How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine

Trond Runar Hagen, Martin O. Henriksen, Jon M. Hjelmervik, and Knut–Andreas Lie

Summary. The paper has two main themes: The first theme is to give the reader an introduction to modern methods for systems of conservation laws. To this end, we start by introducing two classical schemes, the Lax–Friedrichs scheme and the Lax–Wendroff scheme. Using a simple example, we show how these two schemes fail to give accurate approximations to solutions containing discontinuities. We then introduce a general class of semi-discrete finite-volume schemes that are designed to produce accurate resolution of both smooth and nonsmooth parts of the solution. Using this special class we wish to introduce the reader to the basic principles used to design modern high-resolution schemes. As examples of systems of conservation laws, we consider the shallow-water equations for water waves and the Euler equations for the dynamics of an ideal gas.

The second theme in the paper is how programmable graphics processor units (GPUs or graphics cards) can be used to efficiently compute numerical solutions of these systems. In contrast to instruction driven micro-processors (CPUs), GPUs subscribe to the data-stream-based computing paradigm and have been optimised for high throughput of large data streams. Most modern numerical methods for hyperbolic conservation laws are explicit schemes defined over a grid, in which the unknowns at each grid point or in each grid cell can be updated independently of the others. Therefore such methods are particularly attractive for implementation using data-stream-based processing.

1 Introduction

Evolution of conserved quantities such as mass, momentum and energy is one of the fundamental physical principles used to build mathematical models in the natural sciences. The resulting models often appear as systems of quasi-linear hyperbolic partial differential equations (PDEs) in divergence form

$$Q_t + \nabla \cdot F(Q) = 0, \tag{1}$$

where $Q \in \mathbb{R}^m$ is the set of conserved quantities and F is a (nonlinear) flux function. This class of equations is commonly referred to as 'hyperbolic conservation laws' and govern a broad spectrum of physical phenomena in acoustics, astrophysics, cosmology, combustion theory, elastodynamics, geophysics, multiphase flow, and nonlinear material science, to name a few [9, 28, 54, 8, 33, 50, 15]. The simplest example is the linear transport equation,

$$q_t + aq_x = 0,$$

which describes the passive advection of a conserved scalar quantity q. However, the most studied case is the nonlinear Euler equations that describe the dynamics of a compressible and inviscid fluid, typically an ideal gas.

In scientific and industrial applications, the evolution of conserved quantities is often not the only modelling principle. Real-life models therefore tend to be more complex than the quasilinear equation (1). Chemical reactions, viscous forces, spatial inhomogeneities, phase transitions, etc, give rise to other terms and lead to more general evolution models of the form

$$A(Q)_t + \nabla \cdot F(x,t,Q) = S(x,t,Q) + \nabla \cdot \big(D(Q,x,t)\nabla Q\big).$$

Similarly, the evolution equation (1) is often coupled with other models and side conditions. As an example, we refer to the discussion of reservoir simulation elsewhere in this book [1, 2], where injection of water into an oil reservoir is described by a system consisting of an elliptic equation for the fluid pressure and a hyperbolic equation for the transport of fluid phases. Here we will mainly focus on simplified models of the form (1) and discuss some of the difficulties involved in solving such models numerically.

The solutions of nonlinear hyperbolic conservation laws exhibit very singular behaviour and admit various kinds of discontinuous and nonlinear waves, such as shocks, rarefactions, propagating phase boundaries, fluid and material interfaces, detonation waves, etc. Understanding these nonlinear phenomena has been a constant challenge to mathematicians, and research on the subject has strongly influenced developments in modern applied mathematics. In a similar way, computing numerically the nonlinear evolution of possibly discontinuous waves has proved to be notoriously difficult. Two seminal papers are the papers by Lax [29] from 1954, Godunov [16] from 1959, and by Lax and Wendroff [30] from 1960. Classical high-order discretisation schemes tend to generate unphysical oscillations that corrupt the solution in the vicinity of discontinuities, while low-order schemes introduce excessive numerical diffusion. To overcome these problems, Harten [19] introduced so-called high-resolution methods in 1983. These methods are typically based upon a finite-volume formation and have been developed and applied successfully in many disciplines [33, 50, 15]. High-resolution methods are designed to capture discontinuous waves accurately, while retaining high-order accuracy on smooth parts of the solution. In this paper we present a particular class of high-resolution methods and thereby try to give the reader some insight into a fascinating research field.

High-resolution methods can also be applied to more complex flows than those described by (1), in which case they need to be combined with additional numerical techniques. We will not go into this topic here, but rather refer the interested reader to e.g., the proceedings from the conference series "Finite Volumes for Complex Applications" [6, 53, 21].

High-resolution methods are generally computationally expensive; they are often based on explicit temporal discretisation, and severe restrictions on the time-steps are necessary to guarantee stability. On the other hand, the methods have a natural parallelism since each cell in the grid can be processed independently of its neighbours, due to the explicit time discretisation. High-resolution methods are therefore ideal candidates for distributed and parallel computing. The computational domain can be split into overlapping sub-domains of even size. The overlaps produce the boundary-values needed by the spatial discretisation stencils. Each processor can now update its sub-domain independently; the only need for communication is at the end of the time-step, where updated values must be transmitted to the neighbouring sub-domains. To achieve optimal load-balancing (and linear speedup), special care must be taken when splitting into sub-domains to reduce communication costs.

In this paper we will take a different and somewhat unorthodox approach to parallel computing; we will move the computation from the CPU and exploit the computer's graphics processor unit (GPU)—commonly referred to as the 'graphics card' in everyday language—as a parallel computing device. This can give us a speedup of more than one order of magnitude. Our motivation is that modern graphics cards have an unrivalled ability to process floating-point data. Most of the transistors in CPUs are consumed by the L2 cache, which is the fast computer memory residing between the CPU and the main memory, aimed at providing faster CPU access to instructions and data in memory. The majority of transistors in a GPU, on the other hand, is devoted to processing floating points to render (or rasterize) graphical primitives (points, lines, triangles, and quads) into frame buffers. The commodity GPUs we will use in this study (NVIDIA GeForce 6800 and 7800) have 16 or 24 parallel pipelines for processing fragments of graphical primitives, and each pipeline executes operations on vectors of length four with a clock frequency up to 500 MHz. This gives an *observed* peak performance of 54 and 165 Gflops, respectively, on synthetic tests, which is one order of magnitude larger than the *theoretical* 15 Gflops performance of an Intel Pentium 4 CPU. How we actually exploit this amazing processing capability for the numerical solution of partial differential equations is the second topic of the paper. A more thorough introduction to using graphics cards for general purpose computing is given elsewhere in this book [10] or in a paper by Rumpf and Strzodka [46]. Currently, the best source for information about this thriving field is the state-of-the-art survey paper by Owens et al. [42] and the GPGPU website `http://www.gpgpu.org/`.

The aim of the paper is two-fold: we wish to give the reader a taste of modern methods for hyperbolic conservation laws and an introduction to the use of graphics cards as a high-performance computational engine for partial differential equations. The paper is therefore organised as follows: Section 2 gives an introduction to hyperbolic conservation laws. In Section 3 we introduce two classical schemes and discuss how to implement them on the GPU. Then in Section 4 we introduce a class of high-resolution schemes based upon a semi-discretisation of the conservation law (1), and show some examples of implementations on GPUs. In Sections 3.5 and 5 we give numerical examples and discuss the speedup observed when moving computations from the CPU to the GPU. Finally, Section 6 contains a discussion of the applicability of GPU-based computing for more challenging problems.

2 Hyperbolic Conservation Laws

Systems of hyperbolic conservation laws arise in many physical models describing the evolution of a set of conserved quantities $Q \in \mathbb{R}^m$. A conservation law states that the rate of change of a quantity within a given domain Ω equals the flux over the boundaries $\partial\Omega$. When advective transport or wave motion are the most important phenomena, and dissipative, second-order effects can be neglected, the fluxes only depend on time, space and the conserved quantities. In the following we assume that $\Omega \in \mathbb{R}^2$ with outer normal $n = (n_x, n_y)$ and that the flux has two components (F, G), where each component is a function of Q, x, y, and t. Thus, the conservation law in integral form reads

$$\frac{d}{dt}\iint_{\Omega} Q\,dxdy + \int_{\partial\Omega} (F,G)\cdot(n_x,n_y)\,ds = 0. \qquad (2)$$

Using the divergence theorem and requiring the integrand to be identically zero, the system of conservation laws can be written in differential form as

$$\partial_t Q + \partial_x F(Q,x,y,t) + \partial_y G(Q,x,y,t) = 0. \qquad (3)$$

In the following, we will for simplicity restrict our attention to cases where the flux function only depends on the conserved quantity Q. When the Jacobi matrix $\partial_Q(F,G)\cdot n$ has only real eigenvalues and a complete set of real eigenvectors for all unit vectors n, the system is said to be hyperbolic, and thus belongs to the class of 'hyperbolic conservation laws'. These equations have several features that in general make them hard to analyse and solve numerically. For instance, a hyperbolic system may form discontinuous solutions even from smooth initial data and one therefore considers weak solutions of (3) that satisfy the equation in the sense of distributions; that is, $Q(x,y,t)$ is a weak solution of (3) if

$$\iiint\limits_{\mathbb{R}^2\times\mathbb{R}^+} \Big(Q\partial_t\phi + F(Q)\partial_x\phi + G(Q)\partial_y\phi\Big)\,dxdydt = 0. \qquad (4)$$

for any smooth test function $\phi(x,y,t) \in \mathbb{R}^m$ with compact support on $\mathbb{R}^2 \times \mathbb{R}^+$. Weak solutions are not necessarily unique, which means that the conservation law in general must be equipped with certain side-conditions to pick the physically correct solution. These conditions may come in the form of entropy inequalities, here in weak integral form

$$\iiint\limits_{\mathbb{R}^2\times\mathbb{R}^+} \Big(\eta(Q)\phi_t + \psi(Q)\phi_x + \varphi(Q)\phi_y\Big)\,dxdydt \\ + \iint\limits_{\mathbb{R}^2} \phi(x,y,0)\eta(Q(x,y,0))\,dxdy \geq 0,$$

where $\eta(Q)$ denotes the entropy, (ψ,φ) the associated pair of entropy fluxes, and ϕ is a positive test function. The concept of entropy inequalities comes from the second law of thermodynamics, which requires that the entropy is nondecreasing. Generally, entropy is conserved when the function Q is smooth and increases over discontinuities. For scalar equations it is common to use a special family of entropy functions and fluxes due to Kružkov [24],

$$\eta_k(q) = |q-k|, \qquad \begin{cases} \psi_k(q) = \operatorname{sign}(q-k)[F(q)-F(k)], \\ \varphi_k(q) = \operatorname{sign}(q-k)[G(q)-G(k)], \end{cases}$$

where k is any real number.

A particular feature of hyperbolic systems is that all disturbances have a finite speed of propagation. For simplicity, let us consider a linear system in one-dimension,

$$U_t + AU_x = 0$$

with initial data $U(x,0) = U_0(x)$. Hyperbolicity means that the constant coefficient matrix A is diagonalisable with real eigenvalues λ_i, i.e., we can write A as $R\Lambda R^{-1}$, where $\Lambda = \operatorname{diag}(\lambda_1,\ldots,\lambda_m)$, and each column r_i of R is a right eigenvector of A. Premultiplying the conservation law by R^{-1} and defining $V = R^{-1}U$, we end up with m independent scalar equations

$$\partial_t V + \Lambda\partial_x V = 0.$$

Here V is called the vector of *characteristic variables*, and the decomposition $R\Lambda R^{-1}$ is called a *characteristic decomposition.*

Each component v_i of V satisfies a linear transport equation $(v_i)_t + \lambda_i(v_i)_x = 0$, with initial data $v_i^0(x)$ given from $V^0 = R^{-1}U_0$. Each linear transport equation is solved by $v_i(x,t) = v_i^0(x-\lambda_i t)$, and hence we have that

$$U(x,t) = \sum_{i=0}^{m} v_i^0(x-\lambda_i t)r_i.$$

This equation states that the solution consists of a superposition of m simple waves that are advected independently of each other along so-called characteristics. The propagation speed of each simple wave is given by the corresponding eigenvalue. In the nonlinear case, we can write $Q_t + A(Q)Q_x = 0$, where $A(Q)$ denotes the Jacobi matrix dF/dQ. However, since the eigenvalues and eigenvectors of $A(Q)$ now will depend on the unknown vector Q, we cannot repeat the analysis from the linear case and write the solution in a simple closed form. Still, the eigenvalues (and eigenvectors) give valuable information of the local behaviour of the solution, and we will return to them later.

Before we introduce numerical methods for conservation laws, we will introduce two canonical models. The interested reader can find a lot more background material in classical books like [9, 28, 54, 8] or in one of the recent textbooks numerical methods for conservation laws[15, 22, 33, 50] that introduce both theory and modern numerical methods. A large number of recent papers on mathematical analysis and numerical methods for hyperbolic conservation laws can be found on the *Conservation Laws Preprint Server* in Trondheim, `http://www.math.ntnu.no/conservation/`. Excellent images of a large number of flow phenomena can be found in Van Dyke's *Album of Fluid Motion* [11] or by visiting e.g., the *Gallery of Fluid Mechanics*, `http://www.galleryoffluidmechanics.com/`.

2.1 The Shallow-Water Equations

In its simplest form, the shallow-water equations read

$$Q_t + F(Q)_x = \begin{bmatrix} h \\ hu \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}_x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{5}$$

The two equations describe conservation of mass and momentum. This system of equations has applications to a wide range of phenomena, including water waves in shallow waters, (granular) avalanches, dense gases, mixtures of materials, and atmospheric flow. If the application is water waves, h denotes water depth, u the depth-averaged velocity, and g is the acceleration of gravity.

The shallow-water equations have the following eigenvalues

$$\lambda(Q) = u \pm \sqrt{gh}. \tag{6}$$

and possess only nonlinear waves in two families corresponding to the slow and the fast eigenvalue. These nonlinear waves may be continuous transitions called rarefactions or propagating discontinuities called shocks. A rarefaction wave satisfies a differential equation on the form

$$\frac{dF(Q)}{dQ} Q'(\xi) = \lambda(Q) Q'(\xi), \qquad \xi = \xi(x,t) = (x - x_0)/(t - t_0),$$

whereas a shock between right and left values Q_L and Q_R satisfies an algebraic equation of the form

$$(Q_L - Q_R)s = F(Q_L) - F(Q_R),$$

where s is the shock speed. (This algebraic equation is called the Rankine–Hugoniot condition, see [31].)

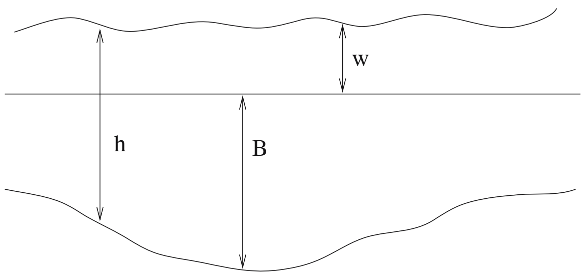

Fig. 1. The quantities involved in the shallow-water equations.

The shallow-water equations form a depth-integrated model for free surface flow of a fluid under the influence of gravity, see Figure 1. In the model, it is assumed that the height of the top surface above the bottom is sufficiently small compared with some characteristic length of motion. Moreover, vertical motion within the fluid is ignored, and we have hydrostatic balance in the vertical direction. For water waves over a bottom topography (bathymetry) $B(x,y)$ in two spatial dimensions, this gives rise to the following inhomogeneous system

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix}. \qquad (7)$$

The right-hand side of (7) models the influence of a variable bathymetry. Figure 1 illustrates the situation and the quantities involved. The function $B = B(x,y)$ describes the bathymetry, while $w = B(x,y,t)$ models the water surface elevation, and we have that $h = w - B$. For brevity, we will refer to this system as

$$Q_t + F(Q)_x + G(Q)_y = H(Q,Z).$$

The system has the following eigenvalues,

$$\lambda(Q,n) = (u,v)\cdot n \pm \sqrt{gh},\ (u,v)\cdot n$$

where a linear shear wave with characteristic wave-speed v has been introduced in addition to the two nonlinear waves described above. For simplicity, we will henceforth assume a flat bathymetry, for which $H(Q,Z) \equiv 0$. Example 6 contains a short discussion of variable bathymetry.

2.2 The Euler Equations

Our second example of a hyperbolic system is given by the Euler equations for an ideal, polytropic gas. This system is probably *the* canonical example of a hyperbolic system of conservation laws that describes conservation of mass, momentum (in each spatial direction) and energy. In one spatial dimension the equations read

$$Q_t + F(Q)_x = \begin{bmatrix} \rho \\ \rho u \\ E \end{bmatrix}_t + \begin{bmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{bmatrix}_x = 0. \tag{8}$$

and similarly in two dimensions

$$\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}_t + \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(E+p) \end{bmatrix}_x + \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(E+p) \end{bmatrix}_y = 0. \tag{9}$$

Here ρ denotes density, u and v velocity in x- and y- directions, p pressure, and E is the total energy (kinetic plus internal energy) given by

$$E = \tfrac{1}{2}\rho u^2 + p/(\gamma - 1), \qquad \text{or} \qquad E = \tfrac{1}{2}\rho(u^2 + v^2) + p/(\gamma - 1).$$

The gas constant γ depends on the species of the gas, and is, for instance, equal to 1.4 for air. The eigenvalues of (8) are

$$\lambda(Q, n) = (u, v) \cdot n \pm \sqrt{\gamma p/\rho},\ (u, v) \cdot n,$$

giving three waves: a slow nonlinear wave, a linear contact wave, and a fast nonlinear wave. In two space dimensions $\lambda(Q, n) = (u, v) \cdot n$ is a double eigenvalue corresponding both to a contact wave and a shear wave. A contact wave is characterised by a jump in density together with no change in pressure and velocity, i.e., ρ is discontinuous and p and $(u, v) \cdot n$ are constant, whereas a shear wave means a discontinuity in the tangential velocity $(u, v) \cdot n^{\perp}$.

