Computers & Fluids 99 (2014) 156–171

Contents lists available at ScienceDirect

Computers & Fluids

journal homepage: www.elsevier.com/locate/compfluid

Solving seven-equation model for compressible two-phase flow using multiple GPUs

Shan Liang [a,b,c], Wei Liu [a], Li Yuan [a,*]

[a] LSEC and NCMIS, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, China
[b] Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China
[c] State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214000, China

ARTICLE INFO

Article history:
Received 17 October 2013
Received in revised form 17 April 2014
Accepted 20 April 2014
Available online 28 April 2014

Keywords:
Compressible multiphase flow
Seven-equation model
HLLC
TVD Runge–Kutta
GPU computing

ABSTRACT

In this paper, the application of an HLLC-type approximate Riemann solver in conjunction with the third-order TVD Runge–Kutta method to the seven-equation compressible two-phase model on multiple Graphics Processing Units (GPUs) is presented. Based on the idea proposed by Abgrall et al. that "a multiphase flow, uniform in pressure and velocity at $t = 0$, will remain uniform on the same variables during time evolution", discretization schemes for the non-conservative terms and for the volume fraction evolution equation are derived in accordance with the HLLC solver used for the conservative terms. To attain high temporal accuracy, the third-order TVD Runge–Kutta method is implemented in conjunction with operator splitting technique, in which the sequence of operators is recorded in order to compute free surface problems robustly. For large scale simulations, the numerical method is implemented using MPI/Pthread-CUDA parallelization paradigm for multiple GPUs. Domain decomposition method is used to distribute data to different GPUs, parallel computation inside a GPU is accomplished using CUDA, and communication between GPUs is performed via MPI or Pthread. Efficient data structure and GPU memory usage are employed to maintain high memory bandwidth of the device, while a special procedure is designed to synchronize thread blocks so as to reduce frequencies of kernel launching. Numerical tests against several one- and two-dimensional compressible two-phase flow problems with high density and high pressure ratios demonstrate that the present method is accurate and robust. The timing tests show that the overall speedup of one NVIDIA Tesla C2075 GPU is 31× compared with one Intel Xeon Westmere 5675 CPU core, and nearly 70% parallel efficiency can be obtained when using 8 GPUs.

1. Introduction

Compressible two-phase flows exist broadly in nature and industry (like bubbles in ocean, cavitations in hydraulic machinery, flows in chemical reactors and cooling circuits of power plants). Numerical simulations of compressible two-phase flows are important research topics. As this type of flows are complex and diverse, a variety of two-phase models with various levels of complexity were proposed in literature, like the complete seven-equation model [1–3], the reduced six-equation model [4], and the more reduced five-equation model [5], to mention just a few. Most two-phase models are derived from integrating individual balance equations weighted by a characteristic function for each phase. This volume average procedure removes the interfacial detail while introducing additional non-conservative terms for describing interactions between phases. The resultant two-phase models pose challenge to numerical solutions mainly due to the complicated characteristics of the equation system and the troublesome non-conservative terms.

In this paper, we are interested in numerical solution of the compressible seven-equation two-phase model [2,3]. In this model, each phase is assumed to have its own velocity, pressure and density, which satisfy respective balance equations. The evolution equation of volume fraction is introduced from integrating the characteristic function to describe how fluid compositions change with time. Due to non-equilibrium of velocity and pressure, drag forces appear between phases causing momentum and energy exchange. In the case of one space dimension, the model has seven equations (two sets of mass, momentum, and energy equations, one volume fraction evolution equation). The advantage of this model is that it is unconditionally hyperbolic, and can treat a wide range of applications including non-equilibrium dispersive multiphase flows as well as free-surface multi-fluid flows [3]. For the latter case, the velocity and pressure of all phases on each side of the interface must be in equilibrium from a physical point of view

* Corresponding author. Tel.: +86 10 62625219.
E-mail address: lyuan@lsec.cc.ac.cn (L. Yuan).

http://dx.doi.org/10.1016/j.compfluid.2014.04.021
0045-7930/

[2,3,6], which can be realized by infinite pressure and velocity relaxation process in the model. As the volume fraction only stands for the constitutive fluid distribution, the material surface is indirectly represented by location where large gradient occurs. The bulk interface is tracked without considering the details even when the distortion is complicated (cavitation, breakdown and coalescence of bubbles, etc.). Of course, the computational cost is larger than a free-surface oriented method, e.g., it is about three times as that of the ghost fluid method according to our experience.

