Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–17

DOI: 10.1177/1094342016630800
hpc.sagepub.com
SAGE

Kepler shuffle for real-world flood simulations on GPUs

Zsolt Horváth[1,2], Rui AP Perdigão[1], Jürgen Waser[2], Daniel Cornel[2], Artem Konev[2] and Günter Blöschl[1]

Abstract

We present a new graphics processing unit implementation of two second-order numerical schemes of the shallow water equations on Cartesian grids. Previous implementations are not fast enough to evaluate multiple scenarios for a robust, uncertainty-aware decision support. To tackle this, we exploit the capabilities of the NVIDIA Kepler architecture. We implement a scheme developed by Kurganov and Petrova (KP07) and a newer, improved version by Horváth et al. (HWP14). The KP07 scheme is simpler but suffers from incorrect high velocities along the wet/dry boundaries, resulting in small time steps and long simulation runtimes. The HWP14 scheme resolves this problem but comprises a more complex algorithm. Previous work has shown that HWP14 has the potential to outperform KP07, but no practical implementation has been provided. The novel shuffle-based implementation of HWP14 presented here takes advantage of its accuracy and performance capabilities for real-world usage. The correctness and performance are validated on real-world scenarios.

Keywords

Shallow water equations, high-performance computing, GPU, CUDA, CFD

1. Introduction

Flood risk assessment is of key importance in minimizing damages and economical losses caused by flood events. These are challenging to predict due to nonlinear interactions between driving processes (Perdigão and Blöschl, 2014). The first step in flood risk studies is the identification of flood prone areas (EU, 2007). This requires the implementation of hydrodynamic models that enable one to quantify the evolution of a flood and its hydraulic representative variables, for example, water level and velocity.

The shallow water equations (SWEs) are capable of modeling many flood phenomena such as levee breaches, flood plain inundations, tsunamis, dam breaks, or river flows in both rural and urban areas. Our primary interest is in decision-making systems, where we evaluate many scenarios and select the solution with the best outcome (Waser et al., 2014). Modern approaches allow the user to evaluate the uncertainties of the parameters considered in the simulation, requiring the simulation of many different scenarios, which is usually computationally expensive. Therefore, simulation runs have to be as fast as possible to reduce the overall time of finding the best solution.

The SWEs are a set of hyperbolic partial differential equations, described by the Saint-Venant system, where the fluid motion is introduced by the gravitational acceleration. They are derived from depth, integrating the Navier–Stokes equations and represent wave propagation with a horizontal length scale much greater than the vertical length scale.

In this article, we focus on schemes that are defined on Cartesian grids and present a new implementation of two shallow water schemes which are both able to handle "dry lake" and "lake at rest" steady-state solutions. We discuss how to exploit the capabilities of modern graphics processing units (GPUs) using the CUDA programming model. We focus on the NVIDIA Kepler architecture and its newest features to achieve peak performance. We build upon the schemes of Kurganov and Petrova (2007) (KP07) and

[1]Vienna University of Technology, Karlsplatz, Austria
[2]VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH, Vienna, Austria

Corresponding author:
Zsolt Horváth, Vienna University of Technology, A-1040 Vienna, Karlsplatz 13/222, Austria.
Email: zhorvath@vrvis.at

Horváth et al. (2014) (HWP14), which improve the former by avoiding spurious high velocities at the dry/wet boundaries. The HWP14 scheme is more complex, and hence it requires more GPU resources. Nevertheless, previous work has shown that the HWP14 scheme has the potential to outperform KP07, since it can use longer time step sizes (Horváth et al., 2014). Until now, no implementation has been given to exploit this potential in practice. To accomplish this, we present a new implementation based on recent advancements in the GPU technology. In summary, this article contributes the following:

- a new implementation of the HWP14 scheme that outperforms the less accurate KP07 scheme,
- exploitation of the Kepler shuffle instructions that allows for an increased GPU occupancy and speedup by requiring less shared memory and less explicit block synchronizations, and
- extensive performance benchmarks of the implemented schemes and validation against real-world scenarios.

2. Related work

In the numerical treatment of the SWE, the spatial domain is discretized. For this purpose, one can use an unstructured mesh or a structured mesh (Cartesian). Unstructured meshes have demonstrated good properties for the simulation of rainfall/runoff events (Lacasta et al., 2015). Since cells do not establish preferential directions, this type reduces the mesh influence on the numerical results. Also, it is easy to adapt unstructured cells to capture local structures. The disadvantage of using unstructured meshes on the GPU is related to the mesh disorder, which makes it expensive to load and share data between threads (Lacasta et al., 2014). Structured meshes have proved suitable for GPU computations (Brodtkorb et al., 2012). Indeed, the GPUs have been optimized for memory access when Cartesian grids are used. The main advantage of structured meshes is the inherent order existent in their creation. On Cartesian grids, every cell knows its neighbors implicitly, no information on the topography is needed to maintain data dependencies. To overcome the limitation regarding capturing local structures, adaptive mesh refinement (AMR) (Sætra et al., 2014) or the cut cell approach (Causon et al., 2001) can be used.

Explicit numerical schemes are well suited for parallel execution on the GPU to solve hyperbolic partial differential equations (Brandvik and Pullan, 2008; Hagen et al., 2006; Klöckner et al., 2009; Wang et al., 2010). Hagen et al. (2005) were among the first to deliver a GPU solver for the SWEs based on the first-order Lax–Friedrichs and the second-order central-upwind schemes. They achieve a 15–30 times speedup compared to an equivalently tuned CPU version. Lastra et al. (2009) implement a first-order well-balanced finite volume solver for one-layer SWE using OpenGL and Cg languages. Acuña and Aoki (2009) propose a multi-GPU solution for tsunami simulations. They use overlapping kernels to compute the solution for x and y directions concurrently. Liang et al. (2009) add a friction slope to the conservation of momentum to simulate tsunami inundation using the MacCormack method. De la Asunción et al. (2010, 2011) demonstrate solvers for 1-D and 2-D SWE using the CUDA Toolkit. They show that an optimized CUDA solver is faster than a GPU version which is based on a graphics-specific language. Brodtkorb et al. (2010) implement three second-order schemes on the GPU. They discuss how single- and double-precision arithmetics affect accuracy, efficiency, scalability, and resource utilization. They show that double precision is not necessary for the second-order SWE (Brodtkorb et al., 2010). Their implementation demonstrates that all three schemes map very well to the GPU hardware. Simulating real-world dam-break scenarios, de la Asunción et al. (2013) and Brodtkorb and Sætra (2012) report relevant computational speedups in comparison with sequential codes. Moreover, Sætra and Brodtkorb use a multiple GPU solver to tackle large domain simulations and reduce both the memory footprint and the computational burden using sparse computation and sparse memory approaches. Sætra et al. (2014) are able to increase the grid resolution locally for capturing complicated structures or steep gradients in the solution. They employ AMR which recursively refines the grid in parts of the domain. Vacondio et al. (2014) implement an implicit local ghost cell approach that enables the discretization of a broad spectrum of boundary conditions. They also present an efficient block deactivation procedure in order to increase the efficiency of the numerical scheme in the presence of wetting–drying fronts.

3. Numerical schemes

In this section, we summarize the underlying numerical theory of the implemented schemes. The hyperbolic conservation law described by the 2-D SWEs of the Saint-Venant system can be written as:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}\Big/C^2 \\ -gv\sqrt{u^2+v^2}\Big/C^2 \end{bmatrix}, \quad (1)$$

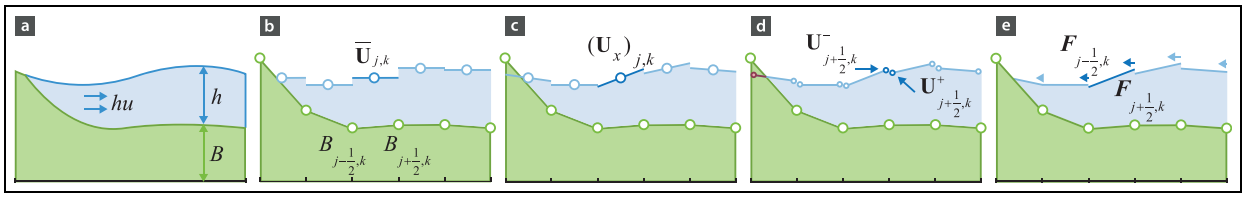

Figure 1. Schematic view of a shallow water flow, definition of the variables, and flux computation. (a) Continuous variables. (b) The conserved variables $\mathbf{U}$ are discretized as cell averages $\bar{\mathbf{U}}_{j,k}$. The bathymetry function B is approximated at cell interface midpoints. (c) Slopes of the water surface $(\mathbf{U}_x)_{j,k}$ are reconstructed using the minmod flux limiter. (d) Left- and right-sided point values are computed at cell interface midpoints. The red circle indicates that a negative water height is computed. Since water heights cannot be negative, they are corrected before the flux computation. (e) Fluxes are computed using the central-upwind flux function at the cell center interfaces.

where h represents the water height, hu is the discharge along the x axis, hv is the discharge along the y axis (Figure 1(a)), u and v are the average flow velocities, g is the gravitational constant, B is the bathymetry, and C is the Chézy friction coefficient. We use the relation $C = (1/n) * h^{(1/6)}$, where n is Manning's roughness coefficient. Subscripts represent partial derivatives, that is, $\mathbf{U}_t$ stands for $\partial\mathbf{U}/\partial t$.

In vector form, the system can be written as:

$$\mathbf{U}_t + \mathbf{F}(\mathbf{U}, B)_x + \mathbf{G}(\mathbf{U}, B)_y = \mathbf{S}_B(\mathbf{U}, B) + \mathbf{S}_f(\mathbf{U}), \quad (2)$$

where $\mathbf{U} = [h, hu, hv]$ is the vector of conserved variables, $\mathbf{F}$ and $\mathbf{G}$ are flux functions, $\mathbf{S}_B$ and $\mathbf{S}_f$ represent the bed slope and bed friction source terms, respectively.

Herein, we focus on two numerical schemes, the first one developed by Kurganov and Petrova (2007) and its improved version developed by Horváth et al. (2014). A good numerical scheme should be able to exactly preserve both lake at rest and dry lake steady states and their combinations. The methods that exactly preserve these solutions are termed "well-balanced" (Bollermann et al., 2013; Duran et al., 2013; Hou et al., 2013; Li et al., 2013; Noelle et al., 2006; Russo, 2005; Xing and Shu, 2005).