3 Two Classical Finite Volume Schemes

A standard approach to numerical schemes for PDEs is to look for a *point-wise* approximation and replace temporal and spatial derivatives by finite differences. Alternatively, we can divide the domain Ω into a set of finite sub-domains Ω_i and then use the integral form of the equation to compute approximations to the *cell average* of Q over each grid cell Ω_i. In one spatial dimension, the cell-average is given by

$$Q_i(t) = \frac{1}{\Delta x_i} \int_{x_{i-1/2}}^{x_{i+1/2}} Q(x, t)\, dx, \tag{10}$$

and similarly in two dimensions

$$Q_i(t) = \frac{1}{|\Omega_i|} \iint_{\Omega_i} Q(x,y,t)\,dxdy. \tag{11}$$

The corresponding schemes are called *finite-volume schemes*.

To derive equations for the evolution of cell averages $Q_i(t)$ in one spatial dimension, we impose the conservation law on integral form (2) on the grid cell $\Omega_i = [x_{i-1/2}, x_{i+1/2}]$, which then reads

$$\frac{d}{dt} \int_{x_{i-1/2}}^{x_{i+1/2}} Q(x,t)\,dx = F\big(Q(x_{i-1/2},t)\big) - F\big(Q(x_{i+1/2},t)\big) \tag{12}$$

Using the integral form of the equation rather than the differential form is quite natural, as it ensures that also the discrete approximations satisfy a conservation property.

From (12) we readily obtain a set of ordinary differential equations for the cell averages $Q_i(t)$,

$$\frac{d}{dt} Q_i(t) = -\frac{1}{\Delta x_i}\big[F_{i+1/2}(t) - F_{i-1/2}(t)\big], \tag{13}$$

where the fluxes across the cell-boundaries are given by

$$F_{i\pm1/2}(t) = F\big(Q(x_{i\pm1/2},t)\big). \tag{14}$$

To derive schemes for the two-dimensional shallow-water and Euler equations we will follow a similar approach. We will assume for simplicity that the finite volumes are given by the grid cells in a uniform Cartesian grid, such that

$$\Omega_{ij} = \big\{\, (x,y) \in \big[(i-1/2)\Delta x, (i+1/2)\Delta x\big] \times \big[(j-1/2)\Delta y, (j+1/2)\Delta y\big] \,\big\}.$$

To simplify the presentation, we henceforth assume that $\Delta x = \Delta y$. Then the semi-discrete approximation to (2) reads

$$\begin{aligned} \frac{d}{dt} Q_{ij}(t) = L_{ij}(Q(t)) = &-\frac{1}{\Delta x}\Big[F_{i+1/2,j}(t) - F_{i-1/2,j}(t)\Big] \\ &-\frac{1}{\Delta y}\Big[G_{i,j+1/2}(t) - G_{i,j-1/2}(t)\Big], \end{aligned} \tag{15}$$

where $F_{i\pm1/2,j}$ and $G_{i,j\pm1/2}$ are numerical approximations to the fluxes over the cell edges

$$\begin{aligned} F_{i\pm1/2,j}(t) &\approx \frac{1}{\Delta y} \int_{y_{j-1/2}}^{y_{j+1/2}} F\big(Q(x_{i\pm1/2},y,t)\big)\,dy, \\ G_{i,j\pm1/2}(t) &\approx \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} G\big(Q(x,y_{j\pm1/2},t)\big)\,dx. \end{aligned} \tag{16}$$

Equation (15) is a set of ordinary differential equations (ODEs) for the time-evolution of the cell-average values Q_{ij}.

Modern high-resolution schemes are often derived using a finite-volume framework, as we will see in Section 4. To give the reader a gentle introduction to the concepts behind high-resolution schemes we start by deriving two classical schemes, the Lax–Friedrichs [29] and the Lax–Wendroff schemes [30], using the semi-discrete finite-volume framework. The exact same schemes can be derived using a finite-difference framework, but then the interpretation of the unknown discrete quantities would be different. The distinction between finite-difference and finite-volume schemes is not always clear in the literature, and many authors tend to use the term 'finite-volume scheme' to denote conservative schemes derived using finite differences. By conservative we mean a scheme that can be written on conservative form (in one spatial dimension):

$$Q_i^{n+1} = Q_i^n - r(F_{i+1/2} - F_{i-1/2}), \tag{17}$$

where $Q_i^n = Q_i(n\Delta t)$ and $r = \Delta t/\Delta x$. The importance of the conservative form was established through the famous Lax–Wendroff theorem [30], which states that if a sequence of approximations computed by a consistent and *conservative* numerical method converges to some limit, then this limit is a weak solution of the conservation law.

To further motivate this form, we can integrate (12) from t to $t+\Delta t$ giving

$$\begin{aligned} Q_i(t+\Delta t) &= Q_i(t) \\ &\quad - \frac{1}{\Delta x_i}\Big[\int_t^{t+1\Delta t} F(Q(x_{i+1/2},t))\,dt - \int_t^{t+1\Delta t} F(Q(x_{i-1/2},t))\,dt\Big]. \end{aligned}$$

By summing (17) over all cells (for $i = -M : N$) we find that the cell averages satisfy a discrete conservation principle

$$\sum_{i=-M}^{N} Q_i^{n+1}\Delta x = \sum_{i=-M}^{N} Q_i^n \Delta x - \Delta t(F_{N+1/2} - F_{-M-1/2})$$

In other words, our discrete approximations mimic the conservation principle of the unknown function Q since the sum $\sum_i Q_i \Delta x$ approximates the integral of Q.

3.1 The Lax–Friedrichs Scheme

We will now derive our first numerical scheme. For simplicity, let us first consider one spatial dimension only. We start by introducing a time-step Δt and discretise the ODE (13) by the forward Euler method. Next, we must evaluate the flux-integrals (14) along the cell edges. We now make the assumption that $Q(x, n\Delta t)$ is piecewise constant and equals the cell average $Q_i^n = Q_i(n\Delta t)$ inside each grid cell. Then, since all perturbations in a hyperbolic system travel with a finite wave-speed, the problem of estimating the solution at the cell interfaces can be recast as a set of locally defined initial-value problems on the form

$$Q_t + F(Q)_x = 0, \qquad Q(x,0) = \begin{cases} Q_L, & x < 0, \\ Q_R, & x > 0. \end{cases} \tag{18}$$

This is called a Riemann problem and is a fundamental building block in many high-resolution schemes. We will come back to the Riemann problem in Section 4.1. For now, we take a more simple approach to avoid solving the Riemann problem. For the Lax–Friedrichs scheme we approximate the flux function at each cell-boundary by the average of the fluxes evaluated immediately to the left and right of the boundary. Since we have assumed the function to be piecewise constant, this means that we will evaluate the flux using the cell averages in the adjacent cells,

$$F(Q(x_{i+1/2}, n\Delta t)) = \frac{1}{2}\Big(F(Q_i^n) + F(Q_{i+1}^n)\Big), \quad \text{etc.}$$

This approximation is commonly referred to as a *centred* approximation.

Summing up, our first scheme reads

$$Q_i^{n+1} = Q_i^n - \frac{1}{2}r\Big[F\big(Q_{i+1}^n\big) - F\big(Q_{i-1}^n\big)\Big].$$

Unfortunately, this scheme is notoriously unstable. To obtain a stable scheme, we can add the artificial diffusion term $(\Delta x^2/\Delta t)\partial_x^2 Q$. Notice, however, that by doing this, it is no longer possible to go back to the semi-discrete form, since the artificial diffusion terms blow up in the limit $\Delta t \to 0$. If we discretise the artificial diffusion term by a standard central difference, the new scheme reads

$$Q_i^{n+1} = \frac{1}{2}\Big(Q_{i+1}^n + Q_{i-1}^n\Big) - \frac{1}{2}r\Big[F\big(Q_{i+1}^n\big) - F\big(Q_{i-1}^n\big)\Big]. \tag{19}$$

This is the classical first-order Lax–Friedrichs scheme. The scheme is stable provided the following condition is fulfilled

$$r \max_{i,k} |\lambda_k^F(Q_i)| \leq 1, \tag{20}$$

where λ_k^F are the eigenvalues of the Jacobian matrix of the flux function F. The condition is called the CFL condition and simply states that the domain of dependence for the exact solution, which is bounded by the characteristics of the smallest and largest eigenvalues, should be contained in the domain of dependence for the discrete equation.

A two-dimensional scheme can be derived analogously. To approximate the integrals in (16) one can generally use any numerical quadrature rule. However, the midpoint rule is sufficient for the first-order Lax–Friedrichs scheme. Thus we need to know the point values $Q(x_{i\pm1/2}, y_j, n\Delta t)$ and $Q(x_i, y_{j\pm1/2}, n\Delta t)$. We assume that $Q(x, y, n\Delta t)$ is piecewise constant and equal to the cell average within each grid cell. The values of $Q(x, y, n\Delta t)$ at the midpoints of the edges can then be estimated by simple averaging of the one-sided values, like in the one dimensional case. Altogether this gives the following scheme

$$\begin{aligned} Q_{ij}^{n+1} =&\frac{1}{4}\Big(Q_{i+1,j}^n + Q_{i-1,j}^n + Q_{i,j+1}^n + Q_{i,j-1}^n\Big) \\ &- \frac{1}{2}r\Big[F\big(Q_{i+1,j}^n\big) - F\big(Q_{i-1,j}^n\big)\Big] - \frac{1}{2}r\Big[G\big(Q_{i,j+1}^n\big) - G\big(Q_{i,j-1}^n\big)\Big], \end{aligned} \tag{21}$$

which is stable under a CFL restriction of 1/2.

The Lax–Friedrichs scheme is also commonly derived using finite differences as the spatial discretisation, in which case Q_{ij}^n denotes the point values at $(i\Delta x, j\Delta y, n\Delta t)$. The scheme is very robust. This is largely due to the added numerical dissipation, which tends to smear out discontinuities. A formal Taylor analysis of a single time step shows that the truncation error of the scheme is of order two in the discretisation parameters. Rather than truncation errors, we are interested in the error at a fixed time (using an increasing number of time steps), and must divide with Δt, giving an error of order one. For smooth solutions the error measured in, e.g., the L^∞ or L^1-norm therefore decreases linearly as the discretisation parameters are refined, and hence we call the scheme *first-order.*

A more fundamental derivation of the Lax–Friedrichs scheme (21) is possible if one introduces a *staggered* grid (see Figure 9). To this end, let Q_{i+1}^n be the cell average over $[x_i, x_{i+2}]$, and Q_i^{n+1} be the cell average over the staggered cell $[x_{i-1}, x_{i+1}]$. Then (19) is the *exact* evolution of the piecewise constant data from time t^n to t^{n+1} followed by an averaging onto the staggered grid. This averaging introduces the additional diffusion discussed above. The two-dimensional scheme (21) can be derived analogously.

3.2 The Lax–Wendroff Scheme

To achieve formal second-order accuracy, we replace the forward Euler method used for the integration of (13) by the midpoint method, that is,

$$Q_i^{n+1} = Q_i^n - r\Big[F_{i+1/2}^{n+1/2} - F_{i-1/2}^{n+1/2}\Big]. \tag{22}$$

To complete the scheme, we must specify the fluxes $F_{i\pm1/2}^{n+1/2}$, which again depend on the values at the midpoints at time $t^{n+1/2} = (n + \frac{1}{2})\Delta t$. One can show that these values can be *predicted* with acceptable accuracy using the Lax–Friedrichs scheme on a grid with grid-spacing $\frac{1}{2}\Delta x$,

$$Q_{i+1/2}^{n+1/2} = \frac{1}{2}\Big(Q_{i+1}^n + Q_i^n\Big) - \frac{1}{2}r\Big[F_{i+1}^n - F_i^n\Big]. \tag{23}$$

Altogether this gives a second-order, predictor-corrector scheme called the Lax–Wendroff scheme [30]. Like the Lax–Friedrichs scheme, the scheme in

(22)–(23) can be interpreted both as a finite difference and as a finite volume scheme, and the scheme is stable under a CFL condition of 1. For extensions of the Lax–Wendroff method to two spatial dimensions, see [44, 15, 33]. To get second-order accuracy, one must include cross-terms, meaning that the scheme uses nine points. A recent version of the two-dimensional Lax–Wendroff scheme is presented in [37].

We have now introduced two classical schemes, a first-order and a second-order scheme. Before we describe how to implement these schemes on the GPU, we will illustrate their approximation qualities.

Example 1. Consider a linear advection equation on the unit interval with periodic boundary data

$$u_t + u_x = 0, \qquad u(x,0) = u_0(x), \qquad u(0,t) = u(1,t).$$

As initial data $u_0(x)$ we choose a combination of a smooth squared sine wave and a double step function,

$$u(x,0) = \sin^2\Big(\frac{x-0.1}{0.3}\pi\Big)\chi_{[0.1,0.4]}(x) + \chi_{[0.6,0.9]}(x).$$

Figure 2 shows approximate solutions after ten periods ($t = 10$) computed by Lax–Friedrichs and Lax–Wendroff on a grid with 100 cells for $\Delta t = 0.95\Delta x$. Both schemes clearly give unacceptable resolution of the solution profile. The first-order Lax–Friedrichs scheme smears both the smooth and the discontinuous part of the advected profile. The second-order Lax–Wendroff scheme preserves the smooth profile quite accurately, but introduces spurious oscillations at the two discontinuities.

Consider next the nonlinear Burgers' equation,

$$u_t + \big(\tfrac{1}{2}u^2\big)_x = 0, \qquad u(x,0) = u_0(x), \qquad u(0,t) = u(1.5,t),$$

with the same initial data as above. Burgers' equation can serve as a simple model for the nonlinear momentum equation in the shallow-water and the Euler equations. Figure 3 shows approximate solutions at time $t = 1.0$ computed by Lax–Friedrichs and Lax–Wendroff on a grid with 150 cells for $\Delta t = 0.95\Delta x$. As in Figure 2, Lax–Friedrichs smooths the nonsmooth parts of the solution (the two shocks and the kink at $x = 0.1$), whereas Lax–Wendroff creates spurious oscillations at the two shocks. On the other hand, the overall approximation qualities of the schemes are now better due to self-sharpening mechanisms inherent in the nonlinear equation.

To improve the resolution one can use a so-called composite scheme, as introduced by Liska and Wendroff [37]. This scheme is probably the simplest possible high-resolution scheme and consists of e.g., three steps of Lax–Wendroff followed by a smoothing Lax–Friedrichs step. The idea behind this scheme is that the numerical dissipation in the first-order Lax–Friedrichs scheme should dampen the spurious oscillations created by the second-order Lax–Wendroff scheme. In Figure 4 we have recomputed the results from Figures 2 and 3 using the composite scheme.

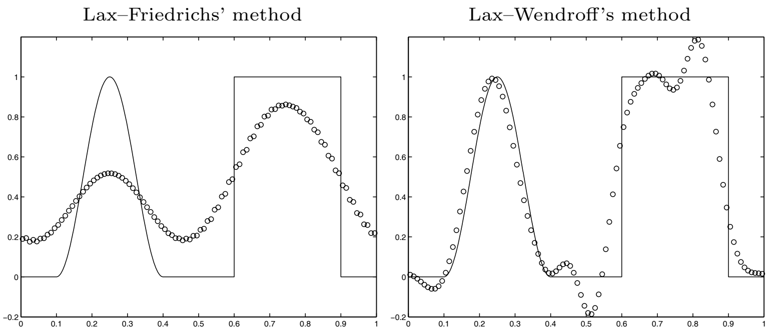

Fig. 2. Approximate solutions at time $t = 10.0$ for the linear advection equation computed by the Lax–Friedrichs and the Lax–Wendroff schemes.

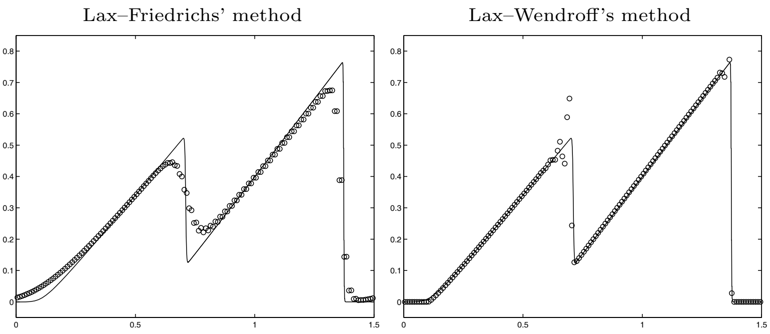

Fig. 3. Approximate solutions at time $t = 1.0$ for Burgers' equation computed by the Lax–Friedrichs and the Lax–Wendroff schemes.

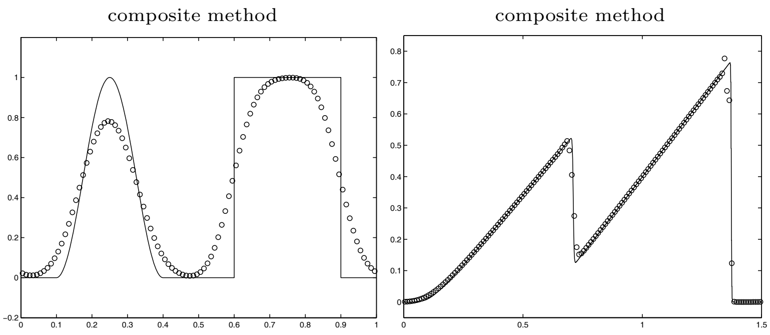

Fig. 4. Approximate solution at time $t = 10.0$ for the linear advection equation (left) and at time $t = 1.0$ for Burgers' equation (right) computed by a composite scheme.

The previous example demonstrated the potential shortcomings of classical schemes, and such schemes are seldom used in practice. On the other hand, they have a fairly simple structure and are therefore ideal starting points for discussing how to implement numerical methods on the GPU. In the next subsection, we therefore describe how to implement the Lax–Friedrichs scheme on the GPU.

