Visual Simulation of Shallow-Water Waves

T.R. Hagen, J.M. Hjelmervik, K.–A. Lie, J.R. Natvig,
M. Ofstad Henriksen

SINTEF ICT, Applied Math., P.O. Box 124 Blindern, NO-0314 Oslo, Norway

Abstract

A commodity-type graphics card (GPU) is used to simulate nonlinear water waves described by a system of balance laws called the shallow-water system. To solve this hyperbolic system we use explicit high-resolution central-upwind schemes, which are particularly well suited for exploiting the parallel processing power of the GPU. In fact, simulations on the GPU are found to run 15–30 times faster than on a CPU. The simulated cases involve dry-bed zones and non-trivial bottom topographies, which are real challenges to the robustness and accuracy of the discretization.

Key words: Water waves, high-resolution schemes, graphics processing unit, parallel processing, physics-based visualization
PACS: 02.60.Cb, 02.70.Bf

1 Introduction

The introduction of programmable GPUs made it feasible to exploit the computational power of the GPU for non-graphical purposes. In [1] a method for physically-based visual simulation using some of the earliest programmable GPUs was proposed. Recently, fragment processors with floating-point precision have become available. This has opened up a wide variety of usages of the GPU including more advanced methods for solving PDEs such as complex explicit and implicit schemes. Methods for employing GPUs to solve linear systems of equations, which is required when using implicit schemes, have been proposed in [2] and [3]. Here we use the GPU for physically-based simulations by applying state-of-the-art explicit schemes to solve systems of balance laws known as the shallow water equations.

Free-surface flow over a variable bottom topography under the influence of

gravity can be modeled by the shallow-water (or Saint–Venant) equations

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -gh\frac{\partial B}{\partial x} \\ -gh\frac{\partial B}{\partial y} \end{bmatrix}, \tag{1}$$

which we write on short form as

$$Q_t + F(Q)_x + G(Q)_y = H(Q, \nabla B).$$

Here $B(x, y)$ is the bottom topography, $h(x, y, t)$ is the distance from the bottom to the (wavy) surface, $[\, u, v \,]$ is the depth-averaged velocity, and g is the gravitational acceleration. The shallow-water equations are derived from the depth-averaged incompressible Navier–Stokes equations for the case where the surface perturbation is much smaller than the typical horizontal length scale.

To compute solutions of (1) it is common to use high-resolution schemes [4] with explicit temporal discretization. Such schemes have an obvious and natural parallelism in the sense that each grid cell can be processed independently of its neighbors and are therefore ideal candidates for an implementation using data-based stream processing on a graphics processing unit (GPU). In this paper we shall compute and compare approximate solutions to (1) using the GPU and the CPU. We demonstrate that the GPU gives a speedup of more than one order of magnitude, while retaining sufficient accuracy to make the GPU an interesting and inexpensive alternative to high-performance computers for qualitative and quantitative simulations of physical phenomena on large grid models. Although our main focus is on the usability and applicability of GPUs in scientific computing, we also try to demonstrate that numerical solution of the shallow-water equations can be used to create semi-realistic, nonlinear wave effects in interactive visual applications. To get a sufficiently high framerate, it is necessary to use small grid models, viz. 10^4–10^5 grid points, and/or simple schemes of low order.

2 Numerical Methods

The system (1) has two key features that makes it difficult to solve numerically. First of all, the solution may contain discontinuities that correspond to breaking waves. Classical schemes will therefore typically either smear the discontinuous parts or introduce spurious oscillations that pollute computed solutions. A common approach is therefore to use high-resolution schemes [4]. Here we will use a semi-discrete finite-volume scheme [5] to ensure high order of approximation for smooth waves and sharp resolution of discontinuities

without the creation of spurious oscillations.

The second difficulty with (1) is that this system of balance laws admits steady-state solutions in which the source terms are exactly balanced by nonzero flux gradients. Capturing such balances is a challenging task for any numerical scheme. Here we will use a well-balanced treatment of the source terms [6] to accurately resolve surfaces that are steady-state solutions or small perturbations thereof.