3.1 Kurganov–Petrova scheme (KP07)

The KP07 scheme is based on a 2-D central-upwind second-order numerical scheme developed by Kurganov and Petrova (2007). It allows for a discontinuous bathymetry and is more suitable for GPU implementation than its predecessor. (Kurganov and Levy, 2002). To avoid negative water heights h and preserve well-balancedness, the KP07 scheme changes from the variables $[h, hu, hv]$ to $[w, hu, hv]$, where $w = h + B$ represents the water surface (Figure 1(a)). Kurganov and Levy use a non-oscillatory conservative piecewise linear reconstruction of w, which is properly corrected near dry areas, without switching to a reconstruction of h there. The correction relies on the fact that the original bottom function B is replaced with its continuous piecewise linear approximation. However, near dry areas and at the dry–wet boundaries, the scheme can still create large errors in the flux calculations, since the water height can become very small or even 0. Due to $u = hu/h$ and $v = hv/h$, these computations may lead to large errors in the partially flooded cells for small water heights and they have a singularity at zero water height $(h = 0)$. To deal with this problem, Kurganov and Levy use the following desingularization:

$$u = \frac{\sqrt{2}h(hu)}{\sqrt{h^4 + \max(h^4, \epsilon)}}; \quad v = \frac{\sqrt{2}h(hv)}{\sqrt{h^4 + \max(h^4, \epsilon)}}, \quad (3)$$

where ε is a small apriori chosen positive number. This has a dampening effect on the velocities as the water height approaches 0. Determining a proper value for ε is a difficult task. High values lead to large errors in the simulation results, while low values give small time steps. There is one more problem related to the partially flooded cells. A correction to the reconstruction has to be applied. This correction solves the positivity problem and guarantees that all water heights are non-negative. However, at the partially flooded cells, it can lead to large errors for small water heights and the flow velocity will grow smoothly in these formerly dry areas, since the correction is not well-balanced there. High velocities in the domain cause small time steps, thus poor performance of the scheme and slower simulation. Another issue related to this correction is that the water climbs up on the shores at the dry/wet boundaries. Finally, if a cell becomes wet, it will almost never be completely dry again.

3.2 Horváth–Waser–Perdigão scheme (HWP14)

The HWP14 scheme is an improved version of the KP07 scheme. It introduces a new reconstruction and the draining time step technique (DTST) to restore the well-balancedness for the partially flooded cells. This scheme is well-balanced, positivity preserving, and

handles dry states. The latter is ensured using the DTST in the time integration process, which guarantees nonnegative water depths. Unlike the KP07 scheme, the new technique does not generate high velocities at the dry–wet boundaries, which are responsible for small time step sizes and slow simulation runs. The new scheme preserves lake at rest steady states and guarantees the positivity of the computed fluid depth in the partially flooded cells. Due to the new reconstruction procedure, this scheme has some additional computations compared to the KP07, and thus it is computationally more expensive. It detects partially flooded cells and computes a new waterline for them. This information is used to reconstruct proper point values at the cell interfaces and to ensure well-balancedness. It reduces the nonphysical high velocities at the dry–wet boundaries and allows for longer time steps that result in shorter simulation runtimes.

3.3 Spatial discretization

The same spatial discretization is applied to both schemes on a uniform grid (Figure 2), where the conserved variables $\mathbf{U}$ are defined as cell averages (Figure 1(b)). The bathymetry is given as a piecewise bilinear surface defined by the values at the cell vertices. The fluxes are computed at the integration points, that is, at the midpoints of the cell interfaces. The central-upwind semi-discretization of equation (1) can be written down as the following system of time-dependent, ordinary differential equations (ODEs):

$$\begin{aligned}\frac{d}{dt}\bar{\mathbf{U}}_{j,k} = &-\left[\mathbf{F}\left(\mathbf{U}_{j+\frac{1}{2},k}\right)-\mathbf{F}\left(\mathbf{U}_{j-\frac{1}{2},k}\right)\right]\\ &-\left[\mathbf{G}\left(\mathbf{U}_{j,k+\frac{1}{2}}\right)-\mathbf{G}\left(\mathbf{U}_{j,k-\frac{1}{2}}\right)\right]\\ &+\mathbf{S}_B\left(\bar{\mathbf{U}}_{j,k},B_{j,k}\right)+\mathbf{S}_f\left(\bar{\mathbf{U}}_{j,k}\right),\\ &=\mathbf{R}_{F+G}(\bar{\mathbf{U}})_{j,k}+\mathbf{S}_{B+f}\left(\bar{\mathbf{U}}_{j,k},B_{j,k}\right)\end{aligned} \tag{4}$$

where $\mathbf{U}_{j\pm1/2,k}$ and $\mathbf{U}_{j,k\pm1/2}$ are the reconstructed point values at the cell interface midpoints.

In order to compute the fluxes $\mathbf{F}$ and $\mathbf{G}$ across the cell interfaces, we start with reconstructing the water surface. A planar surface is computed for each cell using the cell averages $\bar{\mathbf{U}}$ and a piecewise bilinear approximation, which determines the slope of the water in the cells (Figure 1(c)). This slope reconstruction is performed using the generalized minmod flux limiter:

$$(\mathbf{U}_d)_{j,k}=\text{minmod}(\theta b,c,\theta f), \tag{5}$$

where $(\mathbf{U}_d)_{j,k}$ are the derivatives of the conserved variables in x or y direction, b, c, and f are the backward, central, and forward difference approximations of the derivative, respectively, that is, the slope of the water surface within cell $C_{j,k}$. The parameter $\theta\in[1,2]$ is used to control the numerical viscosity of the scheme. Here we use $\theta=1.3$ as suggested by Kurganov and Levy (2002). The index symbol d indicates the direction of the derivation, which can be x or y in our case. The minmod flux limiter is defined as:

$$\text{minmod}(z_1,z_2,z_3)\begin{cases}\min_j\{z_j\}, & \text{if } z_j>0 \quad \forall j,\\ \max_j\{z_j\}, & \text{if } z_j<0 \quad \forall j,\\ 0, & \text{otherwise}\end{cases} \tag{6}$$

and is applied in a componentwise manner to all three elements $[\bar{w},\bar{hu},\bar{hv}]$ of the vector $\bar{\mathbf{U}}$.

In general, the slope reconstruction produces negative water heights h at the integration points in the partially flooded cells. If the water height becomes negative, the computation breaks down since the eigenvalues of the system are $u\pm\sqrt{gh}$ and $v\pm\sqrt{gh}$. Hence, these values have to be corrected while maintaining mass conservation. The KP07 correction solves the positivity problem by raising and lowering the water level at the left and right side of the cell according to the bathymetry function. This guarantees that all water heights are nonnegative. However, this violates the well-balancedness of the scheme at the dry/wet boundary causing high velocities to appear. The HWP14 scheme uses a more complex correction, which preserves the well-balancedness and reduces the spurious velocities. The slope reconstruction is followed by the point value computation. For every cell interface at the integration points, point values are reconstructed

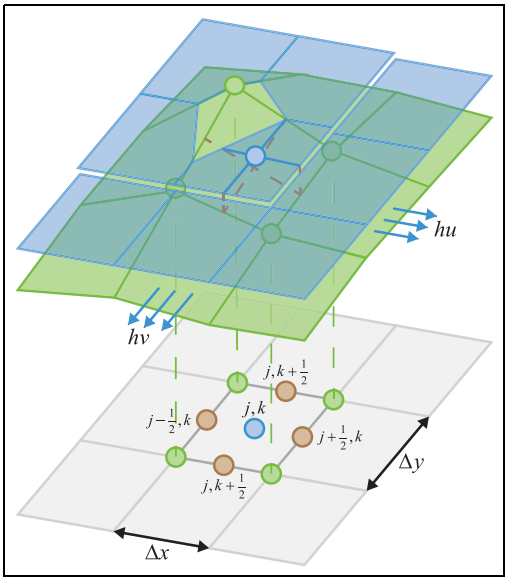

Figure 2. Two-dimensional grid-based representation of the discretized variables of the shallow water equations. Cell averages $\bar{\mathbf{U}}_{j,k}$ are defined at cell centers (blue dots). Green dots indicate the sampling points of the bathymetry function B. Brown dots indicate the approximated values of the bathymetry function at the cell interface midpoints (Horváth et al., 2014).

(Figure 1(d)). Each cell interface has two point values, one from the cell on the left and one from the right. Based on these point values, we compute the fluxes using the central-upwind flux function (Figure 1(e)). In order to do so, we need to calculate the local speed values $u = hu/h$ and $v = hv/h$ at the integration points. This leads to large errors as the water height h approaches 0 and can produce high velocities and instabilities. Since the time step size is calculated using the maximum velocity component in equation (8), it is also affected by this problem. To reduce the effect of these high velocities, we have to apply the desingularization as in equation (3). Finally, we compute the bed slope source term S_B and the friction source term S_f.

3.4 Temporal discretization

We have discussed the spatial reconstruction of the point values and the flux computation. In the following, we continue with the time-quadrature for the fluxes applied to both schemes. We discretize the semi-discrete scheme (equation (1)) in time and advance by Δt using a standard second-order accurate total variation diminishing Runge-Kutta scheme (Gottlieb and Shu, 1988; Shu, 1988):

$$\begin{aligned}\bar{\mathbf{U}}^{*}_{j,k} &= \bar{\mathbf{U}}^{n}_{j,k} + \Delta t\left(\mathbf{R}_{F+G}\left(\bar{\mathbf{U}}^{n}\right)_{j,k} + \mathbf{S}_{B+f}\left(\bar{\mathbf{U}}^{n}_{j,k}, B_{j,k}\right)\right),\\ \bar{\mathbf{U}}^{n+1}_{j,k} &= \frac{1}{2}\bar{\mathbf{U}}^{n}_{j,k}\\ &+ \frac{1}{2}\left[\bar{\mathbf{U}}^{*}_{j,k} + \Delta t\left(\mathbf{R}_{F+G}\left(\bar{\mathbf{U}}^{*}\right)_{j,k} + \mathbf{S}_{B+f}\left(\bar{\mathbf{U}}^{*}_{j,k}, B_{j,k}\right)\right)\right],\end{aligned} \tag{7}$$

where Δt is the time step. In order to keep the numerical integration stable, the time step size is limited by the Courant–Friedrichs–Lewy (CFL) condition (Courant et al., 1967; Gottlieb and Tadmor, 1991; Kurganov and Petrova, 2007):

$$\Delta t \leq \frac{1}{4}\min\left(\frac{\Delta x}{\max_{\Omega}(u \pm \sqrt{gh})}, \frac{\Delta y}{\max_{\Omega}(v \pm \sqrt{gh})}\right), \tag{8}$$

where Ω represents the whole simulation domain.