3.3 Implementation on the GPU

Before we start discussing the actual implementation the Lax–Friedrichs scheme, let us look briefly into the design of a graphics card and its associated programming model.

GPUs operate in a way that is similar to a shared-memory parallel computer of the single-instruction-multiple-data (SIMD) type. In a SIMD computer, a single control unit dispatches instructions to a large number of processing nodes, and each node executes the same instruction on its own local data. SIMD computers are obviously particularly efficient when the same set of operations can be applied to all data items. If the execution involves conditional branches, there may be a performance loss since processing nodes in the worst case (depending on the underlying hardware) must execute both branches and thereafter pick the correct result. In a certain sense, explicit high-resolution schemes are good candidates for a SIMD implementation, since the update of each grid-cell involves a fixed set of given arithmetic operations and seldom introduces conditional branches in the instruction stream.

Let us now look at how computations are dispatched in a GPU. The main objective of the GPU is to render geometrical primitives (points, lines, triangles, quads), possibly combined with one or more textures (image data), as discrete pixels in a frame buffer (screen or off-screen buffer). When a primitive is rendered, the rasterizer samples the primitive at points corresponding to the pixels in the frame buffer. Per vertex attributes such as texture coordinates are set by the application for each vertex and the rasterizer calculates an interpolated value of these attributes for each point. These bundles of values then contain all the information necessary for calculating the colour of each point, and are called *fragments*. The processing pipeline is illustrated in Figure 5. We see that whereas a CPU is instruction-driven, the programming model of a GPU is data-driven. This means that individual data elements of the data-stream can be pre-fetched from memory before they are processed, thus avoiding the memory latency that tends to hamper CPUs.

Graphics pipelines contain two programmable parts, the *vertex processor* and the *fragment processor*. The vertex processor operates on each vertex without knowledge of surrounding vertices or the primitive to which it belongs. The fragment processor works in the same manner, it operates on each fragment in isolation. Both processors use several parallel pipelines, and thus work on several vertices/fragments simultaneously. To implement our numerical schemes (or a graphical algorithm) we must specify the operations in the

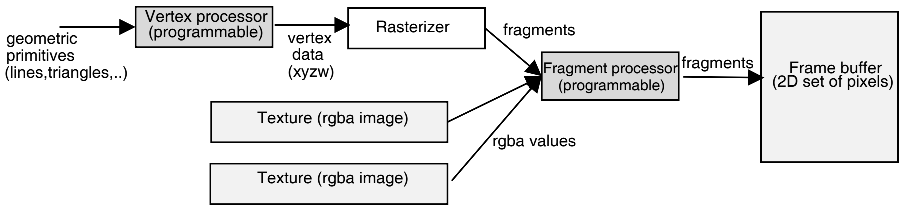
geometric primitives (lines,triangles,..)
Vertex processor (programmable)
vertex data (xyzw)
Rasterizer
fragments
Texture (rgba image)
Texture (rgba image)
rgba values
Fragment processor (programmable)
fragments
Frame buffer (2D set of pixels)

Fig. 5. A schematic of the graphics pipeline.

vertex and fragment processors. To this end, we make one or more *shaders*; these are codes written in a high-level, often C-like, language like Cg or Open GL Shading Language (GLSL), and are read and compiled by our graphical application into programs that are run by the vertex and fragment processors. Consult the references [14] and [45] for descriptions of these languages. The compiled shaders are executed for each vertex/fragment in parallel.

Our data model will be as follows: The computational domain is represented as an off-screen buffer. A geometric primitive is rendered such that a set of $N_x \times N_y$ fragments is generated which covers the entire domain. In this model, we never explicitly loop over all cells in the grid. Instead, processing of each single grid-cell is invoked implicitly by the rasterizer, which interpolates between the vertices and generates individual data for each fragment. Input field-values are realised in terms of 2D textures of size $N_x \times N_y$. Between iterations, the off-screen buffer is converted into a texture, and another off-screen buffer is used as computational domain. To solve the conservation law (3), the fragment processor loads the necessary data and performs the update of cell averages, according to (21) for the Lax–Friedrichs scheme. In this setting, the vertex processor is only used to set texture coordinates that are subsequently passed to the fragment processor. The fragment processor is then able to load the data associated with the current cell and its neighbours from the texture memory.

In the following subsection we will discuss the implementation of fragment shaders in detail for the Lax–Friedrichs scheme. However, to start the computations, it is necessary to set up the data stream to be processed and to configure the graphics pipeline. This is done in a program running on the CPU. Writing such a program requires a certain familiarity with the graphics processing jargon, and we do not go into this issue—the interested reader is referred to [46, 13, 43].

Shaders for Lax–Friedrichs

Listing 1 shows the vertex and fragment shader needed to run the Lax–Friedrichs scheme (21) for the two-dimensional shallow-water equations (7) with constant bottom topography $B \equiv 0$. In the listing, all entities given in boldface are keywords and all entities starting with `gl_` are built-in variables, constants, or attributes.

Listing 1. Fragment and vertex shader for the Lax–Friedrichs scheme written in GLSL.

```
[Vertex shader]

varying vec4 texXcoord;
varying vec4 texYcoord;
uniform vec2 dXY;

void main(void)
{
  texXcoord=gl_MultiTexCoord0.yxxx+vec4(0.0,0.0,−1.0,1.0)*dXY.x; // j, i, i−1, j+1
  texYcoord=gl_MultiTexCoord0.xyyy+vec4(0.0,0.0,−1.0,1.0)*dXY.y; // i, j, j−1, j+1

  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}


[Fragment shader]

varying vec4 texXcoord;
varying vec4 texYcoord;

uniform sampler2D QTex;

uniform float r;
uniform float halfG;

vec4 fflux(in vec4 Q)
{
  float u=Q.y/Q.x;
  return vec4( Q.y, (Q.y*u + halfG*Q.x*Q.x), Q.z*u, 0.0 );
}

vec4 gflux(in vec4 Q)
{
  float v=Q.z/Q.x;
  return vec4( Q.z, Q.y*v, (Q.z*v + halfG*Q.x*Q.x), 0.0 );
}

void main(void)
{
  vec4 QE = texture2D(QTex, texXcoord.wx);
  vec4 QW = texture2D(QTex, texXcoord.zx);
  vec4 QN = texture2D(QTex, texYcoord.xw);
  vec4 QS = texture2D(QTex, texYcoord.xz);

  gl_FragColor = 0.25*(QE + QW + QN + QS)
    − 0.5*r*(fflux(QE) − fflux(QW))
    − 0.5*r*(gflux(QN) − gflux(QS));
}
```

The vertex shader is executed once for all vertices in the geometry. Originally, the geometry is defined in a local coordinate system called object-space or model coordinates. The last line of the vertex shader transforms the vertices from object-space to the coordinate system of the camera (view coordinates, or eye-space). The two coordinate systems will generally be different. During the rendering phase only points inside the camera view (i.e., inside our computational domain) are rendered. In the shader, the model coordinates are given by `gl_Vertex` and the projected view coordinates are returned by `gl_Position`. The coordinate transformation is an affine transformation given by a global matrix called `gl_ModelViewProjectionMatrix`. (Since the camera does not move, this transformation is strictly speaking not necessary to perform for every time step and could have been done once and for all on the CPU).

The first two lines of the vertex shader compute the texture coordinates. These coordinates will be used by the fragment shader to fetch values in the current cell and its four nearest neighbours from one or more texture. The keywords `vec2` and `vec4` declare vector variables with 2 or 4 elements respectively. The construction `varying vec4 texXcoord` means that the four-component vector `texXcoord` will be passed from the vertex shader to the fragment shader. This happens as follows: The vertex shader defines one vector-value `V` at each vertex (the four corners of our computational rectangle), and the rasterizer then interpolates bilinearly between these values to define corresponding values for all fragments within the camera view. In contrast, `uniform vec2 dXY` means that `dXY` is constant for each primitive; its value must be set by the application program running on the CPU. Notice how we can access individual members of a vector as `V.x`, `V.y`, `V.z` and `V.w`, or several elements e.g., `V.xy` or `V.wzyx`. The last example demonstrates 'swizzling'; permuting the components of the vector.

We now move on to the fragment shader. Here we see the definitions of the texture coordinates `texXcoord` and `texYcoord`, used to pass coordinates from the vertex processor. `QTex` is a texture handle; think of it as a pointer to an array. The constants `r` and `halfG` represent $r = \Delta t/\Delta x$ and $\frac{1}{2}g$, respectively. To fetch the cell-averages from positions in the texture corresponding to the neighbouring cells, we use the function `texture2D`. The new cell-average is returned as the final fragment colour by writing to the variable `gl_FragColor`.

To ensure a stable scheme, we must ensure that the time step satisfies a CFL condition of the form (20). This means that we have to gather the eigenvalues from all points in the grid to determine the maximum absolute value. In parallel computing, such a gather operation is usually quite expensive and represents a bottleneck in the processing. By rethinking the gather problem as a graphics problem, we can actually use hardware supported operations on the GPU to determine the maximal characteristic speed. Recall that the characteristic speed of each wave is given by the corresponding eigenvalue. To this end we pick a suitably sized array, say 16×16, and decompose the computational domain $N_x \times N_y$ into rectangular sub-domains (quads) so that each sub-domain is associated with a 16×16 patch of the texture containing the Q-values. All quads are moved so that they coincide in world-coordinates, and hence we get $N_x N_y/16^2$ overlapping quads. By running the fragment shader in Listing 2 they are rendered to the *depth buffer*. The purpose of the depth buffer is to determine whether a pixel is behind or in front of another. When a fragment is rendered to the depth buffer, its depth value is therefore compared with the depth value that has already been stored for the current position,

Listing 2. Fragment shader written in GLSL for determining the maximum characteristic speed using the depth buffer.

```
uniform sampler2D QnTex;
uniform float scale;
uniform float G;

void main(void)
{
  vec4 Q = texture2D(QnTex, gl_TexCoord[0].xy);
  Q.yz /= Q.x;
  float c = sqrt( G*Q.x );
  float r = max(abs(Q.y) + c, abs(texQ.z) + c);
  r *= scale;

  gl_FragDepth = r;
}
```

and the maximum value is stored. This process is called *depth test.* In our setting this means that when all quads have been rendered, we have a 16×16 depth buffer, where each value represents the maximum over $N_x N_y / 16^2$ values. The content of the 16×16 depth buffer is read back to the CPU, and the global maximum is determined. Since the depth test only accepts values in the interval $[0, 1]$, we include an appropriate scaling parameter `scale` passed in from the CPU.

3.4 Boundary Conditions

In our discussions above we have tacitly assumed that every grid cell is surrounded by neighbouring cells. In practice, however, all computations are performed on some bounded domain and one therefore needs to specify boundary conditions. On a CPU, an easy way to impose boundary conditions is to use *ghost cells*; that is, to extend the domain by extra grid cells along the boundaries, whose values are set at the beginning of each time step (in the ODE solver). See [33] for a thorough discussion of different boundary conditions and their implementation in terms of ghost cells.

There are many types of boundary conditions:

Outflow (or absorbing) boundary conditions simply let waves pass out of the domain without creating any reflections. The simplest approach is to extrapolate grid values from inside the domain in each spatial direction.

Reflective boundary conditions are used to model walls or to reduce computational domains by symmetry. They are realised by extrapolating grid values from inside the domain in each spatial direction and reversing the sign of the velocity component normal to the boundary.

Periodic boundary conditions are realised by copying values from grid cells inside the domain at the opposite boundary.

Inflow boundary conditions are used to model a given flow into the domain through certain parts of the boundary. These conditions are generally more difficult to implement. However, there are many cases for which inflow can be modelled by assigning fixed values to the ghost cells.

In this paper we will only work with outflow boundary conditions on rectangular domains. The simplest approach is to use a zeroth order extrapolation, which means that all ghost cells are assigned the value of the closest point inside the domain. Then outflow boundaries can be realised on the GPU simply by specifying that our textures are *clamped*, which means that whenever we try to access a value outside the domain, a value from the boundary is used. For higher-order extrapolation, we generally need to introduce an extra rendering pass using special shaders that render the correct ghost-cell values to extended textures.

3.5 Comparison of GPU versus CPU

We will now present a few numerical examples that compare the efficiency of our GPU implementation with a straightforward single-threaded CPU implementation of the Lax–Friedrichs scheme. The CPU simulations were run on a Dell Precision 670 which is a EM64T architecture with a dual Intel Xeon 2.8 GHz processor with 2.0 GB memory running Linux (Fedora Core 2). The programs were written in C and compiled using `icc -O3 -ipo -xP` (version 8.1). The GPU simulations were performed on two graphics cards from consecutive generations of the NVIDIA GeForce series: (i) the GeForce 6800 Ultra card released in April 2004 with the NVIDIA Forceware version 71.80 beta drivers, and (ii) the GeForce 7800 GTX card released in June 2005 with the NVIDIA Forceware version 77.72 drivers. The GeForce 6800 Ultra has 16 parallel pipelines, each capable of processing vectors of length 4 simultaneously. On the GeForce 7800 GTX, the number of pipelines is increased to 24.

In the following we will for simplicity make comparisons on $N \times N$ grids, but this is no prerequisite for the GPU implementation. Rectangular $N \times M$ grids work equally well. Although the current capabilities of our GPUs allow for problems with sizes up to 4096×4096, we have limited the size of our problems to 1024×1024 (due to high computational times on the CPU).

Example 2 (Circular dambreak). Let us first consider a simple dambreak problem with rotational symmetry for the shallow-water equations; that is, we consider two constant states separated by a circle

$$
\begin{aligned}
h(x,y,0) &= \begin{cases} 1.0, & \sqrt{x^2+y^2} \le 0.3, \\ 0.1, & \text{otherwise}, \end{cases} \\
u(x,y,0) &= v(x,y,0) = 0.
\end{aligned}
$$

We assume absorbing boundary conditions. The solution consists of an expanding circular shock wave. Within the shock there is a rarefaction wave

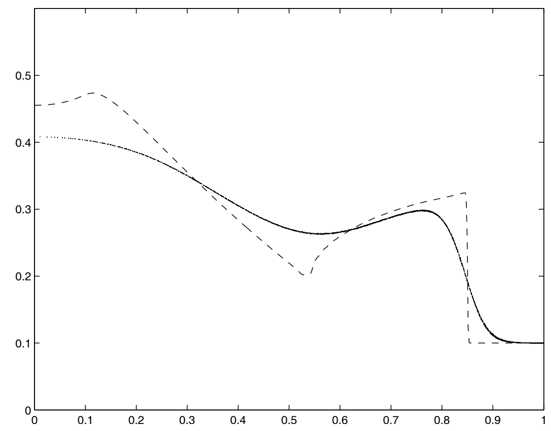

Fig. 6. Scatter plot of the circular dambreak problem at $t = 0.5$ computed by Lax–Friedrichs scheme on a 128×128 grid. The dashed line is a fine-grid reference solution.

transporting water from the original deep region out to the shock. In the numerical computations the domain is $[-1.0, 1.0] \times [-1.0, 1.0]$ with absorbing boundary conditions. A reference solution can be obtained by solving the reduced inhomogeneous system

$$\begin{bmatrix} h \\ hu_r \end{bmatrix}_t + \begin{bmatrix} hu \\ hu_r^2 + \frac{1}{2}gh^2 \end{bmatrix}_r = -\frac{1}{r}\begin{bmatrix} hu_r \\ hu_r^2 \end{bmatrix},$$

in which $u_r = \sqrt{u^2 + v^2}$ denotes the radial velocity. This system is nothing but the one-dimensional shallow-water equations (5) with a geometric source term.

Figure 6 shows a scatter plot of the water height at time $t = 0.5$ computed by the Lax–Friedrichs. In the scatter plot, the cell averages of a given quantity in all cells are plotted against the radial distance from the origin to the cell centre. Scatter plots are especially suited for illustrating grid-orientation effects for radially symmetric problems. If the approximate solution has perfect symmetry, the scatter plot will appear as a single line, whereas any symmetry deviations will show up as point clouds. The first-order Lax–Friedrichs scheme clearly smears both the leading shock and the tail of the imploding rarefaction wave. This behaviour is similar to what we observed previously in Figures 2 and 3 in Example 1 for the linear advection equation and the nonlinear Burgers' equation.

Table 1 reports a comparison of average runtime per time step for GPU versus CPU implementations of the Lax–Friedrichs scheme. Here, however, we see a that the 7800 GTX card with 24 pipelines is about two to three times faster than the 6800 Ultra card with 16 pipelines. The reason is as follows: For the simple Lax–Friedrichs scheme, the number of (arithmetic) operations

Table 1. Runtime per time step in seconds and speedup factor for the CPU versus the GPU implementation of Lax–Friedrichs scheme for the circular dambreak problem run on a grid with $N \times N$ grid cells, for the NVIDIA GeForce 6800 Ultra and GeForce 7800 GTX graphics cards.

N	CPU	6800	speedup	7800	speedup
128	2.22e-3	7.68e-4	2.9	2.33e-4	9.5
256	9.09e-3	1.24e-3	7.3	4.59e-4	19.8
512	3.71e-2	3.82e-3	9.7	1.47e-3	25.2
1024	1.48e-1	1.55e-2	9.5	5.54e-3	26.7

performed per fragment in each rendering pass (i.e., the number of operations per grid point) is quite low compared with the number of texture fetches. We therefore cannot expect to fully utilise the computational potential of the GPU. In particular for the 128×128 grid, the total number of fragments processed per rendering pass on the GPU is low compared with the costs of switching between different rendering buffers and establishing each pipeline. The cost of these context switches will therefore dominate the runtime on the 6800 Ultra card, resulting in a very modest speedup of about 3–4. For the 7800 GTX, the cost of context switches is greatly reduced due to improvements in hardware and/or drivers, thereby giving a much higher speedup factor.

Example 3 (Shock-bubble interaction). In this example we consider the interaction of a planar shock in air with a circular region of low density.