Although the seven-equation model is unconditional hyperbolic, the numerical solution has particular difficulties because it is hard to solve the associated Riemann problem with a large system of equations, and careless approximations to the non-conservative terms in the momentum and energy equations and the non-conservative evolution equation (the volume fraction equation) will often lead to failure in computation. Therefore, the key in numerical solution is to construct an accurate and efficient approximate Riemann solver and at the same time derive corresponding discretization schemes for the non-conservative terms and the non-conservative volume fraction equation.

Many studies have devoted to numerical solution of compressible two-phase models in various variants. Saurel et al. [2,3] used operator splitting approach to treat the hyperbolic part and the relaxation terms of their seven-equation model, but the adopted HLL approximate Riemann solver led to excessive numerical diffusion of contact discontinuities due to the use of only two waves in lieu of full waves. Li et al. [7] developed a simple HLLC scheme for the seven-equation model, but they only considered the subsonic case, and used Roe average for the unknown intermediate state of the volume fraction. Zein et al. [8] also presented a simple HLLC-type scheme for the seven-equation model that took into account the heat and mass transfer through relaxation effects. Combining the thin layer theory with special choice for interfacial variables in liquid–solid problems, Tokareva and Toro [9] proposed a HLLC-type approximate Riemann solver which took into account full waves for the Baer-Nunzatio model. Tian et al. [10] implemented the path-conservative method and a simple HLLC solver for the reduced five-equation model. Yeom and Chang [4] presented a modified HLLC-type scheme for a six-equation model which restores the characteristic fields that have been neglected in the Zein's simple HLLC-type scheme [8]. A more thorough effort to construct approximate Riemann solver for the Saurel-Abgrall model was made recently by Ambroso et al. [11]. Their definition of Riemann problem included not only convective terms and non-conservative terms, but also source terms associated with gravity and drag forces (the drag force source is often separately treated as velocity relaxation process), while pressure relaxation process was split from them alone. In all the work mentioned above, the multiphase flow equations were approximated by numerical methods, but a strategy proceeded in the opposite way was proposed by Abgrall [12], which dealt with mixtures and interfaces under a unique formulation. They started from the pure phase Euler equations at the microscopic level, and gave corresponding numerical approximations via the Godunov scheme and the HLLC flux. After randomization, ensemble average procedures and estimation of the various coefficients of these approximations, numerical scheme for the averaged multiphase flow equations was derived.

In this study, the first objective is to develop a robust high resolution numerical method for the Saurel-Abgrall's seven-equation compressible two-phase model, which has the simple form of a conventional HLLC scheme and the high temporal accuracy of the third-order TVD Rung-Kutta method. We advance the solution with the third-order TVD Runge–Kutta method, inserting the splitting strategy [3] for the hyperbolic operator and the relaxation operators into every sub-step of the Runge–Kutta method. We reorder the sequence of the split operators so that the resulting Runge–Kutta method can work robustly for extreme compressible gas–liquid two-fluid flow problems with high density and high pressure ratios. To obtain a simple scheme, we apply the conventional HLLC flux to the conservative part of the two-phase model in a way similar to Li [7] and Zein [8], and then utilize the homogeneity idea for a multi-phase system [6] to derive discrete formulas for the non-conservative terms and the non-conservative evolution equation corresponding to the HLLC flux used. Our derivation takes into account both subsonic and supersonic cases of the HLLC scheme rather than only subsonic case as did in Ref. [7].

