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

Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations

Martin L. Sætra · André R. Brodtkorb · Knut-Andreas Lie

Received: date / Accepted: date

Abstract The shallow-water equations model hydrostatic flow below a free surface for cases in which the ratio between the vertical and horizontal length scales is small and are used to describe waves in lakes, rivers, oceans, and the atmosphere. The equations admit discontinuous solutions, and numerical solutions are typically computed using high-resolution schemes. For many practical problems, there is a need to increase the grid resolution locally to capture complicated structures or steep gradients in the solution. An efficient method to this end is adaptive mesh refinement (AMR), which recursively refines the grid in parts of the domain and adaptively updates the refinement as the simulation progresses. Several authors have demonstrated that the explicit stencil computations of high-resolution schemes map particularly well to many-core architectures seen in hardware accelerators such as graphics processing units (GPUs). Herein, we present the first full GPU-implementation of a block-based AMR method for the second-order Kurganov–Petrova central scheme. We discuss implementation details, potential pitfalls, and key insights, and present a series of performance and accuracy tests. Although it is only presented for a particular case herein, we believe our approach to GPU-implementation of AMR is transferable to other hyperbolic conservation laws, numerical schemes, and architectures similar to the GPU.

Keywords AMR (adaptive mesh refinement) · GPU Computing · Shallow-Water Simulation · Conservation Law · Domain Decomposition

Mathematics Subject Classification (2000) 35L65 · 65M08 · 65M50 · 76B15 · 68N19

1 Introduction

The shallow-water equations are able to accurately capture the required physics of many naturally occurring phenomena such as dam breaks, tsunamis, river floods, storm surges, and tidal waves. Most of these phenomena typically have some parts of the domain that are more interesting than others. In the case of a tsunami hitting the coast, one is primarily interested in obtaining a high-resolution solution along the coastline where the tsunami hits, whilst a coarser grid may be sufficient to describe the long-range wave propagation out at sea. Similarly, for dam breaks the most interesting part of the domain is downstream from the failed dam, where one wants correct arrival times of the initial wavefront and reliable estimates of maximum water height during flooding.

Adaptive mesh refinement (AMR) [3,2] is a standard technique that was developed to address this particular problem. The basic idea behind AMR is to recursively refine the parts of the domain that require high resolution, and adaptively update the refinement as the simulation progresses. By utilizing AMR and refining only the areas of interest,

Martin L. Sætra
Centre of Mathematics for Applications, University of Oslo, P.O. Box 1053 Blindern, NO-0316 Oslo, Norway.
Norwegian Meteorological Institute, P.O. Box 43 Blindern, NO-0313 Oslo, Norway.
Tel.: +47 22 96 30 25
Fax: +47 22 96 33 80
E-mail: M.L.Satra@cma.uio.no

André R. Brodtkorb
SINTEF ICT, Department of Applied Mathematics, P.O. Box 124, Blindern, NO-0314 Oslo, Norway.

Knut-Andreas Lie
Centre of Mathematics for Applications, University of Oslo, P.O. Box 1053 Blindern, NO-0316 Oslo, Norway.
SINTEF ICT, Department of Applied Mathematics, P.O. Box 124, Blindern, NO-0314 Oslo, Norway.

the required accuracy can be achieved locally at a considerably lower cost than by increasing the resolution of the full domain. To further accelerate the simulation, we propose to move the hierarchical computation of the AMR method to a modern graphics processing (GPU) architecture, which has proved to be particularly efficient for performing the type of stencil computations that are used in high-resolution shallow-water simulators. Herein, our starting point will be a second-order, semi-discrete, non-oscillatory, central-difference scheme that is well-balanced, positivity preserving, and handles wet-dry interfaces and discontinuous bottom topography [16], which we previously have shown can be efficiently implemented on GPUs [7].

Over the last decade or so, GPUs have evolved from being purely graphics co-processors into many-core general purpose computing engines. Today, a GPU can significantly speed up a wide range of applications in a multitude of different scientific areas, ranging from simulations of protein folding to the formation of black holes [29,4,28]. Accelerators like the GPU have been increasingly utilized in supercomputers and adopted by the high-performance computing community as well. If one considers the Top 500 list [23], the first accelerated systems appeared in 2008, and today over 10% use accelerator technology [1]. Compared to the CPU, GPUs generally have much smaller caches and far less hardware logic, and focus most hardware resources on floating point units. This enables execution of thousands to millions of parallel threads, and "scratchpad-type" memory shared between clusters of threads enables fast collaboration. While the CPU is optimized for latency of individual tasks, the GPU is optimized for throughput of many similar tasks. Alongside the development of the GPU hardware, the programming environment has been steadily growing and improving as well. Although GPU development still is somewhat cumbersome and time-consuming, it has been greatly simplified by more expressive high-level languages, tools such as debuggers and profilers, and a growing development and research community utilizing GPUs [5].

There are several software packages that implement AMR for different problems on the CPU. Some of the most common, free, block-based codes are PARAMESH [22], SAMRAI [13], BoxLib [17], Chombo [9], and AMRCLAW [1]. In particular, LeVeque et al. [19] describe in detail the implementation of AMR in the GeoClaw software package to capture transoceanic tsunamis modeled using the shallow-water equations. There are also a few AMR codes that utilize the GPU [35,32,25,8], but these tend to handle most of the AMR algorithm on the CPU and only perform the stencil computations on a single or a group of Cartesian subgrids on the GPU. This involves uploading and downloading large amounts of data, when it would be much more efficient to keep the data on the GPU at all times. To the best of our knowledge, Wang et al. [35] were the first to map an AMR solver to the GPU based on the Enzo hydrodynamics code [34] in combination with a block-structured AMR. Here, a single Cartesian patch in the grid hierarchy was the unit that is sent to the GPU for computing, resulting in a ten times speedup when using one GPU compared to a single CPU core. Schive et al. [32] present a GPU-accelerated code named "GAMER", which is also an astrophysics simulator implemented in CUDA. Here, the AMR implementation is based on an oct-tree hierarchy of grid patches, in which each patch consists of eight by eight cells. The patches are copied to the GPU for solving, and the results are copied back to the CPU again. However, by using asynchronous memory copies and CUDA streams to solve patches at the same refinement level in parallel, they alleviate some of the overhead connected with the data transfer between the CPU and the GPU and report a 12.2 times speedup using a single GPU on a 128^3 grid compared to a single multi-core CPU, and 10.5 times speedup using 16 GPUs with 256^3 cells compared to 16 multi-core CPUs. Six different levels of refinement were used in both cases. Burstedde et al. [8] discuss a hybrid CPU–GPU version of their elastic wave propagation code, dGea, in which the wave propagation solver runs on the GPU and the AMR operations are executed on the CPU. They report a 50 times speedup on an NVIDIA FX 5800 GPU relative to a single Intel Xeon 5150 core running at 2.666 MHz. CLAMR [25] is developed as a testbed for hybrid CPU-GPU codes using MPI and OpenCL [15], and the developers report a performance speedup of approximately 30 times on the GPU versus the CPU using NVIDIA Tesla 2090s and Intel CPUs, for shallow-water simulations.

Herein, we propose to take the development one step further and move all parts of a block-based AMR algorithm, except for the instantiation of new subgrid patches, to the GPU. Our work is, to the best of our knowledge, the first block-based AMR algorithm that has been *fully implemented* on the GPU, so that all simulation data are kept in GPU memory at all times. Our work is also the first to extend the second-order accurate Kurganov–Petrova scheme [16] to an AMR framework. Although the discussion herein will focus on a specific high-resolution shallow-water solver, most of the choices made in our implementation should be easily transferable to other numerical schemes, other modern accelerators (such as the Intel Xeon Phi), and even other systems of hyperbolic conservation laws.

2 Shallow-Water Simulations on the GPU

We have developed our AMR code based on an existing GPU-accelerated shallow-water simulator [7] that has been thoroughly tested, verified, and validated both on synthetic test cases and against measured data. The simulator is written in C++ and CUDA [26] and has a clean and simple API, which makes it possible to set up and execute a simulation using

[1] On the June 2013 list there were 43 GPU-powered machines, and 12 machines using the Intel Xeon Phi co-processor.

about 5–10 lines of code. A brief overview of this simulator and its mathematical model will be given in this section. For a complete description, we refer the reader to [7,31,30]. A set of best practices for harvesting the power of GPUs for this type of simulation can be found in [6].

2.1 Mathematical Model

The shallow-water equations are derived by depth-averaging the Navier–Stokes equations. By adding a bed shear-stress friction term to the standard shallow-water equations, we get the model used in our simulator. In two dimensions on differential form it can be written as:

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

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.

Our numerical scheme is based on the semi-discrete, second-order, central scheme by Kurganov and Petrova [16], which has been extended to include physical friction terms [7]. The spatial discretization of the scheme is well-balanced, positivity preserving, and handles wet-dry interfaces and discontinuous bottom topography. In semi-discrete form it can be written as:

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

Here, Q_{ij} is the vector of conserved variables averaged over the grid cell centered at $(i\Delta x, j\Delta y)$, H_B is the bed slope source term, H_f is the bed-shear stress source term, and F and G represent numerical fluxes along the abscissa and ordinate, respectively. The fluxes are calculated explicitly, based on one-sided point-values $Q_{i\pm1/2,j}$ and $Q_{i,j\pm1/2}$ evaluated from a piecewise-linear reconstruction from the cell averages with slopes limited by the nonlinear, generalized minmod function [18,21,24,33]. The bed-slope source term H_B is also calculated explicitly, and the discretization is designed carefully to ensure that the numerical fluxes exactly balance the bed-slope source term for a lake at rest. Finally, to avoid numerical problems with dry states, the scheme uses a mass-conservation equation formulated using water elevation rather than water depth.