The finite-volume scheme is defined over a regular Cartesian grid with grid cells Ω_{ij} and seeks approximations in the form of cell-averages, $Q_{ij} = \frac{1}{|\Omega_{ij}|}\int_{\Omega_{ij}} Q$. The simplest possible scheme is the first-order Lax–Friedrichs scheme

$$\begin{aligned} Q_{ij}^{n+1} = {} & \frac{1}{4}\Big(Q_{i+1,j}^n + Q_{i-1,j}^n + Q_{i,j+1}^n + Q_{i,j-1}^n\Big) + \Delta t S_{ij}^n \\ & - \frac{\Delta t}{2\Delta x}\Big[F\Big(Q_{i+1,j}^n\Big) - F\Big(Q_{i-1,j}^n\Big)\Big] - \frac{\Delta t}{2\Delta y}\Big[G\Big(Q_{i,j+1}^n\Big) - G\Big(Q_{i,j-1}^n\Big)\Big]. \end{aligned} \tag{2}$$

This is a very robust scheme, which unfortunately gives excessive smearing of nonsmooth parts of the solution.

To obtain better accuracy for nonsmooth solutions, we introduce a high-resolution scheme based upon a semi-discrete formulation where the cell averages are evolved in time according to

$$\frac{dQ_{ij}}{dt} = -\Big(F_{i+1/2,j} - F_{i-1/2,j}\Big) - \Big(G_{i,j+1/2} - G_{i,j-1/2}\Big) + S_{ij}, \tag{3}$$

and the fluxes over the edges are approximated using a numerical quadrature. In [5,6], the authors use the three-point Simpson rule. Here we will use a standard two-point Gaussian quadrature (see Figure 1)

$$\begin{aligned} F_{i+1/2,j}(t) &= \frac{1}{|\Omega_{ij}|}\int_{y_{j-1/2}}^{y_{j+1/2}} F\Big(Q(x_{i+1/2}, y, t)\Big)\, dy \\ &\approx \frac{1}{2\Delta x}\Big[F\Big(Q\Big(x_{i+1/2}, y_j + \frac{\Delta y}{2\sqrt{3}}, t\Big)\Big) + F\Big(Q\Big(x_{i+1/2}, y_j - \frac{\Delta y}{2\sqrt{3}}, t\Big)\Big)\Big], \end{aligned} \tag{4}$$

which is more accurate and reduces the number of flux computations from three to two for each edge. To evaluate the integrand, we start with the cell-averages Q_{ij} and *reconstruct* a function that is piecewise polynomial inside each grid cell, see [5,7]. A componentwise, piecewise linear function is used:

$$\begin{aligned} Q_{ij}(x,y) = Q_{ij} &+ L\Big(Q_{i+1,j} - Q_{i,j}, Q_{i,j} - Q_{i-1,j}\Big)\frac{x - x_i}{\Delta x} \\ &+ L\Big(Q_{i,j+1} - Q_{i,j}, Q_{i,j} - Q_{i,j-1}\Big)\frac{y - y_j}{\Delta y}. \end{aligned} \tag{5}$$

Here L is a so-called limiter function, whose purpose is to construct the linear slope within each grid cell as a nonlinear average of the forward and the

