Efficient Shallow Water Simulations on GPUs: Implementation, Visualization, Verification, and Validation

André R. Brodtkorb[a,*], Martin L. Sætra[b], Mustafa Altinakar[c]

[a] SINTEF ICT, Department of Applied Mathematics, P.O. Box 124, Blindern, NO-0314 Oslo, Norway.
[b] Center of Mathematics for Applications, University of Oslo, P.O. Box 1053 Blindern, NO-0316 Oslo, Norway.
[c] National Center for Computational Hydroscience and Engineering, University of Mississippi, Carrier Hall, Room 102 University, MS 38677.

Abstract

In this paper, we present an efficient implementation of a state-of-the-art high-resolution explicit scheme for the shallow water equations on graphics processing units. The selected scheme is well-balanced, supports dry states, and is particularly suitable for implementation on graphics processing units. We verify and validate our implementation, and show that use of efficient single precision hardware is sufficiently accurate for real-world simulations. Our framework further supports real-time visualization with both photorealistic and non-photorealistic display of the physical quantities. We present performance results showing that we can accurately simulate the first 4000 seconds of the Malpasset dam break case in 27 seconds using over 480000 cells ($dx = dy = 15\ m$), in which our simulator runs at an average of 530 megacells per second.

1. Introduction

We present a state-of-the-art implementation of a second-order explicit finite-volume scheme for the shallow water equations with bed slope and bed shear stress friction terms. Our implementation is verified against analytical data, validated against experimental data, and we show extensive performance benchmarks. We start by giving an introduction to graphics processing units (GPUs), the shallow water equations, and the use of GPUs for simulation of conservation laws in this section. We continue in Section 2 by presenting the mathematical model and discretization of the selected scheme. In Section 3 we present our implementation, followed by numerical and performance experiments in Section 4, and we give our concluding remarks in Section 5.

1.1. Graphics Processing Units

Research on using GPUs for scientific computing started over a decade ago with simple academic tests that demonstrated the use of GPUs for non-graphics applications. Since then, the hardware has evolved from simply accelerating a set of predefined graphics functions for games to being used in the worlds fastest supercomputers [35]. The Chinese Nebulae supercomputer, for example, uses Tesla GPUs from NVIDIA. These cards offer over 500 gigaflops in double precision performance, twice that in single precision, and up-to 6 GB RAM accessible at 148 GB/s. Compared to the state-of-the-art six core Intel Core i7-980X, this is roughly six times the performance and bandwidth. In addition to having a higher peak performance, these GPUs also offer fast, albeit less accurate, versions of trigonometric and other functions, that can be exploited for even higher performance. Researchers have efficiently exploited these GPU features to accelerate a wide range of algorithms, and speedups of 5-50 times over equivalent CPU implementations have been reported (see e.g., [6, 30]).

Today, most scientific articles utilizing GPUs use cards from NVIDIA and program them using CUDA, a C-based programming language for parallel execution on GPUs. CUDA has been widely used by both industrial and academic

*Corresponding author
Email addresses: Andre.Brodtkorb@sintef.no (André R. Brodtkorb), m.l.satra@cma.uio.no (Martin L. Sætra), altinakar@ncche.olemiss.edu (Mustafa Altinakar)

groups, and over 1100 applications and papers in a wide range of fields have been added to the NVIDIA CUDA Showcase [27] since 2007. Browsing through the CUDA Showcase, it quickly becomes apparent that there is a race to report the largest attained speedup. At the time of writing of the present article, there were 465 papers reporting speedups. Out of these 115 claimed a speedup of 100 or more, 18 claimed a speedup of 500 or more, and one reported a stunning speedup of 2600. With a theoretical performance gap on the order of six times for the fastest available hardware today, we believe many of these claims to be misleading, at best. In fact, Lee et al. [22] support this view in a recent comparison of 14 algorithmic kernels, and report an average speedup of only 2.5 times. Whilst one can argue that a paper featuring Intel-engineers only might favor Intel CPUs, and comparison of a mid 2008 GPU with a late 2009 CPU can be rightfully criticized, their benchmarks clearly illustrate that a 100 times speedup is not automatically achieved by simply moving an algorithm to the GPU. We believe that fair comparisons between same-generation hardware of the same performance class, with reporting of the percent of peak performance, would be considerably more meaningful and useful than escalating the speedup race. Such comparisons make it easier to compare efficiency across algorithms and for different hardware.

Both the strengths and the weaknesses of GPUs come from their architectural differences from CPUs. While CPUs are traditionally optimized for single-thread performance using complex logic for instruction level parallelism and high clock frequencies, GPUs are optimized for net throughput. Today's GPUs from NVIDIA, exemplified by the GeForce GTX 480, consist of up-to 15 SIMD cores called streaming multiprocessors (SM). Each SM holds 2×16 arithmetic-logic units that execute 2×32 threads (two *warps*) in SIMD-fashion over two clock cycles. The SMs can keep multiple warps active simultaneously, and instantly switch between these to hide memory and other latencies. One SM can hold a total of 48 active warps, meaning we can have over 23 thousand hardware threads in flight at the same time [28]. This highly contrasts with the relatively low value of 48 operations for current CPUs (6 cores $\times$ 4-way SIMD $\times$ 2 hardware threads). Thus, the programming model of GPUs is very different from that of CPUs, and to quote Bo Kågström, we especially need to take the algorithm and architecture interaction into account for efficient hardware utilization. For a detailed overview of GPU hardware and software, we refer the reader to [6, 30].

1.2. The Shallow Water Equations

The shallow water equations describe gravity-induced motion of a fluid with free surface, and can model physical phenomena such as tidal waves, tsunamis, river flows, dam breaks, and inundation. The system is a set of hyperbolic partial differential equations, derived from the depth-averaged Navier-Stokes equations. As such, solutions of the shallow water equations are only valid for problems where the vertical velocity of the fluid is negligible compared to the horizontal velocities. Luckily, this criteria applies to many situations in hydrology and fluid dynamics, where the shallow water equations are widely used. For hyperbolic equations in general, the domain of dependence is always a bounded set. For the shallow water equations, this means we can solve the system using an explicit scheme, since the waves travel at a finite speed. Furthermore, the intrinsic parallelism of explicit schemes makes them particularly well suited for implementation on modern GPUs.

1.3. Conservation Laws on Graphics Processing Units

Using GPUs for the numerical simulation of conservation laws is not a new idea. In fact, the use of GPUs for such simulations was introduced at least as early as 2005, when one had to map the computations to operations on graphical primitives [14]. Since then, there have been multiple publications regarding conservation and balance laws on GPUs [13, 15, 4, 5, 18, 38, 3, 1].

Several authors have published implementations of explicit schemes for the shallow water equations. Hagen et al. [14] implemented multiple explicit schemes on the GPU using OpenGL, including a high-resolution second-order finite volume scheme very similar to the one we examine here, and demonstrated a 15-30 times speedup over an equivalently tuned CPU implementation on same-generation hardware. Liang et al. [25] present a second-order MacCormack scheme including friction described by the Manning equation implemented using OpenGL. They report a speedup of two for their whole algorithm on a laptop, and up-to 37000 for one of their kernels compared to the CPU. They also visualize their results, by first copying the data to the CPU, and then transferring it back to the GPU again. Lastra et al. [21] implemented a first-order, finite-volume scheme using the graphics languages OpenGL and Cg, presenting over 200 times speedup for a 2008 CPU versus a 2004 GPU. de la Asunción et al. [9] explore the same scheme