The second objective of this study is to efficiently reduce the simulation time posed by numerical solution of the seven-equation two-phase model. To this end, we implement our numerical method using CUDA-GPU parallel computing technology. CUDA (Compute Unified Device Architecture) [13] is a programming model for realizing general purpose GPU (Graphics Processing Unit) computing. Recently there is a surge in hybrid CPU/GPU computations using rapidly evolving GPU architectures and CUDA programming paradigm [14]. In single GPU computing, Ref. [15] accelerated a solver for the Euler equations, and observed 29× and 16× speedups for 2D and 3D problems respectively. Ref. [16] pioneered GPU acceleration of problems on non-uniform and irregular grids for the 3D compressible Euler equations, and obtained 15× to 40× speedups using NVIDIA 8800GTX GPU compared with a single Intel Core 2 Duo E6600(2.4 GHz) CPU. When conducting GPU computing on multiple GPUs, three parallel modes are available: single-thread multi-stream mode (CUDA 4.0 and above), in which every involved device is bounded to a CUDA stream; multi-thread multi-GPU mode (Pthread- or OpenMP-CUDA), in which more than one threads are invoked and each thread controls one GPU; multi-process multi-GPU mode (MPI-CUDA). The former two modes are applicable only to a shared memory machine with several GPUs, while the third one is also applicable to a cluster with many GPUs. Ref. [17] implemented 3D incompressible Navier–Stokes equations in CUDA with the help of Pthread. Using four Tesla C870 GPUs, the computation time was reduced to 1/100 of single AMD Opteron 2.4 GHz CPU, and 1/3 of a single GPU. Ref. [18] obtained linear speedup for fewer than four GPUs when solving a 3D equation using a high order finite difference method. Ref. [19] optimized a finite difference code for direct numerical simulations of turbulence on a GPU accelerated cluster, and obtained a speedup of 20× for 192 M2070 Fermi GPUs vs. 192 Xeon Westmere 2.93 GHz CPU cores. In their implementation, all computations were packed in kernel functions for running in the devices (GPUs), and the communication between devices was done through MPI. The boundary cells of each subdomain were dealt with first, and data copy as well as message passing process were synchronized with inner cell computation, through which the time latency due to data copy and message passing was hidden efficiently.

In our implementation, we use both hybrid MPI-CUDA and Pthread-CUDA parallelization on a shared memory AMAX machine with 2× Intel Xeon Westmere 5675 3.06 GHz six-core CPUs, connected via PCIE2 slot to 8× NVIDIA Tesla C2075 Fermi GPUs. Each computational grid point is mapped to a GPU thread, and appropriate data structure is adopted to exploit the high memory bandwidth of GPU. We design a special procedure including atom operator to synchronize thread blocks. This make reduction operations like *min* or *max* execute completely inside GPU without exiting a kernel, thus eliminating numerous kernel-launching overheads. Besides, we use domain decomposition method for multi-GPU computing. The computation of every subdomain is assigned to a device, while the communication between devices is performed through MPI or Pthread. Both modes are compared in numerical tests.

The paper is organized as follows. We begin with a brief discussion of PDEs of the seven-equation model and their physical explanation in Section 2. Then, in Section 3, we give the HLLC numerical flux in 1D case, and derive corresponding discrete schemes for the non-conservative terms and the volume fraction evolution equation so as to construct a Godunov-type scheme for the hyperbolic part. And then we describe the combination of the third-order TVD Runge–Kutta method with the operator splitting in a way to realize robust computation of free-surface flows. The implementation details of the above numerical method on single and multiple GPUs are introduced in Section 4. Numerical tests involving multiphase mixtures and two pure fluids are conducted in Section 5, in which we compare our method with the Real Ghost Fluid Method (RGFM) [20] and the HLL scheme [2,3] for the same seven-equation model, and several large scale problems are simulated using up to 8 GPUs. The last section is a conclusion.

2. Seven-equation model

For the sake of simplicity, we only consider 1D case and extension of the algorithm to 2D Cartesian grids is traditional (e.g., [3]) which is not discussed in this paper. Suppose that two kinds of fluid exist in the system, and there is neither mass transfer nor heat exchange between them. The compressible seven-equation two-phase model [2] can be written in the following form:

$$\frac{\partial \mathbf{U}}{\partial t}+\frac{\partial \mathbf{F}(\mathbf{U})}{\partial x}=\mathbf{H}(\mathbf{U})\frac{\partial \alpha_1}{\partial x}+\mathbf{S}_v(\mathbf{U})+\mathbf{S}_p(\mathbf{U}), \tag{1}$$