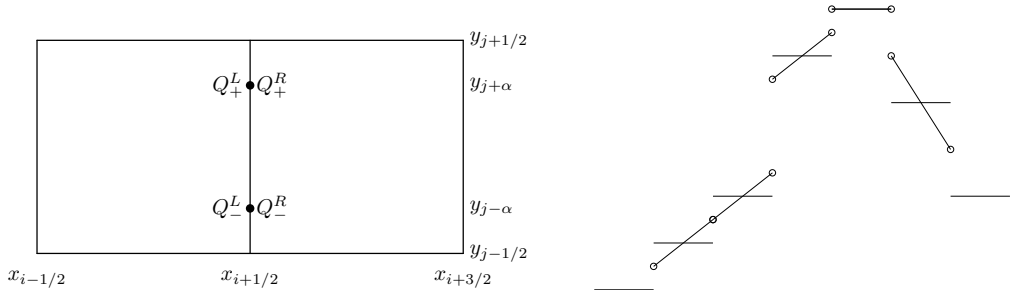

Fig. 1. (Left) Integration points in the Gaussian quadrature. (Right) Reconstruction of a piecewise linear function based upon the cell averages and the limiter (6) with $\theta = 1$. The horizontal lines indicate the cell averaged values and the piecewise linear function is formed by the segments with small circles at the ends. The effect of the limiter is clearly visible: given two candidate slopes based upon a right-sided and a left-sided estimate, the limiter chooses the slope giving the least steep reconstruction if the signs of the two slopes are equal, and chooses a zero slope otherwise.

backward differences, and prevent the creation of overshoots at local extrema; see the right-hand part Figure 1. In this paper, we use the family of generalized minmod limiters

$$L(a, b) = \mathrm{MM}(\theta a, \tfrac{1}{2}(a + b), \theta b)$$

where the minmod function MM is given by

$$\mathrm{MM}(z_1, \ldots, z_n) = \begin{cases} \max_i z_i, & z_i < 0 \ \forall i, \\ \min_i z_i, & z_i > 0 \ \forall i, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

At each integration point $(x_{i+1/2}, y_{j\pm\alpha})$, we have a left-sided and a right-sided point value, Q^L and Q^R (cf. Figure 1). To compute the flux, we could use the average of the flux evaluated at the two one-sided point values. However, to get a high-resolution scheme, we use the central-upwind flux-function [5]

$$\begin{gathered} \mathcal{F}(Q^L, Q^R) = \frac{a^+ F(Q^L) - a^- F(Q^R)}{a^+ - a^-} + \frac{a^+ a^-}{a^+ - a^-}(Q^R - Q^L), \\ a^+ = \max\Big(0, \lambda^+(Q^L), \lambda^+(Q^R)\Big), \quad a^- = \min\Big(0, \lambda^-(Q^L), \lambda^-(Q^R)\Big), \end{gathered} \tag{7}$$

where $\lambda^\pm(Q) = u \pm \sqrt{gh}$ denotes the eigenvalues of dF/dQ.

The ordinary differential equations (3) for the cell averages are integrated using a second-order TVD Runge–Kutta method [8],

$$\begin{aligned} Q_{ij}^{(1)} &= Q_{ij}^n + \Delta t R_{ij}(Q^n), \\ Q_{ij}^{n+1} &= \frac{1}{2} Q_{ij}^n + \frac{1}{2}\Big[Q_{ij}^{(1)} + \Delta t R_{ij}(Q^{(1)})\Big], \end{aligned} \tag{8}$$

where R_{ij} denotes the right-hand side of (3). The time step is restricted by a CFL-condition, which states that disturbances can travel at most one half

grid cell each time step, i.e., $\max(a^+, -a^-)\Delta t \leq \Delta x/2$, and similarly in y.

As noted above, (1) admits steady-state solutions where non-zero flux gradients are balanced by the topographical source terms, i.e.,

$$[hu^2 + \frac{1}{2}gh^2]_x + [huv]_y = -ghB_x,$$

and similarly in the y-direction. Many physically interesting phenomena are perturbations of steady states. To accurately compute the time evolution of such perturbations, we must ensure that our scheme does not produce errors of the same magnitude as the waves we want to resolve. We would therefore like the spatial truncation error to be zero at steady states. This is an important and challenging problem.