Replacing the second-order Runge-Kutta scheme with the first-order Euler scheme, the time integration in equation (6) reduces to:

$$\bar{\mathbf{U}}^{n+1}_{j,k} = \bar{\mathbf{U}}^{n}_{j,k} + \Delta t\left[\mathbf{R}_{F+G}\left(\bar{\mathbf{U}}^{n}\right)_{j,k} + \mathbf{S}_{B+f}\left(\bar{\mathbf{U}}^{n}_{j,k}, B_{j,k}\right)\right]. \tag{9}$$

We note that the HWP14 scheme applies a different time step computation for the partially flooded cells. It is based on the flux divergence of these cells. For more details on the draining time step computation, we refer the reader to the related work by Horváth et al. (2014).

4. GPUs and the CUDA platform

In this section, we introduce the concepts of the CUDA programming model which are most relevant for our implementation of the aforementioned schemes. We discuss performance and memory considerations from the perspective of our flux computation kernel, which is by far the most time-consuming and resource-intensive computational step in our algorithm.

On a GPU, parallel tasks are called threads. These are scheduled and executed simultaneously in groups referred to as warps. One warp contains 32 threads, which is the number of threads effectively processed in parallel by one CUDA streaming multiprocessor (SMM). GPUs have many SMMs, for example, a GeForce Titan has 14 SMMs running in parallel to increase the effective parallelism. Furthermore, threads are organized into larger structures called blocks, and blocks are organized into grids.(Wilt, 2013).

4.1 Memory usage

Understanding the CUDA memory model is often the key to achieving high performance on an NVIDIA GPU. The model consists of different regions with varying scopes, latencies and bandwidths. All threads have access to the same global memory. Shared memory is visible to all threads within a block with the same life time as the block. Each thread has its private registers and local memory.

Global memory resides in the device memory, which is the slowest, but the largest one. It is allocated and freed by the programmer. The device memory is accessed via 32, 64, or 128 byte memory transactions. These memory transactions must be aligned. Only the 32, 64, or 128 byte segments of device memory, all of them of the same size, can be read or written by memory transactions. In our implementation, we use a structure of 1-D arrays (SoA) to store elements of a vector, instead of using an array of structures, since performance is significantly affected by memory access patterns (Cook, 2012; NVIDIA Corporation, 2015; Wilt, 2013). Therefore, when using SoA, all threads in a warp access the same relative address when loading and storing the same element of a vector.

Local memory is often mentioned as virtual memory. It is allocated by the compiler, for example, if one uses large structures or an array that would consume too much register space. It resides in the global memory, and it is cached in L1 and L2 caches. If done right, one can use local memory to avoid costly memory transfers, but if the caches are overused and data are often evicted, the memory traffic and instruction count will be increased with a negative effect on the performance. Therefore, it is always necessary to profile the application. The best practice is to avoid the usage of the local memory whenever possible.

In this work, we focus on the on-chip memory, that is, registers and shared memory. *Shared memory* has much higher bandwidth and much lower latency than the local or the global memory. To achieve high bandwidth, it is divided into equally sized memory modules, called banks, which can be accessed simultaneously by the threads. On the Kepler architecture, the shared memory has 48 KB, and it is organized into 32 banks. However, if two threads in a warp access different memory from the same bank, a bank conflict occurs, and the access is serialized. To get maximum performance, we have to minimize these bank conflicts. In our flux kernel, we store temporal variables in shared memory using 2-D arrays. Bank conflicts can happen when accessing elements in the y direction, but not in the x direction. To circumvent this, we allocate arrays of $blocksize^*(blocksize + 1)$. By adding a padding element of 1, we avoid the bank conflicts.

With the introduction of the *Kepler shuffle* instructions, programmers have been provided with a new memory feature to increase performance. As a faster alternative to shared memory, the Kepler shuffle can be used to efficiently share values between threads in a warp-synchronous manner. On earlier hardware, this could only be done using shared memory. This involved writing data to the shared memory, synchronizing threads, and then reading the data back from the shared memory. The Kepler shuffle enables a thread to directly read a register from another thread in the same warp. This allows threads in a warp to collectively exchange or broadcast data. In our case, every warp is split into two rows of 16 grid cells (Figure 3). The shuffle up instruction sends a value of a cell to the next cell on the right. Since the right most cell does not have any cell on the right, it will send its values to the first one, indicated by the dashed lines in Figure 3. By applying the shuffle down instruction, we can shift data in the opposite direction.

4.2 Block size and occupancy

Choosing a good block size is essential for achieving high performance. The number of threads in the block should be a multiple of the warp size, which is currently 32. Blocks should contain at least 192 threads to hide arithmetic latency (Wilt, 2013). One of the keys to good performance is to keep the multiprocessors on the device as busy as possible. The occupancy value is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Higher occupancy does not always imply higher performance. Above a certain level, additional occupancy does not improve performance. Factors that determine occupancy are the number of registers used per thread and the amount of shared memory used per block. Excess consumption of these resources limits the number of blocks and warps per multiprocessor. Low occupancy always interfers with the ability to hide memory latency, resulting in performance degradation. In our flux kernel, we use 16×16 threads per block. On devices with compute capability 3.5, each multiprocessor is able to schedule 2048 threads simultaneously, that is, 8 blocks per multiprocessor in our case. Furthermore, we require 32 registers per thread and, consequently, 8096 registers per block. Since devices with compute capability 3.5 have 65,536 registers per multiprocessor, 8 blocks could be scheduled in a single multiprocessor. Regarding shared memory, when using the shuffle instructions, the flux kernel requires 4 temporary floating-point variables, which leads to a total of $16 \times 17 \times 4 \times 4 = 4352$ bytes of shared memory per block (the factor 17 stems from the padding of 1 to avoid bank conflicts). This means, we can achieve nearly 100% occupancy as we are limited neither by shared memory nor by register usage. However, if we do not use the shuffle instructions, we need 18 temporary floating-point variables to store in the shared memory, which results in $16 \times 17 \times 18 \times 4 = 19,584$ bytes, and the occupancy drops to 25%.

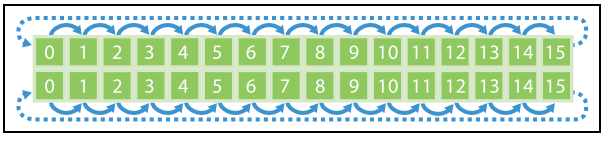

Figure 3. Illustration of the Kepler shuffle up instruction of width 16, used in the flux kernel of block size 16×16. The shuffle up instruction is applied to a single warp executed on two rows of a block. We use it to shift data from cell $C_{j,k}$ to the next cell $C_{j+1,k}$.

5. Implementation

We implemented a shallow water simulator as a plug in node for the Visdom framework (http://visdom.at/). Visdom is a software framework that integrates simulation, visualization, and analysis to assist decision making. The simulator was implemented with C++ and the NVIDIA CUDA Toolkit 6.5. The framework comprises multiple modules responsible for processing various tasks. Each module can be a node of a generic data flow network (Schindler et al., 2013). Nodes can directly import data or access the results produced by other nodes. In our case, the bathymetry node loads and samples the terrain for the shallow water node. The shallow water node simulates the fluid flow, and the simulation results are visualized using the OpenGL-based view node.

Our simulation node comprises several classes and uses C++ templates on both CPU and GPU sides to achieve the highest runtime performance. Being supplied with all the inputs, the node computes the

required amount of memory needed for the simulation and allocates the buffers. The simulation parameters such as the domain and cell size are uploaded to the constant memory of the GPU, where all kernels can access them. Our simulator supports dynamic modification of the input parameters, for example, to simulate the effects of barriers that can be placed into the scene. For the shallow water simulation, we need 11 buffers of the size of the simulation domain, 2 to hold the bathymetry values, 3 + 3 to hold the values of $\bar{\mathbf{U}}_{j,k}$ and $\bar{\mathbf{U}}^*_{j,k}$, and 3 to hold the flux values combined with the source terms. We need one more floating-point value per block to hold the maximum speed value of a block. In addition, smaller buffers are used to provide other features, but their sizes are insignificant compared to the domain size. The shallow water node supports hydrograph-based simulations. A hydrograph is a function describing the development of local water level and discharges over time, often used for specifying time-dependent boundary conditions. The hydrographs are stored as indices of the affected cells, along with time-varying data of the water elevations and velocities.

5.1 Domain partitioning

We coarsely decompose the problem into subproblems that can be solved independently in parallel by blocks of threads. Each subproblem is then split into finer pieces that can be solved cooperatively in parallel by all threads within a block. In case of our flux kernel, each block contains 16×16 threads or cells. The flux kernel loads data for all cells, but produces fluxes only for the inner cells (inner block), as shown in Figure 4. The inner block size is defined by the computational stencil which is the number of neighboring cells needed to compute the fluxes at a particular cell. The computation of the flux for the inner block requires data from more cells than it contains. Additional cells from adjacent inner blocks, called ghost or halo cells, are needed. Therefore, the outer blocks must be bigger and must overlap each other. The stencil differs for the KP07 and HWP14 schemes. In case of KP07, the flux kernel outputs fluxes for the inner 12×12 cells. When using the HWP14 scheme, the inner block is smaller and contains 10×10 cells, since it requires one more ghost cell in every direction.

5.2 Simulation steps

Figure 5 shows the main computation steps of our simulation process. The computation starts by calculating the fluxes ① for all cells in the domain. The time step ② is computed next. The maximum speed value in the domain is used in order to compute the time step size according to equation (8). After computing the fluxes and the maximum time step size, the time integration ③ follows. In addition, each block containing wet cells is marked as active. The last of the four main simulation steps is the application of the boundary conditions ④. At this point, the process can exit the simulation loop if the desired simulation time is reached. Otherwise, the computation proceeds. If no hydrograph is supplied to the simulation node and the sparse computation is disabled, the simulation skips the steps ⑤ and ⑥ and jumps to the flux computation ① again. However, if the hydrograph input is present, its properties are computed for the current time step ⑤. Another feature of the simulation node is the sparse block computation, adopted from Sætra (2013). Initially, all blocks are marked as wet. After evolving the solution in time, the integration kernel checks whether there is at least one wet cell in the block and stores the information in the global memory. If a hydrograph is supplied and the sparse computation is enabled, the blocks containing hydrograph cells should be activated as well. If a block is active, its neighbors could be flooded in the next time step, thus, all adjacent blocks are activated, too. Indices of active blocks are compacted ⑥ and used by the flux and time integration kernels to process only wet blocks.

