Solving the Euler Equations on Graphics Processing Units

Trond Runar Hagen[1,2], Knut-Andreas Lie[1,2], and Jostein R. Natvig[1,2]

[1] SINTEF, Dept. Applied Math., P.O. Box 124 Blindern, N-0314 Oslo, Norway
[2] Centre of Mathematics for Applications (CMA),University of Oslo, Norway
{trr, knl, jrn}@sintef.no
http://www.sintef.no/gpgpu

Abstract. The paper describes how one can use commodity graphics cards (GPUs) as a high-performance parallel computer to simulate the dynamics of ideal gases in two and three spatial dimensions. The dynamics is described by the Euler equations, and numerical approximations are computed using state-of-the-art high-resolution finite-volume schemes. These schemes are based upon an explicit time discretisation and are therefore ideal candidates for parallel implementation.

1 Introduction

Conservation of physical quantities is a fundamental physical principle that is often used to derive models in the natural sciences. In this paper we will study one such model, the Euler equations describing the dynamics of an ideal gas based on conservation laws for mass, momentum, and energy. In three spatial dimensions the Euler equations read

$$\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ E\end{bmatrix}_t + \begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ \rho uw\\ u(E+p)\end{bmatrix}_x + \begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ \rho vw\\ v(E+p)\end{bmatrix}_y + \begin{bmatrix}\rho w\\ \rho uw\\ \rho vw\\ \rho w^2+p\\ w(E+p)\end{bmatrix}_z = \begin{bmatrix}0\\ 0\\ 0\\ g\rho\\ g\rho w\end{bmatrix}. \tag{1}$$

Here ρ denotes the density, (u, v, w) the velocity vector, p the pressure, g the acceleration of gravity, and E the total energy (kinetic plus internal energy) given by $E = \rho(u^2+v^2+w^2)/2+p/(\gamma-1)$. In all computations we use $\gamma = 1.4$. The Euler equations are one particular example of a large class of equations called *hyperbolic systems of conservation laws*, which can be written on the form

$$Q_t + F(Q)_x + G(Q)_y + H(Q)_z = S(Q). \tag{2}$$

This class of PDEs exhibits very singular behaviour and admits various kinds of discontinuous and nonlinear waves, such as shocks, rarefactions, phase boundaries, fluid and material interfaces, etc. Resolving propagating discontinuities accurately is a difficult task, to which a lot of research has been devoted in the

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part IV, LNCS 3994, pp. 220–227, 2006.

last 2–3 decades. Today, a successful numerical method will typically be of the high-resolution type (see e.g., [3]) and be able to accurately capture discontinuous waves and at the same time offer high-order resolution of smooth parts of the solution.

Modern high-resolution methods for nonstationary problems are typically based upon explicit temporal discretisation. In explicit methods there is *no* coupling between unknowns in different grid cells, and one therefore avoids the use of linear system solvers, which is a typical bottleneck in many fluid dynamics algorithms. High-resolution methods are therefore relatively easy to parallelise, using e.g., domain decomposition. In this paper we will discuss parallel implementation of gas-dynamics simulations on commodity graphics cards (GPUs) residing in recent desktop computers and workstations. The idea of using GPUs for numerical simulation is far from new—cf. e.g., `http://www.gpgpu.org`—but except for our previous work [1] on shallow water waves, this is the first paper to consider a GPU implementation of high-resolution schemes for models on the form (2).

From a computational point-of-view, a modern GPU can be considered as a single-instruction, multiple-data processor capable of parallel processing of floating-point numbers. Whereas an Intel Pentium 4 CPU has a theoretical performance of at most 15 Gflops, performance numbers as high as 165 Gflops have been reported for the NVIDIA GeForce 7800 cards. The key to this unrivalled processing power is the fact that current GPUs contain up to 24 parallel pipelines that each are capable of processing vectors of length four simultaneously. By exploiting this amazing computational power for 2D and 3D gas-dynamics simulations, we observe speedup factors of order 10—20 on a single workstation.