The setup is as follows (see Figure 7): A circle with radius 0.2 is centred at $(0.4, 0.5)$. The gas is initially at rest and has unit density and pressure. Inside the circle the density is 0.1. The incoming shock wave starts at $x = 0$ and propagates in the positive x-direction. The pressure behind the shock is 10, giving a 2.95 Mach shock. The domain is $[0, 1.6] \times [0, 1]$ with symmetry about the x axis. Figure 7 shows the development of the shock-bubble interaction in terms of (emulated) Schlieren images. Schlieren imaging is a photo-optical technique for studying the distribution of density gradients within a transparent medium. Here we have imitated this technique by depicting $(1 - |\nabla\rho|/\max|\nabla\rho|)^p$ for $p = 15$ as a greymap.

Figure 8 shows the result of a grid refinement study for Lax–Friedrichs. Whereas the scheme captures the leading shock on all grids, the weaker waves in the deformed bubble are only resolved satisfactorily on the two finest grids.

To study the efficiency of a GPU implementation versus a CPU implementation of the Lax–Friedrichs scheme, we reduced the computational domain to $[0, 1] \times [0, 1]$ and the final simulation time to $t = 0.2$. Runtime per time step and speedup factors are reported in Table 2. Comparing with the circular dambreak case in Table 1, we now observe that whereas the runtime per time step on the CPU increases by a factor between 1.3–1.4, the runtime on the GPU increases at most by a factor 1.1. The resulting increase in speedup factor is thus slightly less than 4/3. This can be explained as follows: Since the

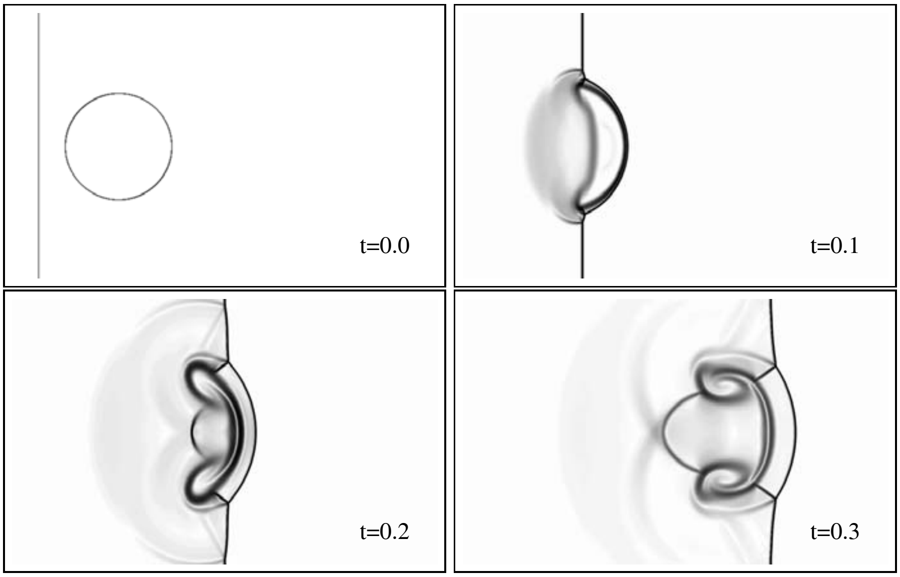

Fig. 7. Emulated Schlieren images of a shock-bubble interaction. The approximate solution is computed by the composite scheme [37] on a 1280×800 grid.

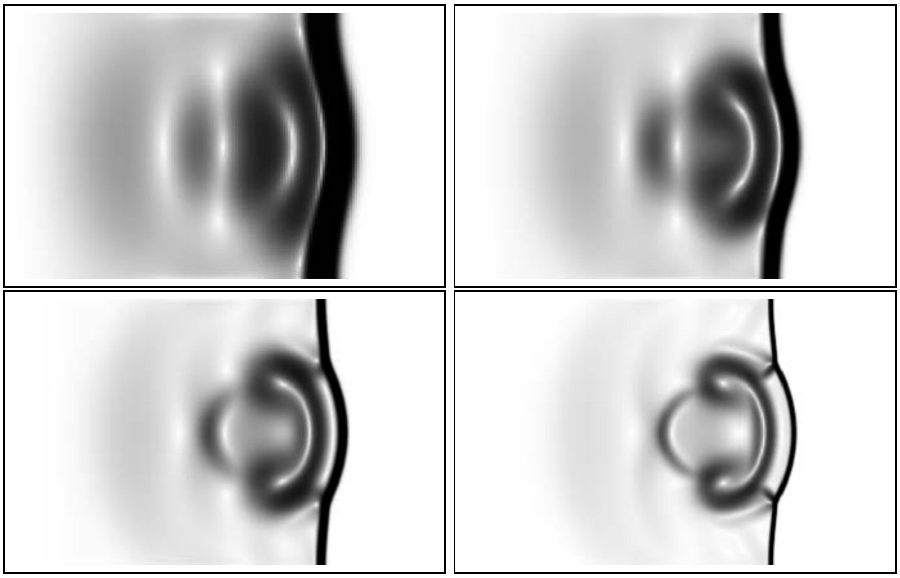

Fig. 8. Grid refinement study of the shock-bubble case at time $t = 0.3$ computed with the Lax–Friedrichs scheme for $\Delta x = \Delta y = 1/100$, $1/200$, $1/400$, and $1/800$.

Table 2. Runtime per time step in seconds and speedup factor for the CPU versus the GPU implementation of Lax–Friedrichs for the shock-bubble problem run on a grid with $N \times N$ grid cells, for the NVIDIA GeForce 6800 Ultra and GeForce 7800 GTX graphics cards.

N	CPU	6800	speedup	7800	speedup
128	3.11e-3	7.46e-4	4.2	2.50e-4	12.4
256	1.23e-2	1.33e-3	9.3	5.56e-4	22.1
512	4.93e-2	4.19e-3	11.8	1.81e-3	27.2
1024	2.02e-1	1.69e-2	12.0	6.77e-3	29.8

shallow-water equations have only three components, the GPU only exploits 3/4 of its processing capabilities in each vector operation, whereas the Euler equations have four component and can exploit the full vector processing capability. However, since the flux evaluation for the Euler equations is a bit more costly in terms of vector operations, and since not all operations in the shaders are vector operations of length four, the increase in speedup factor is expected to be slightly less than 4/3.

3.6 Floating Point Precision

When implementing PDE solvers on the CPU it is customary to use double precision. The current generations of GPUs, however, are only able to perform floating point operations in single precision (32 bits), but double precision has been announced by NVIDIA to appear in late 2007. The single precision numbers of the GPU have the format 's23e8', i.e., there are 6 decimals in the mantissa. As a result, the positive range of representable numbers is $[1.175 \cdot 10^{-38}, 3.403 \cdot 10^{38}]$, and the smallest number ϵ_s, such that $1 + \epsilon_s - 1 > 0$, is $1.192 \cdot 10^{-7}$. The double precision numbers of our CPU have the format 's52e11', i.e., mantissa of 15 decimals, range of $[2.225 \cdot 10^{-308}, 1.798 \cdot 10^{308}]$, and $\epsilon_d = 2.220 \cdot 10^{-16}$.

Ideally, the comparison between the CPU and the GPUs should have been performed using the same precision. Since CPUs offer single precision, this may seem to be an obvious choice. However, on current CPUs, double precision is supported in hardware, whereas single precision is emulated. This means that computations in single precision may be *slower* than the computations in double precision on a CPU. We therefore decided not to use single precision on the CPU.

The question is now if using single precision will effect the quality of the computed result. For the cases considered herein, we have computed all results on the CPU using both double and single precision. In all cases, the maximum difference of any cell average is of the same order as ϵ_s. Our conclusion based on these *quantitative studies* is that the GPU's lack of double precision does not deteriorate the computations for the examples considered in Sections 3.5 and

5. Please note that the methods considered in this paper are stable, meaning they cannot break down due to rounding errors.

4 High-Resolution Schemes

Classical finite difference schemes rely on the computation of point values. In the previous section we saw how discontinuities lead to problems like oscillations or excessive smearing for two basic schemes of this type. For pedagogical reasons we presented the *derivation* of the two schemes in a semi-discrete, finite-volume setting, but this does not affect (in)abilities of the schemes in the presence of discontinuities. Modern high-resolution schemes of the Godunov type aim at delivering high-order resolution of all parts of the solution, and in particular avoiding the creation of spurious oscillations at discontinuities. There are several approaches to high-resolution schemes. Composite schemes form one such approach, see [37]. In the next sections we will present another approach based on geometric considerations.

4.1 The REA Algorithm

Let us now recapitulate how we constructed our approximate schemes in Section 3. Abstracted, our approach can be seen to consist of three basic parts:

1. Starting from the known cell-averages Q_{ij}^n in each grid cell, we *reconstruct* a piecewise polynomial function $\widehat{Q}(x, y, t^n)$ defined for all (x, y). For a first-order scheme, we let $\widehat{Q}(x, y, t^n)$ be constant over each cell, i.e.,

$$\widehat{Q}(x, y, t^n) = Q_{ij}^n, \qquad (x, y) \in [x_{i-1/2}, x_{i+1/2}] \times [y_{j-1/2}, y_{j+1/2}].$$

   In a second-order scheme we use a piecewise bilinear reconstruction, and in a third order scheme a piecewise biquadratic reconstruction, and so on. The purpose of this reconstruction is to go from the discrete set of cell-averages we use to *represent* the unknown solution and back to a globally defined function that is to be evolved in time. In these reconstructions special care must be taken to avoid introducing spurious oscillations into the approximation, as is done by classical higher-order schemes like e.g., Lax–Wendroff.
2. In the next step we *evolve* the differential equation, exactly or approximately, using the reconstructed function $\widehat{Q}(x, y, t^n)$ as initial data.
3. Finally, we *average* the evolved solution $\widehat{Q}(x, y, t^{n+1})$ onto the grid again to give a new representation of the solution at time t^{n+1} in terms of cell averages Q_{ij}^{n+1}.

This three-step algorithm is often referred to as the REA algorithm, from the words reconstruct-evolve-average.

The Riemann Problem

To evolve and average the solution in Steps 2 and 3 of the REA algorithm, we generally use a semi-discrete evolution equation like (15). We thus need to know the fluxes out of each finite-volume cell. In Section 3.1 we saw how the problem of evaluating the flux over the cell boundaries in one spatial dimension could be recast to solve a set of local Riemann problems of the form

$$Q_t + F(Q)_x = 0, \qquad Q(x,0) = \begin{cases} Q_L, & x < 0, \\ Q_R, & x > 0. \end{cases}$$

The Riemann problem has a similarity solution of the form $Q(x,t) = V(x/t)$. The similarity solution consists of constant states separated by simple waves that are either continuous (rarefactions) or discontinuous (shocks, contacts, shear waves, etc) and is therefore often referred to as the Riemann fan. Thus, to compute the flux over the cell edge, all we need to know is $V(0)$. However, obtaining either the value $V(0)$ or a good approximation generally requires detailed knowledge of the underlying hyperbolic system. The algorithm used to compute $V(0)$ is called a Riemann solver. Riemann solvers come in a lot of flavours. Complete solvers, whether they are exact or approximate, use detailed knowledge of the wave structure and are able to resolve all features in the Riemann fan. Exact Riemann solvers are available for many systems, including the Euler and the shallow-water equations, but are seldom used in practice since they tend to be relatively complicated and expensive to compute. Incomplete solvers, like the central-upwind or HLL solver presented in Section 4.5, lump all intermediate waves and only resolve the leading waves in the Riemann fan, thereby giving only a rough estimate of the centre state. A significant amount of the research on high-resolution schemes has therefore been devoted to developing (approximate) numerical Riemann solvers that are fast and accurate. The Riemann problem is thoroughly discussed in the literature, the interested reader might start with [31] and then move on to [50, 51] for a thorough discussion of different Riemann solvers for the Euler and the shallow-water equations.

4.2 Upwind and Centred Godunov Schemes

The REA algorithm above is a general approach for constructing the so-called Godunov schemes that have ruled the ground in conservation laws during the last decades.

Godunov schemes generally come in two flavors: upwind or centred schemes. To illustrate their difference, we will make a slight detour into fully discrete schemes. For simplicity, we consider only the one-dimensional scalar case. Instead of the cell average (10) we introduce a sliding average

$$\bar{Q}(x,t) = \frac{1}{\Delta}x \int_{x-\Delta x/2}^{x+\Delta x/2} Q(\xi,t)\, d\xi.$$

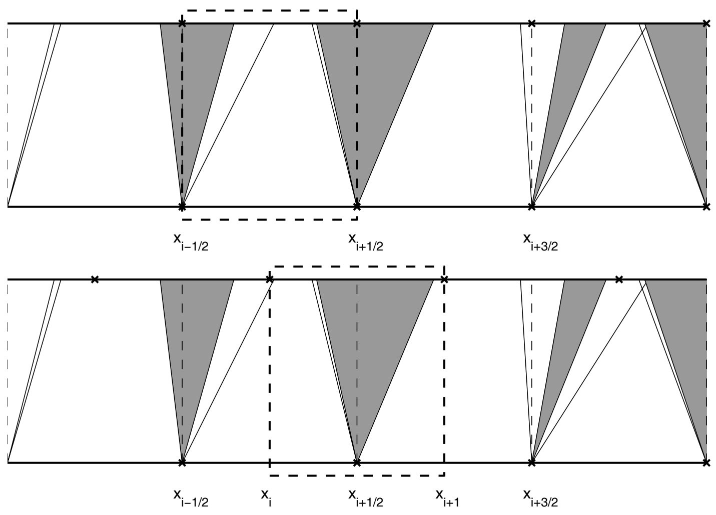

Fig. 9. Sliding average for upwind schemes (top) and centred schemes (bottom). The dashed boxes show the integration volume in the (x, t) plane. The shaded triangles and solid lines emanating from $x_{i-1/2}$, $x_{i+1/2}$ and $x_{i+3/2}$ illustrate the self-similar Riemann fans, whereas the dashed lines indicate the cell edges at time t^n.

Analogously to (13), we can derive an evolution equation for the sliding average

$$\begin{aligned}\bar{Q}(x, t + \Delta t) &= \bar{Q}(x, t)\\ &\quad - \frac{1}{\Delta}x \int_t^{t+\Delta t} \left[F\big(Q(x + \tfrac{\Delta x}{2}, \tau)\big) - F\big(Q(x - \tfrac{\Delta x}{2}, \tau)\big)\right] d\tau. \qquad (24)\end{aligned}$$

Within this setting, we obtain an upwind scheme if we choose $x = x_i$ in (24) and a centred scheme for $x = x_{i+1/2}$. Figure 9 illustrates the fundamental difference between these two approaches. In the upwind scheme, the flux is integrated over the points $x_{i\pm1/2}$, where the reconstruction is generally discontinuous. To integrate along the cell interface in time we will therefore generally have to find the self-similar solution of the corresponding Riemann problem (18) along the space-time ray $(x - x_{i\pm1/2})/t = 0$. To this end, one uses either exact or approximate Riemann solvers, as discussed above.

Centred schemes, on the other hand, use a sliding average defined over a *staggered* grid cell $[x_i, x_{i+1}]$, cf. the dashed box in the lower half of Figure 9. Thus the flux is integrated at the points x_i and x_{i+1}, where the solution is continuous if Δt satisfies a CFL condition of 1/2. This means that no explicit wave propagation information for the Riemann fans is required to integrate the flux, since in this case the Riemann fan is inside confinement of the associated staggered cells. Since they use less information about the specific conservation law being solved, centred schemes tend to be less accurate than upwind schemes. On the other hand, they are much simpler to implement and work pretty well as black-box solvers.

The first staggered high-resolution scheme based upon central differences was introduced by Nessyahu and Tadmor [40], who extended the staggered Lax–Friedrichs scheme (19) discussed in Section 3.1 to second order. Similarly, the scheme has later been extended to higher order and higher dimensions by e.g., Arminjon et al. [3], Russo et al. [7, 34], and Tadmor et al. [38, 23]. Joint with S. Noelle, the fourth author (Lie) has contributed to the research on staggered schemes, see [35, 36]. A complete overview of staggered high-resolution methods is outside the scope of the current presentation, instead we refer the reader to Tadmor's *Central Station*

`http://www.cscamm.umd.edu/~tadmor/centralstation/`

which has links to virtually all papers written on high-resolution central schemes, and in particular, on staggered schemes.

Non-staggered centred schemes can be obtained if the new solution at time $t + \Delta t$ is first reconstructed over the staggered cell and then averaged over the original cell to define a new cell-average at time $t + \Delta t$. In certain cases, semi-discrete schemes can then be obtained in the limit $\Delta t \to 0$. We do not present derivations here; the interested reader is referred to e.g., [26, 25].

4.3 Semi-Discrete High-Resolution Schemes

Let us now return to the semi-discretised equation (15). Before we can use this equation for computing, we must answer two questions: how do we solve the ODEs evolving the cell-average values Q_{ij}, and how do we compute the edge fluxes (16)? The ODEs are usually solved by an explicit predictor-corrector scheme of Runge–Kutta type. In each stage of the Runge–Kutta method the right-hand side of (15) is computed by evaluating the flux integrals according to some standard quadrature rule. To apply a quadrature rule, two more points need to be clarified: how to compute point-values of Q from the cell-averages, and how to evaluate the integrand of the edge fluxes at the integration points. Listing 3 shows a general algorithm based on (15).

Different schemes are distinguished by the three following points: the reconstruction used to compute point-values, the discretisation and evaluation of flux integrals, and the ODE-solver for (15). In the following we will describe all these parts in more detail. We start with the ODE-solver, move on to the flux evaluation, and in the next section consider the reconstruction. We return to the flux evaluation in Section 4.5.