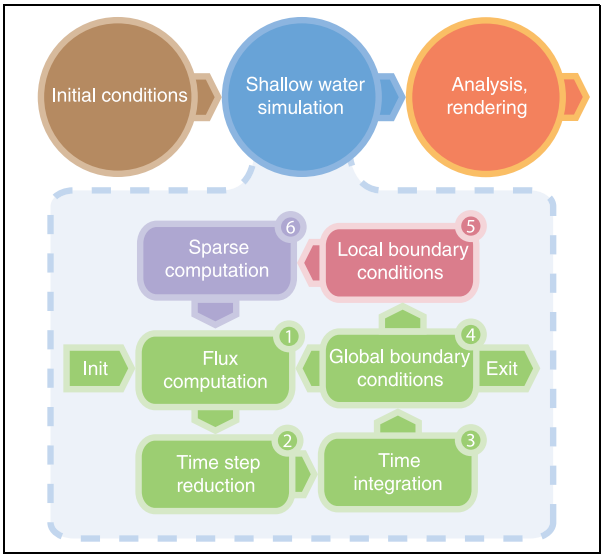

Figure 4. Domain partitioning and the computational stencil of the flux kernel. (a) Domain decomposition into blocks processed independently on the GPU. Fluxes are computed for inner block cells only. The number of ghost cells differs for the KP07 and HWP14 schemes. (b) Computation stencils for the pink cells. KP07 requires two cells in each direction, HWP14 needs three. Blue dots are variables at the cell center, that is, $\bar{\mathbf{U}}_{j,k}$ and $B_{j,k}$, brown dots are values at the cell interface midpoints $\mathbf{U}_{j\pm1/2,k}$ and $\mathbf{U}_{j,k\pm1/2}$, and the bathymetry values $B_{j\pm1/2,k\pm1/2}$ are defined at the green dots. GPU: graphics processing unit; KP07: Kurganov–Petrova scheme; HWP14: Horváth–Waser–Perdigão scheme.

The system can perform the first-order Euler or second-order Runge-Kutta time integrations. For the

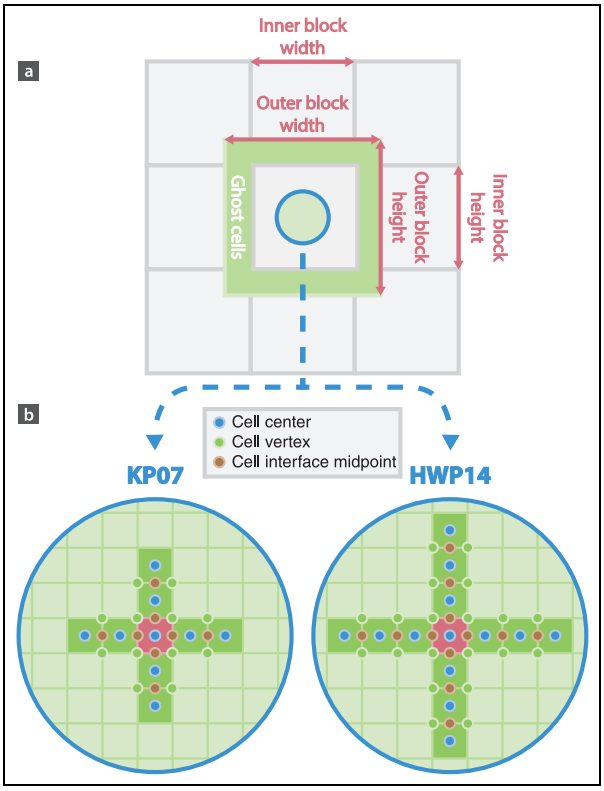

Figure 5. Data flow of the simulation system and steps of the simulation loop. Green boxes ① to ④ are the main steps of the simulation. Boxes ⑤ and ⑥ are optional. Box ⑤ is activated if there is a hydrograph attached. Box ⑥ is an optional optimization that skips dry cells.

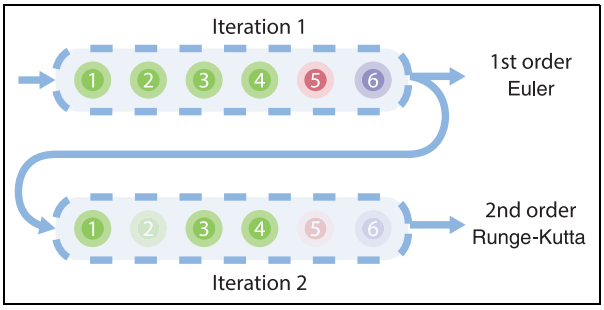

Figure 6. Simulation steps for the first- and second-order time integrations. ① Flux computation, ② time step reduction, ③ time integration, ④ global boundary conditions, ⑤ local boundary conditions, and ⑥ sparse computation. In the first iteration, all computations are active, in the second, some steps are skipped (desatured circles).

second-order accuracy, we make two simulation iterations, whilst for the first-order Euler method, only the first iteration is needed. The first iteration follows all computation steps of the simulation loop. The second iteration requires only three computation steps (Figure 6).

①*Flux computation*: The flux kernel is computationally the most expensive of our parallel routines. It is responsible for computing the source terms and the fluxes over all four interfaces of each cell. We have identified the six most important steps in the code and labeled them from **(a)** to **(f)** in Figures 7 and 8. Figure 8 provides an overview of the code. Figure 7 illustrates the role of the register and shared memory spaces of a block while the kernel is being executed. Our code design is guided by the principle of achieving a maximum amount of warp-synchronous computations. On the GPU, each warp processes 2×16 cells, that is, two rows of a block at once. Due to this, threads of different columns are able to share data with each other without any performance loss. In contrast, the communication between threads across different rows is slower since it requires explicit warp synchronizations. Whenever possible, threads should exchange data across columns only. We suggest a method of data *transposition* to layout the required data in x direction before the actual computations are done.

The kernel starts by allocating in shared memory four 2-D arrays required to hold the temporary variables. **(a)** The conserved variables $\bar{\mathbf{U}}_{j,k}$ and the bathymetry values at the right cell interface midpoints $B_{j,k+1/2}, B_{j+1/2,k}$ are loaded from global memory into registers, totaling to five floating-point numbers per cell. **(b)** At this stage, we can directly proceed with the warp-synchronuous computation of the fluxes in x direction without waiting for thread synchronization, since all required data have been loaded by the same warp. The computation exploits multiple shuffle instructions, factored out into the reusable *calcFlux* subroutine (details follow subsequently). **(c)** Next, we copy the bathymetry value in y direction and the conserved variables to the shared memory. Notice, that they are assigned to the 2-D array element by swapping the x and y indices associated with the thread (transposition). Now we have to wait until this process has been completed by all threads from all warps within the block. An explicit synchronization barrier assures that all data are available for computing the flux in y direction. Then we simply load the data from shared memory without swapping indices. **(d)** The stencil for the y-flux is now arranged in x direction of the block, and we can reuse our efficient shuffle-based (SB) *calcFlux* function. One might argue that the necessary data in y direction could be loaded from global memory again. However, this would significantly lower the performance, not only because the same values have to be loaded twice but also due to coalescing rules. Parallel threads running the same access instruction to consecutive locations in global memory achieve the most efficient memory transfer. Since the data in global memory is organized row after row in a linear array, the access in the y direction is not favorable. The flux kernel proceeds with **(e)** transposing the computed y-fluxes. This way, the y-fluxes are now stored at the correct cell

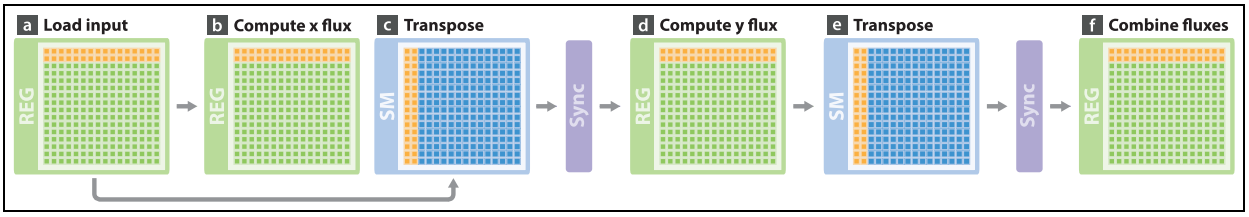

Figure 7. Overview of the flux kernel code from the perspective of SM and REG spaces of the block. Orange squares show the memory occupation for the current warp. Shuffle instructions operate on registers only in *x* dimension. To exploit them for the computation of the fluxes in both directions, we require data transpositions via the SM. Explicit thread synchronization barriers are shown in purple. SM: shared memory; REG: register.

```
// Allocate shared memory for temporary variables
// A padding of 1 avoids bank conflicts
__shared__ float smWaterLevels [blockSize][blockSize + 1];
__shared__ float smBathY [blockSize][blockSize + 1];
…

// (a) Load input. Store conserved variables and bathymetry in variables (registers)
float waterLevel = waterLevels[globalIdx];                          ← w_{j,k}
…
float bathRightInterfaceY = bathymetriesY [globalIdx];              ← B_{j,k+1/2}
float bathRightInterfaceX = bathymetriesX [globalIdx];              ← B_{j+1/2,k}

// (b) Compute x flux. No need for prior thread synchronization because we only
//     require data loaded within the same warp
float3 fluxX = calcFlux (waterLevel, dischargeX, dischargeY, bathRightInterfaceX );

// (c) Transpose. Store conserved variables and bathymetry (in y-direction only)
//     inside shared memory in a transposed way
smWaterLevels [threadIdx.x][threadIdx.y] = waterLevel;
…
smBathY [threadIdx.x][threadIdx.y] = bathRightInterfaceY;

// Make sure that all transposed data is available from other threads
__syncthreads();

// Load data from shared memory
waterLevel = smWaterLevels [threadIdx.y][threadIdx.x];
…
bathRightInterfaceY = smBathY [threadIdx.y][threadIdx.x];

// (d) Compute y flux. The previous transposition allows us to use the same
//     shuffle-based technique
float3 fluxY = calcFlux (waterLevel, dischargeX, dischargeY, bathRightInterfaceY);

// (e) Transpose. Store fluxes in y-direction at the correct cells in shared memory
//                (we re-use the previous buffers which are no longer needed)
smWaterLevels [threadIdx.x][threadIdx.y] = fluxY.x;
…

// Synchronize after transposing back to shared memory
__syncthreads();

// (f) Combine fluxes. Write the results into global memory
```

Figure 8. Overview of the flux kernel code.

locations in shared memory. Next, we need to synchronize the threads to ensure that all threads have finished flux computations. **(f)** The kernel concludes by loading these values without swapping indices and combines them with the *x* flux and the source terms. The result is stored as a three-element vector in global memory.