2 Numerical Methods

To solve the Euler equations in two and three dimensions, we will use a family of semi-discrete finite-volume schemes on a regular Cartesian grid and seek approximations to (2) in terms of the cell-averages $Q_{ijk} = \frac{1}{|\Omega_{ijk}|}\int_{\Omega_{ijk}} Q\,dV$. Integrating (2) over Ω_{ijk}, we obtain an evolution equation for the cell-averages

$$\begin{aligned}\frac{dQ_{ijk}}{dt} = &-\big(F_{i+1/2,jk} - F_{i-1/2,jk}\big) - \big(G_{i,j+1/2,k} - G_{i,j-1/2,k}\big)\\ &- \big(H_{ij,k+1/2} - H_{ij,k-1/2}\big) + S_{ijk},\end{aligned} \tag{3}$$

where $F_{i\pm1/2,jk}$ denote the flux over the surfaces with normal along the x-axis, etc. The fluxes are approximated using a standard Gaussian quadrature (fourth order, tensor product rule):

$$\begin{aligned}F_{i+1/2,jk}(t) &= \frac{1}{|\Omega_{ijk}|}\int_{y_{j-1/2}}^{y_{j+1/2}}\int_{z_{k-1/2}}^{z_{k+1/2}} F\big(Q(x_{i+1/2},y,z,t)\big)\,dydz\\ &\approx \frac{1}{4\Delta x}\sum_{n,m=\{-1,1\}} F\Big(Q\Big(x_{i+1/2},\, y_j + n\frac{\Delta y}{2\sqrt{3}},\, z_k + m\frac{\Delta z}{2\sqrt{3}},\, t\Big)\Big).\end{aligned} \tag{4}$$

To evaluate the integrand, we need to *reconstruct* a continuously defined function from the cell-averages. To this end, we will use a function that is piecewise continuous inside each grid cell. In a first-order method, one would use a piecewise constant function. To obtain second-order (on smooth solutions), we use a piecewise linear reconstruction for each component in Q

$$\begin{aligned}\hat{Q}_{ijk}(x,y,z) = {} & Q_{ij} + L\big(D_x^+ Q_{ijk}, D_x^- Q_{ijk}\big)\frac{x-x_i}{\Delta x} \\ & + L\big(D_y^+ Q_{ijk}, D_y^- Q_{ijk}\big)\frac{y-y_j}{\Delta y} + L\big(D_z^+ Q_{ijk}, D_z^- Q_{ijk}\big)\frac{z-z_k}{\Delta z},\end{aligned} \tag{5}$$

where $D_x^{\pm} = \pm(Q_{i\pm1,jk} - Q_{ijk})$, etc. The so-called limiter L is a nonlinear function of the forward and backward differences, whose purpose is to prevent the creation of overshoots at local extrema. Here we use the family of generalised minmod limiters

$$L(a,b) = \mathrm{MM}(\theta a, \tfrac{1}{2}(a+b), \theta b), \quad \mathrm{MM}(z_1,\ldots,z_n) = \begin{cases} \max_i z_i, & z_i < 0\ \forall i, \\ \min_i z_i, & z_i > 0\ \forall i, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

The reconstruction $\hat{Q}(x,y,z)$ is discontinuous across all cell-interfaces, and thus gives a left-sided and right-sided point value, Q_L and Q_R, at each integration point in (4). To evaluate the flux across the interface at each integration point, we use the central-upwind flux [2]

$$\begin{aligned}\mathcal{F}(Q^L,Q^R) &= \frac{a^+F(Q^L) - a^-F(Q^R)}{a^+ - a^-} + \frac{a^+a^-}{a^+ - a^-}(Q^R - Q^L), \\ a^+ &= \max\big(0, \lambda^+(Q^L), \lambda^+(Q^R)\big), \quad a^- = \min\big(0, \lambda^-(Q^L), \lambda^-(Q^R)\big),\end{aligned} \tag{7}$$

where $\lambda^{\pm}(Q)$ are the slow and fast eigenvalues of dF/dQ, given analytically as $u \pm \sqrt{\gamma p/\rho}$.

