Noname manuscript No.
(will be inserted by the editor)

Simulation and Visualization of the Saint-Venant System using GPUs

André R. Brodtkorb ·
Trond R. Hagen ·
Knut-Andreas Lie ·
Jostein R. Natvig

Received: date / Accepted: date

Abstract
We consider three high-resolution schemes for computing shallow-water waves as described by the Saint-Venant system and discuss how to develop highly efficient implementations using graphical processing units (GPUs). The schemes are well-balanced for lake-at-rest problems, handle dry states, and support linear friction models. The first two schemes handle dry states by switching variables in the reconstruction step, so that that bilinear reconstructions are computed using physical variables for small water depths and conserved variables elsewhere. In the third scheme, reconstructed slopes are modified in cells containing dry zones to ensure non-negative values at integration points. We discuss how single and double-precision arithmetics affect accuracy and efficiency, scalability and resource utilization for our implementations, and demonstrate that all three schemes map very well to current GPU hardware. We have also implemented direct and close-to-photo-realistic visualization of simulation results on the GPU, giving visual simulations with interactive speeds for reasonably-sized grids.

Keywords GPU · Shallow Water · Saint-Venant · Conservation Laws · Visualization · Finite Volume · High-Resolution Scheme

1 Introduction

Accurate simulations of shallow water waves as described by the Saint-Venant system are highly important in many application areas. Herein, our primary interest is flood simulation and interactive studies of multiple scenarios for flood prevention, for which a main challenge is simulation time: the computational power of standard CPUs cannot provide the performance needed for grids with sufficient resolution. In an effort to overcome this problem, we present efficient implementations of three high-resolution schemes for the Saint-Venant system [11, 7, 13] on graphical processing units (GPUs). Leveraging the computational power of a GPU can potentially provide close to real-time simulation and visualization, thereby significantly improving user interactivity. Our work contributes a state-of-the-art implementation of explicit finite-volume schemes on modern graphics hardware, including interactive visualization with photo-realistic effects. The particular schemes capture steady states like the lake-at-rest case, support dry states, and include simple source terms accounting for bottom friction. Verification against analytical solutions and validation against experimental data for the Malpasset dambreak are described in a separate paper [5].

Graphical processing units have in recent years developed from being hardware accelerators of computer graphics into high-performance computational engines. The use of GPUs in scientific computing has gone from early proof-of-concept studies around ten years ago (e.g., matrix operations carried out by using graphics operations [14]), to the current widespread use. Examples from a wide range of applications show how one can obtain a significant computational speedup by harnessing the computational power of a GPU [20]. A comprehensive description of current state-of-the-art GPU technology, including hardware, software, and algorithms, can be found in Brodtkorb et al. [4].

GPUs are stream processors that operate in parallel by running a single kernel on multiple instances of a data stream. This type of parallelization is particularly well suited for the stencil computations that constitute an explicit high-resolution scheme. The idea of using GPUs to accelerate (high-resolution) schemes for systems of conservation and balance laws is not new. To the best of our knowledge, it was first suggested in 2005 by Hagen et al. [7] for the Saint-Venant system and then later for the Euler equations of ideal gas dynamics [8]. Using OpenGL, the authors demonstrated how the stencil computations of several classical and high-resolution schemes could be implemented as operations in the fragment processing units, see [6]. Moreover, for systems of conservation laws, one could utilize the vector operations of four-component graphics (RGBA) to obtain acceleration beyond the number of parallel pipelines. Compared with a highly tuned CPU implementation, speedup factors in the range 15–30 were observed. Another important observation was that explicit schemes for hyperbolic conservation laws and balance laws are memory bound, and hence larger speedups were observed for high-resolution schemes

SINTEF ICT, Dept. Appl. Math., P.O. Box 124, Blindern, NO-0314 Oslo, Norway. {Andre.Brodtkorb, Trond.R.Hagen, Knut-Andreas.Lie, Jostein.R.Natvig}@sintef.no

that are more compute intensive than classical schemes like Lax–Friedrichs, Lax–Wendroff, etc. Since then, there have been several publications devoted to the use of GPUs for the shallow-water equations and other conservation and balance laws, see e.g., [2,3,22,16,10,26,15,1].

In the current paper, we revisit the shallow-water simulations from [7], now using implementations in CUDA rather than OpenGL to give an up-to-date demonstration of the feasibility of GPU computing for the Saint-Venant system. The two main points in the paper are: (i) a discussion of how to implement high-resolution schemes as efficiently as possible on current GPUs, and (ii) a comparison of the efficiency of GPU implementations of the Kurganov–Levy [11] and Kurganov–Petrova [13] schemes. In assessing computational efficiency, it has become quite popular to report speedup factors compared with a CPU implementation, and the literature is filled with optimistic figures that report several orders of magnitude speedups. Unfortunately, most of these findings are overly optimistic (and not examples of good science); by comparing theoretical performance numbers for GPUs and CPUs, it is easy to see that speedup factors exceeding 100 are very unlikely on current hardware. Herein, we will therefore instead consider the degree of resource utilization, which in our opinion is a better measure of how well a particular algorithm maps to the GPU.

2 Model Equations

Waves in shallow waters can be described by the following Saint-Venant system

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2+\frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2+\frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x-\kappa(h)u \\ -ghB_y-\kappa(h)v \end{bmatrix}. \quad (1)$$

Here h is the water depth, hu is the discharge along the x-axis, hv is the discharge along the y-axis, g is the gravitational constant, and B is the bathymetry (see Figure 1). On vector form, we can write the equation as

$$Q_t + F(Q)_x + G(Q)_y = H(Q, \nabla B), \quad (2)$$

where Q is the vector of conserved variables, F and G are flux functions, and H represents the source terms. The friction source term is assumed to be linear in velocity with a constant of proportionality that depends on h,

$$\kappa(h) = \frac{\alpha h}{1+\beta h}. \quad (3)$$

For all synthetic test cases considered herein, α and β have been set, rather haphazardly, to 10^{-2} and 10^{2}, respectively.

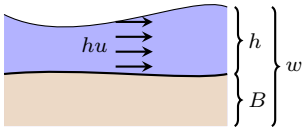


Fig. 1: Variables in the shallow-water equations in one dimension: h is the water depth, B is the bathymetry, w is the total water elevation, and hu is the discharge.

3 Numerical Schemes

There are many aspects to consider when studying numerical methods for the Saint-Venant system. First of all, it is important for any numerical method to be conservative. Second, the method should be accurate on smooth parts of the solution and not create spurious oscillations near discontinuities or sharp transitions in the solution. Moreover, many simulations are perturbations of a steady state. Consider, for example, a lake at rest, in which the hydrostatic contributions to the flux in (1) perfectly balances the bathymetry gradient in the source term. An ideal method should therefore be well balanced in the sense that source terms and fluxes balance exactly also in the discretized equations for zero velocities.

Likewise, to simulate inundating (flooding), we require that the scheme does not break down in the presence of dry states ($h = 0$) and that it is well-behaved in *shoal* zones (h very small). Solving the Saint-Venant system numerically with dry states is difficult. To compute numerical fluxes, one will typically have to divide quantities by the water depth h. As h approaches zero, we get divisions by very small numbers, resulting in large errors in the fluxes. To make matters worse, if the water depth becomes negative, the whole computation breaks down since the eigenvalues of the system are $u \pm \sqrt{gh}$.