To evolve the solution in time, one can choose between the simple first-order, forward Euler method and a second-order, total-variation-diminishing Runge–Kutta method. The friction source term, H_f, is discretized semi-implicitly, which gives rise to the following numerical scheme for forward Euler time integration

$$Q_{ij}^{k+1} = \left(Q_{ij}^k + \Delta t R(Q^k)_{ij}\right) / \left(1 + \Delta t \alpha\right), \tag{3}$$

in which α is the semi-implicit friction source term. The two steps in the Runge–Kutta method are on the same form. For a detailed derivation of the numerical scheme, we refer the reader to [16,7].

2.2 Shallow-Water Simulations on Cartesian Grids

The execution model of the GPU is perfectly suited for working with structured grids since the GPUs have been designed mainly to calculate the color values of regularly spaced pixels covering the computer screen. Conceptually, we "replace" the screen with a computational domain and the colored points by cells (see Figure 1). By structuring the computations so that every cell can be solved independently, we can solve for all cells in parallel.

The shallow-water simulator uses four *CUDA kernels* to evolve the solution one time step:

```
1: while t < T do
2:     for i=1:order do
3:         Flux: reconstruct piecewise continuous solution, compute F, G, and H_B(Q_ij, ∇B),
4:                 and compute upper bound on wave speeds
5:         if order==1 then
6:             Max time step: use parallel reduction to find global limit on time step Δt
7:         end if
8:     Time integration: evolve solution forward in time
```

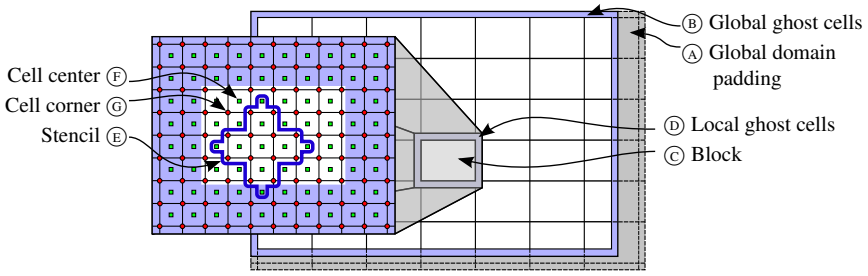


Fig. 1: Domain decomposition and variable locations. The global domain is padded Ⓐ to fit an integer number of CUDA blocks, and global ghost cells Ⓑ are used for boundary conditions. Each block Ⓒ has local ghost cells Ⓓ that overlap with other blocks to satisfy the data dependencies dictated by the stencil Ⓔ. Our data variables Q, R, H_B, and H_f are given at grid cell centers Ⓕ, and B is given at grid cell corners Ⓖ.

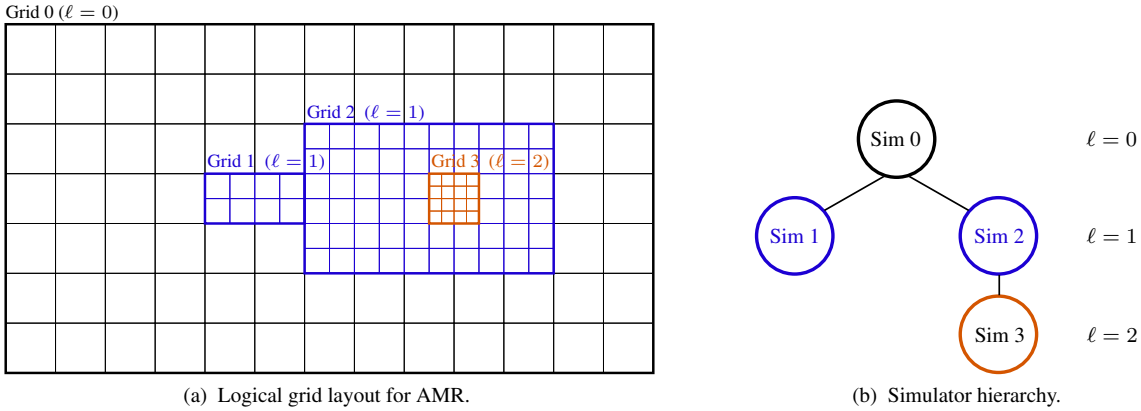
9: *Boundary condition:* update all global boundary conditions
10: **end for**
11: $t = t + \Delta t$
12: **end while**

The flux kernel is also responsible for finding an upper bound on the maximum speed of propagation per CUDA *block*, which will be used to limit the time step according to the CFL condition. After the flux kernel has determined an upper bound, the max time-step kernel finds the global limiting factor using a parallel reduction [12]. In the boundary-conditions kernel, each of the four global boundaries may use different boundary conditions. Executing all four CUDA kernels once, constitutes one full time step if we use forward Euler for time integration. With second-order Runge–Kutta, all kernels are executed twice (i.e., in two substeps), except for the max time-step kernel which needs only be executed in the first substep. To enable maximal utilization of the GPU accelerator, the simulation data are kept on the GPU at all times. Even when interactive visualization is switched on, the simulation data are simply copied from CUDA memory space to OpenGL memory space, never leaving GPU memory.

3 Adaptive Mesh Refinement

The simulator discussed in the following has two novel components: formulation of the Kurganov–Petrova scheme in an AMR framework, and efficient implementation of this framework on a GPU many-core system. In this section we will give algorithmic and implementation details, focusing mainly on challenges related to the many-core GPU implementation. The basic AMR data structure is a sequence of nested, logically rectangular meshes [2] on which balance law (1) is discretized as in (2). There are two main strategies for local refinement: A *cell-based* strategy will refine each cell based on a given refinement criteria, whereas a *block-based* strategy will group cells together and refine collections of cells. We have chosen a block-based strategy, which we will refer to as *tile-based* since we operate in two dimensions. The tile-based approach has several advantages over the cell-based AMR and is, in particular, a good fit for the GPU architecture with regards to cache locality and memory access patterns. It also fits nicely with using ghost cells to handle boundary conditions. To reduce the overhead of the regridding algorithm, one will typically want to extend the refinement area so that waves can propagate for a few time steps before they reach the boundaries of a newly added patch. In other words, padding of new refined areas is necessary and this comes automatically when using tiles. An advantage of using cell-based refinement is a finer granularity, but this can also be achieved with tile-based refinement by adjusting the size of the tiles. The tiles could consist of as little as one cell, and the tile size should be adjusted according to the necessary degree of granularity and efficiency for each simulation scenario. It should also be noted that although the refinement is tile-based, criteria for refining are enforced on the cell-level.

To explain the AMR algorithm, we first need to introduce the grid hierarchy (see Figure 2) and establish some terms that will be used consistently for the rest of the paper. Each grid is linked to an instance of the simulator presented in Section 2. For a standard simulation on a single grid, we will have one simulator instance linked to one grid (see Figure 1). This is referred to as the *root grid* in an AMR context, and the root grid is the only grid that covers the full simulation domain. For a two-level AMR simulation, the root will also have a vector of *children*, each of them covering some part of the domain with twice the resolution of the root grid. For more than two levels this becomes recursive, so that all grids, except the root grid and the grids with the highest resolution (leaf node grids), have one parent grid, and one or more children.



(a) Logical grid layout for AMR.

(b) Simulator hierarchy.

Fig. 2: The AMR grid hierarchy with three levels of refinement and four grids in total, including the root grid. Each grid node is connected to a separate simulator instance which internally uses the domain decomposition shown in Figure 1.

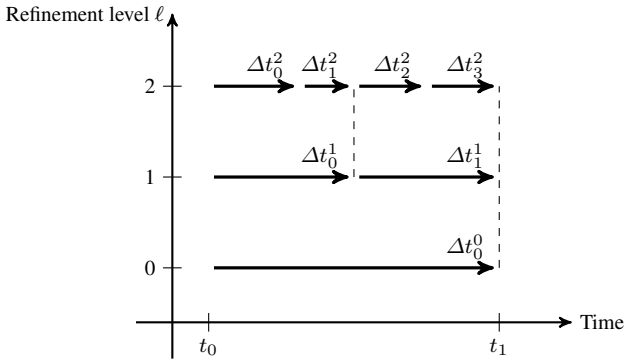


Fig. 3: Different time-step sizes Δt_i^ℓ for grids at different refinement levels ℓ. Global time synchronization occurs after every root time step. This means that the simulation time at root level after time step Δt_0^0 will be identical to the simulation time at refinement level 1 after time step Δt_1^1, and simulation time on refinement level 2 after time step Δt_3^2, and so on. Notice that the time-step sizes in the same refinement level are slightly different due to the CFL condition.

In addition to the tree of simulation grids, there are two main components to the AMR algorithm: the time integration and the regridding. The time integration evolves the solution in time on the parent grid, sets the boundary cells, and initiates the time integration on all descendants of the current simulator instance. The regridding process, on the other hand, adaptively creates and destroys children recursively based on the chosen refinement criteria.

3.1 Time integration

The time-integration function evolves the solution on the current grid up to the current time t on the parent grid. In doing so, the last time step must often be reduced to reach t exactly (see Figure 3). No grid on refinement level ℓ can be integrated in time before all grids on levels $k = 0, 1, ..., \ell - 1$ are at least one time step ahead of level ℓ. Except for this restriction, every grid may be integrated in parallel, independently of all other grids. The time-step size is computed independently on each child grid to ensure optimal computational efficiency for the update on each subgrid. That is, subgrids containing slowly propagating waves can be updated using larger time steps than subgrids on the same level that contain fast waves.