where

$$\mathbf{U}=\begin{pmatrix}\alpha_1\\ \alpha_1\rho_1\\ \alpha_1\rho_1u_1\\ \alpha_1\rho_1E_1\\ \alpha_2\rho_2\\ \alpha_2\rho_2u_2\\ \alpha_2\rho_2E_2\end{pmatrix},\quad \mathbf{F}(\mathbf{U})=\begin{pmatrix}0\\ \alpha_1\rho_1u_1\\ \alpha_1\rho_1u_1^2+\alpha_1p_1\\ \alpha_1u_1(\rho_1E_1+p_1)\\ \alpha_2\rho_2u_2\\ \alpha_2\rho_2u_2^2+\alpha_2p_2\\ \alpha_2u_2(\rho_2E_2+p_2)\end{pmatrix},\quad \mathbf{H}(\mathbf{U})=\begin{pmatrix}-u_I\\ 0\\ p_I\\ u_Ip_I\\ 0\\ -p_I\\ -u_Ip_I\end{pmatrix},$$

$$\mathbf{S}_v(\mathbf{U})=\begin{pmatrix}0\\ 0\\ \lambda(u_2-u_1)\\ \lambda u_I(u_2-u_1)\\ 0\\ -\lambda(u_2-u_1)\\ -\lambda u_I(u_2-u_1)\end{pmatrix},\quad \mathbf{S}_p(\mathbf{U})=\begin{pmatrix}-\mu(p_2-p_1)\\ 0\\ 0\\ \mu p_I(p_2-p_1)\\ 0\\ 0\\ -\mu p_I(p_2-p_1)\end{pmatrix}.$$

Here ρ, u, p and $E=e+\frac{1}{2}u^2$ represent the density, velocity, pressure and total specific energy respectively, and subscripts $k=1$ or 2 is related to phase k. α_k is the volume fraction ranged from 0 to 1, and satisfies the saturation constraint $\alpha_1+\alpha_2=1$. When $\alpha_k=0$ there is no k-th fluid. Actually, to avoid infinite density and velocity computed by mass and energy conservation equations, α_k is set a small value like 10^{-7} in lieu of $\alpha_k=0$ at the initial time. μ and λ are the pressure and velocity relaxation coefficients, and p_I and u_I are the interfacial pressure and the interfacial velocity respectively. Various choices are possible, for example, in the gas–liquid (or solid) problem, $p_I=p_{\text{gas}}$ and $u_I=u_{\text{liquid}}$. Here we follow Ref. [3]:

$$p_I=\sum\alpha_kp_k,\quad u_I=\sum(\alpha_k\rho_ku_k)/\sum\alpha_k\rho_k. \tag{2}$$

3. Numerical method

Based on the operator splitting method [3], Eq. (1) can be decomposed into three parts: the ODE system of pressure relaxation $\mathbf{U}_t=\mathbf{S}_p(\mathbf{U})$, that of velocity relaxation $\mathbf{U}_t=\mathbf{S}_v(\mathbf{U})$, and the non-conservative hyperbolic equations $\mathbf{U}_t+\mathbf{F}(\mathbf{U})_x=\mathbf{H}(\mathbf{U})\alpha_{1x}$. The solution of Eq. (1) at $n+1$ time step was obtained by the succession of operators:

$$\mathbf{U}^{n+1}=L_p^{\Delta t}L_v^{\Delta t}L_H^{\Delta t}(\mathbf{U}^n), \tag{3}$$

where L_p and L_v denote the pressure and velocity relaxation operators respectively [2,3,6], and L_H denotes the non-conservative hyperbolic operator. Scheme (3) is only first order accurate in time and higher accuracy can be obtained by adopting either Strange splitting or Runge–Kutta methods. In this paper, the third-order TVD Runge–Kutta method is used, and the implementation detail will be described in Section 3.2. The relaxation operators can be solved by the instantaneous relaxation procedures [3] as we assume the parameters μ and λ to be infinite in this study. In the following we will focus on the numerical scheme for the non-conservative hyperbolic operator.

3.1. Hyperbolic operator

As done in earlier studies [2,3,7,8], the building of a numerical scheme for multi-phase model is based on the idea proposed in [6] that"a two phase system, uniform in velocity and pressure at $t=0$ will remain uniform on the same variables during time evolution". Nevertheless, there are some freedom in choosing approximate Riemann solvers.