Finally, we need to specify how to integrate the ODEs (3) for the cell-averages. To this end, we use a second-order TVD Runge–Kutta method [5]

$$\begin{aligned}Q_{ij}^{(1)} &= Q_{ij}^n + \Delta t R_{ij}(Q^n), \\ Q_{ij}^{n+1} &= \frac{1}{2}Q_{ij}^n + \frac{1}{2}\big[Q_{ij}^{(1)} + \Delta t R_{ij}(Q^{(1)})\big],\end{aligned} \tag{8}$$

where R_{ij} denotes the right-hand side of (3). The time step is restricted by a CFL-condition, which states that disturbances can travel at most one half grid cell each time step, i.e., $\max(a^+, -a^-)\Delta t \le \Delta x/2$, and similarly in y and z.

3 GPU Implementation

The data-driven programming model of GPUs is quite different from the instruction-driven programming model most people are used to on a CPU. On a CPU,

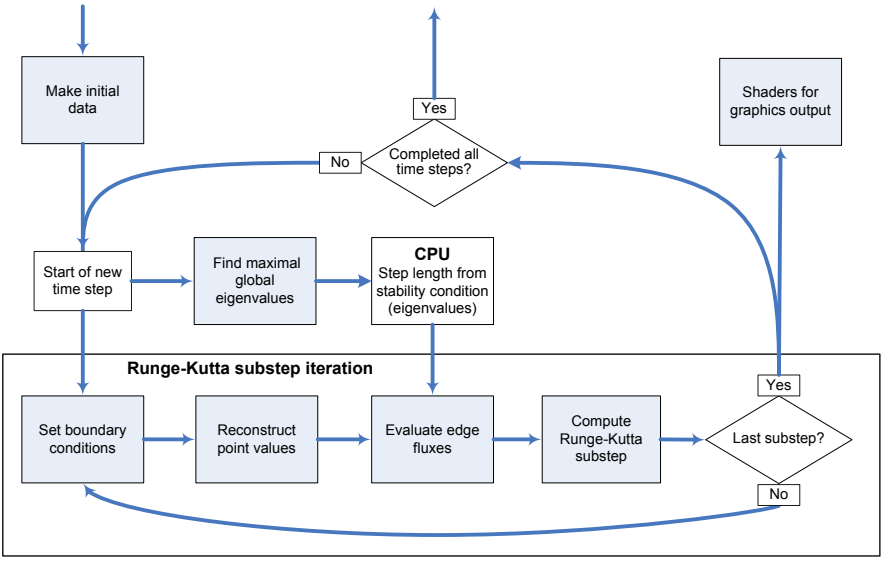

Fig. 1. Flow chart for the GPU implementation of the semi-discrete finite-volume scheme. Gray boxes are executed on the GPU and white boxes on the CPU.

a computer program for the algorithm in Section 2 would consist of a set of arrays and the processing is performed as series of loops that march through all cells to compute reconstructions, integrate fluxes, compute flux differences and evolve the ODEs, etc. On the GPU, each grid cell is associated with a *pixel* (*fragment*) in an off-screen frame buffer. The data stream (cell-averages, fluxes, etc.) is given as *textures* and is invoked by rendering a geometry to a frame buffer. The data stream is processed by a series of kernels (fragment shaders, in graphics terminology) using the fragment-processing capabilities in the rendering pipeline. Writing each computational kernel using Cg (or GLSL) is straightforward for any computational scientist capable of writing C/C++. However, setting up the graphics pipeline requires some familiarity with computer graphics (in our case, OpenGL).