The resulting AMR time integration can be outlined in seven steps:

1. Set $\ell = 0$.
2. Take one time step of length Δt^ℓ on the grid(s) at level ℓ.
3. Using the solution at the beginning and end of this time step: Perform space-time interpolation to determine ghost cell values for all the intermediate time steps on all level $\ell + 1$ grids. The time interpolation is performed during time stepping, as we do not know the time-step sizes a priori.

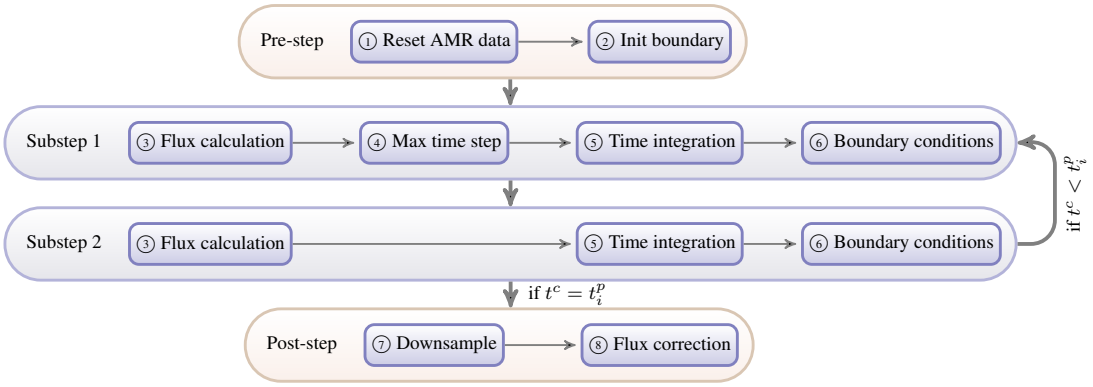
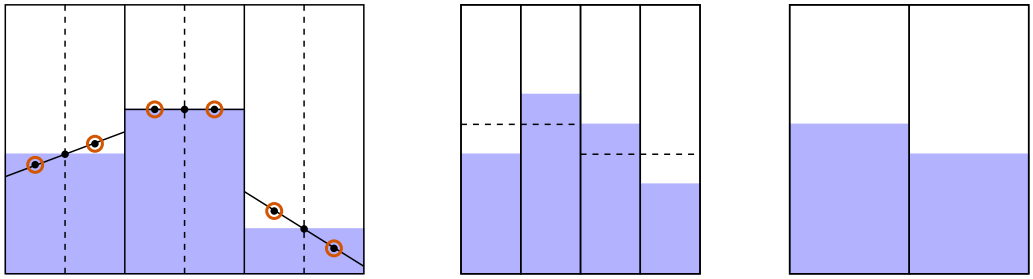


Fig. 4: Conceptual overview of all CUDA kernels used to evolve the solution from time t_{i-1}^p to t_i^p. In the pre-step stage, function ① resets all AMR data for the local solve, and sets the boundary values for the start of the next time-step series at time t_{i-1}^p to the boundary values used at the end of the previous time-step series. Kernel ② computes the boundary values for the end of the next time-step series using the solution from the parent grid at time and t_i^p. In the first substep, kernel ③ calculates F, G, and H_B; kernel ④ finds the maximum Δt; kernel ⑤ calculates α and Q^{k+1}; and kernel ⑥ enforces boundary conditions on Q^{k+1}. For all other than the root grid, the boundary values are a linear interpolation between the reconstructed solutions on the parent grid at times t_{i-1}^p and t_i^p. Last, kernel ⑦ replaces the parent solution by the child solution where they overlap, while kernel ⑧ maintains conservation of mass. As indicated in the figure, Substeps 1 and 2 are repeated until the solution on the current grid has been advanced to the same time level $t^c = t_i^p$ as the parent grid.



(a) 1D minmod reconstruction.

(b) Averaging from level ℓ to level $\ell - 1$ in 1D.

Fig. 5: Visualization of the calculations done by the init-boundaries kernel (A) and the downsample kernel (B). The black dots are input, the encircled black dots are output, and the black lines are the reconstructed surface.

4. Time step on all level $\ell + 1$ grids to bring these grids up to the current time on the level ℓ grid.
5. For any grid cell at level ℓ that is covered by a level $\ell + 1$ grid, replace the solution in this cell by an average of the values from the cells in the level $\ell + 1$ grid that covers this cell.
6. Adjust the values at cells which interface a different grid to maintain conservation of mass.
7. Set $\ell = \ell + 1$, and repeat Steps 2–7 for all grids at this refinement level.

Figure 4 illustrates one simulation cycle for one grid using this algorithm. Here, kernel ① represents a reset of AMR data before a new cycle of time steps, while kernels ② to ⑧ represent one CUDA kernel each. We will go through each step, with emphasis on what is special for AMR. A thorough explanation of kernels ③–⑥ for non-AMR simulations can be found in [7].

Before a simulator instance can start updating its grid, the solution on all parent grids must be one time step ahead. That is, for all simulator instances that do not hold the root grid we can assume that there is a coarser solution available from the end-time t_i^p at the parent grid that can be used to interpolate boundary conditions on the child grid. To prepare for the next time steps on the child grid, the *reset AMR data* kernel resets the accumulated fluxes across the child-parent interface and the accumulated time-step sizes to zero. The pointers to the two arrays containing initial and end-time boundary values are swapped so that the solution at the end of the last computed step on the parent grid (at time t_{i-1}^p) becomes the initial solution for the sequence of time steps on the current child grid. The array holding boundary data at time $t^c = t_i^p$ is set in the *init-boundaries* kernel by reconstructing the parent solution from time t_i^p and then evaluating

the point values at the center points of the child boundary cells (see Figure 5a). For the root grid, the boundary data are obtained from the boundary conditions of the problem.

Once the data structures holding boundary data is properly set, we can evolve all children to time $t^c = t_i^p$: Kernels ③ to ⑥ in the two Runge–Kutta substeps are oblivious to the AMR grid hierarchy and treats each grid as if it were a single-grid simulation, with only minor adaptations for AMR. In kernel ③, *flux-calculation*, we accumulate the fluxes for each of the two substeps, weighing each of them with the Runge–Kutta weight 0.5. This kernel also finds the limiting factor for the time-step size per block. Kernel ④, *max time-step*, then reduces the per-block factor to a global value, and computes the maximum time step based on the CFL-condition of the system. This time step is then added to the accumulated time t^c for the current child grid. Then, kernel ⑤, *time integration*, advances the solution in time and accumulates the fluxes going across the interface to the parent grid, weighted with the time-step size. On the root grid, the *boundary condition* kernel (kernel ⑥) works as normal, executing the prescribed boundary condition for the domain. For all other grids, this kernel interpolates linearly between the two arrays containing boundary data prepared from the parent solution as described above, using the accumulated time as the input variable. If Euler time integration is used, the second substep is simply omitted, and in kernel ③ we do not weigh the fluxes with 0.5. The time-integration process is repeated until t^c equals t_i^p (see Figure 3). Now, the new solution must be communicated back to the parent grid. In kernel ⑦, *downsample*, the solution in the child grid is averaged down to the resolution of the parent grid (see Figure 5b) and used to replace the solution in the parent grid where this grid overlaps with the child grid.

The *flux-correction* kernels in ⑧ finally ensure conservation of mass by modifying the values of cells interfacing a different grid. For grids at the same refinement level, the flux correction computes the corrected flux across a shared interface using

$$F_{\text{corr}}(Q_{i+1/2,j}) = \frac{1}{2\Delta t}\left(\sum \Delta t^L F(Q^L_{i+1/2,j}) + \sum \Delta t^R F(Q^R_{i+1/2,j})\right), \qquad \Delta t = \sum \Delta t^L = \sum \Delta t^R.$$

Here, variables with superscript L and R denote that the variable is taken from the left- and right-hand side of the interface, respectively. The sums represent the accumulated fluxes computed on each side of the interface, weighted with their respective time-step sizes, and the corrected flux is then used to adjust the variables in the adjacent cells. For an interface between a parent and a child grid, on the other hand, we assume that the flux computed on the child grid is the most correct. We therefore correct the flux only in the parent grid using

$$F_{\text{corr}}(Q_{i+1/2,j}) = \frac{1}{\Delta t}\sum \left(\Delta t^c F(Q^c_{i+1/2,j}) + \Delta t^c F(Q^c_{i+1/2,j+1})\right), \tag{4}$$

in which superscript c denotes the child grid. The sums represent the accumulated fluxes for the two cells in the child grid that interface with a single cell in the parent grid. (Note that we have to take into account the difference in grid cell size between the child and parent grid when computing the corrected flux.)

3.2 Regridding

The regridding process is performed after a given number of time steps on every grid from the root grid to a prescribed refinement level. The time intervals and grid depth should be set according to the problem type (dam break, river flood, tsunami, etc.) and refinement criteria. We do not perform refinement on a cell-per-cell basis, but rather we consider tiles of 32×32 cells. Tile size may be adjusted to better suit the problem type and domain size. The process starts by running the *refinement-check* kernel that checks all cells in each tile against the given refinement criteria. After a shared-memory reduction, using 32×4 CUDA threads per tile, the kernel outputs the number of cells that fulfilled the refinement criteria per tile to the *refinement map*. This makes it possible to demand that a certain fraction of the cells must fulfill the refinement criteria before a tile is refined. Setting this number low gives better accuracy, but lower efficiency, and vise versa. All new child grids have twice the resolution of the parent grid.