Runge–Kutta ODE Solver

To retain high-order accuracy in time without creating spurious oscillations, it is customary to use so-called TVD Runge–Kutta methods [47] as the ODE-solver. These methods employ a convex combination of forward Euler steps

Listing 3. General algorithmic form of a semi-discrete, high-resolution scheme, written in quasi-C code

```
Q[N_RK]=make_initial_data();

// Main loop
for (t=0; t<T; t+=dt) {
  Q[0] = Q[N_RK];
  dt = compute_Dt_from_CFL(Q[0]);

  // Runge−Kutta steps
  for (n=1; n<=N_RK; n++) {
    Q[n−1] = set_boundary_conditions(Q[n−1]);
    Qp     = reconstruct_point_values(Q[n−1]);
    [F,G]  = evaluate_edge_fluxes(Qp);
    Q[n]   = compute_RK_step(n, dt, Q, F, G);
  }
}
```

to advance the solution in time. They are especially designed to maintain the TVD property (30), i.e., ensure that the solution is total variation diminishing, see [31]. The second-order method (RK2) reads:

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

and similarly for the third-order method (RK3)

$$\begin{aligned} Q_{ij}^{(1)} &= Q_{ij}^n + \Delta t L_{ij}(Q^n), \\ Q_{ij}^{(2)} &= \frac{3}{4} Q_{ij}^n + \frac{1}{4}\big[Q_{ij}^{(1)} + \Delta t L_{ij}(Q^{(1)})\big], \\ Q_{ij}^{n+1} &= \frac{1}{3} Q_{ij}^n + \frac{2}{3}\big[Q_{ij}^{(2)} + \Delta t L_{ij}(Q^{(2)})\big]. \end{aligned}$$

Listing 4 shows the fragment shader implementing one step of the Runge–Kutta solver. The different steps are realised by choosing different values for the parameter c; $c = (0, 1)$ for the first, and $c = (\frac{1}{2}, \frac{1}{2})$ for the second step of RK2. As in the previous shaders, we use handles of type `uniform sampler2D` to sample texture data, and return the computed values in `gl_FragColor`.

Quadrature Rules

To evaluate the edge fluxes (16), we use a standard quadrature rule. For a second-order scheme the midpoint rule suffices, i.e.,

Listing 4. Fragment shader for the Runge–Kutta solver

```
varying vec4 texXcoord;
varying vec4 texYcoord;

uniform sampler2D QnTex;
uniform sampler2D FnHalfTex;
uniform sampler2D GnHalfTex;

uniform vec2 c;
uniform vec2 dXYf;
uniform float dT;

void main(void)
{
  vec4 FE = texture2D(FnHalfTex, texXcoord.yx);               // F_{i+1/2,j}
  vec4 FW = texture2D(FnHalfTex, texXcoord.zx);               // F_{i−1/2,j}
  vec4 GN = texture2D(GnHalfTex, texYcoord.xy);               // G_{i,j+1/2}
  vec4 GS = texture2D(GnHalfTex, texYcoord.xz);               // G_{i,j−1/2}

  vec4 Lij = −((FE−FW)/dXYf.x + (GN−GS)/dXYf.y);             // Right−hand side

  vec4 Q = texture2D(QnTex, texXcoord.yx);                    // Q_{ij} ^{(k)}
  vec4 QFirst = texture2D(QnFirstTex, texXcoord.yx);          // Q_{ij} ^{n}

  gl_FragColor = c.x*QFirst + c.y*(Q + dT*Lij);               // Q_{ij} ^{(k+1)}
}
```

$$\int_{-1/2}^{1/2} f(x)\,dx = f(0). \tag{25}$$

For a third-order scheme, we can use Simpson's rule

$$\int_{-1/2}^{1/2} f(x)\,dx = \frac{1}{6}\Big[f(-\tfrac{1}{2}) + 4f(0) + f(\tfrac{1}{2})\Big], \tag{26}$$

or, as we will do in the following, use the fourth order Gauss quadrature rule

$$\int_{-1/2}^{1/2} f(x)\,dx = \frac{1}{2}\Big[f(\tfrac{-1}{2\sqrt{3}}) + f(\tfrac{1}{2\sqrt{3}})\Big]. \tag{27}$$

Hence, the edge fluxes (16) become

$$\begin{aligned} F_{i\pm1/2,j}(t) &= \frac{1}{2}\Big[F\big(Q(x_{i\pm1/2}, y_{j+\alpha}, t)\big) + F\big(Q(x_{i\pm1/2}, y_{j-\alpha}, t)\big)\Big], \\ G_{i,j\pm1/2}(t) &= \frac{1}{2}\Big[G\big(Q(x_{i+\alpha}, y_{j\pm1/2}, t)\big) + G\big(Q(x_{i-\alpha}, y_{j\pm1/2}, t)\big)\Big], \end{aligned} \tag{28}$$

where $x_{i\pm\alpha}$ and $y_{j\pm\alpha}$ denote the integration points of the Gauss quadrature rule.

At this point it is probably clear to the reader why we need the reconstruction from Step 1 in Section 4.1. Namely, whereas our semi-discrete scheme (15) evolves cell averages, the flux quadrature requires *point values* at the Gaussian integration points $(x_{i\pm1/2}, y_{j\pm\alpha})$ and $(x_{i\pm\alpha}, y_{j\pm1/2})$. The second step in the computation of fluxes therefore amounts to obtaining these point values through a piecewise polynomial reconstruction.

4.4 Piecewise Polynomial Reconstruction

The major challenge in developing high-order reconstructions is to cope with nonsmooth and discontinuous data. We cannot expect to maintain high-order accuracy at a discontinuity. Instead, the aim is to minimise the creation of spurious oscillations. In this section we introduce a simple piecewise linear reconstruction that prevents spurious oscillations, but retains second-order spatial accuracy away from discontinuities. This will be achieved by introducing a so-called limiter function that compares left and right-hand slopes and picks a nonlinear average in such a way that the total variation of the reconstructed function is no greater than that of the underlying cell averages.

The reconstruction will be introduced for a scalar equation, and we tacitly assume that it can be extended in a component-wise manner to nonlinear systems of equations.

Bilinear Reconstruction

For a second-order scheme we can use a bilinear reconstruction $\widehat{Q}(x, y, t^n)$ given by

$$\widehat{Q}_{ij}(x, y, t^n) = Q^n_{ij} + s^x_{ij}(x - x_i) + s^y_{ij}(y - y_j). \tag{29}$$

Here the slopes s^x_{ij} and s^y_{ij} can be estimated from the-cell average values of the neighbouring cells. For simplicity, let us consider the one-dimensional case. Here there are three obvious candidate stencils for estimating the slope:

$$s^-_i = \frac{Q^n_i - Q^n_{i-1}}{\Delta x}, \qquad s^+_i = \frac{Q^n_{i+1} - Q^n_i}{\Delta x}, \qquad s^c_i = \frac{Q^n_{i+1} - Q^n_{i-1}}{2\Delta x}.$$

Which stencil we should choose will depend on the local behaviour of the underlying function, or in our case, on the three cell averages. As an example, assume that $Q^n_k = 1$ for $k \leq i$ and $Q^n_k = 0$ for $k > i$. In this case, $s^-_i = 0$, $s^+_i = -1/\Delta x$, and $s^c_i = -1/2\Delta x$. Hence, $\widehat{Q}(x)$ will remain monotonic if we choose $s^x_i = s^-_i$, whereas a new maximum will be introduced for s^+_i and s^c_i. Solutions of scalar conservation laws are bounded by their initial data in the sup-norm, preserve monotonicity and have diminishing total variation. The last point may be expressed as

$$\mathrm{TV}(Q(\cdot, \tau)) \leq \mathrm{TV}(Q(\cdot, t)), \qquad \text{for } t \leq \tau, \tag{30}$$

where the total variation functional is defined by

$$\mathrm{TV}(v) = \limsup_{h \to 0} \frac{1}{h} \int |v(x) - v(x - h)|\, dx.$$

We want our discrete solution to mimic this behaviour. Therefore we need to put some "intelligence" into the stencil computing the linear slopes. The key to obtaining this intelligence lies in the introduction of a nonlinear averaging function Φ capable of choosing the slope. Given Φ, we simply let

$$\Delta x\, s^x_{ij} = \Phi\big(Q^n_{ij} - Q^n_{i-1,j}, Q^n_{i+1,j} - Q^n_{ij}\big).$$

This function is called a *limiter*, and is applied independently in each spatial direction. By limiting the size of the slopes in the reconstruction, this function introduces a nonlinearity in the scheme that ensures that solutions are nonoscillatory and total variation diminishing (TVD). An example of a robust limiter is the minmod limiter

$$\mathrm{MM}(a,b) = \tfrac{1}{2}\big(\mathrm{sgn}(a) + \mathrm{sgn}(b)\big)\min(|a|,|b|) \tag{31}$$

$$= \begin{cases} 0, & \text{if } ab \le 0, \\ a, & \text{if } |a| < |b| \text{ and } ab > 0, \\ b, & \text{if } |b| < |a| \text{ and } ab > 0, \end{cases} \tag{32}$$

which picks the least slope and reduces to zero at extrema in the data. For systems of conservation laws, solution are not necessarily bounded in sup-norm or have diminishing total variation. Still, it is customary to apply the same design principle as for scalar equations. In our experiments, we have also used the modified minmod limiter and the superbee limiter

$$\mathrm{MM}_\theta(a,b) = \begin{cases} 0, & \text{if } ab \le 0, \\ \theta a, & \text{if } (2\theta - 1)ab < b^2, \\ \tfrac{1}{2}(a+b), & \text{if } ab < (2\theta - 1)b^2, \\ \theta b, & \text{otherwise}, \end{cases}$$

$$\mathrm{SB}_\theta(a,b) = \begin{cases} 0, & \text{if } ab \le 0, \\ \theta a, & \text{if } \theta ab < b^2, \\ b, & \text{if } ab < b^2, \\ a, & \text{if } ab < \theta b^2 \\ \theta b, & \text{otherwise}. \end{cases}$$

Away from extrema, the minmod always chooses the one-sided slope with least magnitude, whereas the other two limiters tend to choose steeper reconstruction, thus adding less numerical viscosity into the scheme. For other families of limiter, see e.g., [36, 50].

Listing 5 shows the fragment shader implementing the bilinear reconstruction. This shader needs to return two vectors corresponding to $\Delta x \partial_x Q$ and $\Delta y \partial_y Q$. At the time of implementation, GLSL was not capable of simultaneously writing to two textures due to problems with our drivers, we therefore chose to implemented the shader in Cg using so-called *multiple render targets*. In our implementation, we write to two textures using the new structure

Listing 5. Fragment shader in Cg for the bilinear reconstruction

```
float4 minmod(in float4 a, in float4 b)      // minmod function
{
  float4 res = min(abs(a), abs(b));
  return res*(sign(a)+sign(b))*0.5;
}

struct f2b {                                 // Return two fragment buffers
  float4 color0  : COLOR;                    // d_x Q
  float4 color1  : COLOR1;                   // d_y Q
};

struct v2f_input {                           // Current stencil
  float4 pos         : POSITION;             // Position
  float4 texXcoord;                          // Offsets in x
  float4 texYcoord;                          // Offsets in y
};

f2b main(v2f_input IN, uniform sampler2D QnTex, uniform float2 dXYf)
{
  f2b OUT;

  float4 Q  = tex2D(QnTex, IN.texXcoord.yx);            // Q_{ij}
  float4 QE = tex2D(QnTex, IN.texXcoord.wx);            // Q_{i+1,j}
  float4 QW = tex2D(QnTex, IN.texXcoord.zx);            // Q_{i-1,j}
  OUT.color0 = minmod(Q-QW, QE-Q);                      // d_x Q_{ij}

  float4 QN = tex2D(QnTex, IN.texYcoord.xw);            // Q_{i,j+1}
  float4 QS = tex2D(QnTex, IN.texYcoord.xz);            // Q_{i,j-1}
  OUT.color1 = minmod(Q-QS, QN-Q);                      // d_y Q_{ij}

  return OUT;
}
```

`f2b`. Moreover, to avoid introducing conditional branches that can reduce the computational efficiency, we have used the non-conditional version (31) rather than the conditional version (32) of the minmod-limiter.

Higher-order accuracy can be obtained by choosing a higher-order reconstruction. Two higher-order reconstructions are presented in Appendix A; the CWENO reconstruction in Appendix A.1 is a truly multidimensional reconstruction giving third order spatial accuracy, whereas the dimension-by-dimension WENO reconstruction in Appendix A.2 is designed to give fifth order accuracy at Gaussian integration points.

4.5 Numerical Flux

In the previous section we presented a reconstruction approach for obtaining one-sided point values at the integration points of the flux quadrature (28). All that now remains to define a fully discrete scheme is to describe how to use these point-values to evaluate the integrand itself. In the following we will introduce a few fluxes that can be used in combination with our high-resolution semi-discrete schemes.

Centred Fluxes

In Sections 3.1 and 3.2 we introduced two classical centred fluxes, the Lax–Friedrichs flux:

$$F^{LF} = \frac{1}{2}\left[F(Q_L) + F(Q_R)\right] - \frac{1}{2}\frac{\Delta}{x}\Delta t\left[Q_R - Q_L\right],$$

and the Lax–Wendroff flux

$$\begin{aligned} F^{LW} &= F(Q^*), \\ Q^* &= \frac{1}{2}\left[Q_L + Q_R\right] - \frac{1}{2}\frac{\Delta}{t}\Delta x\left[F(Q_R) - F(Q_L)\right]. \end{aligned}$$

By taking the arithmetic average of these two fluxes, one obtains the FORCE flux introduced by Toro [50]

$$F^{FORCE} = \frac{1}{4}\left[F(Q_L) + 2F(Q^*) + F(Q_R)\right] - \frac{1}{4}\frac{\Delta}{x}\Delta t\left[Q_R - Q_L\right], \qquad (33)$$

In the above formulae Q_L and Q_R denote the point values extrapolated from left and right at the Gaussian integration points. If used directly with the cell averaged values (and with forward Euler as time integrator), the FORCE flux gives an oscillatory second-order method.

A Central-Upwind Flux

So far, wave propagation information has only been used to ensure stability through a CFL condition. Let us now incorporate information about the largest and smallest eigenvalues into the flux evaluation, while retaining the centred approach. This will lead to so-called central-upwind fluxes, as introduced by Kurganov, Noelle, and Petrova [25]. To this end, we define

$$\begin{aligned} a^+ &= \max_{Q\in\{Q_L,Q_R\}}\Big(\lambda_m(Q), 0\Big), \\ a^- &= \min_{Q\in\{Q_L,Q_R\}}\Big(\lambda_1(Q), 0\Big), \end{aligned}$$

where λ_1 is the smallest and λ_m the largest eigenvalues of the Jacobian matrix of the flux F. If the system is not genuinely nonlinear or linearly degenerate, the minimum and maximum is taken over the curve in phase space that connects Q_L and Q_R. The values (a^-, a^+) are estimates of how far the Riemann fan resulting from the discontinuity (Q_L, Q_R) extends in the negative and positive direction. Using this information, the evolving solution can be divided into two unions of local domains: one covering the local Riemann fans, where the solution is possibly discontinuous and one covering the area in-between, where the solution is smooth, see Figure 10. By evolving and averaging separately in the two sets, the following flux function can be derived [25]

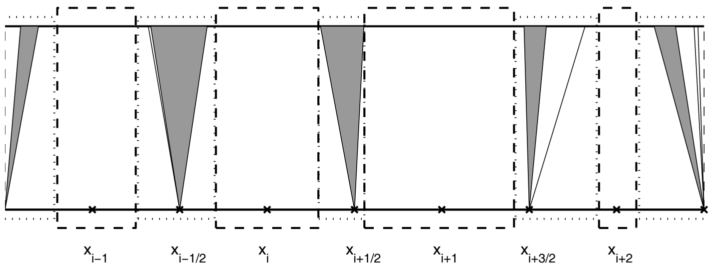

Fig. 10. Spatial averaging in the REA algorithm for the central-upwind scheme.

$$F^{CUW} = \frac{a^+ F(Q_L) - a^- F(Q_R)}{a^+ - a^-} + \frac{a^+ a^-}{a^+ - a^-}\Big(Q_R - Q_L\Big). \tag{34}$$

The name "central-upwind" comes from the fact that for monotone flux functions, the flux reduces to the standard upwind method. For example, for scalar equations with $F'(Q) \geq 0$, a^- is zero and $F^{CUW} = F(Q_L)$. Moreover, the first-order version of the scheme (i.e., for which $Q_L = Q_i$ and $Q_R = Q_{i+1}$) is the semi-discrete version of the famous HLL upwind flux [20]. Finally, if $a^+ = -a^- = \Delta x/\Delta t$, then F^{CUW} reduces to the Lax–Friedrichs flux F^{LF}.

Listing 6 shows the fragment shader implementing the computation of the edge-flux (16) using a fourth-order Gauss quadrature (27), bilinear reconstruction, and central-upwind flux. In the implementation, we compute estimates of the wave-speeds a^+ and a^- at the midpoint of each edge and use them to evaluate the flux function at the two Gaussian integration points. Notice also the use of the built-in inner-product `dot` and the weighted average `mix`.

A Black-Box Upwind Flux

For completeness, we also include a 'black-box' upwind flux, which is based upon resolving the local Riemann problems *numerically.* In the Multi-Stage (MUSTA) approach introduced by Toro [52], local Riemann problems are solved numerically a few time steps using a predictor-corrector scheme with a centred flux. This 'opens' the Riemann fan, and the centre state $V(\xi = 0)$ can be picked out and used to evaluate the true flux over the interface. The MUSTA algorithm starts by setting $Q_L^{(1)} = Q_L$ and $Q_R^{(1)} = Q_R$, and then iterates the following steps 3–4 times:

1. Flux evaluation: set $F_L^{(\ell)} = F(Q_L^{(\ell)})$ and $F_R^{(\ell)} = F(Q_R^{(\ell)})$ and compute $F_M^{(\ell)}$ as the FORCE flux (33).
2. Open Riemann fan

$$Q_L^{(\ell+1)} = Q_L^{(\ell)} - \frac{\Delta}{t}\Delta x\Big(F_M^{(\ell)} - F_L^{(\ell)}\Big), Q_R^{(\ell+1)} = Q_R^{(\ell)} - \frac{\Delta}{t}\Delta x\Big(F_R^{(\ell)} - F_M^{(\ell)}\Big),$$

This approach has the advantage that it is general and requires no knowledge of the wave structure of a specific system.