The flow chart for the simulation algorithm is given in Figure 1. Worth noting is the computation of the maximum eigenvalues to determine the time step. Finding the maximum is implemented using an 'all-reduce' operation utilising the depth buffer combined with a read-back to the CPU; see e.g., [1]. In each of the two Runge–Kutta steps, four basic operations are performed. First we set the boundary data. Then we compute the reconstruction using (5) and (6). The most computationally intensive step is the evaluation of edge fluxes and computation of the source term. Before this calculation can start, the time step Δt must be passed to the shader by the CPU. Finally, the step is completed by adding fluxes and the source term to the cell averages. To complete a full time step, the sequence of operations in the Runge–Kutta box are performed twice.

For simulations in 2D, the vectors with cell-averages, slopes, fluxes, etc. have length four and can each be fitted in a single texture RGBA-element. In 3D, the vectors have length five and do not fit in a single RGBA-element. We therefore chose to split each vector in two three-component textures. Notice that this opens up for adding up to three extra quantities in the vector of unknowns (e.g., to represent two or more gases with different γ's) at a very low computational cost, since the GPU processes 4-vectors simultaneously.

In 2D, the Cartesian grid is simply embedded in a rectangle. In 3D, the grid (padded with ghost-cells to represent the boundary) is unfolded in the z-direction and the 3D arrays of vectors are mapped onto larger 2D textures. Memory limitations on the GPU will restrict the sizes of the grids one can process in a single batch. To be able to run highly resolved simulations in 3D, we will therefore use a domain decomposition approach, in which the domain is divided into smaller rectangular blocks that can be handled separately. The algorithm is straightforward: Each subdomain is extended with one grid-layer of overlap into the neighbouring subdomains for each time-step to be carried out. The initial data is passed by the CPU to the GPU, which performs a given number of time-steps as described above. The result is then read back to the CPU, where it is inserted into the corresponding subdomain in the global solution on the next (global) time level. Since passing of initial data and read-back of computational results can be performed asynchronously between the CPU and the GPU, the performance reduction due to stalls on the GPU will be insignificant. Moreover, this algorithm easily extends to multiple CPU-GPU configurations by implementing some kind of message passing and control on the CPU-side.

4 Numerical Examples

In the following we present a few numerical examples to assess the computational efficiency of our GPU implementation. To this end, we compare runtimes on two NVIDIA GeForce graphics cards (6800 Ultra and 7800 GTX) with runtimes on two different CPUs (a 2.8 GHz Intel Xeon CPU and an AMD Athlon X2 4400+, respectively). The timings are averaged over all timesteps and do not include any preprocessing. The CPU reference codes are implemented in C, using a design that has evolved during 6–7 years research on high-resolution schemes. High computational efficiency has been ensured by carefully minimising the number of arithmetic operations, optimal ordering of loops, use of temporary storage, replacing divisions by multiplications whenever possible, etc. Apart from that, our CPU codes contain no hardware-specific hand-optimisation, but rather rely on general compiler optimisation; `icc -O3 -ipo -xP` (version 8.1) for the Intel CPU and `gcc -O3` for the AMD. To ensure a fair comparison, we have used the same design choices for the GPU implementations, trying to retain a one-to-one correspondence of statements in the CPU and GPU computational kernels.

Another important question is accuracy. The numerical methods considered in the paper are stable and the accuracy will therefore not deteriorate significantly due to rounding errors. In fact, all our tests indicate that the difference between

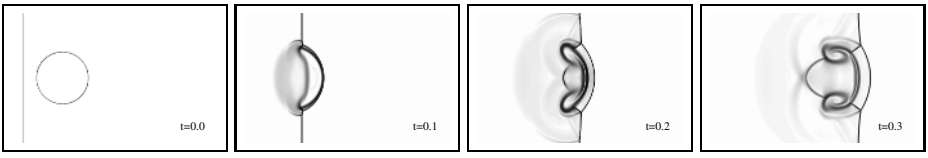

Fig. 2. Emulated Schlieren images of a shock-bubble interaction

Table 1. Runtime per time step in seconds and speedup factor for two CPUs versus two GPUs for the shock-bubble problem run on a grid with $N \times N$ cells for bilinear (upper part) and CWENO reconstruction (lower part)