As seen from Eq. (1), the second through seventh component equations describe mass, momentum and energy conservation of two fluids, respectively. These two sets of balance equations have identical forms except a different sign in the non-conservative terms. For clarity, we only show fluid 1 in Eq. (5) and subsequent HLLC-type scheme. The hyperbolic equations under consideration read (we further suppress subscript 1)

$$\frac{\partial \alpha}{\partial t}+u_I\frac{\partial \alpha}{\partial x}=0, \tag{4}$$

$$\frac{\partial \mathbf{q}}{\partial t}+\frac{\partial \mathbf{f}(\mathbf{q},\alpha)}{\partial x}=\mathbf{h}(\mathbf{q},\alpha)\frac{\partial \alpha}{\partial x}, \tag{5}$$

with $\mathbf{q}=(\alpha\rho,\alpha\rho u,\alpha\rho E)^T$, $\mathbf{f}(\mathbf{q},\alpha)=(\alpha\rho u,\alpha\rho u^2+\alpha p,\alpha u(\rho E+p))^T$, $\mathbf{h}(\mathbf{q},\alpha)=(0,p_I,p_Iu_I)^T$.

As shown by Saurel et al. [2,3], the discretization scheme for the volume fraction evolution Eq. (4) can be derived from the energy equation discretized with the adopted approximate Riemann solver for Eq. (5). We will return to this in Section 3.1.1. For Eq. (5) with both conservative and non-conservative terms, a Godunov-type finite volume method reads

$$\mathbf{q}_j^{n+1}=\mathbf{q}_j^n-\frac{\Delta t}{\Delta x}\left(\mathbf{f}_{j+1/2}^n-\mathbf{f}_{j-1/2}^n\right)+\Delta t\mathbf{h}(\mathbf{q}_j^n)\Theta, \tag{6}$$

where $\mathbf{f}_{j\pm1/2}^n$ is a numerical flux, and Θ represents the discrete form of the non-conservative term $\partial\alpha/\partial x$, which we derive in Section 3.1.1.

3.1.1. HLLC-type approximate Riemann solver

Traditionally, Riemann solvers are constructed according to a set of jump conditions derived by characteristic decomposition technique. This procedure for Eq. (5) would be very complicated due to its non-conservative nature [9,11]. In a simplified sense, we directly adopt the conventional HLLC numerical flux [21,22] for $\mathbf{f}$ in Eq. (6) as Refs. [7,8] did:

$$\mathbf{f}_{j+\frac{1}{2}}^{\text{HLLC}}=\begin{cases}\mathbf{f}_L, & \text{if}\quad 0\leqslant s_L\\ \mathbf{f}_L+s_L(\mathbf{q}_{*L}-\mathbf{q}_L), & \text{if}\quad s_L\leqslant 0\leqslant s_*\\ \mathbf{f}_R+s_R(\mathbf{q}_{*R}-\mathbf{q}_R), & \text{if}\quad s_*\leqslant 0\leqslant s_R\\ \mathbf{f}_R, & \text{if}\quad s_R\leqslant 0\end{cases}. \tag{7}$$

Replacing the conservative variables and pressure in the conventional HLLC scheme with $\mathbf{q}$ and αp respectively, we get the conservative variables near the contact discontinuity, $\mathbf{q}_{*L}$ and $\mathbf{q}_{*R}$:

$$\mathbf{q}_{*K} = \alpha_K \rho_K \frac{s_K - u_K}{s_K - s_*} \cdot \begin{cases} 1 \\ s_* \\ E_K + (s_* - u_K)\left[s_* + \frac{p_K}{\rho_K (s_K - u_K)}\right] \end{cases}, \quad K = L, R. \tag{8}$$