In the *calcFlux* subroutine, the differences between KP07 and HWP14 become apparent (Figure 9). HWP14 exhibits a more elaborate behavior from the GPU perspective. The relevant code sections are

```
SUB: float3 calcFlux (w = waterLevel, dischargeX, dischargeY, bathRightInterfaceX)

// Fetch bathymetry at left interface by shuffling
float bathLeftInterfaceX = __shfl_up(bathRightInterfaceX , 1, blockSize);   ← B_{j-1/2,k}

// Calculate average water level and determine whether the cell is partially flooded
// HWP14 specific
bool isPartiallyFlooded = waterDepth < (cellSize * 0.5f) *
                          abs(bathLeftInterfaceX - bathRightInterfaceX ) / cellSize;

// Reconstruct water levels and discharges at cell interfaces

// Use shuffle to fetch water levels needed for minmod reconstruction
float wCellLeft  = __shfl_up(w, 1, blockSize);                              ← w_{j-1,k}
float wCellRight = __shfl_down(w, 1, blockSize);                            ← w_{j+1,k}
…

// Variables at left-left / right-right cells could be needed and shuffling must be
// done outside the following if-blocks to make sure that all data is available
float wCellLeftLeft      = __shfl_up(w, 2, blockSize);                      ← w_{j-2,k}
float wCellRightRight    = __shfl_down(w, 2, blockSize);                    ← w_{j+2,k}
…

if ( ! isPartiallyFlooded )
{
  // Compute point values for the conserved variables at the inner cell interfaces
  float2 wPointVal = reconstruct (wCellLeft , w, wCellRight);
  …
}
else // HWP14 specific
{
  // Bathymetry steepness needed for cases below
  float bathSteepness = (bathRightInterfaceX - bathLeftInterfaceX) / cellSize;

  // At the partially flooded cells, branching occurs
  if ( isFullyFlooded(leftCell) && bathSteepness > 0 )
  {
    float2 wPointVal = reconstructAtPartial
                       (wCellLeftLeft, wCellLeft , w, bathLeftInterfaceX );
  }
  else if ( isPartiallyFlooded(leftCell) && bathSteepness > 0 )
  …
  … // For the logic of all cases, see Section 4 in [3]
}

// Equivalent reconstruction for point values of discharges
float dischargeXPointVal, dischargeYPointVal  = … ;

// Compute flux at right interface using the point values
// KP07 and HWP14 differ slightly here, see Equations 45, 46 in [3]
float fluxRightInterfaceX = interfaceFlux
                            (wPointVal.y, wPointValCellRight.x, dischargeXPointVal.y, …);

float fluxLeftInterfaceX = __shfl_up(fluxRightInterfaceX, 1, blockSize);

// Conclude by computing the flux at the current cell
return  centralUpwind (fluxLeftInterfaceX, fluxRightInterfaceX);
```

Figure 9. Subroutine *calcFlux* to evaluate the flux in any direction. The function assumes that the stencil is organized in *x* direction, no matter if invoked for the computation of the *y*-flux. HWP14 requires additional work for the correct treatment of partially flooded cells (red parts).

marked in red. HWP14 has to detect the partially flooded cells and treat them differently according to a set of rules. This results in a more complex branching pattern that slows down the process. For the computation of the point values at the cell interfaces (Figure 1(d)), the subroutine acquires all the necessary data from the neighboring cells through shuffling the input values by one index, and in case of HWP14, also by two indices. In KP07, point values are reconstructed in the same way at all cells. HWP14 makes a distinction between partially flooded cells and fully wet cells. Moreover, the treatment of the partially flooded cells depends on the local bathymetry steepness and the amount of water contained in the neighboring cells. Consequently, the execution path divergence is greatly influenced by the current simulation status. There are multiple if-else blocks where we cannot guarantee that half-warps take the same execution path. This breaks the parallel execution of the warp and introduces some overhead compared to KP07. After the reconstruction of all point values, the subroutine proceeds with computing the flux at the right cell interface (Figure 1(e)). Here, the two schemes exhibit slight differences which are not crucial from the implementation point of view. The subroutine continues by acquiring the flux at the left interface through shuffling. The use of the shuffle instruction avoids to compute the same flux twice, first time for cell $C_{i,j}$ and second time for cell $C_{i+1,k}$. Finally, the fluxes at both interfaces are used to evaluate the flux at the current cell according to the central-upwind flux function.

As a side effect, the flux kernel evaluates the maximum speed value inside a block. This value is later needed to compute the time step size for the time integration. Our calculation of the maximum speed value is based on the commonly used butterfly-reduction pattern. We exploit the exclusive-or shuffle instructions to accelerate this process.

②*The time step reduction*: The time step reduction kernel is responsible for finding the highest speed value within the whole domain. Based on the values obtained for each block, this kernel employs a similar butterfly reduction to evaluate the maximum speed value for the entire grid. The maximum allowed time step size is then calculated according to the CFL condition in equation (8).

③*The time integration*: The time integration kernel advances the solution in time and performs one substep of the Runge-Kutta integrator. Since the kernel does not require any ghost cells, its block size has the same dimensions as the inner block of the flux kernel. The kernel starts by reading the conserved variables $\bar{\mathbf{U}}_{j,k}$, the average bathymetry value $B_{j,k}$ at the cell center, and the combined fluxes $\mathbf{R}_{F+G}(\bar{\mathbf{U}})_{j,k}$. After this, the solution advances in time. In addition, one has to ensure that no negative water heights are produced due to floating-point round-off errors.

④*Our global boundary condition*: Our global boundary condition kernel implements four types of boundary conditions using global ghost cells. The supported types are reflective or wall boundaries, inlet discharge, free outlet, and fixed water elevation. To implement these boundaries, we have to manually set both the cell averages and the reconstructed point values at the cell interface midpoints. Similar to Brodtkorb et al. (2012), we exploit the property of the minmod limiter. We recall that the minmod limiter uses the forward, central, and backward difference approximations to the derivative and always selects the least steep slope, or 0 if any of them have opposite signs. We are allowed to set the reconstructed point values to arbitrary values as long as we ensure that the least steep slope is 0. To fulfill this condition, we need two ghost cells in each direction at the boundaries of the domain. Global ghost cells are updated at every step and thus they do not need any special treatment different from the handling of the interior cells.

The wall boundary condition is implemented by mirroring the last two interior cells at the boundary and changing the sign of the velocity component perpendicular to the boundary. The input discharge and fixed water elevations are implemented similarly. For the former, we set a fixed discharge value, whereas for the latter, a fixed water elevation. The free outlet is implemented by copying the values of the cell averages of the last interior cell to the two ghost cells.

⑤*Local boundary conditions*: Inside the domain, we support local boundary conditions to simulate phenomena such as river flow or sewer overflow scenarios. A hydrograph, describing a local water level and discharge over time, can be attached to multiple cells to impose time-varying inlet boundaries. At every time step, we compute the water level and velocity vectors for each affected cell by interpolating the hydrograph function values.

⑥*Spare computation*: The sparse computation is used to exclude blocks that do not contain water. The time integration kernel marks the active blocks that contain wet cells or have cells with nonzero discharges. After this, we compact the indices of the active blocks. On the next iteration, these indices are loaded by the flux and integration kernels to find the offsets of the active blocks. If the sparse block computation is enabled, we have to activate blocks that contain hydrograph cells, since the hydrograph water depths could start at 0 and increase later during the simulation. Therefore, it is necessary to track whether the hydrograph cells become active. If they become inactive again, they will be excluded from the computation.

Table 1. NVCC flags used to compile CUDA source files.

-arch=sm_35	NVIDIA GPU architecture to generate
-maxrregcount=32	Maximum amount of registers per thread
-ftz=true	Flush denormals to 0
-fmad=true	Force fused multiply–add operations
-prec-div=false	Faster, but less accurate division
-prec-sqrt=false	Faster, but less accurate square root
-Xptxas -dlcm=ca	Increases the L1 cache to 48 KB

6. Evaluation

In this section, we provide a performance analysis of our latest, SB implementations of HWP14 and KP07 compared to previous, shared memory-only (SMO) versions. The comparison is done using real-world scenarios. Our benchmarks run on a desktop PC equipped with a 3.4 GHz quad core Intel Xeon CPU and 16 GB RAM. We use an NVIDIA GeForce Titan graphics card which has 6 GB GDDR5 memory. GPU sources are compiled with CUDA version 6.5 using the compiler flags shown in Table 1.

6.1 Validation: Malpasset Dam break

Before testing the simulation speed, we evaluate the correctness of our new solver using the well-known historical Malpasset Dam break event. Our validation is based on the data set available from the TELEMAC samples (http://www.opentelemac.org/). The original data set consists of 104,000 unstructured points. The corresponding simulation grid contains 1149×612 cells, each cell of the size of 15×15 m. We set the desingularization constant $\epsilon = 0.40$ m and use two Manning coefficients $n = 0.025$ m$^{1/3}$/s, and $n = 0.033$ m$^{1/3}$/s. We use CFL $= 0.25$ in all our simulations and simulate the first 4000 s after the actual breach.

We employ the outcome of laboratory experiments on a 1:400 scale model to verify the correctness of our numerical results. In these experiments, researchers have recorded wave front arrival times (Frazao et al., 1999) and maximum water elevations (Hervouet and Petitjean, 1999) at 14 wave gauge locations (S1–S14). Our verification uses only locations S6–S14, since no data are available for the other gauge locations. Figure 10 displays the measurement points and the water extent after 4000 s of simulated model time.