High-resolution schemes. There are many good schemes available in the literature that satisfy the critaria above. Herein, we are mainly interested in problems characterized by strong discontinuities, which typically can be satisfactorily resolved using a well-balanced second-order scheme with capabilities for resolving dry states. For other types of problems involving more smooth phenomena, e.g., the formation of eddies in shelf-slope jets [21], well-balanced schemes of higher order may be required [17]. In choosing among different second-order schemes, our previous experience is that the Kurganov–Levy scheme [11], and its slightly modified version reported in [7], offer a good compromise between simplicity of implementation and efficiency, accuracy, and robustness for the simulation scenarios considered herein. In addition, we consider an improved version developed by Kurganov and Petrova [13], which allows discontinuities in

the bathymetry, contains less branching, requires less shared memory, and has been verified against both analytical and experimental data [24].

The three second-order, semi-discrete, central-difference schemes considered herein are based on the same basic discretization principles on a regular Cartesian mesh, using the generalized minmod flux limiter to obtain a high-resolution [9,25] non-oscillatory reconstruction. We start by integrating (1) over each cell in the mesh to obtain a system of evolutionary equations for the cell averages Q_{ij} of the conserved quantities Q,

$$\begin{aligned}\frac{dQ_{ij}}{dt} =& H(Q_{ij}, \nabla B) - \big[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\big] \\ & - \big[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\big] \\ =& R(Q_{ij}).\end{aligned} \tag{4}$$

Here $F_{i,\pm 1/2,j}$ and $G_{i,j\pm 1/2}$ denote the fluxes over the cell interfaces in the x and y-directions. Then the temporal evolution of cell averages Q_{ij} in cell ij can be approximated using a second-order stability-preserving Runge–Kutta method,

$$\begin{aligned} Q^*_{ij} &= Q^n_{ij} + \Delta t R(Q^n_{ij}) \\ Q^{n+1}_{ij} &= \tfrac{1}{2} Q^n_{ij} + \tfrac{1}{2}\left[Q^*_{ij} + \Delta t R(Q^*_{ij})\right]. \end{aligned} \tag{5}$$

The timestep in the Runge–Kutta solver is restricted by a CFL condition,

$$\Delta t \leq \tfrac{1}{4} \min\left\{ \frac{\Delta x}{\max_\Omega |u \pm \sqrt{gh}|}, \frac{\Delta y}{\max_\Omega |v \pm \sqrt{gh}|} \right\} \tag{6}$$

that limits the propagation of waves to one quarter of a grid cell per timestep.

From (5), we see that we need to compute the flux and source terms twice for each timestep. To compute fluxes, we introduce a quadrature rule for the spatial integration over each cell interface and hence express each flux as a weighted sum of *point values* of the flux functions F or G. To compute these point values, we *reconstruct* a bilinear approximation of Q inside each grid cell. The slope in each spatial direction is computed as a nonlinear combination of the forward, central and backward differences using the cell averages $Q_{ij}, Q_{i\pm 1,j}$ and $Q_{ij}, Q_{i,j\pm 1}$, respectively. In each integration point, we hence obtain two one-sided point values, reconstructed in the two cells on opposite sides of the interface. These two values are combined through a *numerical flux function*; herein we use the central-upwind flux [12]. Finally, the source term can be computed by approximating ∇B from the bathymetry evaluated at the cell vertices, see Figure 2. The resulting stencil is obviously highly parallel, arithmetically intensive, and hence very suitable for GPUs [7].

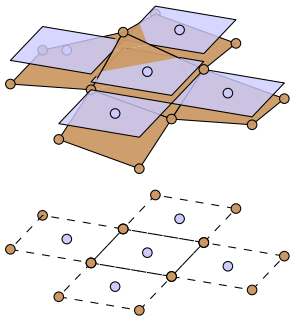


Fig. 2: Reconstruction of surface elevation and bathymetry. For a bilinear reconstruction, the cell averages coincide with the values at the cell centers. The bathymetry is approximated by its values at the cell vertices.

Kurganov–Levy (KL02). To cope with the problem of dry zones, Kurganov and Levy [11] proposed to use a different reconstruction in shoal and wet zones. For the wet zones, they proposed to perform reconstruction and flux calculations based on the variables $U = [h + B, hu, hv]$ rather than on the conserved variables. By using special quadrature rules and discretizing the source term appropriately, reconstruction from these variables leads to a well-balanced and conservative scheme. However, the scheme does not guarantee a non-negative water depth h.

To guarantee non-negative values, they use another reconstruction based on the physical variables $W = [h, u, v]$ in the shoal zones given by $h < K$ for some small prescribed constant K. The resulting scheme is unfortunately not well-balanced and will cause global errors in conservation. Moreover, spurious waves can emerge initially in the shoal zones, but here the solution rapidly reaches a steady state, in which the fluxes balance the source terms. The spurious waves therefore only have a small effect on the global solution.

Modified Kurganov–Levy (KLL05). The Kurganov–Levy scheme (KL02) uses one integration point per cell interfaces to compute fluxes. This limits the scheme to second-order accuracy. Hagen et al. [7] therefore proposed a slightly modified scheme that uses a two-point interior Gaussian quadrature along each interface. This quadrature rule is accurate for reconstructions up to fifth order and thus supports higher-order reconstructions, including the WENO reconstruction [23], as used in [6] for gas dynamics.

Kurganov–Petrova (KP07). Whilst the KL02 and KLL05 schemes avoid negative values for h by switching to physical variables, the Kurganov–Petrova scheme [13] is based on adjusting the reconstruction for cells where the value at the integration points will become negative. If the reconstructed slope creates negative values at the integration

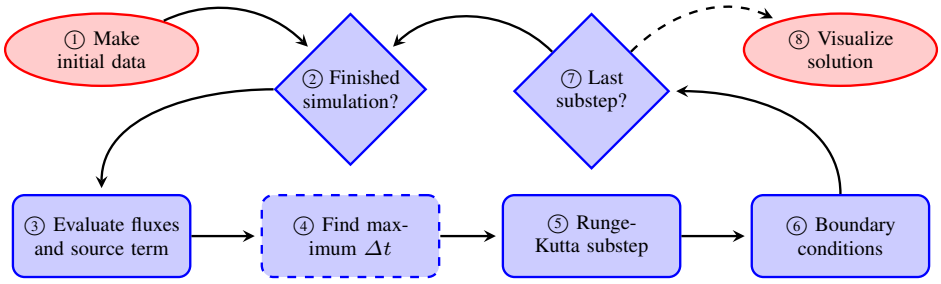


Fig. 3: Program flow for the implemented schemes. Each of the four bottom boxes represents a computational kernel that executes on the GPU. The visualization is also performed directly on the GPU, without copying data over the PCI express bus.

points, the steepness of the water slope is adjusted (reduced or increased) so that the negative value at the integration point becomes zero. This guarantees that all water depths used in the calculations are non-negative. However, very small water depths can still create large errors in the flux calculations. The KP07 scheme handles this by desingularizing the calculated velocity used in the flux calculation for shoal zones:

$$u = \frac{\sqrt{2}h(hu)}{\sqrt{h^4 + max(h^4, \epsilon)}}. \tag{7}$$

This slope fix will ultimately affect the fluxes in shoal zones and thus compromise the well-balanced property. However, as with the KL02 and KLL05 schemes, the fluxes rapidly balance the source term, and these spurious initial waves have small effects on the solution. One thing should be noted, though. The slope fix, and the corresponding errors introduced in the solution, depend strongly on the slope of the bathymetry. If the slope fix is triggered for a cell with a very steep bottom slope, one obtains a very steep water slope as well. Moreover, generalizations of this fix to higher-order reconstructions are difficult, if possible at all. This is because one can only alter the slope to guarantee non-negativeness at one point per cell interface, and higher-order reconstruction require more integration points.