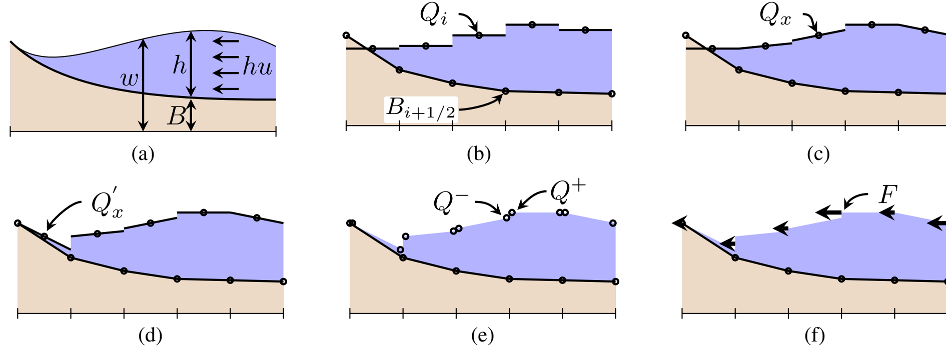


Figure 1: (a) Shallow water flow over a complex bottom topography and definition of variables; (b) conserved variables Q are discretized as cell averages and the bottom topography, B, is represented as a piecewise bilinear function in each cell based on its values at the grid cell intersections; (c) reconstruction of free-surface slopes of each cell using generalized minmod flux limiter; (d) modification of free-surface slopes with wet-dry contact to avoid negative water depth; (e) reconstructed values of the conserved variables on each side of the cell interfaces; and (f) fluxes computed at each cell interface using the central-upwind flux function [19].

using CUDA, and show a speedup of 5.7 in double precision on same-generation hardware. They report a 43% utilization of peak bandwidth and 13% of peak compute throughput for the whole algorithm. They further compared single versus double precision, reporting numbers indicating that single precision calculations are not sufficiently accurate. The same authors have reported similar findings [8] after extending their implementation to support also two-layer shallow water flows. Brodtkorb et al. [7] implemented three second-order accurate schemes on the GPU, comparing single versus double precision, where single precision was found sufficiently accurate for the implemented schemes. They further report an 80% instruction throughput for computation of the numerical fluxes, which is the most time consuming part of the implementation.

1.4. Paper Contribution

We build on the experiences of Brodtkorb et al. [7], and present a completely new state-of-the-art implementation of the Kurganov-Petrova scheme [20]. We have chosen this scheme for several reasons: it is well-balanced, conservative, second order accurate in space, supports dry zones, and is particularly well suited for implementation on GPUs. It has furthermore been shown that single precision arithmetic is sufficiently accurate for this scheme, enabling the use of efficient single precision hardware. Novelties in this paper include: per-block early exit optimization; verification and validation; a 31% decrease in memory footprint; semi-implicit physical friction terms; first order temporal time integration; multiple boundary conditions; simultaneous visualization with multiple visualization techniques; and extensive performance benchmarks.

2. Mathematical Model

The shallow water equations in two dimensions with bed slope and bed shear stress friction terms can be written

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}. \tag{1}$$

Here h is the water depth, and hu and hv are the discharges along the abscissa and ordinate, respectively. Furthermore, g is the gravitational constant, B is the bottom topography measured from a given datum, and C_z is the Chézy friction

coefficient (see also Figure 1a). We employ Manning's roughness coefficient, n, in our scheme using the relation $C_z = h^{1/6}/n$. In vector form, we write the system of equations as

$$Q_t + F(Q) + G(Q) = H_B(Q, \nabla B) + H_f(Q), \tag{2}$$

where Q is the vector of conserved variables, F and G represent fluxes along the abscissa and ordinate, respectively, and H_B and H_f are the bed slope and bed shear stress source terms, respectively.

The scheme we consider here is well-balanced, which means that for $u = v = 0$ the free-surface elevation, w, measured with respect to the same datum as B, remains constant. This requires that the numerical fluxes $F + G$ perfectly balance the bed slope source term H_B. Developing a well-balanced scheme is typically difficult when using the physical variables $[h, hu, hv]^T$, but achievable by switching to the set of derived variables $[w, hu, hv]^T$ (see Figure 1a). Thus for the remainder of this article, we use $Q = [w, hu, hv]^T$, in which (1) is rewritten using $h = w - B$ (see [20] for a detailed derivation).

2.1. Spatial Discretization

The spatial discretization of the Kurganov-Petrova scheme is based on a staggered grid, where Q is given as cell averages, B is given as a piecewise bilinear surface defined by the values at the four cell corners, and fluxes are calculated at integration points at the midpoint of each grid cell interface (see also Figure 4). The spatial discretization can then be written

$$\begin{aligned} \frac{dQ_{ij}}{dt} &= H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) - \big[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\big] - \big[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\big] \\ &= H_f(Q_{ij}) + R(Q)_{ij}. \end{aligned} \tag{3}$$

To compute the fluxes F and G across the cell interfaces, we perform the computations outlined in Figure 1. From the cell averages in Figure 1b we reconstruct a piecewise planar surface for Q in each cell (Figure 1c). A problem with this reconstruction is that we will typically end up with negative values for h at the integration points near dry zones. Without addressing these negative values, our scheme will not handle dry zones, as the eigenvalues of the system are $u \pm \sqrt{gh}$. In the scheme, the slope reconstruction is initially performed using the generalized minmod flux limiter,

$$Q_x = \text{MM}(\theta f, c, \theta b), \qquad \text{MM}(a, b, c) = \begin{cases} \min(a, b, c), & \{a, b, c\} > 0 \\ \max(a, b, c), & \{a, b, c\} < 0 \\ 0, & \end{cases} \tag{4}$$

where $\theta = 1.3$ and f, c, and b are the forward, central, and backward difference approximations to the derivative, respectively. This reconstruction, however, does not guarantee non-negativeness of h. To handle dry zones, Kurganov and Petrova propose to simply alter the slope of w so that the value of h at the integration points becomes non-negative (compare Figure 1c with Figure 1d). Because we have a bilinear bottom topography and planar water elevation, we can guarantee non-negativeness for four integration points per cell when the cell average is non-negative. This, however, limits the spatial reconstruction to second-order accuracy.

After having altered the slopes, we reconstruct point values for each cell intersection from the two adjacent cells (Q^+ and Q^- in Figure 1e), and compute the flux shown in Figure 1f using the central-upwind flux function [19]. A difficulty with these computations related to dry zones is that we need to calculate $u = hu/h$. This calculation leads to large round-off errors as h approaches zero, which in turn can lead to very large velocities and even instabilities. Furthermore, as the timestep is directly proportional to the maximal velocity in the domain, this severely affects the propagation of the solution. To avoid these large velocities, Kurganov and Petrova *desingularize* the calculation of u for *shoal* zones ($h < \kappa$) using

$$u = \frac{\sqrt{2}h(hu)}{\sqrt{h^4 + max(h^4, \kappa)}}. \tag{5}$$

This has the effect of dampening the velocities as the water depth approaches zero, making the scheme well-behaved even for shoal zones. Determining the value of κ, however, is difficult. Using too large a value yields large errors in the results, and setting it too low gives very small timesteps, Kurganov and Petrova used $\kappa = \max\{\Delta x^4, \Delta y^4\}$ in

their experiments. This approach is insufficient for many real-world applications, such as the Malpasset dam break case (used in Section 4) where $\Delta x = \Delta y = 15\ m$. Thus, we suggest using

$$\kappa = K_0 \max\{1, \min\{\Delta x, \Delta y\}\} \tag{6}$$

as an initial guess, with $K_0 = 10^{-2}$ for single precision calculations, and a smaller constant for double precision. This gives a linear proportionality to the grid resolution, which we find more suitable: for example, for a 15 meter grid cell size our approach desingularizes the fluxes for water depths less than 15 cm. Numerical round off errors may also make very small water depths negative, independent of the value of κ. We handle this by setting water depths less than ϵ_m to zero, where ϵ_m is close to machine precision.

To be well-balanced and capture the lake at rest case, the bed slope source term needs to be discretized carefully to match the flux discretization. However, due to the alteration of the slopes of w to guarantee non-negativeness at the integration points, we get nonphysical fluxes emerging from the shores (see Figure 1c and 1d). These fluxes, however, are small, and have minimal effect on the global solution for typical problem setups.