Lake-at-rest is the name given to an important family of steady states for (1); it is characterized by the relations $hu = hv = 0$ and $h + B = \text{Const}$. By reconstructing the surface elevation $w = h + B$ rather than the water depth h, and by a choosing a special quadrature rule for the cell-averaged source term S_{ij}, the spatial part of the truncation error vanishes at these states. The scheme thus preserves steady states numerically, see [6]. The quadrature rule for the second component of the source term is given by

$$\begin{aligned} S_{ij}^{(2)} =& -\frac{1}{|\Omega_{ij}|}\int_{y_{j-1/2}}^{y_{j+1/2}}\int_{x_{i-1/2}}^{x_{i+1/2}} g(w-B)B_x\,dxdy \\ \approx& -\frac{g}{2\Delta x}\Big(h^L_{i+1/2,j-\alpha} + h^R_{i-1/2,j-\alpha}\Big)\Big(B_{i+1/2,j-\alpha} - B_{i-1/2,j-\alpha}\Big) \\ & -\frac{g}{2\Delta x}\Big(h^L_{i+1/2,j+\alpha} + h^R_{i-1/2,j+\alpha}\Big)\Big(B_{i+1/2,j+\alpha} - B_{i-1/2,j+\alpha}\Big), \end{aligned} \tag{9}$$

and a similar construction is used for the third component of S_{ij}.

When water depths approach zero, reconstructing w will not guarantee nonnegative point-values for h. Negative h-values are both physically incorrect and very undesirable in the numerics. On the other hand, by reconstructing h rather than w, the scheme is guaranteed to yield nonnegative water depths (under a more restrictive time step). Therefore, as a reasonable compromise, one can choose a threshold K, and use the following strategy: reconstruct point values from the physical variables $[h, u, v]$ if $h < K$ and from $[w, hu, hv]$ otherwise. A version of this scheme is presented in [6]. Notice that this compromise is not well-balanced in the first component, i.e., it produces nonzero flux terms for lake-at-rest.

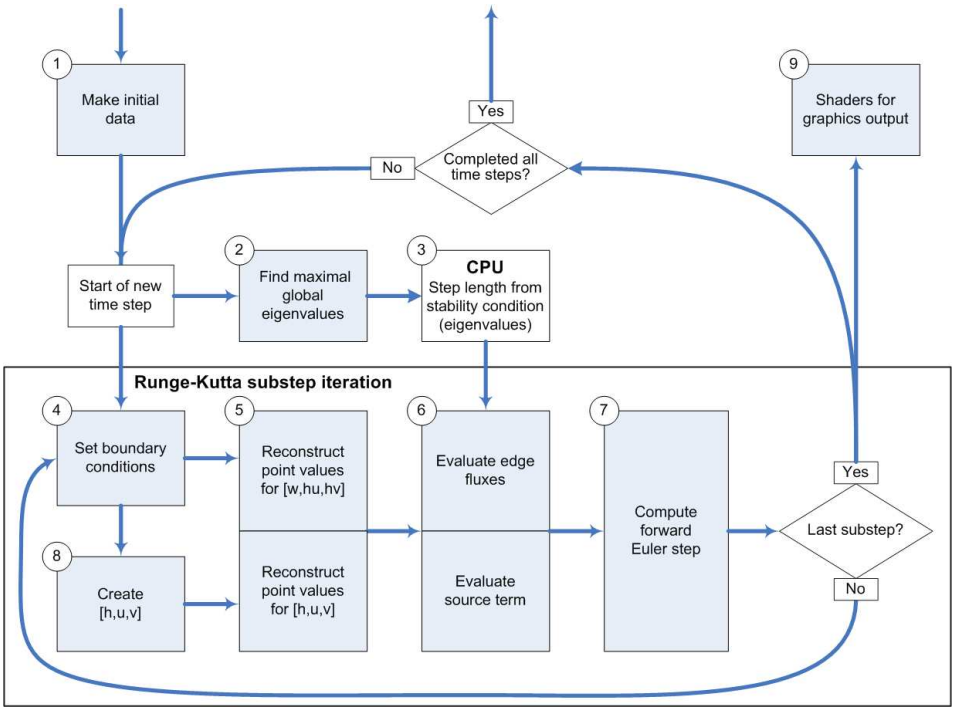

Fig. 2. Flow chart for the GPU implementation of the semi-discrete finite-volume scheme. Gray boxes are executed on the GPU and white boxes on the CPU.