N	Intel	6800	speedup	AMD	7800	speedup
128	4.37e-2	3.70e-3	11.8	1.88e-2	1.38e-3	13.6
256	1.74e-1	8.69e-3	20.0	1.08e-1	4.37e-3	24.7
512	6.90e-1	3.32e-2	20.8	2.95e-1	1.72e-2	17.1
1024	2.95e-0	1.48e-1	19.9	1.26e-0	7.62e-2	16.5
128	1.05e-1	1.22e-2	8.6	7.90e-2	4.60e-3	17.2
256	4.20e-1	4.99e-2	8.4	3.45e-1	1.74e-2	19.8
512	1.67e-0	1.78e-1	9.4	1.03e-0	6.86e-2	15.0
1024	6.67e-0	7.14e-1	9.3	4.32e-0	2.99e-1	14.4

single precision (GPU) and double precision (CPU) results are of the order ϵ_s, where $\epsilon_s = 1.192 \cdot 10^{-7}$ is the smallest number such that $1 + \epsilon_s - 1 > 0$ in single precision. In other words, for the applications considered herein, the *discretisation* errors dominate errors due to lack of precision.

Example 1 (2D Shock-Bubble Interaction). In this example we consider the interaction of a planar 2.95 Mach shock in air with a circular region of low density. The gas is initially at rest and has unit density and pressure. Inside a circle of radius 0.2 centred at $(0.4, 0.5)$ the density is 0.1. The incoming shock-wave starts at $x = 0$ and has a post-shock pressure $p = 10.0$. Figure 2 shows the evolution of the bubble in terms of emulated Schlieren images (density gradients depicted using a nonlinear graymap) as described by the 2D Euler equations, i.e., (1) with $g = w \equiv 0$.

Table 1 reports a comparison of average runtime per time step for GPU versus CPU implementations of the high-resolution scheme using either the bilinear reconstruction in (5) and (6), or the third-order CWENO reconstruction [4]. The corresponding schemes thus have second-order accuracy in time and second and third order accuracy in space, respectively. For the bilinear reconstruction, the resulting speedup factors of order 20 and 15 for the GeForce 6800 and 7800, respectively, are quite amazing since we did not try to optimise the GPU implementation apart from the obvious use of vector operations whenever appropriate.

The CWENO reconstruction is quite complicated, and we have observed on various Intel CPUs that `icc` gives significantly faster code than `gcc`. We therefore

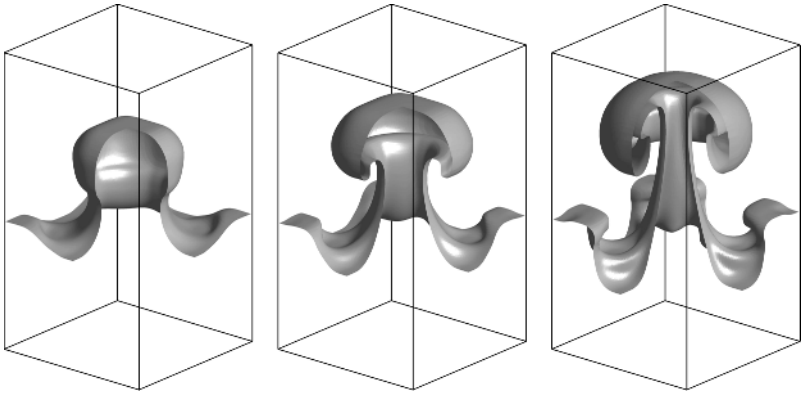

Fig. 3. The Rayleigh–Taylor instability at times $t = 0.5$, 0.6, and 0.7

Table 2. Runtime per time step in seconds and speedup factor for CPU versus GPU for the 3D Rayleigh–Taylor instability run on a grid with $N \times N \times N$ cells

N	AMD	7800	speedup
49	5.23e-1	4.16e-2	12.6
64	1.14e-0	8.20e-2	13.9
81	1.98e-0	1.72e-1	11.5

expect the CWENO code to be suboptimal on the AMD CPU. Moreover, due to the large number of temporary registers required in the CWENO reconstruction, we had to split the computation of edge fluxes in two passes on the GPU: one for the F-fluxes and one for the G-fluxes. This introduces extra computations compared with the CPU code and also extra render target switches and texture fetches. This reduces the theoretical speedup by factor between 25 and 50%, as can be seen in the lower half of Table 1, comparing the 6800 card and the Intel CPU with the `icc` compiler.