2.2. Temporal Discretization

In (3) we have an ordinary differential equation for each cell in the domain. Disregarding the bed shear stress for the time being, we discretize this equation using a standard second-order total variation diminishing Runge-Kutta scheme [33],

$$\begin{aligned} Q^*_{ij} &= Q^n_{ij} + \Delta t R(Q^n)_{ij} \\ Q^{n+1}_{ij} &= \tfrac{1}{2} Q^n_{ij} + \tfrac{1}{2}\left[Q^*_{ij} + \Delta t R(Q^*)_{ij}\right], \end{aligned} \tag{7}$$

where the timestep, Δt, is restricted by a CFL condition that ensures that disturbances travel at most one half grid cell per time step,

$$\Delta t \leq \tfrac{1}{2}\min\Big\{\Delta x/\max_\Omega\big|u \pm \sqrt{gh}\big|, \Delta y/\max_\Omega\big|v \pm \sqrt{gh}\big|\Big\} = \frac{r}{2}. \tag{8}$$

To include the bed shear stress in (7) we use a semi-implicit discretization,

$$\begin{aligned} H_f(Q^*_{ij}) &\approx Q^*_{ij}\tilde{H}_f(Q^n_{ij}), \\ H_f(Q^{n+1}_{ij}) &\approx Q^{n+1}_{ij}\tilde{H}_f(Q^*_{ij}), \end{aligned} \qquad \Bigg| \qquad \tilde{H}_f(Q^k_{ij}) = \begin{bmatrix} 0 \\ -g\sqrt{{u^k_{ij}}^2 + {v^k_{ij}}^2}/h^k_{ij}C_{z\,ij}^2 \\ -g\sqrt{{u^k_{ij}}^2 + {v^k_{ij}}^2}/h^k_{ij}C_{z\,ij}^2 \end{bmatrix} \tag{9}$$

where $\tilde{H}_f(Q^k_{ij})$ is computed explicitly from Q at timestep k. Adding (9) to (7) and reordering, we get

$$\begin{aligned} Q^*_{ij} &= \big[Q^n_{ij} + \Delta t R(Q^n)_{ij}\big]/\big[1 + \Delta t\tilde{H}_f(Q^n_{ij})\big] \\ Q^{n+1}_{ij} &= \Big[\tfrac{1}{2}Q^n_{ij} + \tfrac{1}{2}\big[Q^*_{ij} + \Delta t R(Q^*)_{ij}\big]\Big]/\big[1 + \tfrac{1}{2}\Delta t\tilde{H}_f(Q^*_{ij})\big], \end{aligned} \tag{10}$$

where all terms on the right hand side are explicitly calculated. We can also use a first order accurate Euler scheme, which simply amounts to setting $Q^{n+1}_{ij} = Q^*_{ij}$ in (10).

3. Implementation

We have implemented a cross-platform compatible simulator with visualization using C++, OpenGL [32], and NVIDIA CUDA [28]. The implementation consists of three parts: a C++ interface and CPU code, a set of CUDA kernels that solve the numerical scheme on the GPU, and a visualizer that uses OpenGL to interactively show results from the simulation. The simulation is run solely on the GPU, and data is only transferred to the CPU for file output.

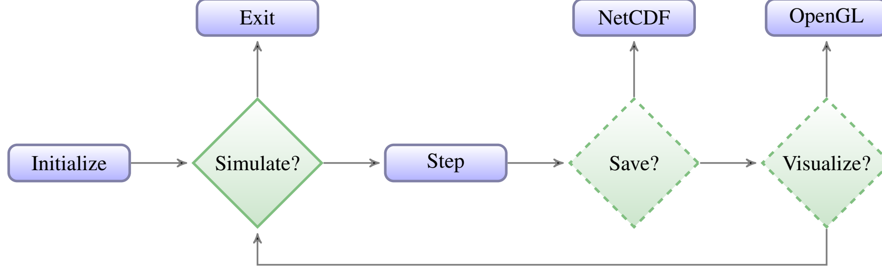


Figure 2: The program flow of our applications. Initialize sets up the simulator class with initial conditions, and step runs one timestep on the GPU. We can also save results in netCDF files and visualize them directly using OpenGL.

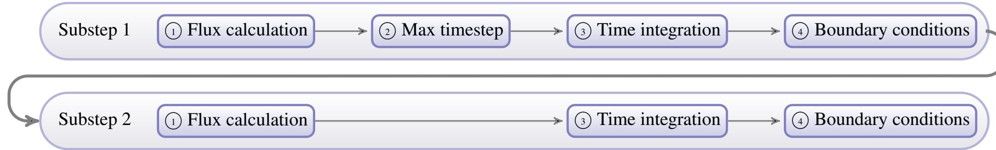


Figure 3: Zoom of the step function in Figure 2 that shows our CUDA kernels. In the first substep, ① calculates R, ② finds the maximum Δt, ③ calculates $\tilde{H}_f$ and Q^*, and ④ enforces boundary conditions on Q^*.

3.1. C++ interface and CPU code

Our C++ interface consists of a relatively simple class that handles data allocation, initialization and deallocation; movement of data between the CPU and the GPU; and invoking CUDA kernels on the GPU. The API for the simulator is easy and clean, and it is possible to set up and run a simulation in about 5–10 lines of code without knowledge of implementation details. We have implemented several applications that use this interface, including one that writes results to netCDF [26] files, an open and standardized file format commonly used to store geophysical data, and one that runs real-time visualization of the simulation using OpenGL. Our applications further supports continuing simulations stored in netCDF files. This is done by initializing the simulator with the bottom topography and the physical variables for the last timestep in the file.

Figure 2 shows the program flow for our applications. The applications start by initializing the simulator class, and continues by entering a loop where we perform timesteps. Because each timestep is variable, we do not know, a priori, how many timesteps we need to perform per unit simulation time. Thus, to save or visualize the results at regular simulation time intervals, we check the current simulation time after each timestep and only exit the simulation loop when the desired simulation time has been reached.

Our C++ class allocates the GPU data required to pass information between kernels. Brodtkorb et al. [7] stored approximately 16 values per grid cell: two for B, three for Q, three for Q^*, three for F, three for G, and two for the non-zero source terms. B was represented at the grid intersections and also at the center of cells as a performance optimization. In our approach, however, we have reduced this to only 11 values per grid cell by combining F, G, and the two non-zero source terms into three values for R. This reduces the number of values we have to transfer to and from global memory dramatically, and reduces the memory footprint by approximately 31%. However, this complicates the implementation of the flux calculation, as will be detailed later. We have used single precision in our kernels, as Brodtkorb et al. [7] have previously shown that this is sufficiently accurate. Using single precision over double has several benefits: data transfers and arithmetic operations can execute at least twice as fast, and all data storage in registers, shared and global memory, takes half the space.

3.2. CUDA Kernels

Figure 3 shows how our numerical scheme is computed by executing seven CUDA kernels in order. One full time-step with the second-order accurate Runge-Kutta scheme runs through both the first and second substep, whilst running only the first substep reduces to the first order accurate Euler scheme. Thus, for second-order accuracy, we

perform around twice the number of calculations. Within each substep we ① calculate the fluxes and bed slope source terms, ② find the maximum timestep, ③ compute bed shear stress source terms and evolve the solution in time, and ④ apply boundary conditions. Inbetween these kernels, we store results in global memory and implicitly perform global synchronization. It would be more efficient if we could perform the full scheme using only a single kernel, however, this is not possible: applying boundary conditions requires the evolved timestep; evolving the solution in time requires the fluxes and maximum timestep; and computing the maximum timestep requires the eigenvalues for all cells that are computed by the flux kernel. One could combine the boundary conditions kernel with either the time integration or the flux kernel, yet this would involve additional branching that quickly outweigh the performance gain from launching one less kernel.

Modern GPUs from NVIDIA consist of a set of SIMD processors that execute *blocks* in parallel. Each block consists of a predefined number of threads, logically organized in a three-dimensional grid. Threads in the same block can cooperate and share data using on-chip *shared memory*. For high performance we need to have a good block partitioning, and choosing a "wrong" block configuration will severely affect kernel performance. However, there are several conflicting criteria for choosing the optimal block size. The optimal block size also varies from one GPU to another, especially between major hardware generations. In the case of the GPU used for the present work, i.e., the NVIDIA GeForce GTX 480, the following factors were observed to affect the performance:

Warp size The hardware schedules the same instruction to 32 threads, referred to as a *warp*, in SIMD fashion. Thus, for optimal performance, we want the number of threads in our block to be a multiple of 32.

Shared memory access Shared memory is organized into 32 *banks*, which collectively service one warp every other clock cycle. If two threads access the same bank simultaneously, however, the transaction takes twice as long. Thus, the width of our shared memory should be a multiple of 33 to avoid bank conflicts both horizontally and vertically.

Shared memory size Keeping frequently used data in shared memory will typically yield performance gains. Thus, we want to maximize the domain kept in shared memory. We also want to keep it as square as possible to maximize the ratio of internal cells to local ghost cells required by the computation (see Figure 4).