The KL02 and KLL05 schemes assume that the bathymetry B is given as a continuous function sampled at the integration points. The KP07 scheme, on the other hand, assumes that the bathymetry is bilinear within each cell. Using this assumption, discontinuous bottom surfaces can be handled and approximated by a piecewise bilinear function.

4 Implementation

We have implemented our solver using C++ and NVIDIA CUDA [18], with heavy use of templates for both the CPU and GPU-parts of the code. We have grouped computations into a set of four kernels, as shown in Figure 3, to best suit the architecture of current GPUs and still fulfill the requirements of the algorithm.

Figure 3 illustrates the program flow of our implementation. We start by initializing the computational domain and data storage between kernels in ①. In total, we need sixteen buffers the size of the computational domain: three to hold U, three to hold Q, two to hold H, six to hold F and G, one to hold B at cell vertices, and one to hold B at the cell centers. We have precomputed B at both cell centers and vertices as a performance optimization: the flux kernel ③ needs the vertex values, whilst the Runge–Kutta kernel ⑤ requires the value at cell centers to ensure non-negative water depths. We also need a buffer to hold the eigenvalues computed in the flux kernel, ③, and used in the maximum Δt kernel, ④, to compute the timestep. This buffer, however, is much smaller than the other buffers, as will be explained later.

After allocating and initializing buffers, we enter the main simulation loop, which contains one or more Runge–Kutta substeps (③–⑥). For each substep, we start by reconstructing a piecewise planar function for each grid cell and evaluate the fluxes and source terms in ③. For the first substep, we also compute the maximum eigenvalues in ③, and reduce them to the global maximum in ④. Then we can compute the maximum timestep satisfying the CFL condition (see (6)) and solve the ODEs (5) in ⑤. Finally, we can set the values of the global ghost cells to impose boundary conditions in ⑥. After the final substep, we may also opt to visualize the current solution.

The following describes in detail our implementation of the kernels used for the KP07 scheme. The other two schemes, KL02 and KLL05, use similar ideas and optimizations.

4.1 Block Decomposition

CUDA uses the concept of blocks to structure computation. Each block will execute independently of all other blocks and consists of a specified number of *threads*, the *block size*. Threads are organized in a logical 2D array, where threads

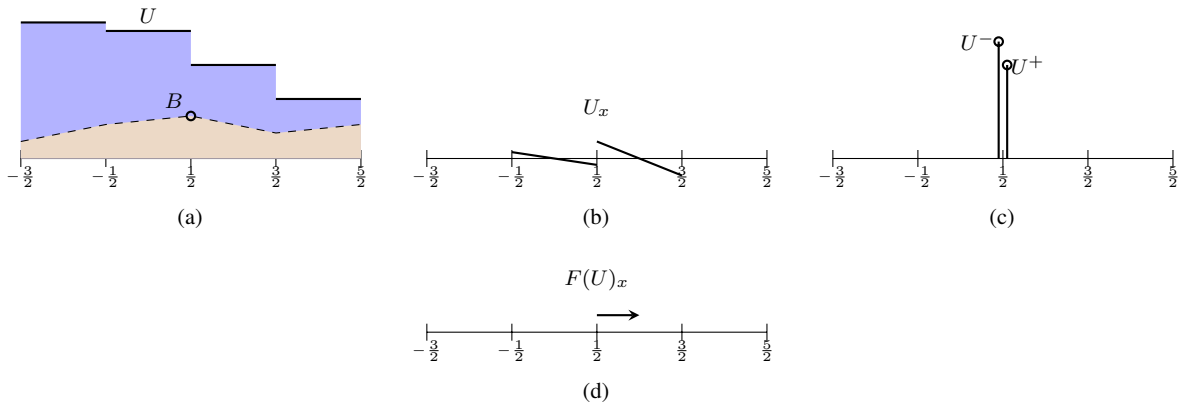


Fig. 4: Data needed to compute the flux across the interface at $i + \frac{1}{2}$. In (a), we have the input water elevation and velocities, from which we reconstruct the slopes (b). We then evaluate the water elevation and velocities at the integration points from the right and left cell in (c). Finally, we compute the flux using the values at the integration points (d).

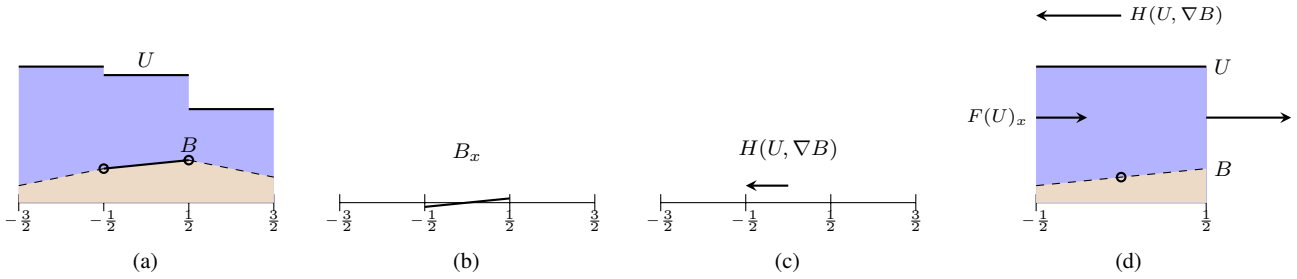
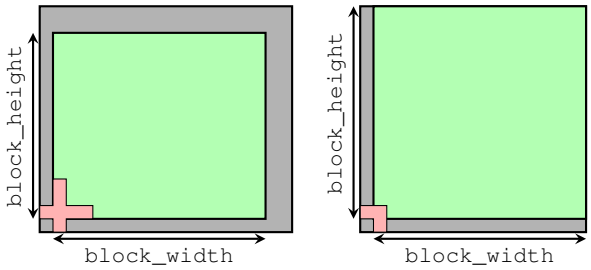


Fig. 5: Data needed to compute the source term at cell i and data needed to perform time integration. In (a), we have the input water elevation from which we reconstruct the slope (b). We then evaluate the source term using the water elevation and bottom topography slope (c). The Runge–Kutta substep kernel (d) simply evolves and averages the solution to the next substep using the original water elevation, computed fluxes and source term, and the average water depth (computed by subtracting B). We also need the bottom elevation to make sure our evolved solution gives non-negative water depths.

belonging to the same block can communicate and cooperate using *shared memory*. Shared memory is an on-chip programmable cache on NVIDIA GPUs, accessible to threads within the same block. Its maximum size is dictated by the physically available memory on each *streaming multiprocessor*, currently 16 KB.

Our finite-volume scheme is in essence a set of complex stencil computations. The fluxes are computed using a neighborhood of four cells, as shown in Figure 4, and the source terms are similarly computed using three cells, shown in Figure 5. The Runge–Kutta substep kernel, shown in Figure 5d, uses the fluxes and source terms from the previous kernel to evolve the solution. This means that we need to use blocks with overlapping input domains for these kernels, as shown in Figures 6 and 7. Finding a good block size is vital for high performance, but determining what this block size should be is a difficult problem with many optimization parameters. We have used a block size of 16×14 for the flux kernel, and 16×16 for the Runge–Kutta kernel. For the KL02 and KLL05 schemes, our block sizes for the flux kernel are even smaller, as they use shared memory for both the physical and the conserved variables. For double precision, all block sizes are effectively cut in half. Our block sizes have been chosen through general optimization guidelines and experimentation, and the following is a rationale behind the choices.

One optimization parameter is shared-memory access. Shared memory is organized into 16 banks, where one thread can access each bank every other cycle. When multiple threads access the same bank (also called *bank conflicts*), their access is serialized. Thus, we should ensure that the block width is a multiple of 17 to avoid bank conflicts horizontally and vertically. We also want to maximize our use of shared memory, which means that we want the ratio of internal cells to ghost cells to be as high as possible for each block. We do this by aiming for a square block size,



(a) Flux and source terms. (b) Runge–Kutta step.