Using the vector of existing child grids, we mask out every tile already overlaid with a child grid in the refinement map to avoid generating new grids that include already refined tiles. Nevertheless, we may end up with overlapping grids in some cases, and this needs to be handled after we have made the *proposed bounding boxes* for new child grids. The new bounding boxes are checked against existing bounding boxes, and any overlap is categorized as one of nine different configurations. The first eight are handled as shown in Figure 6, while the last, which corresponds to complete overlap, is handled by copying the solution from the existing bounding box(es) into the new proposed bounding box. Generating coordinates for new bounding boxes and checking for overlaps are the only tasks performed on the CPU. Likewise, the refinement map may also predict that certain child grids are no longer required. Since the corresponding values have already been averaged onto the parent grid by the downsample kernel, the child grid can be removed and the simulator instance deactivated. Although not yet implemented, it should be straight forward to include the double-tolerance adaptive strategy proposed recently [20], which aims to optimize the overall numerical efficiency by reducing the number of child grids whilst preserving the quality of the numerical solution.

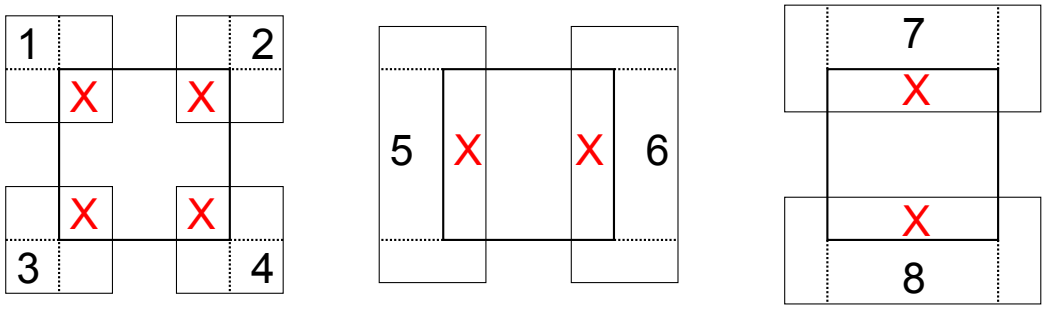


Fig. 6: Showing eight different overlapping configurations and how they are treated. For all configurations, the middle bounding box is the existing one. The dotted lines show how new proposed bounding boxes are repartitioned and the overlapping part, marked with 'X', is discarded.

To initialize a new child grid, we copy and edit the initialization parameters of the parent grid; grid size (Δx, Δy), current simulation time, and all four boundary conditions are set to their correct values. All initial conditions (i.e., the physical variables and bathymetry) remain unset, so no data is uploaded to the GPU when the new child grid simulator instance is constructed. Each new simulator instance also creates its own CUDA stream to enable concurrent execution of grids. After a new simulator instance has been instantiated, the bathymetry and initial conditions need to be set. The bathymetry can be initialized using texture memory, enabling the use of efficient hardware interpolation. The initial conditions are reconstructed from previous cell-averaged values using the generalized minmod limiter, and the array with initial boundary data is filled by running the init-boundary kernel described earlier. (Only one of the two arrays containing boundary values for the start and end of each time-step series is filled in the initialization, and the other one will be filled in the standard time stepping.) Last, we update the neighborhood information for the new simulator instance with any abutting grids at the same refinement level and add it to the vector of children.

To have proper nesting, Berger and Colella [2] require that a child grid starts and ends at the corner of a cell in the parent grid, and that there must be at least one level $\ell - 1$ cell in some level $\ell - 1$ grid separating a grid cell at level ℓ from a cell at level $\ell - 2$ in the north, east, south, and west directions, unless the cell abuts the physical boundary of the domain. These minimum requirements also apply in our GPU code. In addition, we require that no interface between a parent and a child grid, or two child grids at the same refinement level, crosses each other, and on the root grid (level 0), we require three cells between the global domain boundary and all level $\ell > 0$ grids. Without a three-cell margin, one of the steps (parent-child flux correction) of the AMR algorithm interferes with the global wall boundary condition. This last requirement is not imposed by the numerical scheme, but is an assumption introduced to simplify the current implementation of global boundary conditions which should be removed when the simulator is moved beyond its current prototype stage.

3.3 Optimizations

As we have seen, each grid except the root grid depends on its parent to supply the necessary boundary conditions. If we consider Figure 2, this means each grid only depends on its parent grid to complete one time step before its solution can be evolved up to the same time as its parent. Furthermore, this means that we may run simulations on all grids simultaneously, as long as this one dependency if fulfilled. This task parallelism is handled by using *streams* and *events* in CUDA. Each simulator instance has its own stream, in which kernels and memory transfers are issued and executed in order. The synchronization with other simulators is done by issuing events. This synchronization is implemented in the main AMR step function:

```
1: while getCurrentTime() < parent_time do
2:     regrid();
3:     step(parent_time);
4:     cudaEventRecord(main_step_complete, stream);
5:     stepChildGrids();
6: end while
```

assuming that '*parent_time*' is passed as an argument from the parent grid. In the code above, `regrid()` checks the current grid hierarchy, and performs regridding based on the current simulation settings of refinement criteria, regridding frequency, etc. After the `step(...)`-function the grid associated with current simulator instance is advanced in time, and its child grids can then be treated next. If the current grid is the root grid, we only perform one time step, and for all

other grids than the root grid; we do time stepping until the grid has reached the same advanced time as its parent grid. In the last case the child grids of the current grid need to be updated between each time step. The integration of child grids also requires similar synchronization:

```
1: function STEPCHILDGRIDS
2:     for 1 → number_of_child_grids do
3:         cudaStreamWaitEvent(child_grids[i]->stream, main_step_complete);
4:         ...
5:         // calling the main AMR step function
6:         child_grids[i]->step(getCurrentTime());
7:         ...
8:         cudaEventRecord(child_grids[i]->child_steps_complete, child_grids[i]->stream);
9:         cudaStreamWaitEvent(stream, child_grids[i]->child_steps_complete);
10:        ...
11:    end for
12: end function
```

Moreover, it is necessary with additional synchronization within each simulation as described above.

To avoid unnecessary computations it is possible to exit early in dry cells, since the solution will remain constant throughout the whole time step unless the cell is neighbor to a wet cell [7]. Likewise, one can reduce the memory footprint if data values are not stored before they actually contribute in the simulation using a sparse-domain algorithm [30]. These code optimizations have not been included in the AMR implementation, and hence all performance results report the time it takes to simulate every cell in the domain. Some minor adaptations have been made to the simulator described in [7], the most noteworthy being the switch from saving the sum of net fluxes and source term as a vector $R(Q)_{ij}$ (see (2)), to saving them separately and postponing the computation of $R(Q)_{ij}$ to the time integration kernel.

Because each child grid has twice the resolution of its parent, one should in principle use two time steps on the child grid for each time step on the parent grid. However, since the maximum time step allowed by the CFL condition is computed for each time step, and estimates of maximum eigenvalues (local wave-propagation speed) tends to increase with increasing resolution, the allowed step sizes tend to decrease with increasing refinement level. Because we need to synchronize the time between grids at different refinement levels (see Figure 3), we limit the size of the last time step so that all grids at level $\ell + 1$ will exactly hit the current time of level ℓ after some number of time steps. Sometimes this leads to very small last time steps. This is unfortunate since very small time steps cost exactly as much as larger time steps, without significantly advancing the solution in time. By reducing the CFL target a few percent below its maximum allowed value, we are able to avoid many of these small time steps, and thus increase efficiency (see Results 4.5).

A feature called dynamic parallelism has been introduced in the most recent versions of CUDA GPUs. Dynamic parallelism enables a kernel running on the GPU to launch further kernels on the GPU without any CPU involvement, thereby improving performance. One proposed use of dynamic parallelism has been AMR [14], as it enables the GPU to adaptively refine the simulation domain. In cell-based AMR codes, the construction and deletion of grids is highly dynamic, and will therefore benefit greatly from dynamic parallelism. However, our simulator is tile-based, and the overhead connected with the regridding process is already only a small fraction of the run time. The impact of using dynamic parallelism will therefore be negligible, and restrict the simulator to only execute on the most up-to-date CUDA GPUs.

4 Results

How to best report the performance and accuracy of a tile-based AMR code is not self evident. There are many parameters to consider, e.g., refinement criteria, tile size, minimum child grid size, regridding frequency, and maximum levels of refinement. We have constructed seven tests and examples, and the results will be presented in separate sections below.

Our results show that the time integration consumes the majority of the computational time and the time spent on refinement checks and regridding is negligible. The time integration is made up of four kernels, some of them very computationally intensive. The refinement check, on the other hand, is a single, relatively simple kernel and is typically not performed every time step, typically only every 50th or 100th time step. Likewise, the regridding procedure is typically performed on a very small fraction of the global domain. This means that whereas the choice of refinement and regridding parameters will significantly impact the accuracy of the simulation, the hardware utilization of the AMR algorithm will not depend on whether patches are introduced adaptively or statically. Hence, all tests, except those noted, have been run on a statically refined grid with a tile-size of 32×32 cells (measured in the parent grid).