Number of warps per streaming multiprocessor Each multiprocessor can keep up-to 48 active warps simultaneously, as long as there are available hardware resources (such as shared memory and registers). Increasing the number of active warps increases the occupancy, which is a measure of how well the streaming multiprocessor can hide memory latencies. The processor uses this occupancy by instantaneously switching to other warps when the current warp stalls, e.g., due to a data dependency. Thus, we want as many warps per multiprocessor as possible.

Global memory access Global memory is moved into the processor in bulks, reminiscent of the way CPUs transfer full cache lines. For maximum performance, the reads from global memory should be 128 byte sequential, and the start address should be aligned on a 128 byte border (called coalesced accesses). This means that our block width should make sure that global reads start at properly aligned addresses. If we violate these rules, however, we can still benefit from the caching on CUDA compute 2.0 capable cards from NVIDIA.

Flux Calculation. The flux kernel is the computationally most expensive kernel in our implementation. Its main task is to compute R_{ij} from (3). This is done by computing the flux across all the cell interfaces, the bed slope source term for all cells, and summing to find the net contribution to each cell. This contrasts the approach of Brodtkorb et al. [7], where the calculation of the net contribution per cell was done in the time integration kernel. Comparing the two, our new approach stores fewer floating-point values per cell, enabling us to reduce the memory footprint on the GPU.

Before we launch our kernel, we start by performing domain decomposition, yielding high levels of parallelism suitable for the execution model of GPUs. Figure 4 illustrates how our global domain is partitioned into blocks that can be calculated independently by using local ghost cells. We use a block configuration of $16 \times 12 = 192$ threads where we have one thread per cell. This size is a compromise between the above mentioned optimization guidelines that has yielded the best performance. Our block size is a multiple of 32 which fits with the warp size. Our shared memory size uses almost all of the 16 KB available, and it is relatively square (20×16 including local ghost cells). It does not, however, ensure that we have no bank conflicts. Making the width 32, thus 28 threads wide block size, would solve

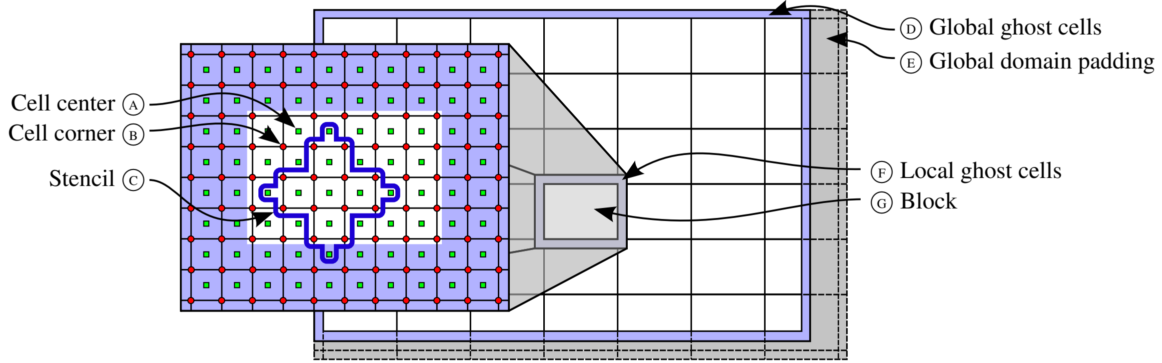


Figure 4: Domain decomposition and variable locations. The global domain is padded Ⓔ to fit an integer number of blocks. Each block Ⓖ has local ghost cells Ⓕ that overlap with other blocks to satisfy the data dependencies dictated by the stencil Ⓒ. Our data variables Q, R, H_B, and H_f are given at grid cell centers Ⓐ, and B is given at grid cell corners Ⓑ.

this at the expense of other optimization parameters such as warp size. For our flux kernel, we also have the option of configuring the shared memory as 48 KB L1 cache and 16 KB shared memory, or 16 KB L1 cache and 48 KB shared memory. We have found that for our kernel, using 48 KB L1 shared memory yielded the best performance. However, we only use 16 KB per block, meaning we can fit several blocks per streaming multiprocessor. Through experimentation, we have found that using three blocks, corresponding to 18 warps, yielded the best performance. Using too many warps exhausts the register file, meaning that some registers are spilled to local memory. Using too few warps may cause the processor not to have enough warps to select from when some are stalled by latencies. Finally, due to the local ghost cells of our blocks, we are unable to fulfill coalescing rules. Fortunately, however, we do benefit from the 16 KB L1 cache, and 768 KB L2 cache. On older graphics cards, one could use the texture cache for better performance of uncoalesced reads. However, using the L1 and L2 cache on compute 2.0 capable cards is superior in performance to using the texture cache [29].

When launching our kernel, we start by reading from global memory into on-chip shared memory. In addition to the interior cells of our block, we need to use data from two neighbouring cells in each direction to fulfill the data dependencies of the stencil. After having read data into shared memory, we proceed by computing the one dimensional fluxes in the x and y directions, respectively. Using the steps illustrated in Figure 1, fluxes are computed by storing all values that are used by more than one thread in shared memory. We also perform calculations collectively within one block to avoid duplicate computations. However, because we compute the net contribution for each cell, we have to perform more reconstructions and flux calculations than the number of threads, complicating our kernel. This is solved in our code by designating a single *warp* that performs the additional computations; a strategy that yielded a better performance than dividing the additional computations between several warps. This comes at the expense of a more complex flux kernel, but it greatly simplifies, and increases the performance, of the time integration kernel.

The first of our calculations is to reconstruct the value of B at each interface midpoint, so that we have the value of B properly aligned with Q. This calculation is performed every timestep as a performance trade-off; in order to reduce memory and bandwidth usage. We continue by reconstructing the slopes of Q using the branchless generalized minmod slope limiter [13]. The slopes of w are then adjusted to guarantee non-negative values at the integration points, according to the scheme. With the slopes calculated we can reconstruct point values at the cell interfaces, Q^+ and Q^-, and from these values compute the fluxes. We also compute the bed slope source term, sum the contributions for each cell, and write the results to global memory.

The flux kernel is also responsible for computing r in (8) for each integration point, where the global minimum is used to calculate the maximum timestep. A problem with calculating r is that for zero water depths, we get division by zero. We solve this in our code by setting $r = \min\{\Delta x / \max(\epsilon_m, u \pm \sqrt{gh}), \Delta y / \max(\epsilon_m, v \pm \sqrt{gh})\}$, where ϵ_m is close to machine epsilon. Furthermore, instead of storing one value per integration point, we perform efficient shared memory reduction to find the minimum r in the whole block at very little extra cost. This reduces the number

of values we need to store in global memory by a factor 192, yielding a two-fold benefit: we transfer far less data, and the maximum timestep kernel needs to consider only one value per block.

Maximum timestep. The maximum timestep kernel is a simple reduction kernel that computes the maximum timestep based on the minimum r for each block. Finding the global minimum is done in a similar fashion to the reduction example supplied with the NVIDIA GPU Computing SDK. We use a single block where thread t_{no} *strides* through the dataset, considering elements $t_{\text{no}} + k \cdot n$, where n is the number of threads. This striding ensures fully coalesced memory reads for maximum bandwidth utilization. Once we have considered all values in global memory, we are left with n values that we reduce using shared memory reduction, and we compute the timestep as $\Delta t = 0.5r$. Also here, the block size has a large performance impact. Using fewer threads than the number of elements gives us a sub-optimal occupancy, and using too many threads launches warps where not a single thread is useful. Thus, to launch a suitable number of threads for varying domain sizes we use template arguments to create multiple realizations of the reduction kernel: we generate one kernel for $1, 2, 4, \dots, 512$ threads, and select the most suitable realization at runtime.

Time integration. The time integration kernel performs a domain partitioning similarly to the flux kernel, but we here use a block size of $32 \times 16 = 512$ threads as we are not limited by shared memory: our computations are embarrassingly parallel, and we do not require any local ghost cells. We can thus achieve full coalescing for maximum bandwidth utilization. This is also an effect of storing only the net contribution, R for each cell, as opposed to storing both the fluxes and non-zero source terms. The kernel starts by reading Q and R into per-thread registers, and then computes the bed shear stress source term, $\tilde{H}_f$. The timestep, Δt, is also read into registers, and we evolve the solution in time.