Fig. 6: Domain decomposition. In (a), we show a single block with ghost cells for the kernel computing flux and source terms. The seemingly asymmetric data dependency is because we compute the flux across the east and north interfaces of each cell, see Figure 4. In (b), we show a single block with ghost cells for the Runge–Kutta kernel.

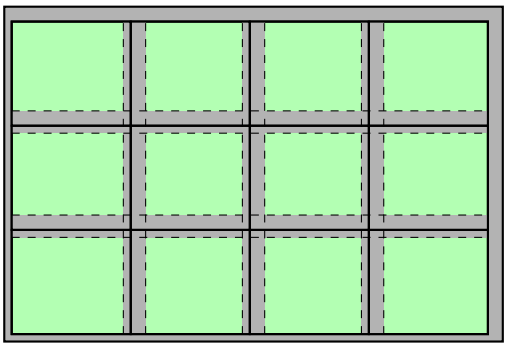


Fig. 7: Grid decomposition for our flux kernel. Within each block (solid lines), we compute the source term for all cells and the flux across the north and east interfaces. The global ghost cells are used to implement boundary conditions. Notice that our computational domain covers one of the global ghost cells to bottom and left of the domain. This is because our kernel computes the flux across the north and east cell interfaces (see also Figure 6). Also notice that the blocks read overlapping data from the global domain to satisfy data dependencies dictated by our stencils.

i.e., trying to equalize height and width. The block size of 16×14 gives a shared memory size of 19×17 which gives almost full use of shared memory, and the size is relatively square. This does not ensure that we have no bank conflicts, but it has been more important to ensure that the number of threads in the block is a multiple of 32, since the GPU executes *warps* of 32 threads in SIMD fashion.

To optimize memory access, we want to achieve *coalesced* reads and writes, which means that the width of data read into the kernel for each *warp* must be a multiple of 128 bytes, and that the starting address is aligned on a 128-byte boundary. Unfortunately, we cannot fulfill these two requirements at the same time for all blocks because our scheme requires overlapping blocks. To lessen the performance impact of slower global memory access, we use the texture cache to fetch data from global memory. It should be noted that the hardware used in our tests does not support double-precision texture fetches. Thus, we employ the standard technique of using an int2 representation during texture fetches, followed by reinterpreting the result as a double.

4.2 CUDA Kernels

Flux and Source Term (③). The kernel that computes flux and source terms is the main computational kernel in our solver. The kernel starts by reading data into shared memory, including the overlapping domain dictated by our stencils, shown in Figure 6a. Each thread (i, j) within each block is responsible for calculating the flux across the east $(i + \frac{1}{2}, j)$ and north $(i, j + \frac{1}{2})$ interface, in addition to the source term for cell (i, j). By examining the dependencies required by both the flux and the source term calculation, we see that we need one ghost cell to the south and west, and two ghost cells to the east and north.

The kernel begins by reading B and U into shared memory. From these, we reconstruct the slopes of U and calculate B at the integration points, totaling to twelve shared-memory variables. These variables are the ones needed by more than one thread. We compute the value of the bathymetry at the integration points in the kernel, as opposed to reading them from a precomputed buffer. This dramatically lessens the burden on the memory subsystem, and adds only a few extra computations. Reconstructing the slope of U is done using the branchless generalized minmod limiter [6], for which we use efficient bit operations to compute the sign function. To guarantee non-negative water depths at the integration points, we also correct the slopes for affected cells. This is done consistently in shared memory by choosing the slope that interpolates the bathymetry (at the negative integration point) and the average grid cell water elevation.

After reconstructing the slopes, we can evaluate U at the east and north integration points and compute the flux and the source term for the cell. If we are at the first Runge–Kutta substep, we also compute the eigenvalues and find the maximum within each block using reduction in shared memory. This is very efficient, as we reduce the number of elements the maximum Δt kernel ④ has to read by a factor $16 \times 14 = 224$ with our current block size. This also reduces the storage requirement, as mentioned in the beginning of this section.

Maximum Δt (④). The kernel that computes maximum Δt simply finds the maximum eigenvalue within the whole computational domain and computes the timestep Δt using (6).

We use a single block, where each thread loops through a strided subset. This ensures coalescing of data reads, thus maximizing memory performance of this memory-bound kernel. Once all eigenvalues for each thread have been considered, we perform in-block reduction between threads using shared memory. Finally, one thread computes and writes the maximum timestep to global memory.

Runge–Kutta (⑤). The Runge–Kutta substep kernel computes one substep of the Runge–Kutta ODE integrator (5). It is a memory-bound kernel that performs few computations, but accesses global memory many times. First, we read the fluxes F and G into shared memory. We then read the source term H, the existing solution U (and Q^* for the second substep), the bathymetry B, and finally the timestep Δt into registers for each thread. We then evolve the solution one substep. We also make sure all water elevations are non-negative, as floating-point round-off errors might cause negative water elevations.

Boundary Conditions (⑥). This kernel is quite similar to the maximum Δt kernel ④. The kernel is memory bound, as it performs very few computations. We only launch one block, which simply fills the ghost cells at the boundary with appropriate values. In our case, we have implemented wall conditions, i.e., we copy the cells closest to the boundary to the ghost cells and change the sign of the perpendicular velocity component. As an alternative to using a separate kernel to set boundary conditions, we could have used an extra buffer to identify boundary cells. This, however, would dramatically increase the load on the memory bus.

Other Optimizations. We need to pass a large amount of parameters to each of the kernels outlined above. For 32-bit operating systems, we can pass them in the normal fashion. However, for 64-bit systems, the size of pointers double, and exceed the maximum size allowed by CUDA. We thus use constant memory, which is auto-coalesced and cached global memory on the GPU. This enabled us to pass the parameters on 64-bit systems, and further proved to be a significant performance boost on 32-bit systems.

4.3 Visualization

The purpose of visualization is to improve our understanding of, and extract information from the simulation results. Hence, what variables to chose and what visualization techniques to use will strongly depend on what features of the solution on is interested in. Herein, we focus on producing a birds-eye view of the water surface and the surrounding terrain. To this end, we have implemented direct visualization of simulation results in OpenGL [19] with photo-realistic effects, as shown in Figure 8d. First, we render the terrain using a quadrilateral mesh where the nodes are displaced according to the height of the bathymetry B. The mesh is then draped with a texture and we use Phong shading (a method for calculating light reflected from surfaces by by interpolating surface normals across rasterized polygons) to compute per-pixel lighting. The water surface is also rendered using a quadrilateral mesh and displaced according to the water elevation w. We use the Fresnel equations to compute the angle of reflected and refracted rays, and the amount refraction. The reflected ray is then used to look up into the environment map (the skybox[1]), and our refracted ray is used to look up into the terrain texture. Environment mapping combined with reflection is a very good tool to spot discrepancies in the simulation, as our eyes rapidly detect imperfections in the mirror-like surface.

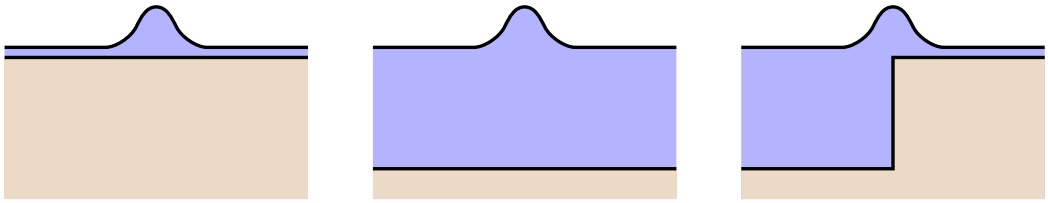
We visualize the bathymetry given at the center of each grid cell as a piecewise bilinear function, and the same is done for the cell averages of the water elevation. This means that the visualization is slightly erroneous and can give rise to visual artifacts where the water depth approaches zero (see along the shore in Figure 8d). However, we find it a reasonable approximation that gives users a good overview of the simulation results.