3 Implementation on the GPU

We have implemented the method presented above using OpenGL and Cg to utilize the GPU's capabilities for floating-point processing. Using GPUs for non-graphical computations are generally known as GPGPU (General-Purpose Computation using Graphics Hardware), and an introduction to techniques commonly used in this field can be found in [9]. The general setup is to render a triangle that covers the entire viewport, and employ a fragment shader as the computational kernel. A fragment shader is a program, which the GPU executes for each pixel. For a quadrilateral domain it is more natural to render a quad than a triangle. However, the GPU is optimized for rendering triangles, and this choice results in a performance gain.

Algorithms must be divided into steps in such a way that each data cell can be calculated independently within a step in the algorithm. Each step can be executed by rendering a triangle covering the viewport—which causes the fragment processor to execute the fragment shader, thereby calculating the new cell values. Figure 2 illustrates the different steps and the data flow in the implemented algorithm.

Initial data ① are created either by using height maps of the bottom topography and water level or by procedural textures. The CFL stability condition is determined by the global maximum of eigenvalues ② in the grid cells. To

find this maximum we use an 'all-reduce' operation utilizing the depth buffer combined with read-back to the CPU for the final calculations ③. Here we divide the domain into smaller sub-domains and render the eigenvalues of each sub-domain into the depth buffer. The depth buffer used has the same dimension as the sub-domains. The effect of this is that one of the values of the depth buffer must be the largest eigenvalue. This depth buffer is then read back to the CPU, which picks the global maximum. We did not use the GPU to determine the global maximum because the application requires the maximum value to be read back to the CPU anyway. Compared to a linear search through all eigenvalues by the CPU, the depth buffer technique gives a performance gain—which mainly is due to the efficiency of the GPU's depth test. With this method, this part of the algorithm represents less than 5% of the total computational cost.

The Runge–Kutta method for advancing the solution in time is comprised of two iterations. Each iteration begins with a fragment shader that sets the boundary conditions ④. The next step is to reconstruct point values ⑤ from the cell averages. For cases with positive water depth, it is only necessary to run the fragment shader for the variables $[w, hu, hv]$. For computations with dry states, we also need to reconstruct point values of $[h, u, v]$. In this case we precompute $[h, u, v]$ at step ⑧, and then apply the reconstruction shader to both sets of variables.

The most computationally intensive step is the evaluation of edge fluxes and source terms ⑥. The time step Δt is passed to this shader by the CPU when the stability calculations of step ③ are done. For the simplest case with constant bottom topography, $\nabla B = 0$, the source terms are zero, and the shader computes only the flux integrals (4). For the more complex case of varying bottom topography, the source-term integrals (9) are also needed. When the simulation involves dry states, branching is needed at this step to determine if one should use the reconstruction of $[w, hu, hv]$ or reconstruction of $[h, u, v]$ for the calculations. The design of the graphics hardware is not optimal for handling branching efficiently, so generally one should aim at using algorithms with as few branches as possible. Eliminating the branch in the scheme used in this paper seems rather difficult—for cases with dry states, computational performance may therefore be better with other schemes; see e.g., [10].

The forward Euler step ⑦ corresponds to equation (8). By repeating the execution of the shaders inside the forward Euler box, one arrives at the second-order Runge–Kutta scheme. A number of different shaders are used for output ⑨ to the screen, exemplified in Figure 3.

Table 1
Runtime in milliseconds per time step and speedup factor ν for a CPU versus a GPU implementation of the Lax–Friedrichs and the second-order central-upwind scheme, both without source terms and dry states. The case is the circular dambreak problem run on a set of uniform grids with $N \times N$ cells.