Boundary conditions. We have selected to implement boundary conditions using global ghost cells. As our scheme is second-order accurate, we need two global ghost cells in each direction that are set for each substep by the boundary conditions kernel. This makes our flux kernel oblivious to boundary conditions, which means we do not have to handle boundaries differently from cells in the interior of our domain. We have implemented four types of boundary conditions: wall, fixed discharge (e.g., inlet discharge), fixed depth, and free outlet. To implement these conditions, we need to fix both the cell averages and the reconstructed point values, Q^+ and Q^-. To do this, we utilize an intrinsic property of the minmod reconstruction. Recall that this reconstruction uses the forward, backward, and central difference approximation to the derivative, and sets the slope to the least steep of the three, or zero if any of them have opposite signs. Thus, by ensuring that the least steep slope is zero, we can fix the reconstructed point values to any given value. Wall boundary conditions are implemented by mirroring the two cells nearest the boundary, and changing the sign of the normal discharge component. Fixed discharge is implemented similarly, but the normal discharge is set to a fixed value, and fixed depth boundaries set the depth instead of the discharge to the requested value. Finally, free outlet boundaries are implemented by copying the cell nearest the boundary to both ghost cells. It should be noted that our free outlet boundary conditions give rise to small reflections in the domain, as is typical for this kind of implementation (see e.g., [23]). We have further developed our fixed discharge and fixed depth boundary conditions to handle time-varying data. By supplying a hydrograph in the form of time-discharge or time-depth pairs, our simulator performs linear interpolation between points and sets the boundary condition accordingly. These types of boundary conditions can easily be used to perform, e.g., tidal wave, storm surge or tsunami experiments.

The boundary conditions are applied to the four global boundaries in a single kernel call in a very efficient fashion. At compile-time, we generate one kernel for each combination of boundary conditions, which we can automatically select at run-time. We have ten different realizations to optimize for different domain sizes (similarly to the maximum timestep kernel), and five for the different boundary conditions for each of the edges. This totals to $10 \times 5^4 = 6250$ optimized kernel realizations, which exceeds what the linker is capable of handling. To circumvent this compiler issue, we reduce the number of realizations by disregarding optimization for domain sizes where the width and height are both less than 64 cells, leading to only 2500 kernel realizations. Disregarding optimizations for the small domain sizes is not a big loss, since using the GPU is most efficient for large domain sizes. It is also possible to completely skip boundary conditions, to gain performance. This can safely be done when the domain has dry boundaries.

A weakness in our current implementation is the lack of support for mixed boundary conditions within each boundary. We can implement mixed boundaries using two auxiliary buffers for each edge. The first buffer describes the type of boundary condition, e.g., using an integer, and the second holds a value to be used by the boundary

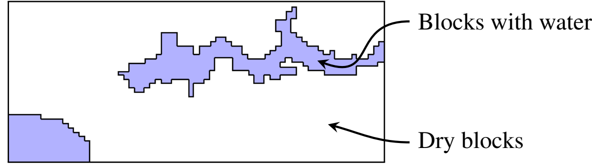


Figure 5: Auxiliary buffer used to store whether blocks contain water or not. The presence of one single wet cell in a block is sufficient to consider the whole block as wet.

condition (e.g., water depth for fixed depth boundaries). However, such an implementation would be quite complex, require more data accesses, and impose additional branches. Thus, it might be a difficult task to implement a very efficient mixed boundary conditions kernel. On the other hand, the ability to run several kernels simultaneously on compute 2.0 capable cards might enable a different approach, where multiple kernels running simultaneously each handles a part of the boundary conditions.

3.2.1. Early Exit Optimization

The flux and time integration kernels both divide the computational domain into blocks that can be computed independently. However, if a block does not contain water at all, we can simply skip all computations for that block. Unfortunately, we do not a priori know whether or not a block contains water. In our approach, we use the same block size for the time integration kernel and the flux calculations kernel and implement early exit on a per block basis. In the time integration kernel, we let each block perform shared memory reduction to find out if this block contains any water at all, and write to an auxiliary buffer (see Figure 5). In the flux kernel we then read from this buffer, and if the block and its four closest neighbouring blocks are dry, we can skip flux calculations all together. We need to check the neighbouring blocks due to our local ghost cells that overlap with these blocks. This optimization has a dramatic effect on the calculation time for domains with a lot of dry cells, as is typical for flooding applications. We have also developed this technique to mark cells where the source terms perfectly balance the fluxes ($R = 0$) as “dry”. This extends the early exit strategy to also optimize for wet blocks where the steady state property holds. However, there is a small penalty to pay for using early exit, as reading and writing the extra data and performing the shared memory reductions takes a small amount of time to complete. To cope with this problem, we therefore perform runtime analysis of the kernel execution time. Once the kernel execution time for the early exit kernel exceeds the execution time for the regular kernel, we swap over to using the regular kernel. In addition, we also perform a probe every hundredth timestep in order to use the most efficient implementation at all times. This approach can also automatically be used where one has sources and sinks in the interior of a domain by implementing sources and sinks as changes to R, thus violating the lake-at-rest property.

3.3. OpenGL Visualizer

We have implemented a direct visualization using OpenGL, which was originally presented in [7]. A new feature in this implementation is the use of a more efficient data path between CUDA and OpenGL, which is enabled by the new graphics interoperability provided by CUDA 3.0. We have also implemented a non-photorealistic visualization that provides a different view of the simulation results. The terrain is rendered by draping it with a user-selectable image (such as a satellite or orthophoto image) and light-set using a static normal map. Several options are available for rendering the water surface. The first method is the photorealistic rendering, where the Fresnel equations are employed to calculate reflection and refraction of rays hitting the water surface as shown in Figure 6. As an alternative, we also provide the possibility of viewing the data using a color transfer function, also shown in Figure 6. We use interpolation between colors in the HSV color space, and the figure shows the water depth where one full color cycle corresponds to $15\ m$ (one Δx in the simulation).

4. Experiments

We have performed several experiments with our implementation. We first present validation against a two-dimensional test problem where there is a known analytical solution. We then present verification against a real-world

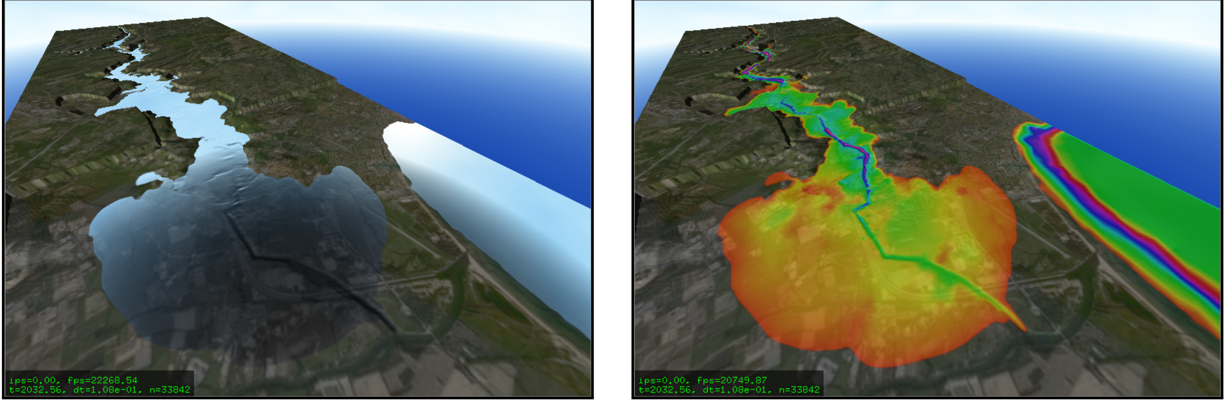


Figure 6: Two visualizations of the Malpasset dam break at $t = 2032.56$ seconds after the breach. We can visualize the results using the Fresnel equations, yielding a water surface with both reflection and refraction (left). We can also visualizes the physical variables using a color transfer function, here show for the water depth (right).

dam break, and finally we present performance experiments with several different datasets to show performance and scalability. Our benchmarks have been run on a machine with a 2.67 GHz quad core Intel Core i7 920 CPU with 6 GiB RAM. The graphics card is an NVIDIA GeForce GTX 480 with 1.5 GiB RAM in a PCI-express 2.0 $\times 16$ slot in the same system. In the results we present, we have compiled our CUDA source code with CUDA 3.0 using the compiler options

```
-arch=sm_20       //Generate compute 2.0 ptx code
-use_fast_math    //Use fast, but less accurate math functions
-ftz=true         //Flush denormal numbers to zero
-prec-div=false   //Use fast, but inaccurate division
-prec-sqrt=false  //Use fast, but inaccurate square root
-Xptxas -dlcm=ca  //Enable L1 and L2 caching of global memory
```

These options sacrifice precision for performance, enabling fast execution on the computational hardware. As will be evident from our verification and validation experiments, however, this does not hinder the models ability to capture analytical solutions and real-world flows.