For each timestep to be visualized, we start by copying simulation results from CUDA memory to OpenGL texture memory. We accomplish this by copying from CUDA simulation memory to a CUDA mapped pointer of an OpenGL pixel-buffer object. Then, we synchronize the pixel-buffer object with an OpenGL texture. One optimization would be to remove the first of these copies and instead run the simulation using a CUDA mapped pointer to an OpenGL pixel-buffer object directly. However, this would increase the code complexity of the simulator and also couple the simulation code with the visualization code. The second copy might also seem superfluous, but is mandatory in current driver versions and is very efficient: it is performed entirely on the GPU, without the need to transfer data over the PCI express bus.

5 Numerical Experiments

To assess the performance of our implementation of the three schemes (KL02, KLL05, and KP07), we consider four different cases, see Figure 8. Case 1 consists of a bell-shaped water elevation over a flat bathymetry. In *Case 1a*, the water elevation is set very low so that the schemes interpret the solution to be in the shoal zone, in which dry-state reconstruc-

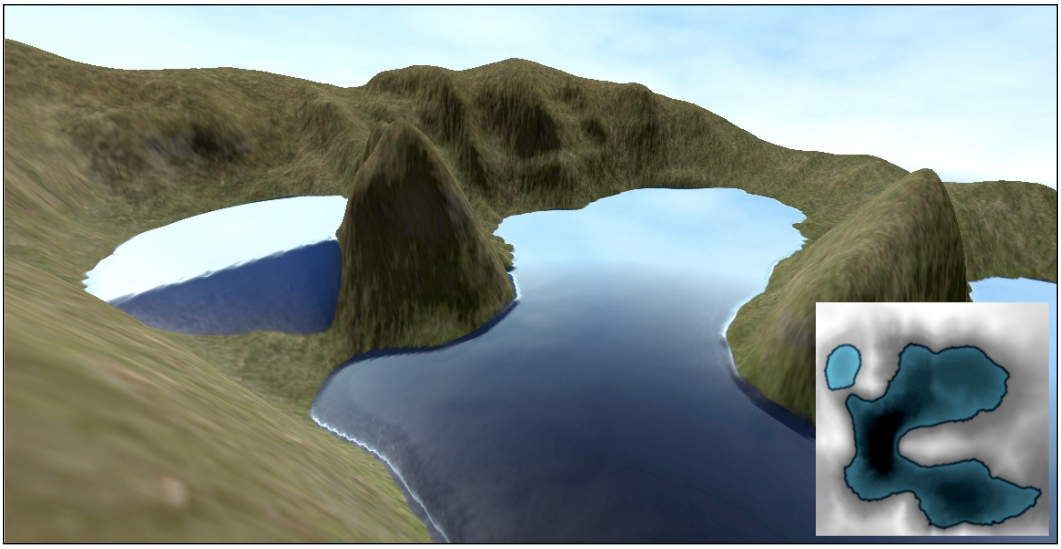
[1] Skyboxes are used in computer graphics to create the illusion that the displayed scene is larger than it actually is. The rendered schene is embedded inside a box, and images of the sky and the distant landscapes are projected on the faces of the box to illude the unreachable 3D space surrounding the scene.



(a) Case 1a: low water elevation.

(b) Case 1b: high water elevation.

(c) Case 2: discontinuous bathymetry.



(d) Case 3: dambreak simulation.

Fig. 8: The different test cases used in our benchmarks. For Cases 1a, 1b and 2, we have a 2D domain with a bell-shaped water elevation at the center of the domain. Case 3 is an synthetic terrain with a breaking dam. The height map is superimposed on the image.

tion is triggered for KL02 and KLL05, and desingularized flux computation (7) is triggered for KP07. In *Case 1b*, the water elevation is so high that the entire solution is in the wet zone. *Case 2* has the same setup, but now with a discontinuous bathymetry; this to illustrate the difference between the KL02/KLL05 and KP07 schemes. Finally, *Case 3* consists of a synthetic bathymetry that defines a "dambreak" simulation in which a high-altitude dam floods an underlying valley and lake terrain, creating a combination of wet regions, shoal regions, and dry states.

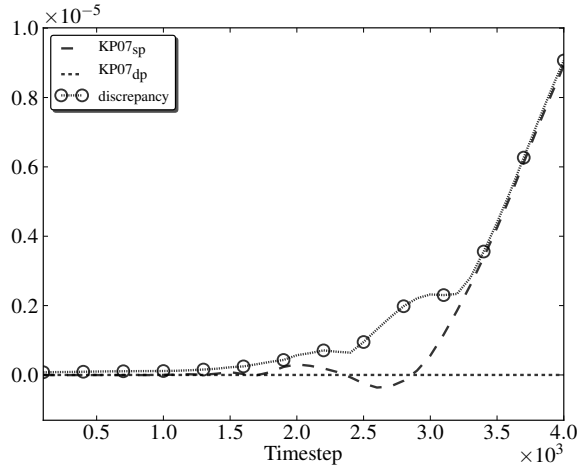
5.1 Float vs. Double Precision

Double-precision arithmetics has so far not been supported very well on GPUs, and when available, has come with a big performance penalty. Hence, it is advantageous if the high-resolution schemes can rely solely on single-precisions arithmetics. We therefore start by investigating how using single-precision influences the accuracy of our schemes. To this end, we consider the relative errors in mass conservation,

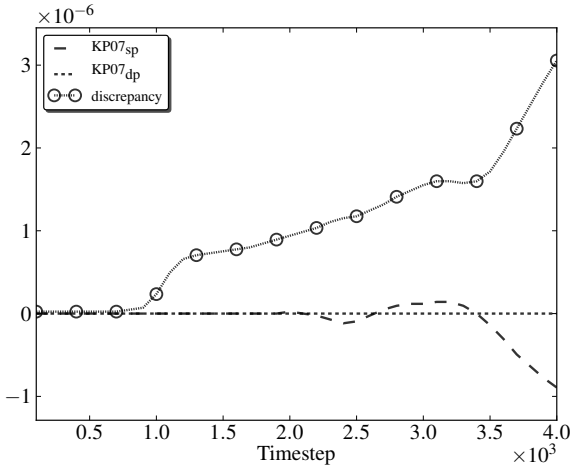
$$E^c = \frac{\int_\Omega h^0 \, dx - \int_\Omega h^n \, dx}{\int_\Omega h^0 \, dx}, \tag{8}$$

where h^0 is the initial water depth and h^n is the water depth at timestep n (using a fixed timestep). Figure 9 reports this error for single-precision (SP) and double-precision (DP) versions of the Kurganov–Petrova scheme (KP07). Likewise, we report the absolute discrepancy between the two solutions for each timestep. It is interesting to note that single precision gives round-off errors that violate conservation of water, both for low and high water elevations, even for flat bottom topographies. The absolute discrepancy between the two solutions is also growing for Cases 1a, 1b, and 2. The two Kurganov–Levy schemes exhibit the exact same behavior and plots are not included.