Listing 6. Fragment shader for computing the edge-flux in the x-direction using central-upwind flux for the two-dimensional Euler equations.

```
varying vec4 texXcoord;
uniform sampler2D QnTex;
uniform sampler2D SxTex;
uniform sampler2D SyTex;
uniform float gamma;

float pressure(in vec4 Q)
{
  return (gamma−1.0)∗(Q.w − 0.5∗dot(Q.yz,Q.yz)/Q.x);
}

vec4 fflux(in vec4 Q)
{
  float u = Q.y/Q.x;
  float p = (gamma−1.0)∗(Q.w − 0.5∗dot(Q.yz,Q.yz)/Q.x);
  return vec4(Q.y, (Q.y∗u)+p, Q.z∗u, u∗(Q.w+p) );
}

void main(void)
{
  // Reconstruction in (i,j)
  vec4 Q  = texture2D(QnTex, texXcoord.yx);
  vec4 Sx = texture2D(SxTex, texXcoord.yx);
  vec4 Sy = texture2D(SyTex, texXcoord.yx);
  vec4 QL = Q + Sx∗0.5;
  vec4 QLp = Q + Sx∗0.5 + Sy∗0.2886751346;
  vec4 QLm = Q + Sx∗0.5 − Sy∗0.2886751346;

  // Reconstruction in (i+1,j)
  vec4 Q1 = texture2D(QnTex, texXcoord.wx);
  vec4 Sxp = texture2D(SxTex, texXcoord.wx);
  vec4 Syp = texture2D(SyTex, texXcoord.wx);
  vec4 QR = Q1 − Sxp∗0.5;
  vec4 QRp = Q1 − Sxp∗0.5 + Syp∗0.2886751346;
  vec4 QRm = Q1 − Sxp∗0.5 − Syp∗0.2886751346;

  // Calculate ap and am
  float c, ap, am;
  c = sqrt(gamma∗QL.x∗pressure(QL));
  ap = max((QL.y + c)/QL.x, 0.0);
  am = min((QL.y − c)/QL.x, 0.0);
  c  = sqrt(gamma∗QR.x∗pressure(QR));
  ap = max((QR.y + c)/QR.x, ap);
  am = min((QR.y − c)/QR.x, am);

  // Central−upwind flux
  vec4 Fp = ((ap∗fflux(QLp) − am∗fflux(QRp)) + (ap∗am)∗(QRp − QLp))/(ap − am);
  vec4 Fm = ((ap∗fflux(QLm) − am∗fflux(QRm)) + (ap∗am)∗(QRm − QLm))/(ap − am)

  gl_FragColor = mix(Fp, Fm, 0.5);
}
```

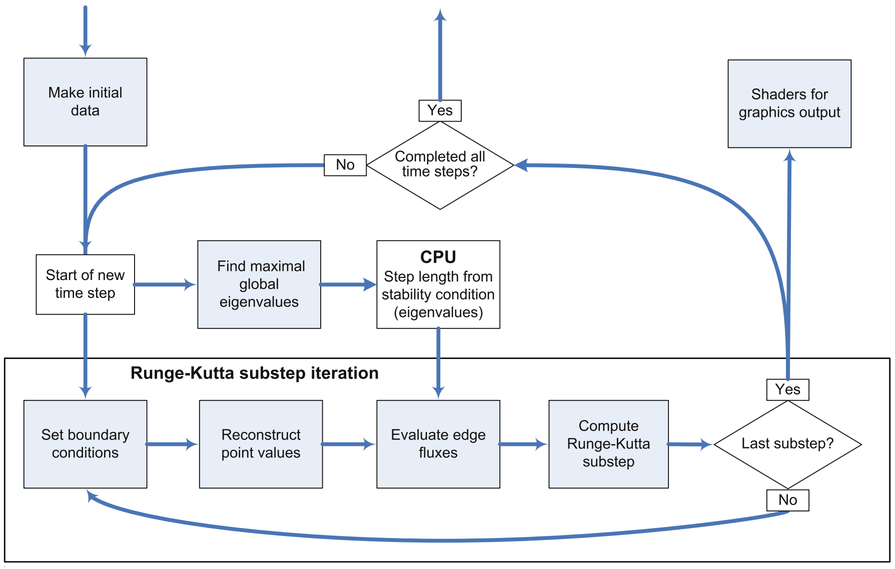
Make initial data
Yes
Completed all time steps?
No
Shaders for graphics output
Start of new time step
Find maximal global eigenvalues
CPU
Step length from stability condition (eigenvalues)
Runge-Kutta substep iteration
Yes
Set boundary conditions
Reconstruct point values
Evaluate edge fluxes
Compute Runge-Kutta substep
Last substep?
No

Fig. 11. Flow-chart for the GPU implementation of a semi-discrete, high-resolution scheme. White boxes refer to operations on the CPU and shaded boxes to operations on the GPU.

4.6 Putting It All Together

In the above sections we have presented the components needed to implement a high-resolution scheme on the GPU. Let us now see how they can be put together to form a full solver. Figure 11 shows a flow-chart for the implementation. In the flow-chart, all the logic and setup is performed on the CPU, whereas all time-consuming operations are performed in parallel on the GPU. The only exception is the determination of the time step, which is performed by postprocessing results from the depth buffer (see Listing 2). The three essential shaders are the reconstruction given in Listing 5, the evaluation of edge fluxes in Listing 6, and the Runge–Kutta steps in Listing 4. To make a complete scheme, we also need two shaders to set initial and boundary data, respectively. These shaders are problem specific and must be provided for each individual test case.

5 Numerical Examples

We now present a few numerical examples to highlight a few features of the high-resolution schemes and assess the efficiency of a GPU versus a CPU implementation. As in Section 3.5, all CPU simulations were run on a Dell Precision 670 with a dual Intel Xeon 2.8 GHz processor and the GPU simulation were performed on two NVIDIA GeForce graphics card, a 6800 Ultra

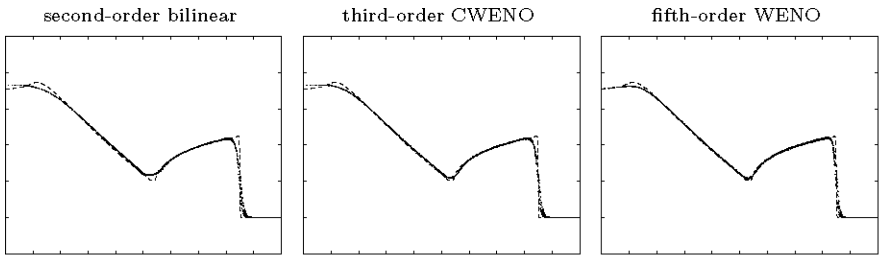

Fig. 12. Scatter plot of the circular dambreak problem at $t = 0.5$ computed on a 128×128 grid with reconstructions: bilinear with minmod limiter, CWENO, and WENO. The dashed line is a fine-grid reference solution.

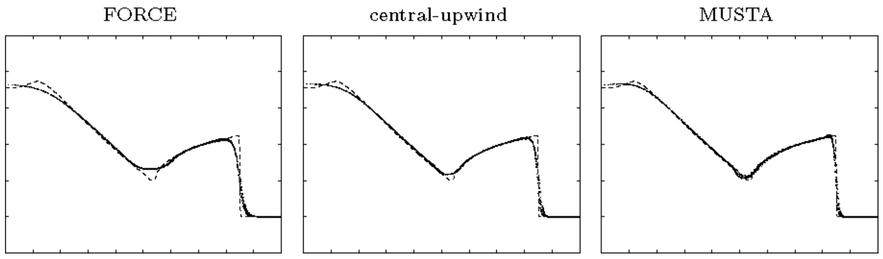

Fig. 13. Scatter plot of the circular dambreak problem at $t = 0.5$ computed on a 128×128 with bilinear reconstruction and FORCE, central-upwind, and MUSTA fluxes. The dashed line is a fine-grid reference solution.

and a 7800 GTX. (See also our two journal papers on shallow-water waves [17] and the 3D Euler equations [18] for supplementary numerical results.)

Example 4 (Circular dambreak). In the first example, we revisit the circular dambreak problem from Example 2. Figure 12 shows approximate solutions computed using three different reconstructions. (To isolate the effects of the reconstruction, all three computations used third-order Runge–Kutta, central-upwind flux, and fourth order Gauss quadrature). Starting with the bilinear reconstruction, we see that this scheme gives superior resolution of both the leading shock and the rarefaction wave compared with the Lax–Friedrichs scheme in Figure 6. By increasing the order of the reconstruction from bilinear to the third-order CWENO, we improve the local minimum at $r \approx 0.5$. Finally, by introducing the fifth-order WENO reconstruction, we also obtain a satisfactory resolution of the constant state near the origin (and a slight improvement near the shock).

Figure 13 shows a comparison of the three different fluxes for the bilinear scheme. The FORCE flux is a centred flux that does not incorporate any information of the local wave propagation. As a result, the corresponding scheme smears the leading shock and both ends of the rarefaction wave (at

Table 3. Runtime per time step in seconds and speedup factor for the CPU versus the GPU implementation of bilinear interpolation with modified minmod limiter for the dambreak problem run on a grid with $N \times N$ grid cells, for the NVIDIA GeForce 6800 Ultra and GeForce 7800 GTX graphics cards. The upper part uses second-order and the bottom part third-order Runge–Kutta time stepping.

N	CPU	6800	speedup	7800	speedup
128	3.06e-2	3.78e-3	8.1	1.27e-3	24.2
256	1.22e-1	8.43e-2	14.5	4.19e-3	29.1
512	4.86e-1	3.18e-2	15.3	1.68e-2	28.9
1024	2.05e-0	1.43e-1	14.3	6.83e-2	30.0
128	4.56e-2	5.58e-3	8.2	1.90e-3	23.9
256	1.83e-1	1.24e-2	14.8	6.23e-3	29.4
512	7.33e-1	4.69e-2	15.6	2.51e-2	29.2
1024	3.09e-0	2.15e-1	14.3	1.04e-1	29.7

$r \approx 0.5$ and $r \approx 0.15$, respectively). The central-upwind flux incorporates local wave propagation information in the form of two-sided estimates of the local wave speeds and therefore gives sharper resolution of both the shock and the rarefaction ends. Finally, the MUSTA flux is an iterative solver for the local Riemann problem that gives very accurate approximation of the Godunov edge-flux. The corresponding scheme therefore gives good resolution of both the leading shock and the rarefaction ends.

In Table 3 we have rerun the experiments from Table 1 in Example 2, now using the high-resolution scheme with central-upwind flux and bilinear reconstruction (and modified minmod limiter with $\theta = 1.3$). Compared with the simple Lax–Friedrichs scheme, the bilinear scheme involves a larger number of arithmetic operations per grid cell. This means that the costs of data fetch and data processing perfectly balance, giving a considerably higher speedup then for Lax–Friedrichs.

Example 5 (Shock-bubble). In the second example, we revisit the shock-bubble interaction from Example 3. We will now consider two different simulation methods using the second-order bilinear reconstruction from Section 4.4 and the third-order CWENO reconstruction from Appendix A.1. Apart from this, both methods use the second Runge–Kutta method, fourth-order Gauss quadrature, and central-upwind flux.

Figure 14 illustrates the effect of choosing different reconstructions. In the bilinear reconstruction, we have used the minmod limiter (31), the modified minmod-θ limiter (with $\theta = 1.3$) and the superbee limiter (with $\theta = 1.5$). The two latter limiters tend to choose steeper slopes in the local reconstructions, thereby introducing less numerical viscosity in the approximate solution. The interface making up the walls of the collapsing bubble is instable and tends to break up for the least dissipative superbee limiter, whereas the minmod limiter has sufficient numerical dissipation to suppress the instability. We notice in

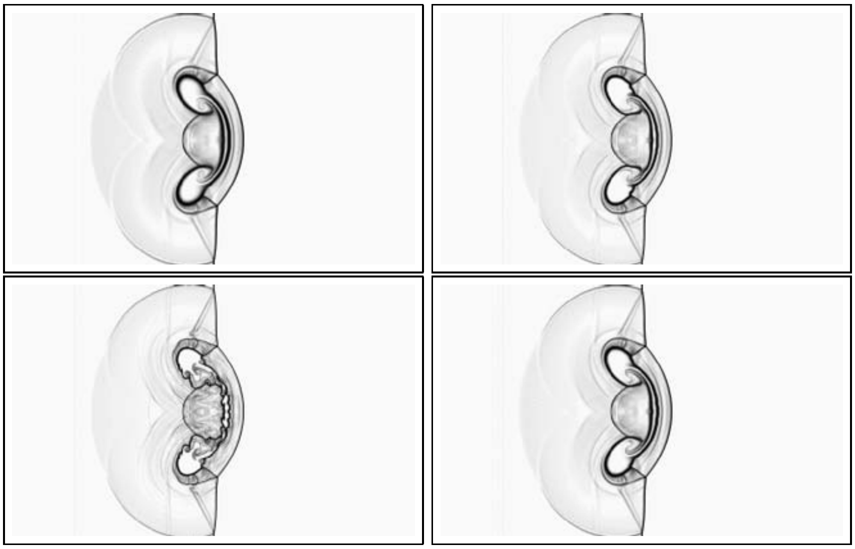

Fig. 14. Approximate solution for $\Delta x = 1/800$ of the shock-bubble problem at time $t = 0.2$ computed with the bilinear scheme for using three different limiters: minmod (top-left), modified minmod (top-right), and superbee (bottom-left) and by the CWENO scheme (bottom-right).

particular that although the CWENO reconstruction has higher order, it also contains more numerical dissipation than the bilinear reconstruction with the superbee limiter and therefore suppresses the breaking of the bubble interface.

In Table 4 we have rerun the experiments from Table 2 in Example 3, now using the high-resolution scheme with bilinear reconstruction (and modified minmod limiter with $\theta = 1.3$). Comparing Tables 3 and 4, we observe that the runtime for the GPU simulations only increases a few percent when going from the shallow-water to the Euler equations, indicating that for the bilinear scheme the runtime is strongly dominated by vector operations that are perfectly parallel. The resulting speedup factors of order 20 and 40 for GeForce 6800 and 7800, respectively, are in fact amazing, since the GPU implementations have not involved any low-level optimisation apart from the obvious use of vector operations whenever appropriate. (Using the widespread `gcc` compiler with full optimisation rather than `icc` to compile the CPU code gave approximately 50% higher runtime on the CPU, and hence speedup factors of magnitude up to 70 for the GeForce 7800 card!)

Table 5 shows corresponding runtimes and speedup factors for the third-order CWENO reconstruction. As seen from the discussion in Appendix A.1, we cannot expect to retain the good speedup observed for the bilinear scheme. Still, a speedup of 8–9 for GeForce 6800 and 23–24 for GeForce 7800 is indeed impressive.

Table 4. Runtime per time step in seconds and speedup factor for the CPU versus the GPU implementation of bilinear interpolation with modified minmod limiter for the shock-bubble problem run on a grid with $N \times N$ grid cells, for the NVIDIA GeForce 6800 Ultra and GeForce 7800 GTX graphics cards. The upper part uses second-order and the bottom part third-order Runge–Kutta time stepping.

N	CPU	6800	speedup	7800	speedup
128	4.37e-2	3.70e-3	11.8	1.38e-3	31.7
256	1.74e-1	8.69e-3	20.0	4.37e-3	39.8
512	6.90e-1	3.32e-2	20.8	1.72e-2	40.1
1024	2.95e-0	1.48e-1	19.9	7.62e-2	38.7
128	6.27e-2	5.22e-3	12.0	1.97e-3	31.9
256	2.49e-1	1.28e-2	19.5	6.44e-3	38.6
512	9.89e-1	4.94e-2	20.0	2.56e-2	38.6
1024	4.24e-0	2.20e-1	19.3	1.13e-1	37.5

Table 5. Runtime per time step in seconds and speedup factor for the CPU versus the GPU implementation of CWENO reconstruction for the shock-bubble problem run on a grid with $N \times N$ grid cells, for the NVIDIA GeForce 6800 Ultra and GeForce 7800 GTX graphics cards. The upper part uses second-order and the bottom part third-order Runge–Kutta time stepping.

N	CPU	6800	speedup	7800	speedup
128	1.05e-1	1.22e-2	8.6	4.60e-3	22.8
256	4.20e-1	4.99e-2	8.4	1.74e-2	24.2
512	1.67e-0	1.78e-1	9.4	6.86e-2	24.3
1024	6.67e-0	7.14e-1	9.3	2.99e-1	22.3
128	1.58e-1	1.77e-2	8.9	6.80e-3	23.2
256	6.26e-1	6.78e-2	9.2	2.59e-2	24.2
512	2.49e-0	2.66e-1	9.4	1.02e-1	24.3
1024	9.98e-0	1.06e-0	9.4	4.45e-1	22.4

Example 6. In the last example, we will consider the full shallow-water equations (7) with a variable bathymetry. A particular difficulty with this system of balance laws (conservation laws with source terms are often called balance laws) is that it admits steady-state or solutions in which the (topographical) source terms are exactly balanced by nonzero flux gradients,

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

and similarly in the y-direction. Capturing this delicate balance without introducing spurious waves of small amplitude is a real challenge for any numerical scheme. Of particular interest is the so-called *lake-at-rest* problem, in which $hu = hv = 0$ and $w = h + B =$ constant.