4.1. Verification: Oscillations in a Parabolic Basin

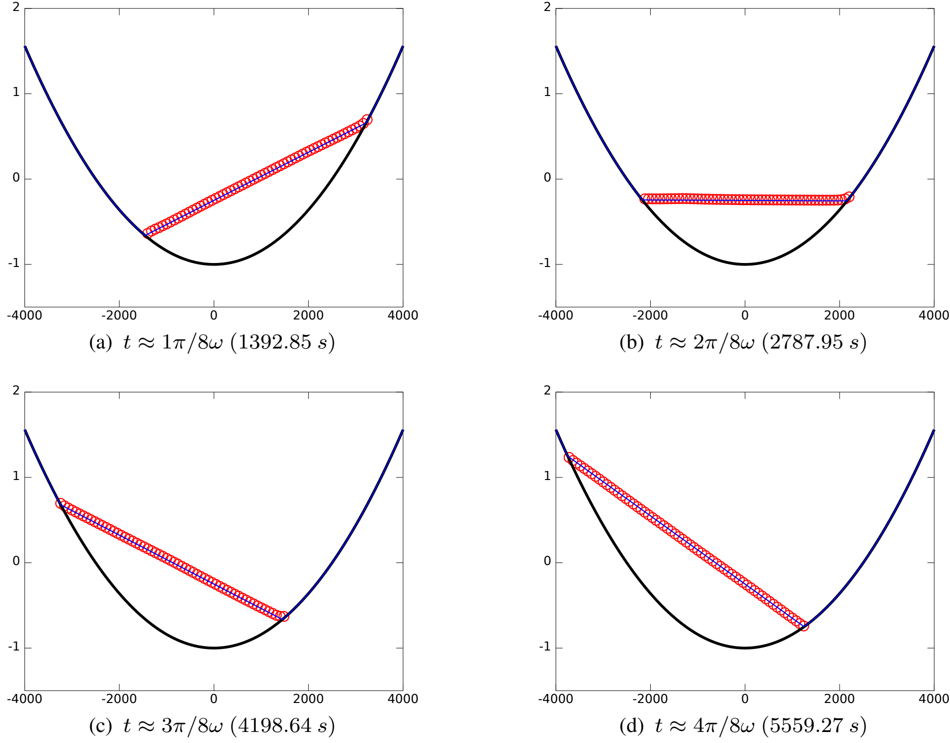
The analytical solution of time dependent motion in a friction-less parabolic basin described by Thacker [34] was chosen as the first test case for model verification. In this two-dimensional case, we have a parabolic basin where a planar water surface oscillates. More recently, Sampson et al. [31] extended the solutions of Thacker to include bed friction, however restricted to one dimension. These test problems have been used by several authors previously (see e.g., [24] and references in Sampson [31]) and, thus, can serve as a comparison with other numerical model results. In our test setup, which is identical to that presented by Holdahl et al. [17], we have the parabolic bottom topography given as

$$B(x,y) = D_0 \left[(x^2 + y^2)/L^2 - 1 \right]. \tag{11}$$

The water elevation and velocities at time t are defined as

$$\begin{aligned} w &= 2AD_0(x \cos \omega t \pm y \sin \omega t + LB_0)/L^2 \\ u &= -A\omega \sin \omega t \\ v &= \pm A\omega \cos \omega t. \end{aligned} \qquad \Bigg| \qquad \omega = \sqrt{2D_0/L^2} \tag{12}$$

We use the parameters $D_0 = 1$, $L = 2500$, $A = L/2$, and $B_0 = -A/2L$, set the manning coefficient $n = 0$, the gravitational constant $g = 1$, the desingularization epsilon $\kappa = 0.01$, and use 100×100 grid cells with the second-order accurate Runge-Kutta time integrator. Figure 7 shows our results for four snapshots in time compared to the



(a) $t \approx 1\pi/8\omega$ ($1392.85\ s$)
(b) $t \approx 2\pi/8\omega$ ($2787.95\ s$)
(c) $t \approx 3\pi/8\omega$ ($4198.64\ s$)
(d) $t \approx 4\pi/8\omega$ ($5559.27\ s$)

Figure 7: Simulation of oscillating water in a parabolic basin, compared to the analytical solution. The solid line is the analytical solution, and the circles are the computed values (only wet cells marked).

respective analytical solutions. The figure clearly indicates that our implementation captures the analytical solution well for the water surface elevation. For the velocities, however, we see that there is a growing error along the wet-dry boundary that eventually also affects the elevation. This error is difficult to avoid, and is found in many schemes (see e.g., [17]).

4.2. Validation: The Malpasset Dam Break

The Malpasset dam, completed in 1954, was a 66.5 meter high double curvature dam with a crest length of over 220 meters that impounded 55 million cubic meters of water in the reservoir. Located in a narrow gorge along the Reyran River Valley, it literally exploded after heavy rainfall in early December 1959. Over 420 casualties were reported due to the resulting flood wave. This dam break is a unique real-world case, in that there exists front arrival time and maximum water elevation data from the original event, in addition to detailed data from a scaled model experiment [12, 10]. However, due to large changes to the terrain as a result of the flood, the digital bottom topography had to be reconstructed from a 1:20000 map dated 1931. The original dataset contains a total of 257622 unstructured points, and our regular Cartesian grid, identical to that of Ying and Wang [39], consists of 1100×440 bottom topography values spaced equally by 15 meters. This results in 1099×439 cells with $\Delta x = \Delta y = 15\ m$. We set the Manning coefficient to $n = 0.033\ m^{1/3}s$, the desingularization epsilon, $\kappa = 40\ cm$, and simulate the first 4000 seconds after the breach using Euler time integration. The desingularization epsilon is set rather high compared to our suggested initial guess of $15\ cm$. This value was chosen through experimentation: setting it to a lower value increased the computational time, while higher values did not dramatically decrease this time. A sensitivity analysis of the parameter would be useful to further justify its setting, but this is outside the scope of the current work.

Figure 8 shows our simulation results compared with the experimental data from a 1:400 scaled model [10]. The largest discrepancy is for gauge 9 for the maximum water elevation, and gauge 14 for the arrival time. These large

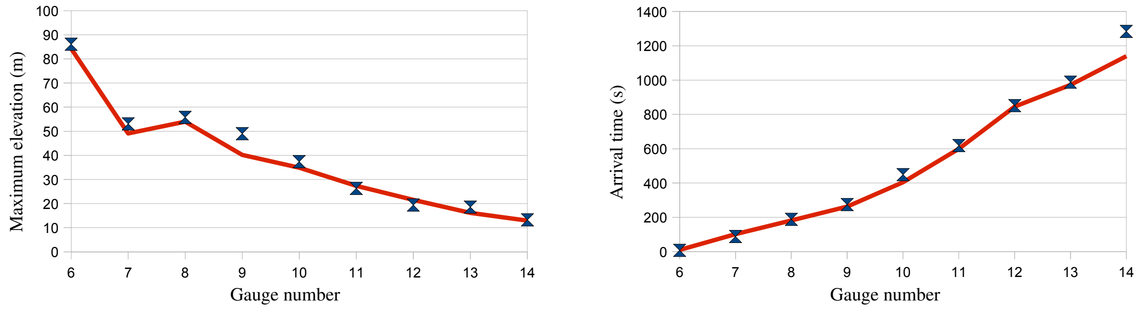


Figure 8: Validation against the maximum water elevation (left) and arrival time (right) from the Malpasset dam break case. The domain consists of 1099×439 cells with $\Delta x = \Delta y = 15$ meters. For both figures, the solid line represents the experimental data, and the symbols are the simulation results.