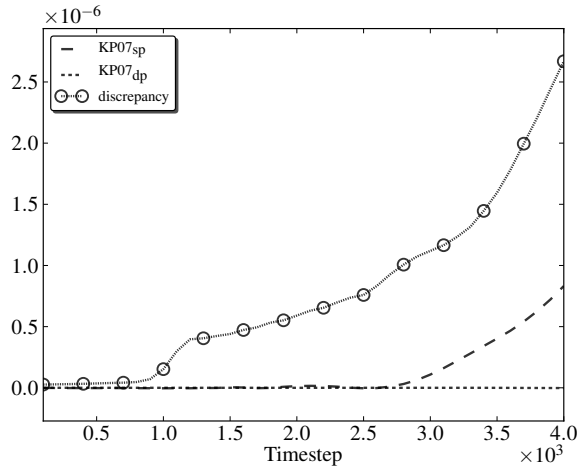
When dry states and varying bathymetry is included (Case 3), we see that error increases significantly and that the overall error of the scheme dominates the effects of sin-



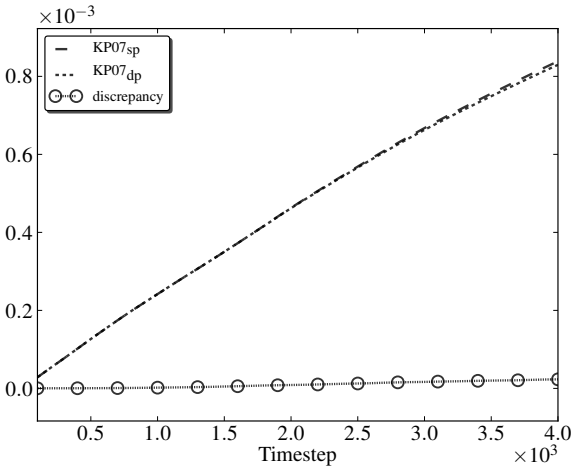
(a) Case 1a: low water elevation.



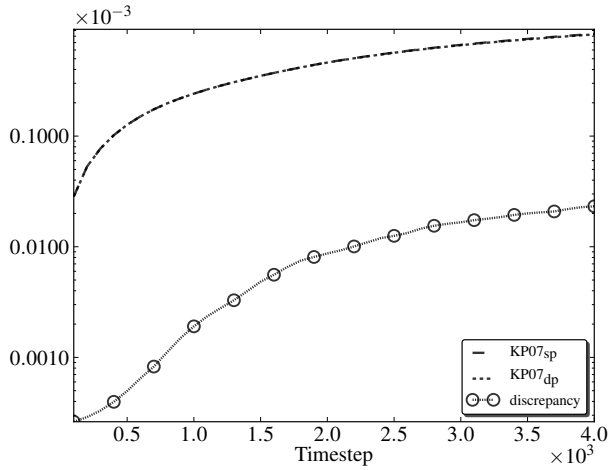
(b) Case 1b: high water elevation.



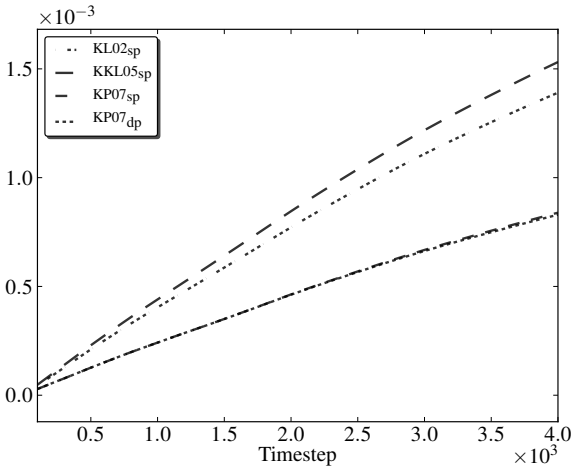
(c) Case 2: discontinuous bathymetry.



(d) Case 3: dambreak simulation.



(e) Case 3: dambreak, semilog scale.



(f) Case 3: dambreak simulation.

Fig. 9: Comparison of single-precision and double-precision versions of the KP07 scheme on a 1024×1024 grid. In (a) to (e), relative errors in mass conservation (E^c in (8)) are shown together with the discrepancy between single and double-precision simulations. Subplot (f) shows the error for all three schemes.

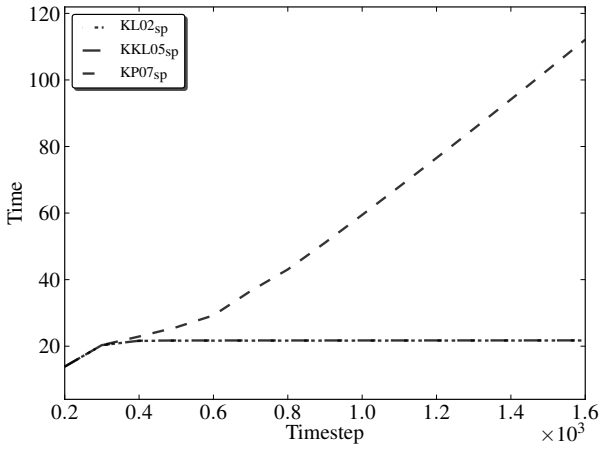


Fig. 10: Plot of time versus number of timesteps for Case 2.

gle versus double precision. The increase in error comes from the switching in the Kurganov–Levy schemes and the slope fix in the Kurganov–Petrova scheme. Hence, we conclude that when the schemes are used for the type of problems they were designed for with shoal zones and dry states, the error induced by floating-point precision is negligible (as originally stated in [7], based on CPU simulations in single and double precision). It is interesting to see that the added integration points in the modified Kurganov–Levy seems to negatively affect the conservation. Our explanation to this is that the modified scheme performs twice the number of flux evaluations, and should thus experiences more round-off errors.

We have further verified our wall boundary conditions by checking their effect on conservation. The boundary conditions did not affect the conservation for the wet-bed test cases.

5.2 Discontinuous Bathymetry

The Kurganov–Levy schemes assume a continuous bathymetry, and a straightforward sampling of a discontinuous bathymetry, as in Case 2, will result in very steep gradients in the bathymetry approximation, which in turn will effect the CFL number and drive the stable timesteps toward zero. This effect is illustrated in Figure 10: when the wave reaches the discontinuity in the bathymetry, the timestep in the Kurganov–Levy schemes decays to zero and even after 5000 timesteps, the simulation is still at time $t \approx 23$. The Kurganov–Petrova scheme, on the other hand, propagates the wave with nearly unaffected timesteps past the discontinuity.

5.3 Efficiency

Ever since the first applications on GPUs were published, there has been a trend to report speedups over the CPU. At the time when [7,8,6] was written, general-purpose computation on graphics processing units (GPGPU) was still in its infancy, and an important statement was to demonstrate that it was possible to use GPUs for general-purpose computation, and to convince the reader that a GPU code could be much faster than a corresponding CPU code. Gigaflops and execution-time metrics have been used extensively, and speedup factors between 2 to 200 are commonly found in articles, still today. However, by examining the theoretical performance numbers for GPUs and CPUs, one quickly realizes that a speedup of over 100 seems unlikely on current hardware. Typically, these figures emerge from comparing an unoptimized (or even worse, claimed to be optimized) CPU code to a highly-tuned GPU implementation. There are cases, e.g., for algorithms dominated by expensive trigonometric computations, where the use of highly efficient, albeit less accurate, hardware implementations found on the GPU can give a speedup larger than what one would expect by only comparing memory speed, clock frequencies, and number of arithmetic units. For most algorithms, however, these high speedups are not attainable. Our view is that quoting such high speedups has had its mission and is now destructive to the reputation of GPU computing.