Figures 11 and 12 show our simulation results compared to the physical data acquired by the laboratory model. Overall, there is good agreement with the measurements. Small discrepancies between the scale model and the numerical results were also reported by others (Brodtkorb et al., 2012; George, 2011; Hou et al., 2014), and our results are consistent with these. We simulated two different roughness values for the terrain and compared the maximum water elevations and wave

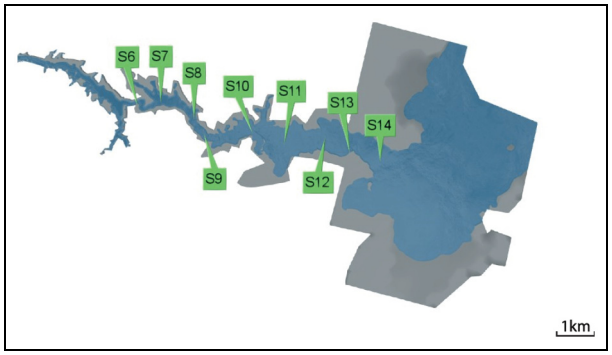

Figure 10. Simulated dam break of Malpasset. The water extent at the time step 4000 s is shown (blue). The simulation results are verified with the experimental data obtained from laboratory models at the displayed locations (green labels).

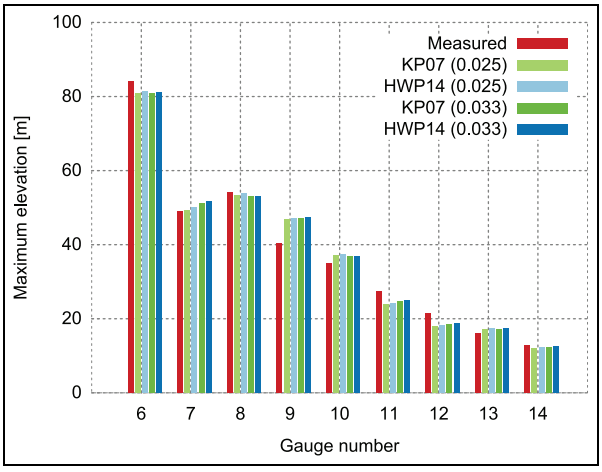

Figure 11. Verification of the maximum water elevations during the Malpasset Dam break event at nine gauge locations (S6–S14) for two roughness values (0.025 and 0.033).

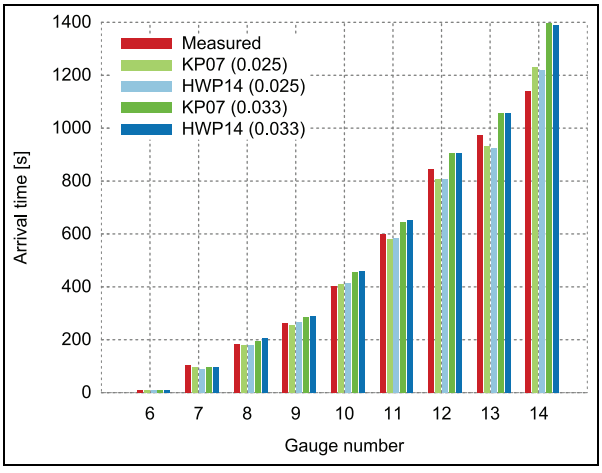

Figure 12. Verification of the wave arrival times during the Malpasset Dam break event at nine gauge locations (S6–S14) for two roughness values (0.025 and 0.033).

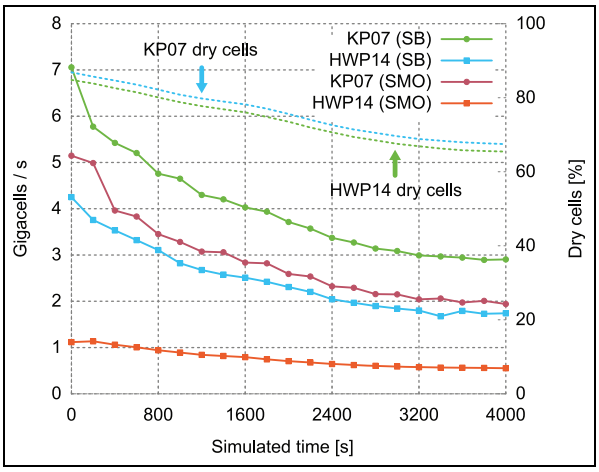

Figure 13. Estimated solver performance, measured in gigacells/s, for the Malpasset Dam break scenario (solid lines). The dashed lines show the percentage of dry blocks within the simulation domain, which is different for KP07 and HWP14. KP07: Kurganov–Petrova scheme; HWP14: Horváth–Waser–Perdigão scheme.

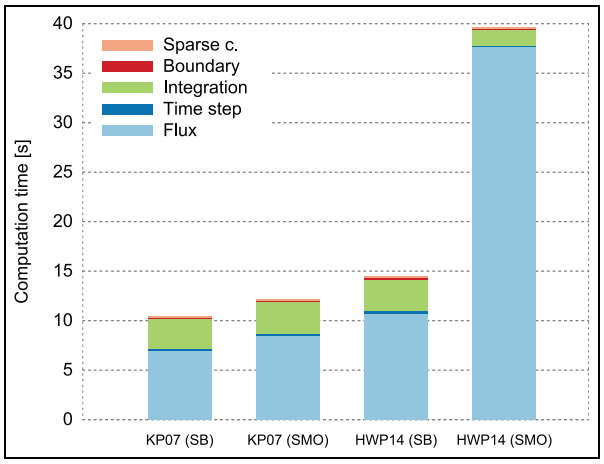

Figure 14. Overall GPU runtime of the implemented solvers for the simulation of 4000 s of the Malpasset Dam break event, including the time distribution over five phases of a single computation iteration. GPU: graphics processing unit.

arrival times. The roughness value 0.025 is associated with gravelly earth channels, and the roughness value 0.033 corresponds to weedy, stony earth channels and floodplains with pasture and farmland. The higher the roughness value, the slower the flow velocity, which causes the water level to increase for the same discharge. This effect is visible in Figure 11, where the elevation values are slightly increased when using the higher roughness value. The wave arrival times are more affected by the roughness value than the water elevations. A higher roughness value results in slower wave propagation. We note that, with a uniform roughness value across the whole domain, we can approximate the real-world process to a certain extent only. Usually, the roughness values vary spatially across the domain.

Performance measurements

To evaluate the performance of the implemented solvers, we first examine the number of gigacells that can be processed per second. This quantity is computed by counting the number of wet cells that have been processed in one second of computational time. Thus, the quantity is independent of the actual number of time steps needed and provides a measure for the kernel performance. Figure 13 shows how the performance decreases over time as the water spreads through the domain and the number of dry blocks decreases. The dashed lines show the percentage of the dry cells during the simulation. The HWP14 solver has smaller blocks, hence, it needs more blocks than the KP07 solver to cover the same domain. This explains the difference in the percentage of dry cells in the graph. One can notice that, due to simpler computations, the KP07 solver can process more cells per second than the HWP14 solver. Later in the text, we will show that the KP07 solver *has to* process much more cells for real-world scenario modeling, since it requires more time steps to simulate the same duration in long simulation runs. In the end, this fact makes the HWP14 solver significantly more efficient.

Figure 14 displays the absolute GPU runtime of the solvers required to simulate the first 4000 s of the Malpasset Dam break event. While the previous SMO implementation of the HWP14 scheme is obviously slower, the new SB approach for HWP14 exhibits clear improvements. Its running times are similar to those of the KP07 implementations. Even though the newly implemented SB solver for KP07 completes the 4000 s simulation slightly faster, the situation changes in real-world scenarios, where hours or days of flooding have to be simulated. The difference becomes dramatic when ensembles of such scenarios are used for uncertainty treatment.

In Figure 14, we see that the computation times for the two KP07 solvers are only slightly different, while there is a significant difference for the HWP14 solvers. For the KP07 (SB) solver, we cannot further improve the GPU utilization by reducing the shared memory footprint, since we are already limited by the number of registers per block, that is, the GPU cannot launch more blocks on a multiprocessor. However, the HWP14 (SMO) solver requires more shared memory than KP07 (SMO), and more synchronization barriers, which are responsible for the performance loss. The use of the shuffle instructions (SB) avoids the need for some of the synchronization barriers, since threads in the same warp are always executed in a synchronous

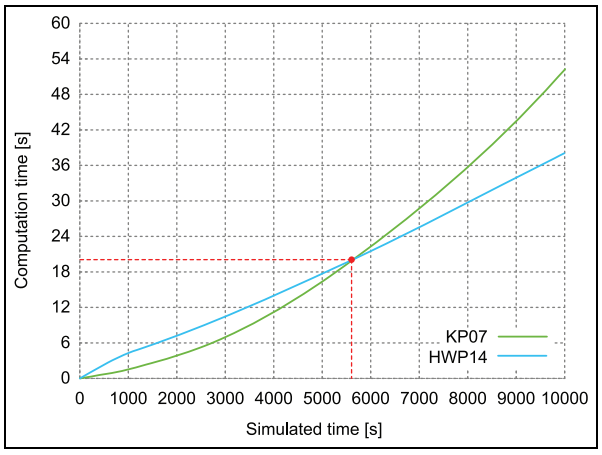

Figure 15. Computation time of the two shuffle-based solvers for the simulation of 10,000 s of the Malpasset Dam break event. The red dot shows the intersection of the simulated and the computation time of the solvers.

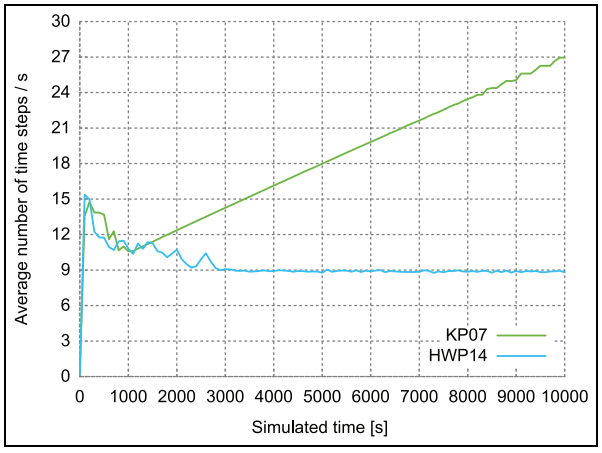

Figure 16. Average number of time steps/s required by the two shuffle-based solvers for the simulation of 10,000 seconds of the Malpasset Dam break event.

way. Moreover, this leads to a reduced shared memory footprint, thus to improve GPU utilization and a high performance gain.

Figures 15 and 16 show the computation time and the average number of time step per second for the first 10,000 s of the Malpasset Dam break event. In Figure 15, we see that the KP07 solver is faster for the first 5600 s. However, over time, it has to perform more and more time steps (Figure 16). Thus, it requires more computation time than HWP14 as the simulation progresses.