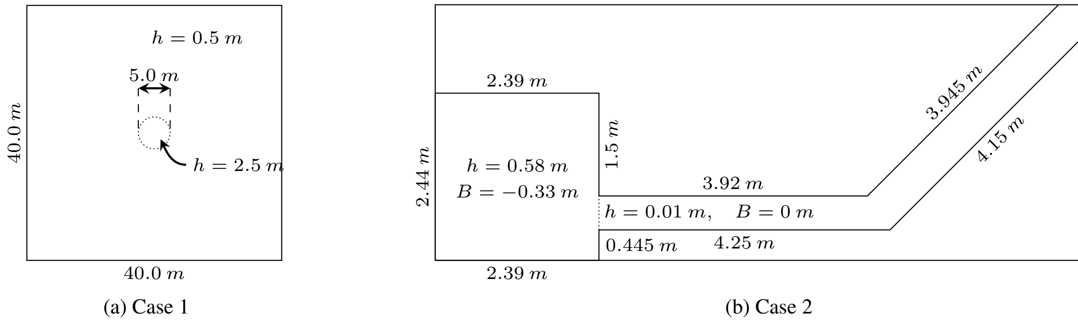


Figure 9: Dam break test cases used for performance benchmarks. The dotted lines indicate the location of the dams that are instantaneously removed at simulation start. In Case 2 the bottom topography of the area outside the reservoir and the canal is set to 2 m.

discrepancies are also found in other published results of this case, and our results compare well with these (see e.g., [2, 16, 37, 39])

4.3. Performance Experiments

To assess the performance of our simulator, we have benchmarked our code on a set of well known test cases. Our first case is an idealised circular dam break [36, 23], meaning that the dam collapses instantaneously. The dam is located at the center of a 40×40 metre domain as shown in Figure 9a, where we employ wall boundaries. The second case is a dam break through a 45° bend, as used in the CADAM project [11]. Experimental set up consists of a reservoir connected to a $\sim 0.5\ m$-wide rectangular channel with a horizontal bottom by a $0.33\ m$-high positive step. The channel makes a sharp 45° bend to the left about $4\ m$ downstream from the reservoir and is initially filled with water up to a depth of $0.01\ m$ (see Figure 9b). The south and west boundaries are set as wall boundaries, and the north and east boundaries are free outlet.

Figure 10 shows the absolute performance of our implementation in megacells (millions of cells processed) per second, where we generate both cases for a large number of different domain sizes. We run the simulator to four seconds simulation time for case 1, in which the wave is roughly three metres from the edge having reached 57% of the domain. For the second case, the percentage of the domain covered by wet cells is constant, and we run our simulator for thirty wall clock seconds. Figure 10 (left) clearly illustrates that small domains do not fully utilize GPU's processing capabilities. However, after five million cells, we see that the GPU has more or less reached the

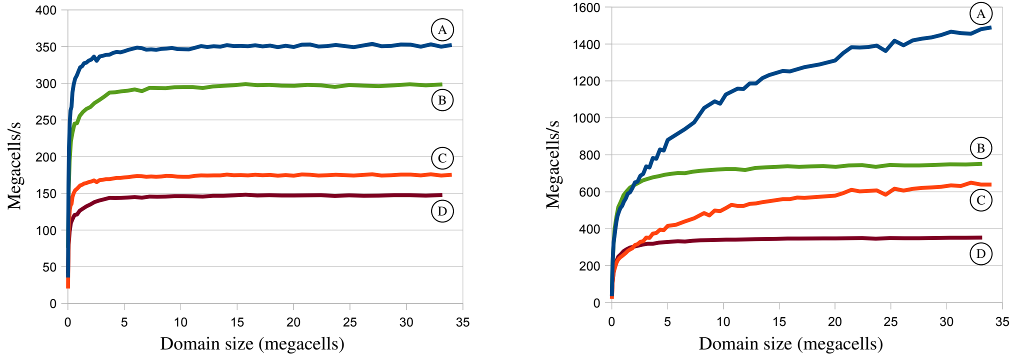


Figure 10: Absolute performance of our implementation as a function of domain size when calculating all cells (left), and when only calculating "wet" cells (right). Ⓐ and Ⓒ correspond to the dam break through 45° bend, and Ⓑ and Ⓓ correspond the to circular dam break case. Ⓐ and Ⓑ use first order accurate Euler time integration, and Ⓒ and Ⓓ use second order accurate Runge-Kutta time integration.

peak performance of up-to 350 million cells per second. For the circular dam break, five million cells corresponds to a domain sized roughly 2200×2200, which should give us almost 26000 blocks. The reason that we need this large number of blocks is not only to fully occupy the GPU hardware, but also in order to make overheads, such as launching kernels, negligible. The largest benchmark we have run is more than 33 million cells (5760×5760 cells for case 1), consuming over 1.4 GB of the 1.5 GB available RAM. We also see that there is a clear difference between the square and rectangular domains, which might seem odd. However, in our tests, we have discovered that this is not in fact due to the size of the domain, but rather connected to the ratio of wet to dry cells. The reason for the difference is that in the flux function, we do not compute the fluxes for dry cells. This is not related at all to the early exit optimization, but to the fact that we set the values at integration points with $h < \epsilon_m$ to zero. Examining Figure 9, we see that we have water in all cells in case 1, whereas we have large parts where we have no water at all for case 2. However, in the dry parts of the second case we still have to compute all of the reconstructions before we find out that the integration point in fact is dry. We can also see that there is a clear distinction between the first order and second order accurate time integrator schemes: running the second order accurate scheme takes twice as long as the first order scheme, as is expected. In Figure 10 (right) we have enabled the early exit optimization, where dry blocks do not even perform the reconstructions. Compared to Figure 10 (left), we immediately see the effect of this optimization for both cases. For case 1, our performance more than doubles by enabling the early exit optimization, and we soon reach a peak performance of 700 to 750 megacells per second for domain sizes larger than five megacells. For case 2, however, we see a much more rough curve, in addition to it rising to a much higher peak performance. This is because we have a fixed number of cells that are marked as "wet", independent of time since small waves emerging from the canal walls ensure that the canal is marked as wet within very few timesteps. Thus, by increasing the resolution, we also increase the relative number of blocks that perform early exit. This illustrates that our early exit strategy is highly suited for domains where there are large dry zones.

Figure 11 further shows the effect of the early exit optimization, where we show the absolute performance as a function of simulation time. This illustrates how the propagation of the solution affects the performance. When calculating the full domain, we see that the performance remains at around 300 megacells per second throughout the simulation. When early exit is enabled, we see that the initial performance is much higher, and gradually decreases as the solution progresses. This is because the waves from the dam break spread out, thus marking more and more blocks as "wet". For case 1 the early exit kernel eventually takes *more* time than the kernel that calculates all cells. However, as we always select the fastest kernel, we are not penalized at all for enabling early exit. For the Malpasset dam break case, on the other hand, we see that the early exit kernel always performs better. This is because the water here follows the valley, thus marking fewer cells as "wet" (see also Figure 5, which is an actual map of "wet" blocks for the Malpasset case). We also here see that for the Malpasset dam break case we reach less than half of the performance of case 1. This is largely due to the domain size which is insufficiently large to mask all overheads.

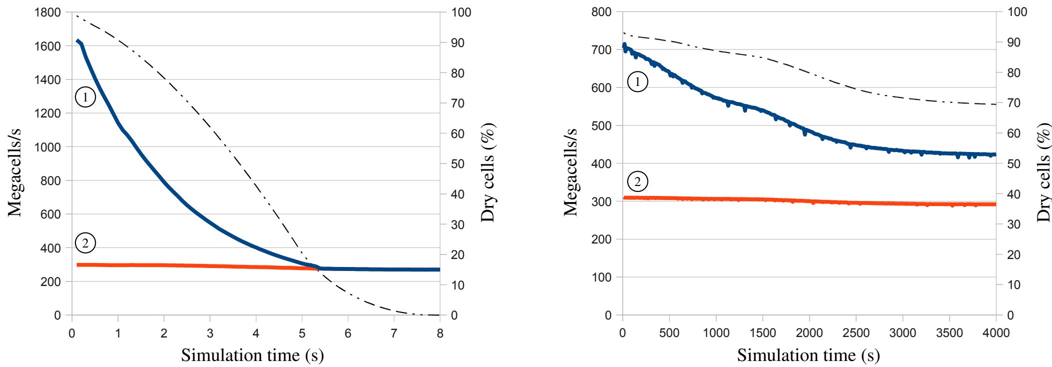


Figure 11: Absolute performance in megacells per second of our implementation for the circular dam break case (left) and the Malpasset dam break case (right). The domains contain 4000×4000 and 1099×439 cells respectively. The graphs show the estimated instantaneous performance as a function of simulation time: ① has enabled the early exit optimization, and ② calculates all cells. The dashed line shows the percentage of dry cells in the domain.