	Lax–Friedrichs			2nd order central-upwind		
N	CPU [ms]	GPU [ms]	ν	CPU [ms]	GPU [ms]	ν
128	2.22	0.23	9.53	30.6	1.27	24.2
256	9.09	0.46	19.8	122	4.19	29.1
512	37.1	1.47	25.2	486	16.8	28.9
1024	148	5.54	26.7	2050	68.3	30.0

4 CPU versus GPU

In this section we compare CPU and GPU implementations of the two schemes presented in Section 2. We measure runtimes and explore the factors that affect the relative speedup. The GPU is a NVIDIA Geforce 7800 GTX and the CPU is a 2.8 GHz Intel Xeon (EM64T).

Case 1. The first test case is presented in [11], and we give the results here for the convenience of the reader. We consider a simple circular dambreak problem over the domain $[-1.0, 1.0] \times [-1.0, 1.0]$ with absorbing boundary conditions. Th water surface is initially at rest with height $h = 1.0$ inside a circle of radius 0.3 and height $h = 0.1$ outside.

We compute the solution using both the first-order Lax–Friedrichs scheme (2) and the second-order central-upwind scheme described in Section 2. Runtimes for CPU and GPU implementations are reported in Table 1. The Lax–Friedrichs scheme requires a smaller number of arithmetic operations per grid cell than does the central-upwind scheme. The timings reported here therefore show that the speedup is better for schemes with a high count of arithmetic operations per time step (similar behavior has been observed by the authors for numerous other cases; a few of these are reported in [11]).

Case 2. We consider lake-at-rest defined by a variable bottom topography $B(x, y) = \max(0, 1 - x^2 - y^2)$ and a stationary flat water surface $h(x, y, 0) = \max(w_0 - B(x, y), 0)$. For $w_0 = 1.01$, the water depth is strictly positive and the solution is stationary and exactly preserved by the central-upwind scheme. Runtimes per time step for the CPU and the GPU are reported in Table 2 along with corresponding speedup factors ν. Compared with Table 1, we notice a slight increase in runtimes, yet the speedup factor ν does not change. This indicates that the added complexity by introducing the well-balanced

Table 2
Runtime in milliseconds per time step and speedup factor ν for a CPU versus a GPU implementation of the second-order central-upwind scheme. The case is 'lake-at-rest' with surface level w_0, computed on a set of uniform grids with $N \times N$ cells. For $w_0 = 1.01$ we used the simpler scheme without the switch for dry-regions.

	without dry states, $w_0 = 1.01$			with dry states, $w_0 = 0.9$		
N	CPU [ms]	GPU [ms]	ν	CPU [ms]	GPU [ms]	ν
128	32.7	1.35	24.2	35.2	2.38	14.7
256	130	4.40	29.5	143	8.09	17.7
512	518	17.2	30.1	599	31.9	18.8
1024	2140	69.8	30.6	3270	142	23.0

treatment of the variable bottom topography in Table 2 has more or less the same effect on the CPU and on the GPU.

For $w_0 = 0.9$, we have zero water depth in parts of the domain. The measured runtimes and speedup factors are reported in Table 2. In this case, the branching described in Section 2 is needed to ensure nonnegative water depth. As pointed out in Section 3, branching is not as natural on the GPU as on the CPU and may therefore increase the runtime per time step. To avoid data-dependent branching in the reconstruction of point values, both sets of variables are reconstructed in the GPU implementation; compare with the right-hand part of Table 1, and notice that the branching almost doubles the runtime. The CPU implementation, on the other hand, only reconstructs the needed variables. Though computations are relatively inexpensive, this contributes to the reduced speedup.

Flood waves caused by a dambreak. In Figure 3, we have included four snapshots of a dambreak simulation on the GPU. Initially, the water in the upper and lower lake is at rest. When the water in the upper lake is released through the narrow canyon, it generates flood waves and vortices in the lower lake. The wave motion and the increased water level in the lower lake makes the water flow into the valley below before it again comes to rest. The simulation is interactive in the sense that it has a fly-through mode that allows the user to inspect the solution while it is being computed.