We now show that for real-world simulations, our SB implementation of HWP14 outperforms KP07, since it actually requires considerably fewer time steps to solve the same problem. As a test case, we use the overtopping of mobile flood protection walls in Cologne, Germany (Figure 17(b)). Such overtopping happens when the water in the Rhine river raises above the level of 11.9 m, marked with a red line in Figure 17(a). A standard way to use uncertain overtopping predictions for action planning is to simulate ensembles of overtopping events. In our test case, we simulate an ensemble of 10 overtopping events corresponding to water levels from 11.95 m to 12.95 m, displayed by a set of hydrographs in Figure 17(a). The simulation domain of approximately 3.1×7.5 km is shown in Figure 17(c). The corresponding grid contained approximately 2.6 million cells of 3×3 m. For each ensemble member, we decided to simulate only the first 2 h of overtopping, since the KP07 solver was too slow. In Figure 17(b) and (c), the expected building damage and water depths, aggregated over the whole ensemble, are displayed with colors. Figure 18(a) shows the computation runtimes for each simulated scenario. These are measured for the SMO and SB implementations of the KP07 and HWP14 schemes. Clearly, the SB implementation of HWP14 exhibits the best performance, closely followed by the SMO implementation of the same scheme. Both implementations of KP07 turn out to be considerably slower. This is due to high velocities eventually generated at dry/wet boundaries and, consequently, smaller time steps sizes. In turn, a larger number of smaller time steps imply more computation effort to advance the simulation up to the required model time. This computation overhead multiplies with the size of the ensemble. For illustration, we compare the number of time steps required by both schemes to simulate every scenario (Figure 18(b)). Note that the number of time steps depends solely on the scheme and not on the implementation (SB or SMO), hence two groups of bar charts instead of four. As one can see in Figure 18(b), there is a large, sometimes up to an order of magnitude, difference between the numbers of time steps required by the two schemes, which proves our assertion.

To sum up, we discuss the main factors responsible for the differences in the number of time steps between the presented case studies. The first factor is the flow velocity. In case of the Malpasset Dam break, the flow velocity is very high due to the high water level at the dam. At the beginning of the simulation, the error in the velocities along the dry/wet boundaries is negligible compared to the high flow velocity in the middle of the stream. As time advances, the reservoir is draining and the flow velocity decreases. This allows for a smaller number of time steps (see the first 1000 s in Figure 16). However, for KP07, the error is growing along the boundaries, causing an increase in the number of time steps again. In the Cologne case, the water depth is very shallow with low flow velocities, which means that the effects of the error accumulation along the dry/wet boundaries occurs sooner. The second factor is the

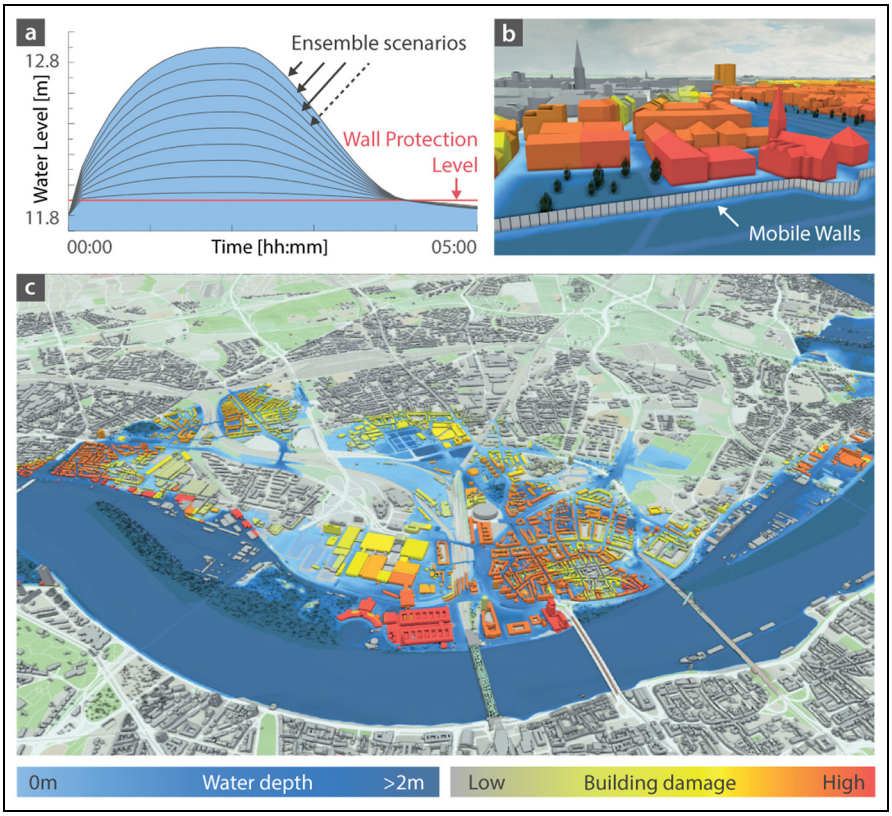

Figure 17. Uncertainty-aware prediction of mobile flood protection wall overtopping in Cologne. (a) Input hydrographs forming an ensemble of 10 different scenarios with varying peak levels. (b, c) Visualization of ensemble results. Buildings are colored according to the expected damage. The terrain is colored according to the average water depth.

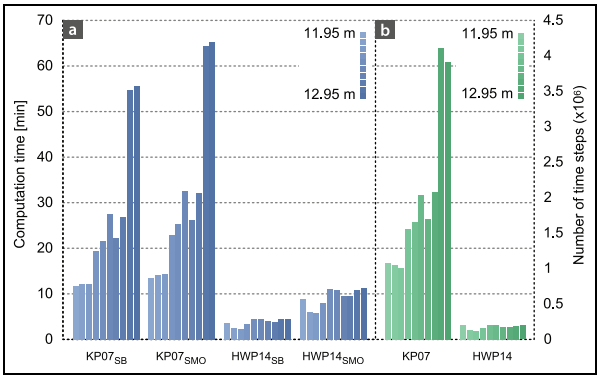

Figure 18. Runtimes and number of time steps for the first 2 h of model time for each overtopping scenario of the simulated ensemble for Cologne.

resolution of the bathymetry. It is 3×3 m in Cologne, which is $5\times$ higher than in the Malpasset case, where we use a cell size of 15×15 m. This increases the rate of the accumulation of the error, since the computation of the time step size depends not only on the maximum velocity but also on the grid resolution as in equation (8).

7. Conclusion

In recent years, a lot of effort has been made toward efficient GPU-based solvers for SWEs on Cartesian grids. However, their application to real-world scenarios is often plagued with significant slowdowns. This happens due to spurious high velocities generated by the corresponding numerical schemes at dry/wet boundaries. These high velocities sometimes lead to time steps so small that the simulation barely advances in time. Such slowdowns are unacceptable for real-world decision making in flood management, where large ensembles of flood scenarios have to be simulated to handle the prediction uncertainty. Keeping this problem in mind, we developed the HWP14 scheme (Horváth et al., 2014) that stabilizes the velocities and increases the time step size but requires a more complex algorithm. It should also be noted that, even in shorter run times where KP07 outperforms HWP14 in runtime efficiency, the HWP14 scheme has a crucial advantage over KP07: it produces simulations that are more physically consistent. For instance, it avoids the buildup of unrealistically high velocities close to boundaries, which actually ends up contributing to the loss of efficiency of KP07 for longer runs.

In this article, we present a novel GPU-based implementation that fully reveals the potential of the HWP14 scheme. The implementation is optimized for the Kepler GPU architecture. In order to achieve high utilization, we exploit the capabilities of the shuffle instructions. Due to their warp-synchronous nature, we can share data between the threads and thus avoid explicit synchronizations. However, we have to reorganize data using the shared memory to be able to use them for the second dimension efficiently. As demonstrated by the evaluation, our SB implementation of HWP14 outperforms the existing alternatives significantly when applied to real-world scenarios. This allows for the use of the presented GPU solver for real-world decision making, where, for instance, imminent floods can be modeled and analyzed in time-critical situations. Moreover, it can be efficiently used for planning purposes where a large number of scenarios need to be explored prior to any flood events.

Nevertheless, open questions remain, motivating further improvements. Reconstruction cases in the presented solver lead to a conditional branch divergence in the code. To avoid performance loss, these need to be carefully optimized. Furthermore, with the new Thrust library version 1.8, which will support CUDA streams, we will be able to increase the GPU utilization. This will further reduce the overall simulation time. Another interesting direction for future work is to provide the solver with the support for adaptive grids for handling even larger simulation domains.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by grants from the Austrian Science Fund (FWF) project number W1219-N22 (Vienna Doctoral Programme on Water Resource Systems) and from the Vienna Science and Technology Fund (WWTF) project number ICT12-009 (Scenario Pool), and from the European Research Council (ERC) Advanced Grant FloodChange, project number 291152. We thank the flood protection centre of Steb Cologne, AöR.

References

Acuña M and Aoki T (2009) Real-time tsunami simulation on multi-node GPU cluster. In: *ACM/IEEE Conference on Supercomputing*, 2009. Oregon Convention Center, Portland, Oregon, 14–20 November 2009.

Bollermann A, Chen G, Kurganov A, et al. (2013) A well-balanced reconstruction of wet/dry fronts for the shallow water equations. *Journal of Scientific Computing* 56(2): 267–290.

Brandvik T and Pullan G (2008) Acceleration of a 3D Euler solver using commodity graphics hardware. In: *46th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, USA, 7–10 January 2008, p. 607.

Brodtkorb AR and Sætra ML (2012) Explicit shallow water simulations on GPUs: Guidelines and best practices. In: *XIX International Conference on Water Resources, CMWR*, Champaign, USA, 17–22 June 2012, pp. 17–22.

Brodtkorb AR, Sætra ML and Altinakar M (2012) Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids* 55: 1–12.

Brodtkorb AR, Hagen TR, Lie K.-A, et al. (2010) Simulation and visualization of the Saint-Venant system using GPUs. *Computing and Visualization in Science* 13(7): 341–353.

Causon DM, Ingram DM and Mingham CG (2001) A cartesian cut cell method for shallow water flows with moving boundaries. *Advances in Water Resources* 24(8): 899–911.

Courant R, Friedrichs K and Lewy H (1967) On the partial difference equations of mathematical physics. *IBM Journal of Research and Development* 11(2): 215–234.

