RRB-solver on the GTX 580 for a 2048×2048 *domain.*

In Table 3 it is shown how the time in the preconditioner step is distributed over the various levels. According to Equation (6) the maximal number of levels is $k_{\max} = 1 + \lfloor (\log_2(\max\{1024, 1024\})) \rfloor = 11$. However, the coarsest 6 levels fit in the cache of the GPU and are therefore handled by 1 SM at once on the 6th level. As we can see only a small fraction of the time is taken by the coarsest levels. So, the typical "Multigrid-issues" such as idle threads on the coarsest levels are not really an issue for the CUDA RRB-solver.

Finally, in Figure 13 an example of the solution of Equation (1) is shown.

7 Conclusions and discussion

To solve the systems of linear equations we have used a solver which is very efficient in itself. In combination with a very efficient implementation

Level	$\frac{1}{2}$#Nodes	Time (μs)	Percentage
1	1024×1024	428	39.2
2	512×512	405	37.1
3	256×256	130	11.9
4	128×128	50	4.6
5	64×64	25	2.3
6-11	32×32	54	4.9
Total		1092	100

Table 3: Time spent per level for a domain of 2048×2048 *nodes.*

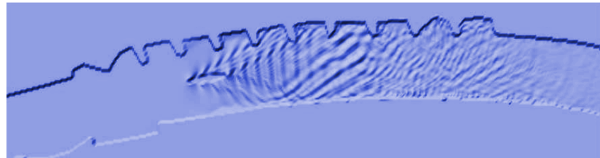

Figure 13: A ship sailing through the Gelderse IJssel.

on the GPU, the solver allows real-time simulation of interactive waves using up to 4 million nodes.

ILU preconditioners tend to parallelize very poorly as the operations usually are inherently sequential. However, we have demonstrated that the RRB-solver allows an efficient parallelization for both the construction and the application of the preconditioner. With the $r_1/r_2/b_1/b_2$ storage format we were able to implement the RRB-solver efficiently on the GPU. By doing so we found speed up factors of more than 30 compared to a sequential implemention on the CPU.

In our experiments we used equidistant Cartesian meshes, but this is not mandatory. With non-equidistant and curvilinear meshes one should be able to simulate even much larger domains in real-time.

Acknowledgement

The authors gratefully acknowledge the contributions of Gert Klopman and Anneke Sicherer-Roetman to the research project.

References

[1] C.W. Brand, An Incomplete-factorization Preconditioning using Repeated Red-Black Ordering, *Numerische Mathematik*, pp. 433–454, 1992.

[2] A. Ditzel, N. Leith, Deep Water Anchor Handling Simulation. MARSIM, Singapore, 2012.

[3] G. Klopman, Variational Boussinesq Modelling of Surface Gravity Waves over Bathymetry, PhD Thesis, University of Twente, Twente, 2010.

[4] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, 2nd edition, Philadelphia, 2003.