The purpose of the simulation is to give a qualitative description of the major flood waves according to the shallow water model. If we were to use the simulation for visual purposes within computer animation, several approaches could have been taken to make the water surface look more realistic. First of all, we would have chosen a less smooth bottom topography and initial water surface. Second, one could add artificial visual water effects, see e.g., [12].

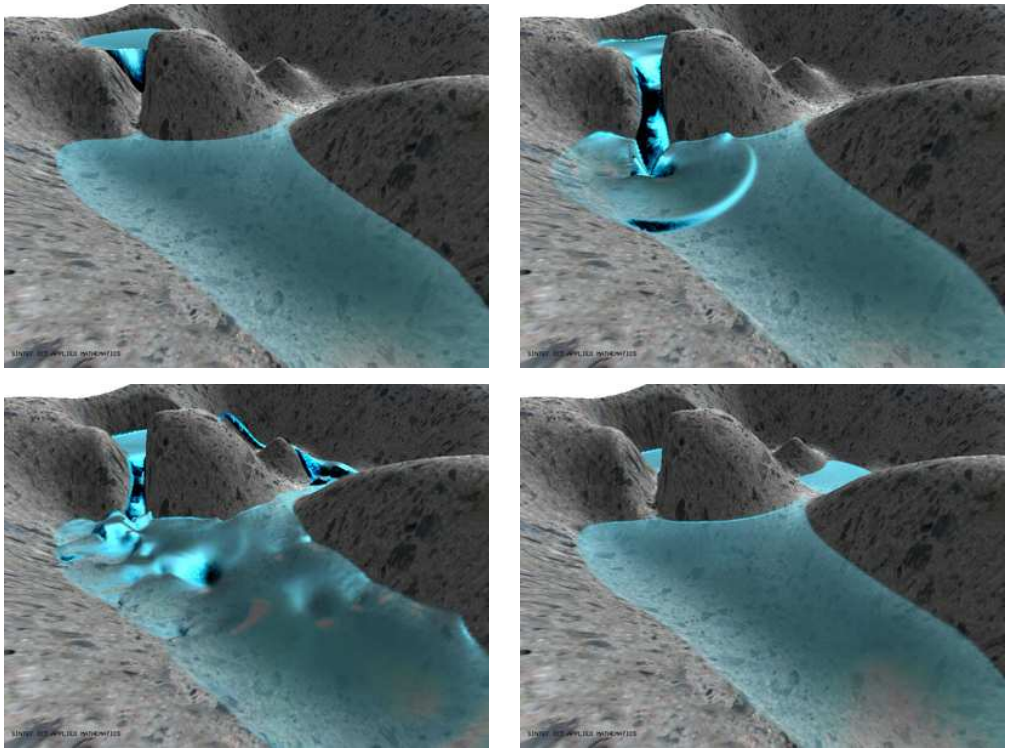

Fig. 3. Snapshots of a dambreak simulation in artificially constructed terrain.

5 Final Comments

In this paper we have demonstrated the applicability of the GPU as a computational resource for PDE-based simulations of gravity-driven surface waves in shallow waters. Modern numerical schemes for such models are inherently parallel in the sense that very little global communication is needed in the computational domain to advance the solution forward in time. Therefore, this application can readily exploit the parallel architecture of modern GPUs. We have seen in practical computations that moving from a serial CPU-based implementation to an implementation on a modern GPU decreases the runtime by more than one order of magnitude. (The runtime of the full dambreak simulation was reduced from two hours to five minutes!) To achieve the same speedup using CPUs, one would have to resort to a cluster of twenty or more processing nodes.