We have run the simulator with both first-order forward Euler time integraion and second-order Runge–Kutta integration. The Runge–Kutta time integration will give overall better hardware utilization and hence improve results on the

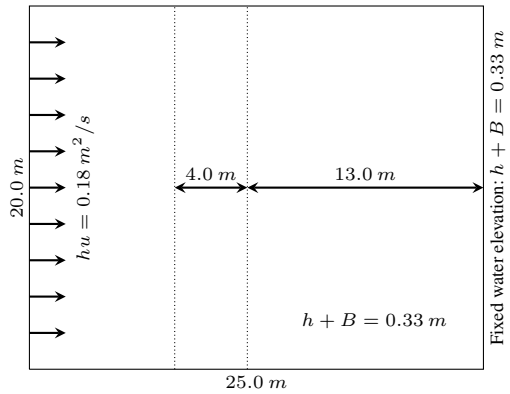


Fig. 7: The SWASHES test setup. The dotted lines indicate the bump in the bathymetry.

efficiency tests as we get a higher compute-to-"AMR-overhead" ratio. Friction terms are neglected for simplicity, unless stated otherwise in the description. The performance tests were run on a node with an Intel i7 3930k CPU @ 3.2 GHz, 32 GB RAM, 64-bit Windows 8, and a GeForce 780 GTX graphics card using CUDA 5.5. All other tests were run on a node with an Intel i7 2657M CPU @ 1.6 GHz, 8 GB RAM, 64-bit Linux, and a GeForce GT 540M graphics card using CUDA 5.

4.1 Verification

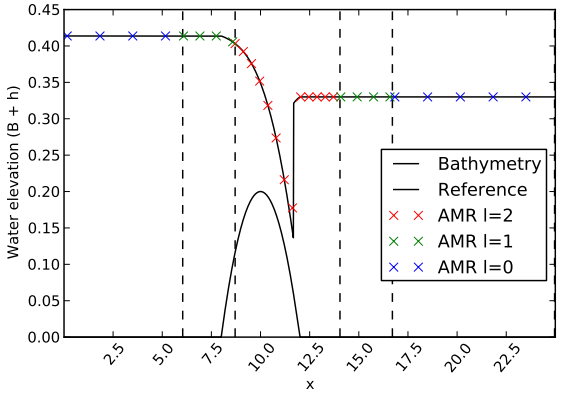
In this test, we will use an analytic solution to verify the accuracy of the original simulator and its new AMR version with different levels of refinement. To this end, we will use the SWASHES code [10] to compute a steady-state reference solution for a transcritical flow with a shock over a bathymetry with a single bump [11]. The domain, as depicted in Figure 7, is 25 m $\times$ 20 m with a bathymetry given by:

$$B(x) = \begin{cases} 0.2 - 0.05(x - 10)^2 & \text{if } 8 \text{ m} < x < 12 \text{ m}, \\ 0 & \text{else.} \end{cases} \tag{5}$$

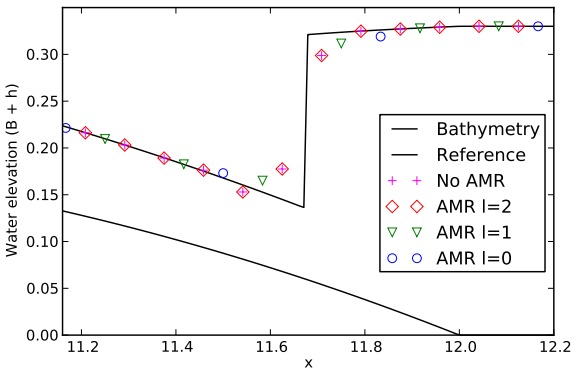
For the AMR code, a new bathymetry is generated for each refinement level to avoid introducing errors due to interpolation or extrapolation. Water elevation is initially set to 0.33 m. Wall boundary conditions are imposed at $y = 0$ m and $y = 20$ m. At $x = 0$ m we have an inflow boundary with a fixed discharge of 0.18 m^2/s in the positive x-direction and at $x = 25$ m we have an outflow boundary with a fixed water elevation at 0.33 m. All simulations are run using first-order Euler time integration until they reach steady state. The AMR algorithm will generate small fluxes in the y-direction across interfaces between grids at different refinement levels, and hence longer simulation times are required before the steady state is reached. Since the SWASHES reference solution is in 1D, we extract an 1D solution from our 2D solution, which is simply a line going through the middle of the domain in the x-direction. From the results, shown in Figure 8, we see that the AMR simulator captures the shock with increasing accuracy for each level of refinement. Furthermore, the AMR solution, in which we only refine a small area around the shock, is almost identical to the global refinement solution of matching resolution.

Using the same setup, we now compare the computational time for the original simulator and the AMR simulator, with resolution set so that the child grids with highest resolution in the AMR simulation runs have the same resolution as the grid in the no AMR simulation runs, in which 300 $\times$ 240 cells are used. The first level child grid is offset by three cells from the $x = 0$ m boundary, and the second level child grid is offset two cells from the boundary of the first child grid. To make the comparison fair, we fit as many tiles as possible into each of the two levels of child grids in the y-direction. In the x-direction, the grid with the highest resolution is just wide enough to capture the bump and the shock, and the its parent grid is just wide enough to contain it. All test runs reach steady state. Results are shown in Figure 9. The AMR simulations are clearly faster than the original simulator, and the discrepancy is increasing. This is caused by the increasing ratio of cells outside the area of interest (the shock) over the total number of grid cells.

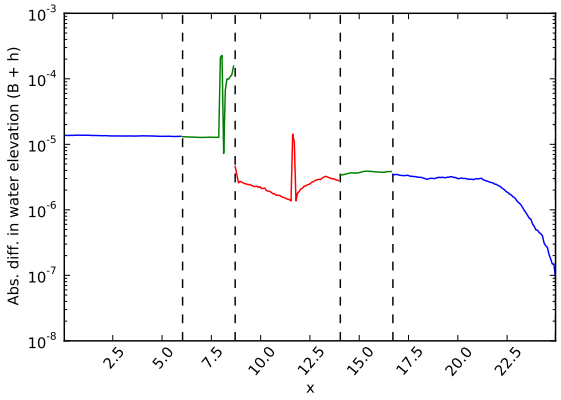
Considering both accuracy and efficiency, we have shown that the AMR-simulation gives the same results for less computations, and several times faster (3.4 times for the highest resolution). In this test, the domain size has been constant at 20 m $\times$ 25 m, and the resolution has been variable. For cases where the area of interest is of fixed size, but the rest of the domain is increased, e.g., if the area of interest is far away from the event origin, the use of AMR will have an even bigger impact on simulation efficiency.



(a) Solution in the full simulation domain.



(b) Solution zoomed in around the shock.



(c) Difference between single-grid and AMR simulation.

Fig. 8: Single-grid simulation and simulations with AMR using up-to three levels of refinement. Every fifth grid point is marked in the full simulation domain solution. The reference solution has been generated using the SWASHES library [10]. Notice that the no AMR and AMR results match up closely, with an error between 10^{-4} and 10^{-6}. The largest discrepancies are located at the shock itself, and at the bump covered by the level one AMR grid (which has half the resolution of the no AMR simulation).

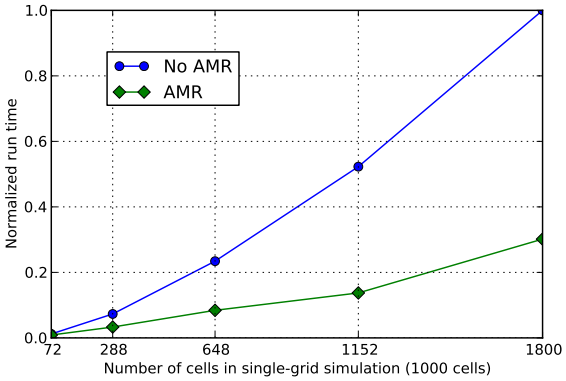
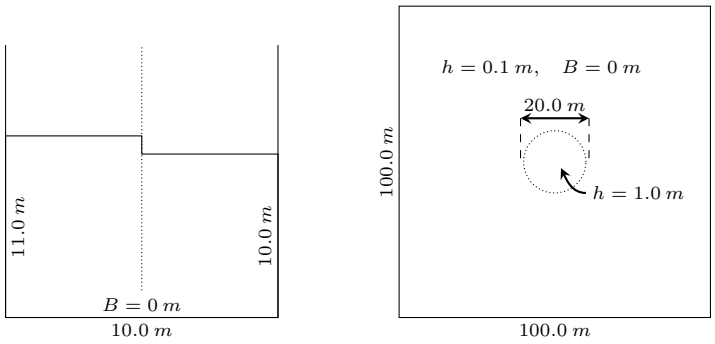


Fig. 9: Comparison of wall clock time for simulation runs with and without AMR. All test runs have been normalized with regards to the single slowest run. The AMR code is over three times faster for larger domain sizes.



(a) Dam break (b) Circular dam break

Fig. 10: Dam break setups used in the tests: The left figure shows a cross-section of the domain (which is 80 m along the intersected dimension), and the right figure shows a top-view of the domain. The dotted lines indicate the location of the dams that are instantaneously removed at simulation start.

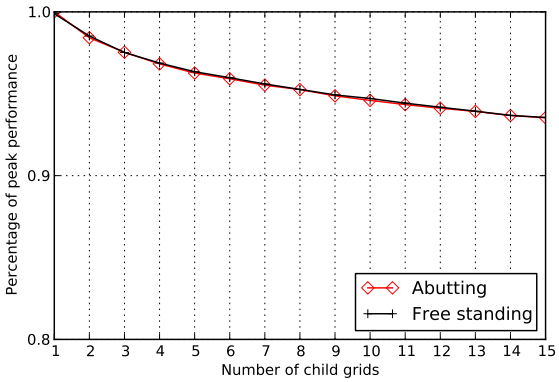


Fig. 11: Overhead connected with an increasing number of added child grids. For the abutting test runs, each new child grid is added next to the last added child grid so that the two grids interface each other along one full edge. All test runs have been normalized with regards to the single fastest run.