We would like to see less of these speedup figures and more figures that show efficient hardware utilization and scalability. Reporting utilization of hardware resources will thus give the user an idea of what to expect, not only on current hardware, but also for future hardware generations. We have measured the efficiency of our implementations on the NVIDIA GeForce GTX 285 using the CUDA Visual Profiler supplied with the CUDA SDK. We have used the profiler to give us detailed statistics over runtime, memory bandwidth utilization, and instruction throughput. The profiler has also been actively used during development, to find and remove bottlenecks.

The flux and source term kernel is the most time-consuming of our kernels. For the KP07 scheme, this kernel takes up 72% of the GPU runtime. It utilizes only 17.6% of peak bandwidth for global store because we violate coalescing rules. For global load, our original version had an efficiency of 11%. However, when using texture fetches, we saw a large performance boost. The kernel has an instruction throughput of 80%, which means we idle 20% of the time. We have profiled this kernel also on other GPUs with less bandwidth relative to compute power, and the instruction throughput has remained at 80%. Our experiments and benchmarks thus indicate that the idling is caused by data dependencies and instruction latencies, and not due to waiting on data from global memory. Thus, we conclude that the

flux and source term kernel is compute bound for the benchmarked GPUs.

For the KL02 and KLL05 schemes, our flux kernel is also the most time consuming, with 82% and 84% of the total runtime, respectively. These two kernels have a memory efficiency equivalent to the KP07 kernel, as they perform the same memory operations. However, the instruction throughput is at a mere 66%. This can be explained by the number of threads we are able to schedule to each multiprocessor. Because we are limited by shared-memory usage, we have fewer threads for these two kernels, meaning we do not have as many other threads to run whilst we wait for data. This is often referred to as the occupancy, and our occupancy is 22% for the KP07 scheme, yet only 12.5% for the KL02 and KLL05 schemes.

The Runge–Kutta substep kernel is our second most time-consuming kernel, with 28%, 18%, and 16% of the GPU runtime for KP07, KL02 and KLL05, respectively. This kernel is heavily memory bound, and penalized for violating the coalescing rules. We achieve 15% global store efficiency, and 11% global load efficiency. However, as with the flux and source term kernel, we use textures for most reads, which again showed to give a substantial performance gain. This kernel has also an instruction throughput of 80%. However, when profiling on GPUs with less relative bandwidth, we see that the instruction throughput is proportional to the bandwidth. Thus, we conclude that the Runge–Kutta kernel is memory bound.

The rest of the GPU time is spent, in decreasing order, on copying data to the GPU, running the maximum Δt kernel, and downloading data to the CPU. As can be seen from the previously presented numbers, these operations are negligible for the GPU runtime. However, they do impose a software overhead. The upload of data is done for each kernel, as we need to upload the parameters to constant memory before each kernel invocation. It should be possible to make this a GPU–GPU copy instead of a CPU–GPU copy, but we have not pursued this option because of the little time it takes. The maximum Δt kernel has a memory efficiency of 20% for store, and 40% for load. The kernel obeys all coalescing rules, but is penalized because of the very few items it considers. Finally, the download to the CPU is to keep track of the global simulation time, as the maximum Δt kernel places the result in GPU memory.

5.4 Performance and Scalability.

Figure 11 shows the performance of our schemes for Case 3 on the NVIDIA GeForce GTX 285. The most important grid sizes are those larger than or equal to 512^2, as this is where we best utilize the hardware. For this grid size, the KP07 implementation is able to run at 387 iterations per second in floating-point precision, and 45 in double precision, i.e., a factor 8.6 slower. The cause for this massive speed-down is that the benchmark GPU only has one double-precision unit per streaming multiprocessor, but eight single-precision units. Further, the double-precision implementation also imposes other overheads, such as more register space and a need for handling texture fetches in a special way. Our double-precision numbers for other grid sizes are similar.

When we increase the workload by four we would expect the number of iterations per second to decrease by four as well. However, going from 512^2 to 1024^2, we get more than one fourth of the performance for single precision. This is partly caused by the need to pad our domain to fit an integer number of blocks, which has a larger impact on the smaller domain sizes. Going from 1024^2 to 2048^2, this penalty becomes negligible, and we attain almost perfect weak scaling, also reflected in the double-precision results. For larger sizes, however, we quickly run out of graphics memory, as our benchmark machine is limited to 1 GB. The maximum simulation size we have been able to run is a 3900×3900 grid, which consumes around 928 MB of graphics memory.

The main reason that the KP07 scheme is faster than KL02 and KLL05 is that the latter needs to reconstruct both from the positivity-preserving *and* the conserved variables, as we do not know, a priori, whether a cell is in the shoal zone or not. This double reconstruction increases the amount of computations dramatically. Furthermore, the KP07 scheme consumes far less shared memory in the flux and source-term kernel. The shared-memory use dictates the maximum block size, and for larger block sizes, the relative size of the overlapping ghost-cell regions goes down, lessening both the number of computations and the burden on the memory subsystem.

Finally, it is interesting to note that there seems to be no performance impact for the added integration points for the KLL05 scheme, which enables the use of higher-order reconstructions up to degree five. Our explanation to this is that the Kurganov–Levy based kernels are memory bound so that we can add more computations without severely affecting performance.

5.5 Visualization

Our simulator can run both with and without visualization. When visualization is enabled, we visualize every fifteenth timestep. This, unfortunately, has a negative impact on the simulator performance, as both the simulator and visualizer use the same hardware resources. To make matters worse, there is an overhead connected with mapping and unmapping OpenGL memory for use with CUDA, in addition to overheads related to the context switch between CUDA and OpenGL. For the 512^2 grid of Case 3, we achieve 300 timesteps per second using the KP07 scheme, which is a 22%

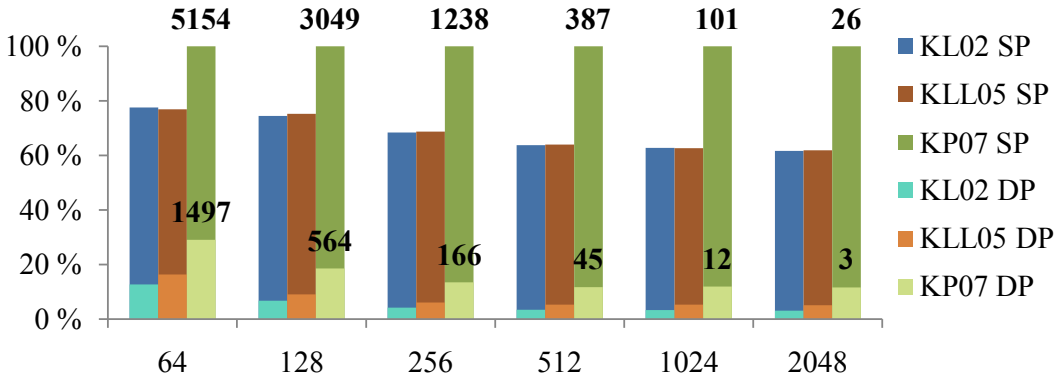


Fig. 11: Performance of the different schemes for Case 3 on grids with $n \times n$ cells running on the NVIDIA GeForce GTX 285. From left to right, the three columns for each domain size display the performance of KL02, KLL05, and KP07, respectively. The graph is normalized relative to single-precision KP07. The figures in bold over each column indicates the number of timesteps for our KP07 implementation in single precision. The superimposed columns are for the same schemes in double precision, and the figures in bold indicate the number of timesteps for the KP07 implementation.

drop from the 389 timesteps per second we get without visualization. Nevertheless, this translates directly to an interactive 20 frames per second. Several examples of the visualization can be found on YouTube: `http://www.youtube.com/user/babrodtk`.