For high-resolution schemes a lot of research has been devoted to develop so-called well-balanced schemes, which are capable of accurately resolving steady-state and near-steady-state solutions; see e.g., [32, 5, 27, 4, 41] and references therein. In [17] we discussed a GPU-implementation of the well-balanced approach from [27]. The key points in this approach is: (i) to reconstruct the surface elevation $w = h + B$ rather than the water depth h as a piecewise polynomial function, and (ii) to use a special quadrature rule for the source term S. For the second component of the source term, the quadrature rule reads

$$\begin{aligned} S_{ij}^{(2)} &= -\frac{1}{|\Omega_{ij}|}\int_{y_{j-1/2}}^{y_{j+1/2}}\int_{x_{i-1/2}}^{x_{i+1/2}} g(w-B)_x\,dxdy \\ &\approx -\frac{g}{2\Delta x}\big(h^L_{i+1/2,j-\alpha}+h^R_{i-1/2,j-\alpha}\big)\big(B_{i+1/2,j-\alpha}-B_{i-1/2,j-\alpha}\big) \\ &\quad -\frac{g}{2\Delta x}\big(h^L_{i+1/2,j+\alpha}+h^R_{i-1/2,j+\alpha}\big)\big(B_{i+1/2,j+\alpha}-B_{i-1/2,j+\alpha}\big), \end{aligned} \tag{35}$$

This source term is then included in the semi-discrete evolution equation for the cell-averages

$$\frac{d}{dt}Q_{ij} = -\frac{F_{i+1/2,j}-F_{i-1/2,j}}{\Delta x} - \frac{G_{i,j+1/2}-G_{i,j-1/2}}{\Delta y} + S_{ij}, \tag{36}$$

where the flux-terms and the ODE are discretised as discussed above.

We will now use this scheme to simulate a dambreak in a mountainous terrain. Figure 15 shows four snapshots from the GPU simulation. Initially, the water in the lake is at rest in the upper lake (steady state). At time zero, the dam breaks and water starts to flood down into the neighbouring valley, where it creates strong flood waves that wash up on the hillsides. As time increases, the simulation *slowly* approaches a new steady state with equal water height in the two valleys.

The presence of dry-bed zones (i.e., areas with $h = 0$) poses extra challenges for the simulation, since these areas must be given special treatment to avoid (nonphysical) negative water heights; see [17].

6 Concluding Remarks

In this paper we have tried to give the reader an introduction to two exciting research fields: numerical solution of hyperbolic conservation laws and general purpose computation using graphics hardware. Research on high-resolution methods for hyperbolic conservation laws is a mature field, evolved during the last three decades, and both its mathematical and numerical aspects are supported by a large body of publications. When it comes to the use of graphics hardware as a computational resource, the situation is the opposite. Although there exist some early papers on the use of fixed-pipeline graphics

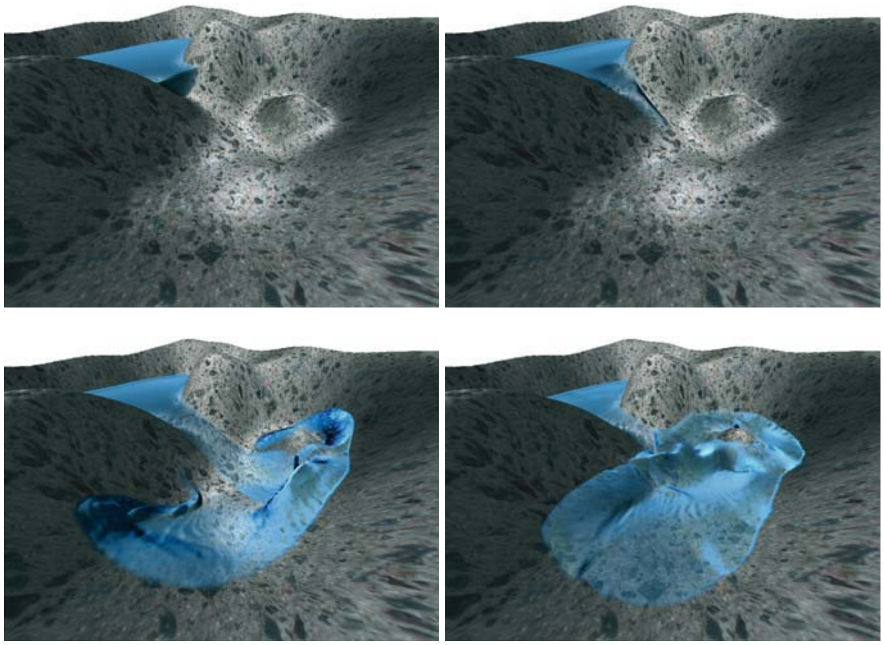

Fig. 15. Snapshots of a dambreak simulation in an artificially constructed terrain.

cards for non-graphical purposes, the interest in this field has exploded since the first GPUs with fully programmable fragment processors were introduced in 2003. Since then, the use of graphics cards for general purpose computing has attracted the interest of researchers in many different fields.

To demonstrate the computing capabilities of the GPU, we chose two common physical models, the Euler equations and the somewhat simpler shallow-water equations. We have attacked these equations with an assembly of finite-volume methods ranging from the basic Lax–Friedrichs scheme to the sophisticated third-order CWENO scheme, which represents state-of-the-art in high-resolution methods. Using standard numerical test cases in square domains, we observe that moving the computations from the CPU to a GPU gives a speedup of at least one order of magnitude. This amazing speedup is possible because the methods we have considered are explicit. Therefore, each cell can be updated independently of its neighbours, using only local points in a texture representing the previous time-step. This makes the methods "brilliantly parallel" and ripe for the data-based stream processing of graphics hardware. On the other hand, we have seen that in order to exploit this potential speedup, the computational algorithms must be recast to graphics terms. In our experience this requires familiarity with both computer graphics and the underlying hardware. However, once the algorithm has been recast to computer graphics, any programmer with a background from scientific computing should be able to write the actual shaders.

In the following subsections we give a short discussion of how GPU computations can be applied to more complex problems. Moreover, we also present a current outlook for GPUs for solving PDEs in scientific and industrial problems.

More Complex Physics

Having seen the success of using GPUs for the shallow water and the Euler equations, one can easily envisage that GPU computations can be used to speed up the computation of other (and more complex) hyperbolic models. As an example, magneto-hydrodynamics in two spatial dimensions is described by a set of seven equations and therefore do not map directly into a single four-component texture. A simple solution would be to split the vector of unknowns between two textures, and similarly for the fluxes, reconstructed slopes, etc. The flux computations would then typically involve a single shader reading from both textures, whereas a componentwise reconstruction could be performed by running the same shader consecutively on the two textures. Similarly, the Euler equations in three spatial dimensions is a 5×5 system and thus could be represented as e.g., a one-component texture for density and a four-component texture for momentum and energy or into two three-component textures as done in [18]. Three-dimensional grids can either be represented directly using the recent feature of 3D textures or by mapping each it to a regular 2D texture [39, 18].

Real-life problems often involve complex internal and external boundaries. A efficient method for representing complex boundaries in fluid flow simulations was presented by Wu et al. [39, 55]. Their idea is to preprocess the computational domain and define texture coordinate offsets to identify the nodes that determine the values of nodes close to the boundaries. Although this approach is introduced for the incompressible Navier–Stokes equations, we believe that it can be adopted for conservation laws as well.

To Whom Will GPU-Computing Be Useful?

From one point of view, the GPU can be considered as a parallel computer of the SIMD (single-instruction, multiple data) type, except that for the GPU there are no expressed communication between the nodes/pipelines anywhere in the code. From a user's point of view, the main attraction with the GPUs is the price-performance ratio (or the ratio of price versus power consumption). Assuming that the memory available on a graphics card is not a limitation, a simple back-of-an-envelope comparison of speedup versus dollar favour the GPU over a cluster of PCs, viz.

Price of graphics card:	\$ 500
Price of 24 computers (\$ 1000 each):	\$ 24 000
Speedup of graphics card:	20 times
Speedup of cluster:	24 times
Dollar/speedup for graphics card:	\$ 25
Dollar/speedup for cluster:	\$ 1000

With these figures, purchasing a 24-pipeline programmable graphics card gives 40 times more speedup per dollar than a 24-node cluster.

Having seen the performance of one single GPU it is tempting to ask what can be gained by exploiting a *cluster* of GPUs. Today, there are several research groups (including us) that try to use clusters equipped with GPUs or even clusters consisting of PlayStations to perform large-scale computing, see e.g., [12]. Although early performance reports are very good, the fact that GPUs are based upon a different programming model may in the end prevent wide-spread use of GPUs for high-performance computing.

On the other hand, the use of graphics cards may be a key factor in bringing PDE simulations from batch mode to interactive mode for desktop-sized applications, thereby opening up for their use in real-time systems (for monitoring and control), computer games and entertainment industry.

Current Technology Trends (Afterword March 2007)

The study reported herein was performed in the period 2004 to early 2006. During that period, the most recent generations of GPUs from NVIDIA (6800 Ultra from 2004/2005 and 7800 GTX from 2005/2006) demonstrated amazing floating-point performance of about 54 Gflops and 165 Gflops, respectively, compared with the typical commodity CPUs; contemporary Intel Pentium 4 CPUs, for instance, had a theoretical performance of at most 15 Gflops.

As the book goes into press (March 2007), the performance of GPUs has increased even further to about 0.5 Tflops for the NVIDIA GeForce 8800 GTX, which has 128 pipelines and 768 MB memory. A bit simplified, the reason behind this tremendous increase in computing power is as follows: Due to the parallel nature of a GPU, it is possible to increase the performance simply by adding more computational units, thereby increasing the number of computations that are made in parallel. Moreover, the architecture of commodity CPUs has changed significantly in the sense that dual-core CPUs have become common. Currently quad-core CPUs are beginning to hit the market. As for the future, Intel has recently showed “Polaris”, a prototype processor with 80 cores, each with its own programmer-managed memory. Last year, AMD bought ATI, the other main manufacturer of GPUs. Recently, AMD announced the development of a hybrid CPU-GPU processor (Fusion) and the Torrenza initiative aimed at easy integration of various hardware accelerator units (like graphics cards). This development points in the direction of heterogeneous computers consisting of traditional CPU type units in combination with specialised accelerator units.

Another important point is the appearance in the mass-marked of other data-parallel processors, in particular the Cell BE processor designed for PlayStation 3 and used in blade servers. The Cell BE processor consists of one general processor connected to eight specialised computational cores, which may remind of stream processors. With this tile architecture, the Cell processor should be quite well-suited for general-purpose computing.

The development in hardware is expected to be followed by a similar development in software tools. NVIDIA recently released CUDA (Compute Unified Device Architecture – `http://developer.nvidia.com/object/cuda.html`), which allows the programmer to treat graphics cards in the GeForce 8000 series as general data-parallel processors without using the graphical API. Similarly, companies like RapidMind and PeakStream are developing software tools for taking advantage of multi-core processors and data-parallel processors like GPUs and Cell.

Altogether, it therefore seems that data-parallel processors from the mass market may offer computing capabilities in the next few years that are hard to ignore, and that utilising these capabilities for scientific computing will be a very exciting field to work in for a scientist.

Acknowledgement. This research was funded in part by the Research Council of Norway under grant no. 158911/I30.

References

1. J. E. Aarnes, T. Gimse, and K.-A. Lie. An introduction to the numerics of flow in porous media using Matlab. In *this book*.
2. J. E. Aarnes, V. Kippe, K.-A. Lie, and A. B. Rustad. Modelling of multiscale structures in flow simulations for petroleum reservoirs. In *this book*.
3. P. Arminjon, D. Stanescu, and M.-C. Viallon. A two-dimensional finite volume extension of the Lax–Friedrichs and Nessyahu–Tadmor schemes for compressible flows. In M. Hafez and K. Oshima, editors, *Proceedings of the 6th International Symposium on CFD, Lake Tahoe*, volume IV, pages 7–14, 1995.
4. E. Audusse, F. Bouchut, M.-O. Bristeau, R. Klein, and B. Perthame. A fast and stable well-balanced scheme with hydrostatic reconstruction for shallow water flows. *SIAM J. Sci. Comp.*, 25:2050–2065, 2004.
5. Derek S. Bale, Randall J. Leveque, Sorin Mitran, and James A. Rossmanith. A wave propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM J. Sci. Comput.*, 24(3):955–978 (electronic), 2002.
6. F. Benkhaldoun and R. Vilsmeier, editors. *Finite volumes for complex applications.* Hermes Science Publications, Paris, 1996. Problems and perspectives.
7. F. Bianco, G. Puppo, and G. Russo. High-order central schemes for hyperbolic systems of conservation laws. *SIAM J. Sci. Comput.*, 21(1):294–322 (electronic), 1999.
8. A. J. Chorin and J. E. Marsden. *A mathematical introduction to fluid mechanics*, volume 4 of *Texts in Applied Mathematics.* Springer-Verlag, New York, third edition, 1993.

9. R. Courant and K. O. Friedrichs. *Supersonic Flow and Shock Waves.* Interscience Publishers, Inc., New York, N. Y., 1948.
10. T. Dokken, T. R. Hagen, and J. M. Hjelmervik. An introduction to general-purpose computing on programmable graphics cards. In *this book.*
11. M. Van Dyke. *An Album of Fluid Motion.* Parabolic Press, 1982.
12. Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.
13. R. Fernando, editor. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Addison Wesley, 2004.
14. R. Fernando and M.J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc., 2003.
15. E. Godlewski and P.-A. Raviart. *Numerical approximation of hyperbolic systems of conservation laws*, volume 118 of *Applied Mathematical Sciences.* Springer-Verlag, New York, 1996.
16. S. K. Godunov. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Mat. Sb. (N.S.)*, 47 (89):271–306, 1959.
17. T. R. Hagen, J. M. Hjelmervik, K.-A. Lie, J. R. Natvig, and M. Ofstad Henriksen. Visual simulation of shallow-water waves. *Simul. Model. Pract. Theory*, 13(8):716–726, 2005.
18. T. R. Hagen, K.-A. Lie, and J. R. Natvig. Solving the Euler equations on graphical processing units. In V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*, volume 3994 of *Lecture Notes in Computer Science (LNCS)*, pages 220–227. Springer Verlag, 2006.
19. A. Harten. High resolution schemes for hyperbolic conservation laws. *J. Comput. Phys.*, 49(3):357–393, 1983.
20. A. Harten, P. D. Lax, and B. van Leer. On upstream differencing and Godunov-type schemes for hyperbolic conservation laws. *SIAM Rev.*, 25(1):35–61, 1983.
21. R. Herbin and D. Kröner, editors. *Finite volumes for complex applications III.* Laboratoire d'Analyse, Topologie et Probabilités CNRS, Marseille, 2002. Problems and perspectives, Papers from the 3rd Symposium held in Porquerolles, June 24–28, 2002.
22. H. Holden and N. H. Risebro. *Front tracking for hyperbolic conservation laws*, volume 152 of *Applied Mathematical Sciences.* Springer-Verlag, New York, 2002.
23. G.-S. Jiang and E. Tadmor. Nonoscillatory central schemes for multidimensional hyperbolic conservation laws. *SIAM J. Sci. Comput.*, 19(6):1892–1917, 1998.
24. S. N. Kružkov. First order quasilinear equations with several independent variables. *Mat. Sb. (N.S.)*, 81 (123):228–255, 1970.
25. A. Kurganov, S. Noelle, and G. Petrova. Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations. *SIAM J. Sci. Comput.*, 23(3):707–740 (electronic), 2001.
26. A. Kurganov and E. Tadmor. New high-resolution semi-discrete central schemes for Hamilton–Jacobi equations. *J. Comp. Phys.*, 160:720–742, 2000.
27. Alexander Kurganov and Doron Levy. Central-upwind schemes for the Saint-Venant system. *M2AN Math. Model. Numer. Anal.*, 36(3):397–425, 2002.

28. L. D. Landau and E. M. Lifshitz. *Fluid mechanics.* Translated from the Russian by J. B. Sykes and W. H. Reid. Course of Theoretical Physics, Vol. 6. Pergamon Press, London, 1959.
29. P. D. Lax. Weak solutions of nonlinear hyperbolic equations and their numerical computation. *Comm. Pure Appl. Math.*, 7:159–193, 1954.
30. P.D. Lax and B. Wendroff. Systems of conservation laws. *Comm. Pure Appl. Math.*, 13:217–237, 1960.
31. R. J. LeVeque. *Numerical Methods for Conservation Laws.* Lectures in Mathematics ETH Zürich. Birkhäuser Verlag, Basel, second edition, 1994.
32. R. J. LeVeque. Balancing source terms and flux gradients in high-resolution Godunov methods: The quasi-steady wave-propagation algorithm. *J. Comput. Phys*, 146:346–365, 1998.
33. R. J. LeVeque. *Finite volume methods for hyperbolic problems.* Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002.
34. D. Levy, G. Puppo, and G. Russo. Compact central WENO schemes for multidimensional conservation laws. *SIAM J. Sci. Comput.*, 22(2):656–672, 2000.
35. K.-A. Lie and S. Noelle. An improved quadrature rule for the flux-computation in staggered central difference schemes in multidimensions. *J. Sci. Comput.*, 18(1):69–81, 2003.
36. K.-A. Lie and S. Noelle. On the artificial compression method for second-order nonoscillatory central difference schemes for systems of conservation laws. *SIAM J. Sci. Comput.*, 24(4):1157–1174, 2003.
37. R. Liska and B. Wendroff. Composite schemes for conservation laws. *SIAM J. Numer. Anal.*, 35(6):2250–2271, 1998.
38. X.-D. Liu and E. Tadmor. Third order nonoscillatory central scheme for hyperbolic conservation laws. *Numer. Math.*, 79(3):397–425, 1998.
39. Y. Liu, X. Liu, and E. Wu. Real-time 3d fluid simulation on GPU with complex obstacles. In *Proceedings of Pacific Graphics 2004*, pages 247–256. IEEE Computer Society, 2004.
40. H. Nessyahu and E. Tadmor. Nonoscillatory central differencing for hyperbolic conservation laws. *J. Comput. Phys.*, 87(2):408–463, 1990.
41. S. Noelle, N. Pankratz, G. Puppo, and J. R. Natvig. Well-balanced finite volume schemes of arbitrary order of accuracy for shallow water flows. *J. Comput. Phys.*, 213(2):474–499, 2006.
42. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.
43. M. Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley Professional, 2005.
44. R. D. Richtmyer and K. W. Morton. *Difference methods for initial-value problems.* Second edition. Interscience Tracts in Pure and Applied Mathematics, No. 4. Interscience Publishers John Wiley & Sons, Inc., New York-London-Sydney, 1967.
45. R. J. Rost. *OpenGL^R Shading Language.* Addison Wesley Longman Publishing Co., Inc., 2004.
46. M. Rumpf and R. Strzodka. Graphics processor units: new prospects for parallel computing. In A.M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 89–134. Springer Verlag, 2006.