4.2 Child grid overhead

Maintaining several simulator instances to represent different subgrid patches may represent a significant computational overhead that may potentially slow down the simulation. In this test, we will therefore measure the added overhead in time stepping connected with an increasing number of child grids. To this end, we consider a rectangular domain represented on a root grid with 4096×512 cells. Initial data are set to be equal 11 m in the left half of the domain and 10 m in the right half (see Figure 10a). Wall boundary conditions are used and all tests are run for two minutes wall-clock time using first-order Euler time integration. The first series of test runs is performed with abutting child grids, centered in the domain, consisting of 256×256 cells (measured in the root grid) laid out so that each new child grid interfaces with the previously added child grid along one full edge. The second set of test runs is performed with a two-cell margin in the root grid between child grids. Results are shown in Figure 11. The difference between the two setups is negligible, showing that the flux correction between neighboring children is efficient. For a total of 15 subgrids, the performance drops to roughly 94% of peak. Kepler-generation GPUs [27] from NVIDIA contain new features for more efficient execution of concurrent kernels, which should decrease this overhead even more.

This test clearly shows that a strategy for minimizing the number of child grids on each level of refinement is important. The current prototype uses a relatively simple strategy for merging neighboring subgrid patches. However, since this operation is performed on the CPU and well separated from the rest of the simulator code, one can easily substitute it by more advanced methods that can be found in generic grid generators.

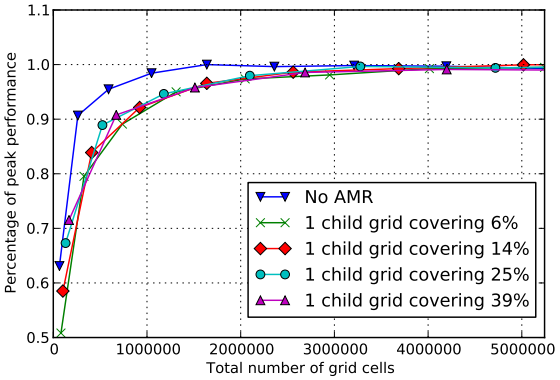


Fig. 12: Comparison between single-grid simulation and AMR simulations using a single child grid covering a varying fraction of the global domain. All test runs have been normalized with regards to the single fastest run. There is an overhead connected with using AMR, but the overhead becomes negligible for larger domain sizes.

4.3 Efficiency

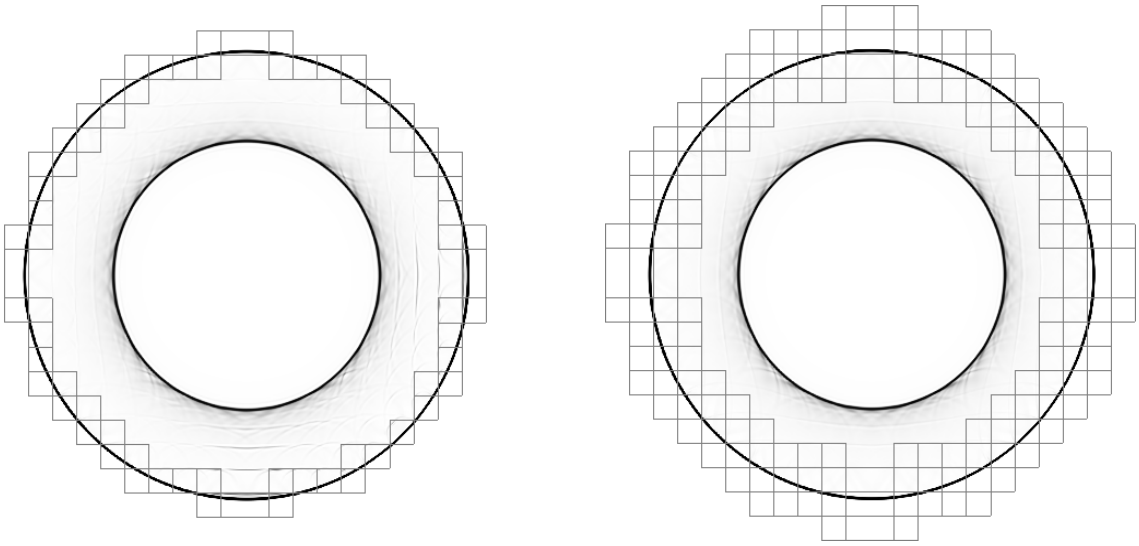
In this test, we compare the efficiency of the original simulator with different simulations with the AMR simulator using a single child grid covering a varying area of the global 100 m $\times$ 100 m domain. Initial conditions are a circular dam break in the center of the domain with a radius of 10 m, in which the water elevation is set to 1.0 m, and 0.1 m in the rest of the domain (see Figure 10b). Wall boundary conditions are used and we simulate the wave propagation of the dam break up to time ten seconds using first-order Euler time integration. Results are shown in Figure 12. As expected, the efficiency of the hardware utilization increases with increasing percentage of the domain covered by the child grid. Likewise, increasing the number of cells in the root grid improves the performance and eventually leads to full utilization of the computing potential of the GPU. From this test we conclude that the overhead associated with a single child grid is small (max 5% for one million cells or more) and diminishing as the child grid covers an increasing portion of the root grid and as the number of total cells increases.

4.4 Shock tracking

In this test, we will demonstrate that the regridding algorithm is capable of tracking propagating waves. We mark all cells in which $\max\left(|h_{i+1,j} - h_{i,j}|, |h_{i,j+1} - h_{i,j}|\right) > 0.1$ m for refinement, in which $h_{i,j}$ is the average water depth in the grid cell centered at $(i\Delta x, j\Delta y)$. The domain is as in the previous test (see Figure 10b). Initial conditions are a circular dam break in the center of the domain with a radius of 10 m, in which the water elevation is set to 1.0 m, and 0.1 m in the rest of the domain. The root grid is 512×512 cells, a refinement check is performed every 10th time step, and the minimum child grid size is set to one tile to accurately track the shock. The test is run using second-order accurate Runge–Kutta time integration until it reaches seven seconds simulation time.

Figure 13a shows that the refinement closely follows the leading shock. The adept reader will notice that there are visible anomalies in the solution. These anomalies are caused by the initializing of new child domains. By initializing a child grid over an existing shock, the inevitable interpolation errors are severely magnified at the location of the shock, which leaves a "ghost" impression of the shock in the solution. These errors are alleviated by simply making sure that child domains are initialized in smooth parts of the solution, thereby minimizing the interpolation errors. One example of this strategy is illustrated in Figure 13b, in which the anomalies are reduced dramatically. Both of these simulations have a large number of child grids. The number of child grids can be dramatically reduced, however, by simply combining multiple neighboring child grids.

To further investigate this issue, we have studied the radial symmetry of the problem. Figure 14 shows the radial symmetry seven seconds into the simulation. We can see that the simulation with 512^2 cells, as expected, is more smeared than the reference solution, especially close to the sharp shock. The 1024^2 simulation, on the other hand, captures the shock much better. In both simulations, however, we see that there is a slight radial dissymmetry, shown by the vertical spread of the points. When we then go to the AMR simulations, we see that these simulations have a larger radial dissymmetry. This is as expected, as selectively refining parts of the domain will inevitably reduce the radial symmetry. For the simulation in which the child domains are initialized *on* the shock, however, there is a large non-physical dissymmetry (clearly visible approximately 25 meters from the center of the domain). When we initialize



(a) Without padding. (b) With padding.

Fig. 13: Emulated Schlieren visualization is used to demonstrate how the refinement follows the shock, enabling a more accurate capturing. Both figures are after seven seconds of simulation time, and the root grid resolution is 1024×1024 cells. Notice the ripples in the solution without padding, which have been significantly reduced when padding is added.

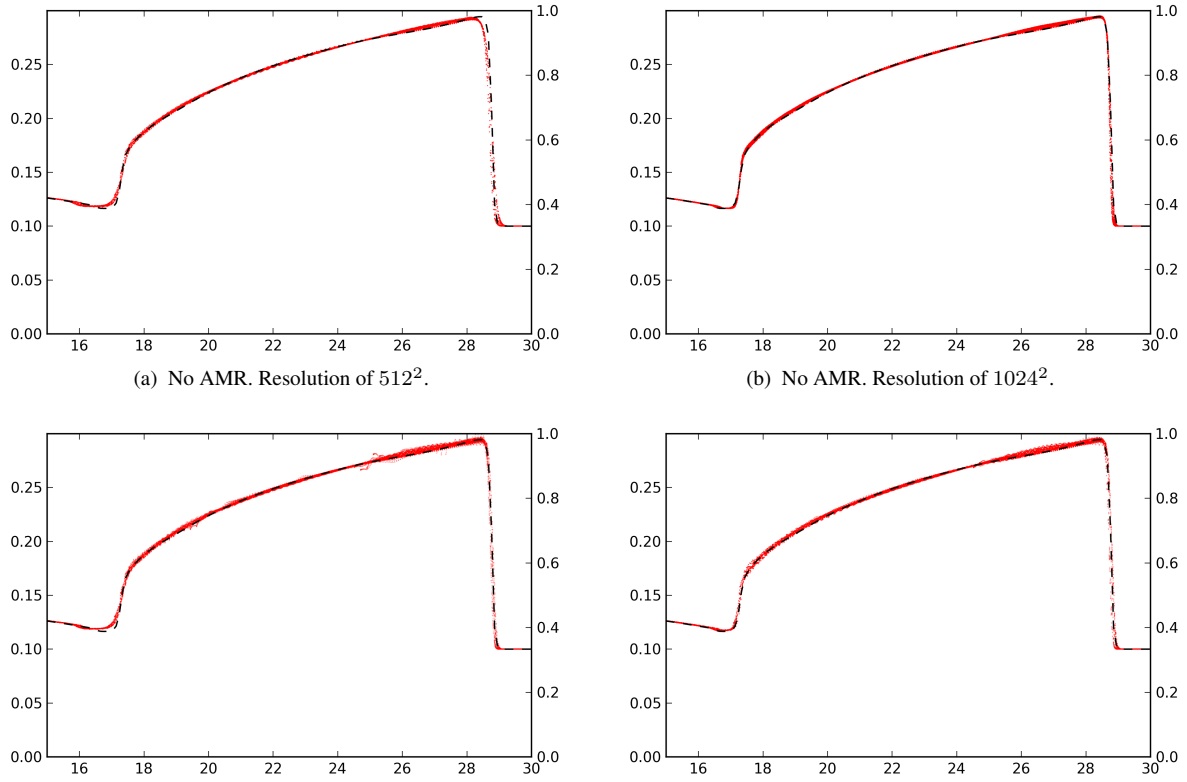
child domains before the shock arrives, however, these anomalies disappear. Even though there still is a larger radial dissymmetry than for the non-AMR simulations, we also see that the shock is captured with the resolution of the high-resolution no AMR reference.