6 Summary

In this paper, we have presented how shallow-water waves, as described by the Saint-Venant system, can be computed efficiently on graphical processing units using three different well-balanced, high-resolution schemes. By implementing direct visualization on the GPU, including various photo-realistic effects, we have developed a visual and interactive simulator.

Current GPU hardware is much more efficient when using single rather than double-precision arithmetics. For simple computational setups with no transitions between wet and shoal zones, round-off errors introduced by single-precision arithmetics cause lack of mass conservation and a significant deviation from the corresponding double-precision solution. However, for more complex cases that contain transitions between wet and shoal zones and/or between shoal and dry zones, the effect of single-precision arithmetics is masked by errors inherent in the schemes' treatment of dry zones. Hence, single-precision arithmetics can mostly likely be used for the typical complex cases the schemes were developed to handle. Preliminary experiments also indicate that use of mixed-precision arithmetics can be a way out to preserve both high accuracy *and* efficiency for single-zone cases.

Of the three schemes considered, the Kurganov–Petrova (KP07) scheme is our method of choice. This scheme has the best resource utilization of current GPU architectures and is hence more efficient, has a better treatment of dry states, and can handle discontinuities in the bathymetry. On the other hand, the treatment of dry states in KP07 only applies to bilinear reconstructions, and hence the scheme cannot be extended to higher spatial order, which may be important when studying smooth effects like eddies and other smooth phenomena.

Our implementations show relatively high utilization of computational resources and memory transfer. Still, there is room for further improvement. Increased memory throughput can for example be achieved by using Morton order for texture fetches. We also anticipate that increases in shared-memory size and a new cache, as in the new Fermi architecture from NVIDIA, will give a significant performance boost. Likewise, the performance of our visualization is likely to benefit from new functionality in CUDA 3.0 Beta for more efficient sharing of data between CUDA and OpenGL.

Our initial interest in simulating shallow-water waves on GPUs was to use high-resolution schemes as an excellent demonstrator of GPU capabilities and to provide a use case of interactive visualization. Lately, however, our interest has moved more towards full-featured shallow-water simulation: realistic dambreak scenarios, storm surges, etc. Verification, validation, and further algorithmic and implementational improvements are described in [5]. Moreover, we have developed a multi-GPU cluster implementation that shows (nearly) perfect scaling; further details will be provided in an upcoming paper.

Acknowledgements. The authors gratefully acknowledge financial support from the Research Council of Norway under grants number 180023/S10 and 186947/I30 and the Center

of Mathematics for Applications, University of Oslo. The authors also thank NVIDA for their continued support.

References

1. de la Asunción, M., Mantas, J.M., Castro, M.J.: Simulation of one-layer shallow water systems on multicore and CUDA architectures. J. Supercomput. (2010)
2. Brandvik, T., Pullan, G.: Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. IMechE Proc C: J. Mech. Engng. Sci. (2007). DOI 10.1243/09544062JMES813FT
3. Brandvik, T., Pullan, G.: Acceleration of a 3D Euler solver using commodity graphics hardware. In: 46th AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2008-607 (2008)
4. Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., Storaasli, O.: State-of-the-art in heterogeneous computing. Scientific Programming **18**(1) (2010)
5. Brodtkorb, A.R., Sætra, M.L., Altinakar, M.: Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. In preparation (2010)
6. Hagen, T., Henriksen, M., Hjelmervik, J., Lie, K.A.: 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.) Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF, pp. 211–264. Springer Verlag (2007)
7. Hagen, T., Hjelmervik, J., Lie, K.A., Natvig, J., Henriksen, M.: Visual simulation of shallow-water waves. Simul. Model. Pract. Theory **13**(8), 716–726 (2005)
8. Hagen, T.R., Lie, K.A., Natvig, J.R.: Solving the Euler equations on graphics processing units. In: Proceedings of the 6th International Conference on Computational Science—ICCS 2006, *Lect. Notes Comp. Sci.*, vol. 3994, pp. 220–227. Springer Verlag, Berlin/Heidelberg (2006)
9. Harten, A.: High resolution schemes for hyperbolic conservation laws. Journal of Computational Physics **49**(3), 357–393 (1983)
10. Klöckner, A., Warburton, T., Bridge, J., Hesthaven, J.: Nodal discontinuous Galerkin methods on graphics processors. J. Comp. Phys. **228**(21), 7863–7882 (2009). DOI 10.1016/j.jcp.2009.06.041
11. Kurganov, A., Levy, D.: Central-upwind schemes for the Saint-Venant system. Mathematical Modelling and Numerical Analysis **36**, 397–425 (2002)
12. Kurganov, A., Noelle, S., Petrova, G.: Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations. SIAM J. Sci. Comput. **23**(3), 707–740 (electronic) (2001)
13. Kurganov, A., Petrova, G.: A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. Communications in Mathematical Sciences **5**, 133–160 (2007)
14. Larsen, E., McAllister, D.: Fast matrix multiplies using graphics hardware. In: Supercomputing, pp. 55–55. ACM, New York, NY, USA (2001). DOI http://doi.acm.org/10.1145/582034.582089
15. Lastra, M., Mantas, J.M., na, C.U., Castro, M.J., García-Rodríguez, J.A.: Simulation of shallow-water systems using graphics processing units. Math. Comput. Simulat. **80**(3), 598 – 618 (2009). DOI 10.1016/j.matcom.2009.09.012
16. Liang, W.Y., Hsieh, T.J., Satria, M., Chang, Y.L., Fang, J.P., Chen, C.C., Han, C.C.: A GPU-based simulation of tsunami propagation and inundation. In: Algorithms and Architectures for Parallel Processing, *Lect. Notes Comp. Sci.*, vol. 5574, pp. 593–603. Springer Verlag, Berlin/Heidelberg (2009). DOI 10.1007/978-3-642-03095-6_56
17. Natvig, J.R., Sebastian, N., Pankratz, N., Puppo, G.: Well-balanced finite volume schemes of arbitrary order of accuracy for shallow water flows. J. Comput. Phys. **13**(2), 474–499 (2006)
18. NVIDIA: NVIDIA CUDA reference manual 2.3 (2009)
19. OpenGL ARB, Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, 6th edition edn. Addison-Wesley (2007)
20. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU computing. Proceedings of the IEEE **96**(5), 879–899 (2008). DOI 10.1109/JPROC.2008.917757
21. Pankratz, N., Natvig, J.R., Gjevik, B., Noelle, S.: High-order well-balanced finite-volume schemes for barotropic flows. development and numerical comparisons. Ocean Modelling **18**(1), 53–79 (2007)
22. Phillips, E.H., Zhang, Y., Davis, R.L., Owens, J.D.: Rapid aerodynamic performance prediction on a cluster of graphics processing units. In: Proceedings of the 47th AIAA Aerospace Sciences Meeting, AIAA 2009-565 (2009)
23. Shu, C.W.: Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. In: Advanced numerical approximation of nonlinear hyperbolic equations (Cetraro, 1997), *Lecture Notes in Math.*, vol. 1697, pp. 325–432. Springer, Berlin (1998)
24. Singh, J., Altinakar, M., Ding, Y.: 2D numerical model for shallow transient free surface flows over natural terrain. In: Proceedings of the International Conference on Hydroscience and Engineering. **Accepted for publication** (2010)
25. Sweby, P.K.: High resolution schemes using flux limiters for hyperbolic conservation laws. Siam Journal of Numerical Analysis **21**(5), 995–1011 (1984)
26. Wang, P., Abel, T., Kaehler, R.: Adaptive mesh fluid simulations on GPU. New Astron. **In press**, – (2009). DOI 10.1016/j.newast.2009.10.002