Example 2 (3D Rayleigh–Taylor Instability). In the next example we simulate a Rayleigh–Taylor instability, which arises when a layer of heavier fluid is placed on top of a lighter fluid and the heavier fluid is accelerated downwards by gravity. Similar phenomena occur more generally when a light fluid is accelerated towards a heavy fluid. In the simulation, we consider the domain $[-1/6, 1/6]^2 \times [0.2, 0.8]$ with gravitational acceleration $g = 0.1$ in the z-direction. The lower fluid has unit density and the upper fluid density $\rho = 2.0$. Initially the two fluids are at rest, in hydrostatic balance, and separated by an interface located at $z = 1/2 + 0.01\cos(6\pi \min(\sqrt{x^2 + y^2}, 1/6))$. Reflective boundary conditions are assumed on all exterior boundaries. Figure 3 shows the evolution of the instability.

Table 2 reports a comparison of average runtime per time step for a GPU versus a CPU implementation for the high-resolution scheme with the trilinear

reconstruction in (5) and (6). Compared with the 2D simulation, the speedup is reduced. There are two factors contributing the the reduced speedup: (i) cache misses on the GPU due to lookup in the unfolded 3D texture, and (ii) use of only three out of four vector components in all basic arithmetic operations. For the 2D solver, all texture fetches are to neighbouring texel locations, whereas the access in the z-direction introduces non-local texture fetches. Similarly, the 2D solver uses a single four-component texture to represent the conserved quantities, whereas the 3D solver needs to use two three-component textures: $(\rho, \rho u, \rho v, \rho w, E)$, plus a passive tracer to distinguish the gases.

5 Concluding Remarks

In this paper we have demonstrated the application of GPUs as high-performance computational engines for compressible gas dynamics simulations in 2D and 3D. Unlike many other fluid dynamics algorithms, the current high-resolution schemes do not involve any linear system solvers, which may be a performance bottleneck in many GPU/parallel implementations. Instead, the schemes are based upon explicit temporal discretisation, for which each cell can be updated independent of the others. This makes the schemes perfect candidates for parallel implementation. Moreover, a high number of arithmetic operations per memory fetch makes these algorithms ideal for high performance data-stream based computer architectures and results in fairly amazing speedup numbers.

The possibility of using a simple domain-decomposition algorithm to handle large simulation models makes it attractive to explore future use of clusters of CPU–GPU nodes for this type of simulations. The communication need between different nodes is low compared with the computations performed on each GPU, and communication bandwidth is therefore not expected to be a major issue.

Acknowledgement

The research is funded by the Research Council of Norway under grants number 158911/I30 (Hagen and Lie) and 139144/431 (Natvig).

References

1. Hagen, T.R., Hjelmervik, J.M., Lie, K.-A., Natvig, J.R., Henriksen, M.O.: Visual simulation of shallow-water waves. Simul. Model. Pract. Theory, **13** (2005) 716–726.
2. Kurganov, A, Noelle, S., Petrova, G.: Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations. SIAM J. Sci. Comput. **23 (3)** (2001) 707–740.
3. LeVeque, R: Finite volume methods for hyperbolic problems, Cambridge Texts in Applied Mathematics, Cambridge University Press, Cambridge, 2002.
4. Levy, D., Puppo, G., Russo. G.:Compact central WENO schemes for multidimensional conservation laws. SIAM J. Sci. Comput. **22(2)** (2000) 656–672.
5. Shu, C.-W.: Total-variation-diminishing time discretisations. SIAM J. Sci. Stat. Comput. **9** (1988) 1073–1084.