Cook S (2012) *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, Applications of GPU Computing Series*. Waltham, MA: Elsevier Science, 2012.

de la Asunción M, Mantas JM and Castro MJ (2010) Programming CUDA-based GPUs to simulate two-layer shallow water flows. In: *Euro-Par 2010-Parallel Processing, 2010*, Italy, August 3–September 3, pp. 353–364. Heidelberg, Germany: Springer.

De La Asunción M, Mantas JM and Castro MJ (2011) Simulation of one-layer shallow water systems on multicore and CUDA architectures. *The Journal of Supercomputing* 58(2): 206–214.

de la Asuncin M, Castro MJ, Fernndez-Nieto E, et al. (2013) Efficient GPU implementation of a two waves TVD-WAF method for the two-dimensional one layer shallow water system on structured meshes. *Computers & Fluids* 80: 441–452.

Duran A, Liang Q and Marche F (2013) On the well-balanced numerical discretization of shallow water equations on unstructured meshes. *Journal of Computational Physics* 235: 565–586.

EU (2007) Directive 2007/60/EC of the European Parliament and of the Council of 23 October 2007 on the assessment and management of flood risks. *Official Journal of the European Union* L288: 27–34.

Frazao SS, Alcrudo F and Goutal N (1999) Dam-break test cases summary. In: *The Proceedings of the 4th CADAM Meeting*, Zaragoza, Spain, 18–19 November 1999.

George D (2011) Adaptive finite volume methods with well-balanced Riemann solvers for modeling floods in rugged terrain: application to the Malpasset dam-break flood (France, 1959). *International Journal for Numerical Methods in Fluids* 66(8): 1000–1018.

Gottlieb S and Shu C-W (1998) Total variation diminishing Runge-Kutta schemes. *Mathematics of Computation of the American Mathematical Society* 67(221): 73–85.

Gottlieb D and Tadmor E (1991) The CFL condition for spectral approximations to hyperbolic initial-boundary value problems. *Mathematics of Computation* 56(194): 565–588.

Hagen TR, Lie K-A and Natvig JR (2006) Solving the Euler equations on graphics processing units. In: Alexandrov VN, van Albada GD, Sloot PMA, et al. (eds) *Computational Science–ICCS 2006*, Reading, UK, 28–31 May 2006, pp. 220–227. Berlin/Heidelberg: Springer.

Hagen TR, Hjelmervik JM, Lie K.-A, et al (2005) Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory* 13(8): 716–726.

Hervouet J-M and Petitjean A (1999) Malpasset dam-break revisited with two-dimensional computations. *Journal of Hydraulic Research* 37(6): 777–788.

Horváth Z, Waser JW, Perdigão RAP, et al. (2014) A two-dimensional numerical scheme of dry/wet fronts for the Saint-Venant system of shallow water equations. *International Journal for Numerical Methods in Fluids*. doi:10.1002/fld.3983. Available at: http://dx.doi.org/10.1002/fld.3983 (accessed 2 February 2016).

Hou J, Liang Q, Zhang H, et al (2014) Multislope MUSCL method applied to solve shallow water equations. *Computers & Mathematics with Applications* 68(2): 2012–2027.

Hou J, Liang Q, Simons F, et al. (2013) A 2d well-balanced shallow flow model for unstructured grids with novel slope source term treatment. *Advances in Water Resources* 52: 107–131.

Klöckner A, Warburton T, Bridge J, et al. (2009) Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics* 228(21): 7863–7882.

Kurganov A and Levy D (2002) Central-upwind schemes for the saint-venant system. *ESAIM: Mathematical Modelling and Numerical Analysis* 36(03): 397–425.

Kurganov A and Petrova G (2007) A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. *Communications in Mathematical Sciences* 5(1): 133–160.

Lacasta A, Morales-Hernández M, Murillo J, et al. (2014) An optimized gpu implementation of a 2d free surface simulation model on unstructured meshes. *Advances in Engineering Software* 78: 1–15.

Lacasta A, Morales-Hernández M, Murillo J, et al. (2015) GPU implementation of the 2D shallow water equations for the simulation of rainfall/runoff events. *Environmental Earth Sciences* 74(11): 7295–7305.

Lastra M, Mantas JM, Ureña C, et al. (2009) Simulation of shallow-water systems using graphics processing units. *Mathematics and Computers in Simulation* 80(3): 598–618.

Li G, Gao J and Liang Q (2013) A well-balanced weighted essentially non-oscillatory scheme for pollutant transport in shallow water. *International Journal for Numerical Methods in Fluids* 71(12): 1566–1587.

Liang W-Y, Hsieh T-J, Satria MT, et al. (2009) A GPU-based simulation of tsunami propagation and inundation. In: Arrems H and Chang S-L (eds) *Algorithms and Architectures for Parallel Processing*. Berlin/Heidelberg: Springer, pp. 593–603.

Noelle S, Pankratz N, Puppo G, et al. (2006) Well-balanced finite volume schemes of arbitrary order of accuracy for shallow water flows. *Journal of Computational Physics* 213(2): 474–499.

NVIDIA Corporation (2015) *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2015.

Open TELEMAC-MASCARET, http://www.opentelemac.org/ (accessed 24 June 2015).

Perdigão RA and Blöschl G (2014) Spatiotemporal flood sensitivity to annual precipitation: evidence for landscape-climate coevolution. *Water Resources Research* 50(7): 5492–5509.

Russo G (2005) Central schemes for conservation laws with application to shallow water equations. In: Rionero S and Romano G (eds) *Trends and Applications of Mathematics to Mechanics*. Milano, Italy: Springer, pp. 225–246.

Sætra M (2013) Shallow water simulation on GPUs for sparse domains. In: Cangiani A, Davidchack RL, Georgoulis EH, et al. (eds) *Numerical Mathematics and Advanced Applications 2011*. Heidelberg, Germany: Springer, pp. 673–680.

Sætra ML, Brodtkorb AR and Lie K-A (2014) Efficient GPU-implementation of adaptive mesh refinement for the shallow-water equations. *Journal of Scientific Computing* 63(1): 23–48.

Schindler B, Waser J, Ribicic H, et al. (2013) Multiverse data-flow control. *Visualization and Computer Graphics, IEEE Transactions on* 19(6): 1005–1019.

Shu C-W (1988) Total-variation-diminishing time discretizations. *SIAM Journal on Scientific and Statistical Computing* 9(6): 1073–1084.

Vacondio R, Dal Palù A and Mignosa P (2014) GPU-enhanced finite volume shallow water solver for fast flood simulations. *Environmental Modelling & Software* 57: 60–75.

Visdom—an integrated simulation and visualization system, http://visdom.at/ (accessed 24 June 2015).

Wang P, Abel T and Kaehler R (2010) Adaptive mesh fluid simulations on GPU. *New Astronomy* 15(7): 581–589.

Waser J, Konev A, Sadransky B, et al. (2014) Many plans: multidimensional ensembles for visual decision support in flood management. *Computer Graphics Forum* 33(3): 281–290. doi:10.1111/cgf.12384. Available at: http://dx.doi.org/10.1111/cgf.12384 (accessed 2 February 2016).

Wilt N (2013) *The CUDA handbook: A comprehensive guide to GPU programming*. Boston, MA: Pearson Education, 2013.

Xing Y and Shu C-W (2005) High order finite difference WENO schemes with the exact conservation property for the shallow water equations. *Journal of Computational Physics* 208(1): 206–227.

Author biographies

Zsolt Horváth is a PhD student at the Vienna University of Technology and a researcher in the VRVis Research Center for Virtual Reality and Visualization. He completed an engineering master's diploma in 2012 at the Faculty of Information Technology at the Brno University of Technology, Czech Republic. His work focused on real-time and

interactive simulation of fluids on general purpose graphics processing units (GPUs) using Smoothed particle hydrodynamics (SPH). In 2012, he joined the Computer Graphics Research Group at the Brno University of Technology and worked in the field of Computer Vision. His current research topics involve water resource management and flood simulations on GPUs.

Rui AP Perdigão is a mathematical physicist and Earth system dynamicist at the Institute of Hydraulic Engineering and Water Resources Management of the Vienna University of Technology. He is responsible for fundamental research on mathematical physical methods and dynamical processes for better understanding and modeling flood regime changes and underlying physical causes. He conducted independent research on mathematical physics, predictability, and uncertainty dynamics, from conceptual and methodological research to applications to the physics of climate. He has also conducted research and development for the EU/ESA Copernicus Program and the North Atlantic Alliance.

Jürgen Waser is a researcher and project leader at the VRVis Research Center for Virtual Reality and Visualization. Since completion of his master's in technical physics, he has been working in the area of flood simulation and visualization. He is the cofounder of the Visdom framework, which combines visualization, simulation, and analysis techniques to assist decision making. His PhD thesis demonstrates Visdom as a decision support system for flood management. His research interests include integrated visualization systems, visual analytics, simulation steering, and decision support.

Daniel Cornel received his BSc degree in media informatics from the Vienna University of Technology in 2011 and continued his studies with the master's program in visual computing, which he completed in 2014. During his studies, he specialized in real-time rendering with a focus on engine development for video games. He joined the Visualization group of the VRVis Research Center for Virtual Reality and Visualization in 2014. Currently, he is a PhD student and focuses on abstract and simplified visualization of time-dependent and uncertain geospatial data.

Artem Konev obtained a BSc degree from the Faculty of Applied Mathematics and Computer Science of Novosibirsk State Technical University, Russia. Later on, he got an MSc in computational logic as a double degree from the Faculty of Computer Science of Free University of Bozen-Bolzano, Italy, and the Faculty of Informatics of Vienna University of Technology, Austria. Currently, he is a researcher at VRVis research center and a PhD student at the Institute of Computer Graphics and Algorithms of Vienna University of Technology. In his research, he focuses on the issues of visual analytics for action planning in the presence of uncertainty.

Günter Blöschl is Head of the Institute of Hydraulic Engineering and Water Resources Management, Director of the Centre for Water Resource Systems of the Vienna University of Technology as well as Chair of the Vienna Doctoral Program on Water Resource Systems. He has received numerous honors during his career including election as a Fellow of the American Geophysical Union, the German Academy of Science and Engineering (Acatech), and the International Water Academy. Recently, he was awarded the Advanced Grant of the European Research Council (ERC). He sits on the Scientific Advisory Council of numerous institutions including the German Federal Institute of Hydrology (BfG), the GeoForschungsZentrum (GFZ) Potsdam, and the NFP61 of the Swiss National Science Foundation. He has been the chair of the Predictions in Ungauged Basins (PUB) initiative of the International Association Hydrological Sciences. From 2013 to 2015, he was the President of the European Geosciences Union.