In our experience, numerical solution of conservation laws and balance laws are as reliable on the GPU as on the CPU; the single-precision arithmetic of the GPU does not negatively affect the computations. Indeed it should not, since the methods we have examined are numerically stable. As we have seen, the ratio of simulation speeds on the GPU and the CPU varies. It is interesting to identify what causes these variations. In this paper two effects seem clear. In the first place, the advantage of the GPU is increased if the ratio

between arithmetic operations and memory access is well balanced. This allows the application to take advantage of both the memory bandwidth and the computational power of the GPU. In our applications, we observe that the speedup is better for the high-resolution scheme than for the arithmetically less demanding Lax–Friedrichs scheme. The reason is probably that the larger number of arithmetic operations allows the GPU to fetch data while calculating.

Secondly, data dependent branching is expensive. There exist several known methods to handle this challenge, but the results are highly dependent of the input data and the graphics hardware. We did not invest a large amount of time to find the best possible solution for our hardware, but aimed for a reasonable compromise. Due to the architecture of currently available GPUs, fragments that are being evaluated simultaneously must execute the same branch. Thus the evaluation of some fragments must wait for its neighbors. In our application, however, the effect of this is small because the outcome of the conditionals are mainly the same for fragments closely related in screen space. It seems that much of the performance loss when data dependent branching is involved happens because the compiler looses some of its freedom to make optimizations. Since data dependent branching in the pixel shader is a relatively new feature of GPUs, compilers are likely to improve in the future when it comes to optimization of the code. The branching intensive shader in our application is compiled to over 400 instructions—therefore we did not make any attempt to optimize the assembly version by hand.

It seems clear that the computing resources on the GPU have many applications outside of computer graphics. First of all, the computing power of desktop computers can be vastly increased with modest expenses. This can have a great impact in end-user software since one may be able to remove bottlenecks in computationally expensive applications. Computations that today are done in batch mode may with the use of the GPU become interactive. With interactivity and fast visualization, new applications of PDE-based physics are possible, ranging from process steering and control, to computer games and educational simulators.

References

[1] M. J. Harris, G. Coombe, T. Scheuermann, A. Lastra, Physically-based visual simulation on graphics hardware, in: HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2002, pp. 109–118.

[2] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation

of numerical algorithms, ACM Transactions on Graphics (TOG) 22 (3) (2003) 908–916.

[3] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the gpu: conjugate gradients and multigrid, ACM Trans. Graph. 22 (3) (2003) 917–924.

[4] R. LeVeque, Finite volume methods for hyperbolic problems, Cambridge Texts in Applied Mathematics, Cambridge University Press, Cambridge, 2002.

[5] A. Kurganov, S. Noelle, G. Petrova, Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations, SIAM J. Sci. Comput. 23 (3) (2001) 707–740.

[6] A. Kurganov, D. Levy, Central-upwind schemes for the Saint-Venant system, M2AN Math. Model. Numer. Anal. 36 (3) (2002) 397–425.

[7] D. Levy, G. Puppo, G. Russo, Compact central WENO schemes for multidimensional conservation laws, SIAM J. Sci. Comput. 22 (2) (2000) 656–672.

[8] C.-W. Shu, Total-variation-diminishing time discretisations, SIAM J. Sci. Stat. Comput. 9 (1988) 1073–1084.

[9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, in: Eurographics 2005, State of the Art Reports, 2005, pp. 21–51.

[10] E. Audusse, F. Bouchut, M.-O. Bristeau, R. Klein, B. Perthame, A fast and stable well-balanced scheme with hydrostatic reconstruction for shallow water flows, SIAM J. Sci. Comp. 25 (2004) 2050–2065.

[11] T. Hagen, M. Henriksen, J. M. Hjelmervik, K.-A. Lie, How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine, in: G. Hasle, K.-A. Lie, E. Quak (Eds.), Geometric Modelling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF, Springer-Verlag, 2005, to appear.

[12] J. Tessendorf, Simulating ocean water, SIGGRAPH 2004 Course Notes.