We have also profiled our code using the CUDA visual profiler, which gives a rough idea of our resource utilization. We run the circular dam break without the early exit optimization using the Runge-Kutta time integration on a 4000×4000 domain, which should make any overheads negligible. For this domain size, we have a 98% utilization of the GPU. Of the 98% percent GPU utilization, our flux kernel is by far most time consuming, using 87.5% of the runtime. Next comes the time integration kernel, with 12%, leaving less than one percent of the time spent on boundary conditions, maximum timestep, and memory copies to set kernel parameters. For our flux kernel, however, we only achieve an instruction throughput of 56%. This relatively low figure comes from the mismatch between the block size and the conflicting optimization parameters. Nevertheless, this is a trade-off where we are able to implement a much more efficient time integration kernel by having a less efficient flux kernel.

Finally, we have tested the performance impact of running real-time visualization of the results. Our implementation ensures a steady frame rate of 30 frames per second visualization, and runs as many simulation steps as possible. For the Malpasset dam break case, for example, our simulator runs a 4000 second simulation in 27 seconds without visualization. Enabling the visualization, however, the same simulation takes only slightly more than 30 seconds, a mere 11% increase. This clearly shows that efficient utilization of GPUs can also be used to offer real-time visualization without a major effect on the simulator performance.

5. Summary

We have presented a highly optimized implementation of the Kurganov-Petrova scheme on GPUs. The implementation has been verified and validated, showing its ability to capture both analytical and real-world shallow water flows, even with first-order accurate time integration. Our implementation contains novel optimization techniques including the especially efficient early exit strategy, clever application of boundary conditions, and a dramatically smaller memory footprint compared to Brodtkorb et al. [7]. Our extensive performance benchmarks show good resource utilization, being able to compute the first 4000 seconds of the Malpasset dam break case in 27 seconds.

Acknowledgements

Part of this work is done under Research Council of Norway project number 180023 (Parallel3D) and 186947 (Heterogeneous Computing). The author from SINTEF would like to acknowledge the continued support from NVIDIA. Partly accomplished at the National Center for Computational Hydroscience and Engineering, this research was also funded by the Department of Homeland Security-sponsored Souteast Region Research Initiative (SERRI) at the U.S. Department of Energy's Oak Ridge National Laboratory.

References

[1] M. A. Acuña and T. Aoki. Real-time tsunami simulation on multi-node GPU cluster. Supercomputing, 2009. [Poster].

[2] F. Alcrudo and E. Gil. The Malpasset dam break case study. In *The Proceedings of the 4th CADAM meeting*, 1999.

[3] A. S. Antoniou, K. I. Karantasis, E. D. Polychronopoulos, and J. A. Ekaterinaris. Acceleration of a finite-difference weno scheme for large-scale simulations on many-core architectures. In *Proceedings of the 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2010.

[4] T. Brandvik and G. Pullan. Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. *Journal of Mechanical Engineering Science*, 221(12):1745–1748, 2007.

[5] T. Brandvik and G. Pullan. Acceleration of a 3D Euler solver using commodity graphics hardware. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, number AIAA 2008-607, 2008.

[6] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Journal of Scientific Programming*, 18(1):1–33, 2010.

[7] A. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. Simulation and visualization of the Saint-Venant system using GPUs. *Computing and Visualization in Science*, 2010. [forthcoming].

[8] M. de la Asunción, J. M. Mantas, and M. J. Castro. Programming CUDA-based GPUs to simulate two-layer shallow water flows. In *Proceedings of the 16th International Euro-Par Conference*, 2010. [accepted for publication].

[9] M. de la Asunción, J. M. Mantas, and M. J. Castro. Simulation of one-layer shallow water systems on multicore and CUDA architectures. *Journal of Supercomputing*, 2010.

[10] S. S. Frazão, F. Alcrudo, and N. Goutal. Dam-break test cases summary. In *The Proceedings of the 4th CADAM meeting*, 1999.

[11] S. S. Frazão, X. Sillen, and Y. Zech. Dam-break flow through sharp bends, physical model and 2D Boltzmann model validation. In *The Proceedings of the 1st CADAM meeting*, 1998.

[12] N. Goutal. The Malpasset dam failure, an overview and test case definition. In *The Proceedings of the 4th CADAM meeting*, 1999.

[13] T. Hagen, M. Henriksen, J. Hjelmervik, and K.-A. Lie. How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine. In G. Hasle, K.-A. Lie, and E. Quak, editors, *Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF*, pages 211–264. Springer Verlag, 2007.

[14] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, and M. Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13(8):716–726, 2005.

[15] T. R. Hagen, K.-A. Lie, and J. R. Natvig. Solving the Euler equations on graphics processing units. In *Proceedings of the 6th International Conference on Computational Science*, volume 3994 of *Lect. Notes Comp. Sci.*, pages 220–227, Berlin/Heidelberg, 2006. Springer Verlag.

[16] J.-M. Hervouet and A. Petitjean. Malpasset dam-break revisited with two-dimensional computations. *Journal of Hydraulic Research*, 37:777–788, 1999.

[17] R. Holdahl, H. Holden, and K.-A. Lie. Unconditionally stable splitting methods for the shallow water equations. *BIT Numerical Mathematics*, 39(3):451–472, 1999.

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

[19] A. Kurganov, S. Noelle, and G. Petrova. Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations. *SIAM Journal of Scientific Computing*, 23(3):707–740, 2001.

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

[21] M. Lastra, J. M. Mantas, C. Ureña, M. J. Castro, and J. A. García-Rodríguez. Simulation of shallow-water systems using graphics processing units. *Mathematics and Computers in Simulation*, 80(3):598 – 618, 2009.

[22] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, New York, NY, USA, 2010. ACM.

[23] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[24] Q. Liang and F. Marche. Numerical resolution of well-balanced shallow water equations with complex source terms. *Advances in Water Resources*, 32(6):873 – 884, 2009.

[25] W.-Y. Liang, T.-J. Hsieh, M. Satria, J.-P. Chang, Y.-L.and Fang, C.-C. Chen, and C.-C. Han. A GPU-based simulation of tsunami propagation and inundation. In *Algorithms and Architectures for Parallel Processing*, volume 5574 of *Lect. Notes Comp. Sci.*, pages 593–603, Berlin/Heidelberg, 2009. Springer Verlag.

[26] NetCDF (network Common Data Form). `http://www.unidata.ucar.edu/software/netcdf/`. [visited 2010-06-01].

[27] NVIDIA. CUDA community showcase. 2010.

[28] NVIDIA. NVIDIA CUDA programming guide 3.0, 2010.

[29] NVIDIA. Tuning CUDA applications for Fermi, 2010.

[30] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[31] J. Sampson, A. Easton, and M. Singh. Moving boundary shallow water flow above parabolic bottom topography. *Australian and New Zealand Industrial and Applied Mathematics Journal*, 49:666–680, 2006.

[32] D. Shreiner and Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley, 7th edition edition, 2007.

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

[34] W. C. Thacker. Some exact solutions to the nonlinear shallow-water wave equations. *Journal of Fluid Mechanics*, 107:499–508, 1981.

[35] Top 500 supercomputer sites. `http://www.top500.org/`, June 2010.

[36] E. F. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, Ltd., 2001.

[37] A. Valiani, V. Caleffi, and A. Zanni. Case study: Malpasset dam-break simulation using a two-dimensional finite volume method. *Journal of Hydraulic Engineering*, 128:460–472, 2002.

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

[39] X. Ying and S. Y. Wang. Modeling flood inundation due to dam and levee breach. In *Proceedings of the US-China Workshop on Advanced Computational Modeling in Hydroscience and Engineering*, 2005.