47. C.-W. Shu. Total-variation-diminishing time discretisations. *SIAM J. Sci. Stat. Comput.*, 9:1073–1084, 1988.
48. C.-W. Shu. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. In *Advanced numerical approximation of nonlinear hyperbolic equations (Cetraro, 1997)*, volume 1697 of *Lecture Notes in Math.*, pages 325–432. Springer, Berlin, 1998.
49. V. A. Titarev and E. F. Toro. Finite-volume WENO schemes for three-dimensional conservation laws. *J. Comput. Phys.*, 201(1):238–260, 2004.
50. E. F. Toro. *Riemann solvers and numerical methods for fluid dynamics.* Springer-Verlag, Berlin, second edition, 1999.
51. E. F. Toro. *Shock-capturing methods for free-sufrace shallow flows.* Wiley and Sons Ltd., 2001.
52. E. F. Toro. Multi-stage predictor-corrector fluxes for hyperbolic equations. Technical Report NI03037-NPA, Isaac Newton Institute for Mathematical Sciences, 2003.
53. R. Vilsmeier, F. Benkhaldoun, and D. Hänel, editors. *Finite volumes for complex applications II.* Hermes Science Publications, Paris, 1999. Problems and perspectives, Papers from the 2nd International Conference held in Duisburg, July 19–22, 1999.
54. G. B. Whitham. *Linear and nonlinear waves.* Wiley-Interscience [John Wiley & Sons], New York, 1974. Pure and Applied Mathematics.
55. E. Wu, Y. Liu, and X. Liu. An improved study of real-time fluid simulation on GPU. *J. of Computer Animation and Virtual World*, 15(3–4):139–146, 2004.

A Higher-Order WENO Reconstructions

A popular approach for making higher-order reconstructions is the *essentially nonoscillatory* (ENO) approach. In Section 4.4 we saw how we could avoid the creation of spurious oscillations for a piecewise linear reconstruction by comparing one-sided slopes and picking the slope that gave the least oscillation. (This was done by the nonlinear limiter function).

The ENO approach is basically the same idea extended to higher-order polynomials constructed using divided differences. Assume for the moment that we have one-dimensional data $\{Q_i\}$. The original ENO idea starts by making a piecewise linear polynomial $P_i^1(x)$ based upon Q_i and Q_{i-1}. Next, we consider the two candidate stencils made out of either $\{Q_{i+1}, Q_i, Q_{i-1}\}$ or $\{Q_i, Q_{i-1}, Q_{i-2}\}$. The piecewise quadratic ENO polynomial $P_i^2(x)$ can be constructed using the three points that will create the least oscillatory polynomial. The correct stencil is picked by comparing the magnitude of the corresponding divided differences. This way, one can continue to recursively construct higher and higher order polynomials by adding points to the left or right, depending on the magnitude of the divided differences.

In the more recent *weighted ENO* (WENO) approach, the polynomials are constructed by weighting all possible stencils rather than choosing the least oscillatory. The idea of WENO is that stencils from smooth parts of the solution are given high weight and stencils from nonsmooth parts are given low weight. More details on (W)ENO reconstructions can be found in the survey paper [48].

There are essentially two different ways to reconstruct multidimensional data: genuinely multidimensional and dimension-by-dimension. Below we will give an example of each type. First we will introduce a multidimensional third-order centred WENO reconstruction. Then we present a dimension-by-dimension, fifth-order WENO reconstruction that is specifically tailored to give maximum resolution at the Gaussian integration points, as proposed by Titarev and Toro [49].

A.1 Third Order CWENO Reconstruction

Third order accuracy is obtained by using a piecewise biquadratic reconstruction

$$\begin{aligned}\widehat{Q}_{ij}(x,y,t^n) = Q_{ij}(x,y) &= V_{ij} + s^x_{ij}(x-x_i) + s^y_{ij}(y-y_j)\\ &+ \frac{1}{2}s^{xx}_{ij}(x-x_i)^2 + \frac{1}{2}s^{yy}_{ij}(y-y_j)^2 + s^{xy}_{ij}(x-x_i)(y-y_j).\end{aligned}$$

For smooth data, one can use an optimal polynomial $Q^{\text{OPT}}(x,y)$ based upon a centred nine-point stencil with

$$\begin{aligned}V_{ij} &= Q^n_{ij} - \frac{1}{24}\big(\Delta x^2 s^{xx}_{ij} + \Delta y^2 s^{yy}_{ij}\big),\\ s^x_{ij} &= \frac{Q^n_{i+1,j} - Q^n_{i-1,j}}{2\Delta x}, \qquad s^{xx}_{ij} = \frac{Q^n_{i+1,j} - 2Q^n_{ij} + Q^n_{i-1,j}}{\Delta x^2},\\ s^y_{ij} &= \frac{Q^n_{i,j+1} - Q^n_{i,j-1}}{2\Delta y}, \qquad s^{yy}_{ij} = \frac{Q^n_{i,j+1} - 2Q^n_{ij} + Q^n_{i,j-1}}{\Delta y^2},\\ s^{xy}_{ij} &= \frac{Q^n_{i+1,j+1} + Q^n_{i-1,j-1} - Q_{i+1,j-1} - Q^n_{i-1,j+1}}{4\Delta x\Delta y}.\end{aligned} \tag{37}$$

For nonsmooth data, a direct application of this reconstruction will introduce spurious oscillations. We therefore present another compact reconstruction called CWENO (centred WENO) that was introduced by Levy, Puppo and Russo [34]. The reconstruction employs a weighted combination of four one-sided piecewise bilinear and a centred piecewise quadratic stencil

$$P_{ij}(x,y) = \sum_k w^k_{ij} P^k_{ij}(x,y), \qquad k \in \{\text{NE},\text{NW},\text{SW},\text{SE},\text{C}\}. \tag{38}$$

By this weighting we seek to emphasize contributions from cell averages in smooth regions and diminish contributions from cell averages in nonsmooth regions.

The four bilinear stencils are

$$
\begin{aligned}
P^{\text{NE}}_{ij}(x,y) &= Q^n_{ij} + \frac{Q^n_{i+1,j}-Q^n_{ij}}{\Delta x}(x-x_i) + \frac{Q^n_{i,j+1}-Q^n_{ij}}{\Delta y}(y-y_j),\\
P^{\text{NW}}_{ij}(x,y) &= Q^n_{ij} + \frac{Q^n_{ij}-Q^n_{i-1,j}}{\Delta x}(x-x_i) + \frac{Q^n_{i,j+1}-Q^n_{ij}}{\Delta y}(y-y_j),\\
P^{\text{SW}}_{ij}(x,y) &= Q^n_{ij} + \frac{Q^n_{ij}-Q^n_{i-1,j}}{\Delta x}(x-x_i) + \frac{Q^n_{ij}-Q^n_{i,j-1}}{\Delta y}(y-y_j),\\
P^{\text{SE}}_{ij}(x,y) &= Q^n_{ij} + \frac{Q^n_{i+1,j}-Q^n_{ij}}{\Delta x}(x-x_i) + \frac{Q^n_{ij}-Q^n_{i,j-1}}{\Delta y}(y-y_j),
\end{aligned}
\tag{39}
$$

and the centred stencil is taken to satisfy

$$
P^{\text{OPT}}(x,y) = \sum_k C^k P^k(x,y), \qquad \sum_k C^k = 1, \qquad k \in \{\text{NE,NW,SW,SE,C}\}.
$$

On smooth data, a third-order reconstruction is obtained if we choose the weights to be $C^k = 1/8$ for $k \in \{\text{NE,NW,SW,SE}\}$ and $C^{\text{C}} = 1/2$. In other words, we choose the centred stencil to be

$$
P^c(x,y) = 2P^{\text{OPT}}(x,y) + \frac{1}{4}\sum_k P^k(x,y), \qquad k \in \{\text{NE,NW,SW,SE}\}.
$$

To tackle discontinuous data, we will introduce a nonlinear weighting procedure, analogously to the limiter defined for the bilinear reconstruction in Section 4.4. The nonlinear weights in (38) are designed so that they are as close as possible to the optimal weights C^k for smooth data. For nonsmooth data, the largest contribution in the reconstruction comes from the stencil(s) that generates the least oscillatory reconstruction. The weights are given as

$$
w^k_{ij} = \frac{\alpha^k_{ij}}{\sum_\ell \alpha^\ell_{ij}}, \qquad \alpha^k_{ij} = \frac{C^k}{(\beta^k_{ij}+\epsilon)^2}, \tag{40}
$$

for $k,\ell \in \{\text{NE,NW,SW,SE,C}\}$. Here ϵ is a small parameter (typically 10^{-6}) that prevents the denominator from vanishing for constant data, and the β^k's are smoothness indicators that are responsible for detecting discontinuities or large gradients

$$
\beta^k_{ij} = \sum_{|\alpha|=1,2} \int_{x_{i-1/2}}^{x_{i+1/2}} \int_{y_{j-1/2}}^{y_{j+1/2}} \Delta x^{2(|\alpha|-1)} \Big(\frac{d^{|\alpha|}}{dx^{\alpha_1} dy^{\alpha_2}} P^k(x,y) \Big)\, dx dy.
$$

For the bilinear stencils the smoothness indicators read

$$
\beta^k_{ij} = \Delta x^2 \Big((s^{x,k}_{ij})^2 + (s^{y,k}_{ij})^2 \Big), \qquad k \in \{\text{NE,NW,SW,SE}\}
$$

with slopes given by (39), and for the quadratic stencil the indicator reads

$$
\beta^{\text{C}}_{ij} = \Delta x^2 \Big((s^x_{ij})^2 + (s^y_{ij})^2 \Big) + \frac{\Delta x^4}{3} \Big(13(s^{xx}_{ij})^2 + 14(s^{xy}_{ij})^2 + 13(s^{yy}_{ij})^2 \Big)
$$

with slopes given by (37).

Compared with the bilinear reconstruction from Section 4.4, the construction of the CWENO polynomials involves a large number or arithmetic operations. Moreover, a large number of variables are needed to represent the reconstruction in each grid cell: 14 polynomial coefficients from (37) and (39) and five weights (40). In the CPU implementation, we therefore chose to recompute the coefficients of the reconstructions rather than storing them. The evaluation of each edge flux involves four one-sided point values taken from the reconstructions in the two adjacent cells. Therefore, in order to compute all edge fluxes, we make a single pass through all grid cells (including one layer of ghost cells) and compute the edge flux between the current cell (i, j) and its neighbour to the east $(i + 1, j)$ and to the north $(i, j + 1)$. Altogether, this means that in order to save computer memory we recompute the coefficients and weights three times in each grid cell.

In the GPU implementation, we were not able to use the same approach due to the lack of temporary registers. We therefore had to split the computation of edge fluxes into two passes: one for the F-fluxes and one for the G-fluxes, meaning that we recompute the coefficients and weights four times in each grid cell. Moreover, by splitting the flux computation into two render passes, we introduce one additional context switch and extra texture fetches. Unfortunately, we therefore reduce the theoretical speedup by a factor between 1/2 and 3/4. We expect the number of temporary registers to increase in future generations of GPUs and thus become less limiting.

A.2 A Fifth Order WENO Reconstruction

We seek the point values $Q(x_{i\pm 1/2}, y_{j\pm\alpha})$ and $Q(x_{i\pm\alpha}, y_{j\pm 1/2})$. The reconstruction is performed in two steps: first by reconstructing averages over the cell edges from the cell-averages, then by reconstructing the point-values from the edge-averages. In each step we use a one-dimensional, piecewise WENO reconstruction consisting of three quadratic stencils

$$V(\xi) = \sum_{k=0}^{2} w_k Q^k(\xi).$$

In the first step, we start from the cell averages and reconstruct one-sided averages over the cell edges in the x-direction

$$Q^L_{ij} = \frac{1}{\Delta} y \int_{y_{j-1/2}}^{y_{j+1/2}} Q(x^-_{i+1/2}, y)\,dy,$$

$$Q^R_{ij} = \frac{1}{\Delta} y \int_{y_{j-1/2}}^{y_{j+1/2}} Q(x^+_{i-1/2}, y)\,dy.$$

To this end, we use linear combinations of three one-dimensional quadratic stencils centred at $(i+1, j)$, (i, j), and $(i-1, j)$. The corresponding smoothness indicators read

$$\begin{aligned}
\beta^0_{ij} &= \frac{13}{12}\big(Q^n_{ij} - 2Q^n_{i+1,j} + Q^n_{i+2,j}\big)^2 + \frac{1}{4}\big(3Q^n_{ij} - 4Q^n_{i+1,j} + Q^n_{i+2,j}\big)^2,\\
\beta^1_{ij} &= \frac{13}{12}\big(Q^n_{i-1,j} - 2Q^n_{ij} + Q^n_{i+1,j}\big)^2 + \frac{1}{4}\big(Q^n_{i-1,j} - Q^n_{i+1,j}\big)^2,\\
\beta^2_{ij} &= \frac{13}{12}\big(Q^n_{i-2,j} - 2Q^n_{i-1,j} + Q^n_{ij}\big)^2 + \frac{1}{4}\big(Q^n_{i-2,j} - 4Q^n_{i-1,j} + 3Q^n_{ij}\big)^2.
\end{aligned}$$

The optimal linear weights for the left extrapolated value Q^L_{ij} are

$$C^0 = \frac{3}{10}, \quad C^1 = \frac{6}{10}, \quad C^2 = \frac{1}{10},$$

and Q^L_{ij} becomes

$$\begin{aligned}
Q^L_{ij} =& \frac{w^0_{ij}}{6}\Big(2Q^n_{ij} + 5Q^n_{i+1,j} - Q^n_{i+2,j}\Big) + \frac{w^1_{ij}}{6}\Big(-Q^n_{i-1,j} + 5Q^n_{ij} + 2Q^n_{i+1,j}\Big)\\
&+ \frac{w^2_{ij}}{6}\Big(2Q^n_{i-2,j} - 7Q^n_{i-1,j} + 11Q^n_{ij}\Big).
\end{aligned}$$

By symmetry, the linear weights for Q^R_{ij} are

$$C^0 = \frac{1}{10}, \quad C^1 = \frac{6}{10}, \quad C^2 = \frac{3}{10},$$

and the edge average itself becomes

$$\begin{aligned}
Q^R_{ij} =& \frac{w^0_{ij}}{6}\Big(11Q^n_{ij} - 7Q^n_{i+1,j} + 2Q^n_{i+2,j}\Big) + \frac{w^1_{ij}}{6}\Big(2Q^n_{i-1,j} + 5Q^n_{ij} - Q^n_{i+1,j}\Big)\\
&+ \frac{w^2_{ij}}{6}\Big(-Q^n_{i-2,j} + 5Q^n_{i-1,j} + 2Q^n_{ij}\Big).
\end{aligned}$$

In the second step, we start from the edge averages defined above and reconstruct point values $Q(x^-_{i+1/2}, y_{j+\pm\alpha})$ at the Gaussian integration points. For the first integration point $(x_{i+1/2}, y_j - \Delta y/2\sqrt{3})$, the optimal weights are

$$C^0 = \frac{210 - \sqrt{3}}{1080}, \quad C^1 = \frac{11}{18}, \quad C^2 = \frac{210 + \sqrt{3}}{1080},$$

and the reconstructed point value reads

$$\begin{aligned}
Q(x_{i+1/2}, y_{j-\alpha}) =& w^0_{ij}\Big(Q^L_{ij} + \big(3Q^L_{ij} - 4Q^L_{i+1,j} + Q^L_{i+2,j}\big)\frac{\sqrt{3}}{12}\Big)\\
&+ w^1_{ij}\Big(Q^L_{ij} - \big(Q^L_{i+1,j} - Q^L_{i-1,j}\big)\frac{\sqrt{3}}{12}\Big)\\
&+ w^2_{ij}\Big(Q^L_{ij} - \big(3Q^L_{ij} - 4Q^L_{i-1,j} + Q^L_{i-2,j}\big)\frac{\sqrt{3}}{12}\Big).
\end{aligned}$$

Similarly, for the second Gaussian point $(x_{i+1/2}, y_j + \Delta y/2\sqrt{3})$

$$C^0 = \frac{210 + \sqrt{3}}{1080}, \quad C^1 = \frac{11}{18}, \quad C^2 = \frac{210 - \sqrt{3}}{1080},$$

and

$$\begin{aligned} Q(x_{i+1/2}, y_{j+\alpha}) =& w_{ij}^0 \Big(Q_{ij}^L - \big(3Q_{ij}^L - 4Q_{i+1,j}^L + Q_{i+2,j}^L\big) \frac{\sqrt{3}}{12} \Big) \\ &+ w_{ij}^1 \Big(Q_{ij}^L + \big(Q_{i+1,j}^L - Q_{i-1,j}^L\big) \frac{\sqrt{3}}{12} \Big) \\ &+ w_{ij}^2 \Big(Q_{ij}^L + \big(3Q_{ij}^L - 4Q_{i-1,j}^L + Q_{i-2,j}^L\big) \frac{\sqrt{3}}{12} \Big). \end{aligned}$$

To reconstruct the point values $Q(x_{i-1/2}^+, y_{j\pm\alpha})$, we start from Q_{ij}^R and repeat the second step. A completely analogous procedure is then used in the y-direction to construct the point values $Q(x_{i\pm\alpha}, y_{j+1/2}^\pm)$.