We have also run this test without flux correction (both parent-child and child-child) and with a fixed time-step size in an attempt to reduce the radial dissymmetry even more. However, the results from these test runs showed no significant difference from the AMR-run with padding. For ten seconds of simulation time, the AMR shock-tracking without padding is over 2.2 times faster than using the highest resolution for the whole domain, at the expense of a few minor artifacts in the solution.

4.5 Optimization: Effect of reducing Δt

In this test, we will investigate the effect of reducing the time-step size on a parent grid to avoid extremely small time steps on its children. The computational setup is as in the previous test (see Figure 10b). One child grid covering 25% of the root grid is used, and the shock never travels out of this child grid during the simulation runs, which are run up to time one second using second-order Runge–Kutta time integration. Different factors for limiting Δt, so that the Courant number stays below its maximal value, have been tested. A representable subset of the tested factors are shown in Figure 15. Considering that the areas of the solution with the highest velocities typically are the same areas one wants to refine, these results are promising. However, we have only tested using two levels of refinement (including the root grid), and it is not trivial to implement this strategy for more than two levels of refinement. This optimization if also highly problem dependent, and further investigation should try to define some factor or range of factors that is proven viable in a broader selection of test setups.

Although further investigation is necessary, some preliminary conclusions can be made. We see that a domain of a certain size (more than 65 000 cells in this example) is necessary before the Δt-limiting produces stable results, and that the stability of the optimization also seems dependent on the size of the limiting factor. The optimal size of the limiting factor would be such that the last very small step on the child grid is completely eliminated, but no larger. Considering that all time-step sizes are varying throughout the simulation, and that some child time-step series will not even have a very small last time step, this is a challenging task to accomplish. It seems likely that this needs to be an auto-tuned parameter, adjusted after a certain number of time steps, analog to the early-exit parameter discussed in [7].



(a) No AMR. Resolution of 512^2.

(b) No AMR. Resolution of 1024^2.

(c) AMR without padding of new child grids.

(d) AMR with padding of new child grids.

Fig. 14: Scatter plot of water elevation values (m) as a function of radius (m) from the center of the circular dam after seven seconds of simulation time. The dotted line is a cross section of the 1024^2 simulation, taken throught the center of the circular dam, and the dots are the simulation results for each of the different runs. Notice that both AMR simulations capture the shock position accurately and that padding reduces the loss of radial symmetry.

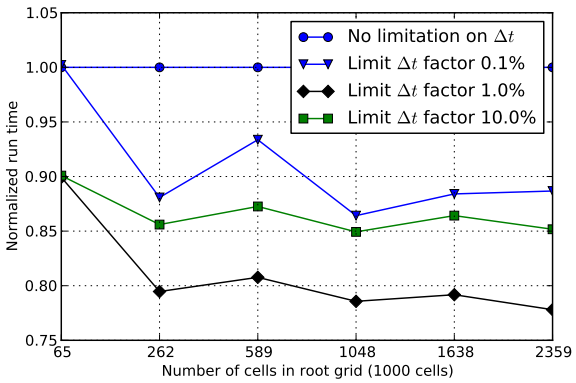


Fig. 15: Comparison of wall clock time for simulation runs with and without limit on root Δt. All time values represent the best performance of five consecutive runs, and have been normalized with respect to the simulation run without limit on root Δt for each domain size.

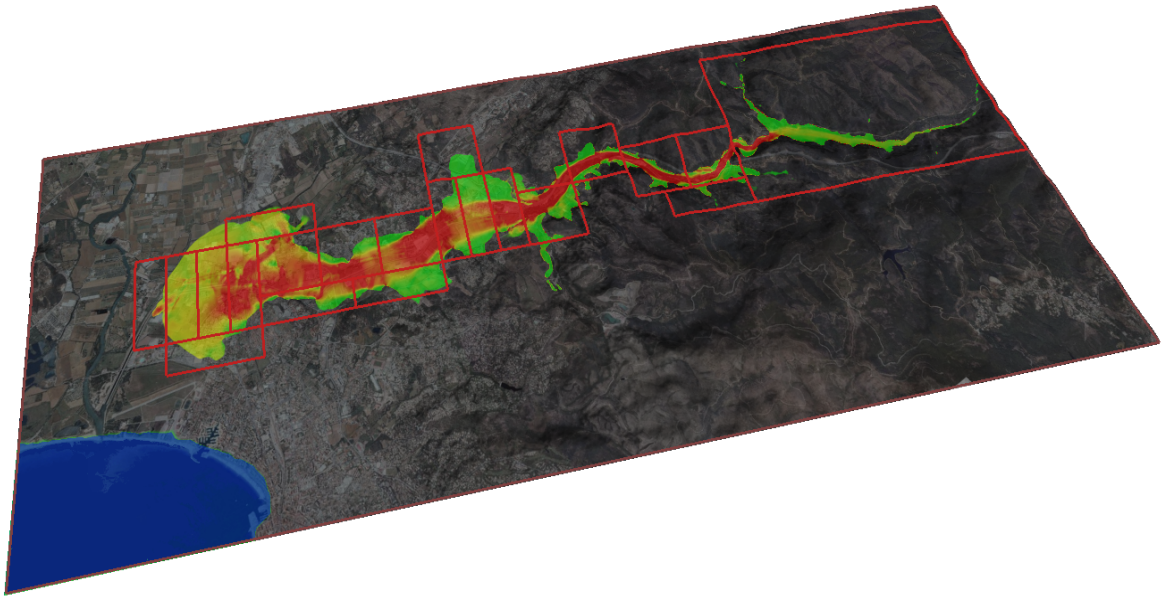


Fig. 16: Visualization of water velocity in the 1959 Malpasset dam break case. Child grids are added dynamically to cover the flood as it progresses.

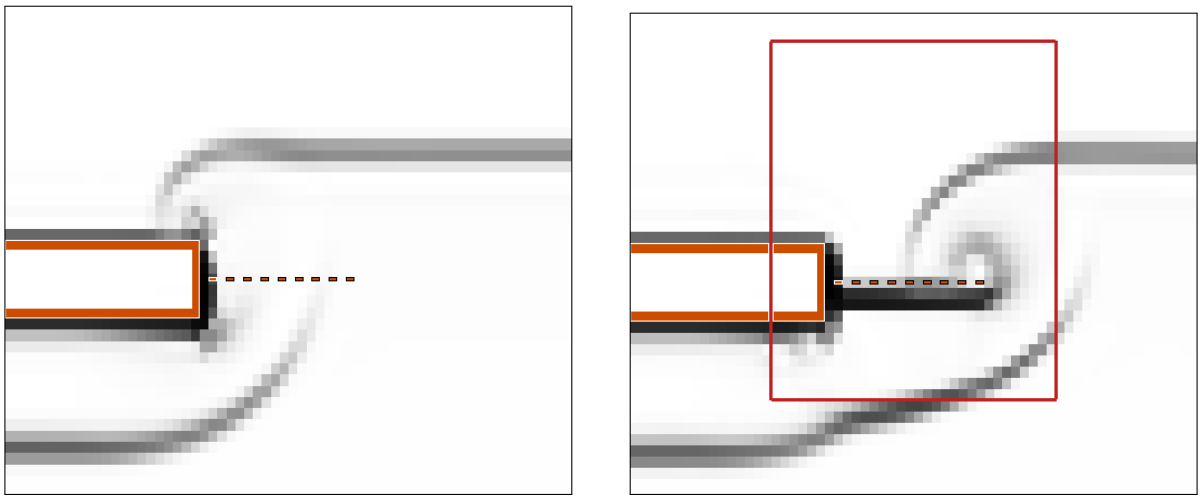
4.6 Real-world case

Simulation of a real-world case; the 1959 Malpasset dam break in southern France. This example demonstrates the code's ability to cope with a real-world case including complex bathymetry, bed shear-stress friction, and high velocities, while still conserving mass. Figure 16 shows the simulation when the flooding has reached the city of Fréjus, and as we can see, the child grids completely covers the flood. To ensure refinement once water enters dry areas, we flag cells for refinement when the water level exceeds 1×10^{-4} m. Initially, the water is contained in the dam visible in the upper child grid, wall boundaries are used, the Manning friction coefficient is set to 0.033, and second-order accurate Runge–Kutta time integration is used. A refinement check is performed every 50th time step, the minimum child grid size is set to three tiles, and two levels of refinement are used (including the root grid). If we let the simulation run to time 20 minutes, the AMR solver runs four times faster than the original simulator. After 30 minutes, a quite large fraction of the global domain has been covered by child grids (as depicted in Figure 16), and the speedup has dropped to 3.6 times.

4.7 Improved spatial resolution

This example is constructed to show one of the main strengths of the AMR algorithm: introducing improved spatial resolution locally enables the simulator to capture more details in both the bathymetry and the conserved quantities, for a less computational cost than increasing the resolution of the global domain. The domain is a modified version of the one shown in Figure 7. All global boundaries are now walls, and the domain size is 250 m $\times$ 200 m. The bump in the bathymetry is raised to model a breakwater that narrows to 2 m in width after 75 m and disappears completely after 125 m. For the AMR simulation, we use one child grid covering 32 $\times$ 32 cells of the root grid, offset by 16 cells from the global boundary in both spatial dimensions. The bathymetry is generated procedurally on each refinement level, and second-order accurate Runge–Kutta time integration is used.

As we can see in Figure 17, because of insufficient resolution, the original simulator fails to properly model the narrow portion of the breakwater. In the AMR simulation, however, the narrow part is present, and dramatically changes the solution. Many real-world cases, e.g., tsunamis and storm surges, feature natural hierarchies of different scales, from the open ocean to coast land, and connected river systems, swamplands, etc. In these cases it would be too computationally demanding to solve the full domain using the highest resolution, and AMR is one possible solution to this problem. It should be noted that this case is synthetic in the sense that a very small root grid resolution is used. This should therefore be viewed as being a very small part of a much bigger domain.



(a) Without AMR. (b) With AMR.

Fig. 17: AMR is used to increase resolution in parts of the domain that hold a subgrid feature representing a narrow breakwater. The breakwater structure cannot be represented at the root grid resolution without AMR, which leads to erroneous simulation results.

5 Conclusions

In this article, we have implemented a tile-based AMR algorithm using a well-balanced, high-resolution finite-volume scheme so that it runs fully on a GPU. The resulting simulator conserves mass and has simple shock-tracking capability. The AMR implementation has been thorougly tested and verified using analytical solutions, synthetic dam breaks, and real data from the Malpasset dam break. The results show that the code has excellent hardware utilization and that the accuracy on the child grids with the highest resolution (herein, we use at most three levels in total) is close to what would be obtained on a grid with full global refinement. The simulator has been carefully designed using modern software principles so that the code should be easy to use and understand and that simulations should be fast to setup and run.

Acknowledgements This work is supported in part by the Research Council of Norway through grant number 180023 (Parallel3D), and in part by the Centre of Mathematics for Applications at the University of Oslo (Sætra and Lie).

References

1. Berger, M., LeVeque, R.: Adaptive mesh refinement for two-dimensional hyperbolic systems and the AMRCLAW software. SIAM J. Numer. Anal **35**, 2298–2316 (1998)
2. Berger, M.J., Colella, P.: Local adaptive mesh refinement for shock hydrodynamics. Journal of Computational Physics **82**, 64–84 (1989). DOI 10.1016/0021-9991(89)90035-1
3. Berger, M.J., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. Journal of Computational Physics **53**(3), 484–512 (1984). DOI 10.1016/0021-9991(84)90073-1
4. Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., Storaasli, O.: State-of-the-art in heterogeneous computing. Journal of Scientific Programming **18**(1), 1–33 (2010)
5. Brodtkorb, A.R., Hagen, T.R., Sætra, M.L.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing **73**(1), 4–13 (2013). DOI 10.1016/j.jpdc.2012.04.003
6. Brodtkorb, A.R., Sætra, M.L.: Explicit shallow water simulations on GPUs: Guidelines and best practices. In: XIX International Conference on Water Resources, CMWR 2012, June 17–22, 2012. University of Illinois at Urbana-Champaign (2012)
7. Brodtkorb, A.R., Sætra, M.L., Altinakar, M.: Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. Computers & Fluids **55**(0), 1–12 (2012). DOI 10.1016/j.compfluid.2011.10.012
8. Burstedde, C., Ghattas, O., Gurnis, M., Isaac, T., Stadler, G., Warburton, T., Wilcox, L.: Extreme-scale AMR. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12. IEEE Computer Society (2010)
9. Colella, P., Graves, D.T., Johnson, J.N., Johansen, H.S., Keen, N.D., Ligocki, T.J., Martin, D.F., McCorquodale, P.W., Modiano, D., Schwartz, P.O., Sternberg, T.D., Van Straalen, B.: Chombo software package for AMR applications design document. Tech. rep., Lawrence Berkeley National Laboratory (2012)

10. Delestre, O., Lucas, C., Ksinant, P.A., Darboux, F., Laguerre, C., Vo, T.N.T., James, F., Cordier, S.: SWASHES: a compilation of shallow water analytic solutions for hydraulic and environmental studies. International Journal for Numerical Methods in Fluids **72**(3), 269–300 (2013). DOI 10.1002/fld.3741
11. Goutal, N., Maurel, F.: Proceedings of the 2nd Workshop on Dam-Break Wave Simulation. Tech. rep., Groupe Hydraulique Fluviale, Département Laboratoire National d'Hydraulique, Electricité de France (1997)
12. Harris, M.: NVIDIA GPU computing SDK 4.1: Optimizing parallel reduction in CUDA (2011)
13. Hornung, R.D., Kohn, S.R.: Managing application complexity in the SAMRAI object-oriented framework. Concurrency and Computation: Practice and Experience **14**(5), 347–368 (2002). DOI 10.1002/cpe.652
14. Jones, S.: Introduction to dynamic parallelism. GPU Technology Conference presentation S0338 (2012)
15. Khronos Group: OpenCL. `http://www.khronos.org/opencl/`
16. Kurganov, A., Petrova, G.: A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. Communications in Mathematical Sciences **5**, 133–160 (2007)
17. Lawrence Berkeley National Laboratory, Center for Computational Sciences and Engineering: BoxLib. `https://ccse.lbl.gov/BoxLib/index.html`
18. van Leer, B.: Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. Journal of Computational Physics **32**(1), 101–136 (1979). DOI 10.1016/0021-9991(79)90145-1
19. LeVeque, R.J., George, D.L., Berger, M.J.: Tsunami modelling with adaptively refined finite volume methods. Acta Numerica **20**, 211–289 (2011). DOI 10.1017/S0962492911000043
20. Li, R., Wu, S.: H-adaptive mesh method with double tolerance adaptive strategy for hyperbolic conservation laws. Journal of Scientific Computing **56**(3), 616–636 (2013). DOI 10.1007/s10915-013-9692-1
21. Lie, K.A., Noelle, S.: On the artificial compression method for second-order nonoscillatory central difference schemes for systems of conservation laws. SIAM Journal on Scientific Computing **24**(4), 1157–1174 (2003)
22. MacNeice, P., Olson, K.M., Mobarry, C., de Fainchtein, R., Packer, C.: PARAMESH: A parallel adaptive mesh refinement community toolkit. Computer Physics Communications **126**(3), 330–354 (2000). DOI 10.1016/S0010-4655(99)00501-9
23. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: Top 500 supercomputer sites. `http://www.top500.org/` (2013)
24. Nessyahu, H., Tadmor, E.: Non-oscillatory central differencing for hyperbolic conservation laws. Journal of computational physics **87**(2), 408–463 (1990)
25. Nicholaeff, D., Davis, N., Trujillo, D., Robey, R.W.: Cell-based adaptive mesh refinement implemented with general purpose graphics processing units. Tech. rep., Los Alamos National Laboratory (2012)
26. NVIDIA: NVIDIA CUDA programming guide 5.0 (2012)
27. NVIDIA: NVIDIA GeForce GTX 680. Tech. rep., NVIDIA Corporation (2012)
28. NVIDIA: CUDA community showcase. `http://www.nvidia.com/object/cuda_showcase_html.html` (2013)
29. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU computing. Proceedings of the IEEE **96**(5), 879–899 (2008). DOI 10.1109/JPROC.2008.917757
30. Sætra, M.L.: Shallow water simulation on GPUs for sparse domains. In: A. Cangiani, R.L. Davidchack, E. Georgoulis, A.N. Gorban, J. Levesley, M.V. Tretyakov (eds.) Numerical Mathematics and Advanced Applications 2011, pp. 673–680. Springer Berlin Heidelberg (2013). DOI 10.1007/978-3-642-33134-3_71
31. Sætra, M.L., Brodtkorb, A.R.: Shallow water simulations on multiple GPUs. In: K. Jónasson (ed.) Applied Parallel and Scientific Computing, *Lecture Notes in Computer Science*, vol. 7134, pp. 56–66. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-28145-7_6
32. Schive, H.Y., Tsai, Y.C., Chiueh, T.: GAMER: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. The Astrophysical Journal Supplement Series **186**(2), 457–484 (2010). DOI 10.1088/0067-0049/186/2/457
33. Sweby, P.K.: High resolution schemes using flux limiters for hyperbolic conservation laws. SIAM journal on numerical analysis **21**(5), 995–1011 (1984)
34. The Enzo Project: Enzo. `http://enzo-project.org/`
35. Wang, P., Abel, T., Kaehler, R.: Adaptive mesh fluid simulations on GPU. New Astronomy **15**(7), 581–589 (2010). DOI 10.1016/j.newast.2009.10.002