The wave speeds s_L, s_R and s_* are common to both fluids (this means that a three-wave approximate Riemann solver is used for the seven-equation model) and are estimated as

$$s_L = \min\{u_{1L} - c_{1L}, u_{2L} - c_{2L}, u_{1R} - c_{1R}, u_{2R} - c_{2R}\},$$
$$s_R = \max\{u_{1L} + c_{1L}, u_{2L} + c_{2L}, u_{1R} + c_{1R}, u_{2R} + c_{2R}\},$$
$$s_* = \frac{\bar{p}_R - \bar{p}_L + \bar{\rho}_L \bar{u}_L (s_L - \bar{u}_L) - \bar{\rho}_R \bar{u}_R (s_R - \bar{u}_R)}{\bar{\rho}_L (s_L - \bar{u}_L) - \bar{\rho}_R (s_R - \bar{u}_R)},$$

where $\bar{\rho}_K, \bar{p}_K$ and $\bar{u}_K$ denote the mixture values for density, pressure, and velocity on the left ($K = L$) and right ($K = R$) sides of a cell interface $j + \frac{1}{2}$:

$$\begin{cases} \bar{\rho}_K = \alpha_{1_K} \rho_{1_K} + \alpha_{2_K} \rho_{2_K} \\ \bar{p}_K = \alpha_{1_K} p_{1_K} + \alpha_{2_K} p_{2_K} \\ \bar{u}_K = (\alpha_{1_K} \rho_{1_K} u_{1_K} + \alpha_{2_K} \rho_{2_K} u_{2_K}) / \bar{\rho}_K \end{cases} \quad K = L, R. \tag{9}$$

Now, the discrete form for Θ in Eq. (6) and the numerical scheme for the fraction evolution Eq. (4) will be deduced. Firstly, make a denotation $\mathbf{f}^{\mathrm{HLLC}}_{j+1/2} = \left(F^{(1)}_{j+1/2}, F^{(2)}_{j+1/2}, F^{(3)}_{j+1/2}\right)^T$. Then the discrete mass and momentum conservation equations in Eq. (6) are

$$(\alpha\rho)^{n+1}_j = (\alpha\rho)^n_j - \frac{\Delta t}{\Delta x}\left(F^{(1)}_{j+1/2} - F^{(1)}_{j-1/2}\right), \tag{10a}$$

$$(\alpha\rho u)^{n+1}_j = (\alpha\rho u)^n_j - \frac{\Delta t}{\Delta x}\left(F^{(2)}_{j+1/2} - F^{(2)}_{j-1/2}\right) + \Delta t (p_I)_j \Theta. \tag{10b}$$

According to Ref. [6], if the two-phase system is uniform on velocity and pressure at $t = t^n$, i.e. $(u_I)^n_j = u^n_{j\pm1/2} = u^n_j = u$ and $(p_I)^n_j = p^n_{j\pm1/2} = p^n_j = p$, then it must remain uniform on the same variables during time evolution such that $u^{n+1}_j = u^n_j = u$ and $p^{n+1}_j = p^n_j = p$. Substitute these relations into Eqs. (10a) and (10b) and do (10b)$-u\times$(10a), we get the discrete form of the non-conservative terms $\partial\alpha/\partial x$:

$$\Theta = \frac{\phi_{j+1/2} - \phi_{j-1/2}}{\Delta x}, \tag{11}$$

where

$$\phi_{j+1/2} \triangleq \frac{F^{(2)}_{j+1/2} - u F^{(1)}_{j+1/2}}{(p_I)_j} = \begin{cases} \alpha_L, & \text{if } 0 \leqslant s_L, \\ \alpha_L \frac{p_L}{(p_I)_j} + s_L \alpha_L \rho_L \frac{s_L - u_L}{s_L - s_*} \frac{(s_* - u_L)}{(p_I)_j}, & \text{if } s_L \leqslant 0 \leqslant s_*, \\ \alpha_R \frac{p_R}{(p_I)_j} + s_R \alpha_R \rho_R \frac{s_R - u_R}{s_R - s_*} \frac{(s_* - u_R)}{(p_I)_j}, & \text{if } s_* \leqslant 0 \leqslant s_R, \\ \alpha_R, & \text{if } s_R \leqslant 0, \end{cases} \tag{12}$$

in which the adopted HLLC flux has been inserted. Noting that $s_* = u_L = u_R, p_I = p_L = p_R$ as a result of the uniformity assumption, Eq. (12) can be simplified to

$$\phi_{j+1/2} = \begin{cases} \alpha_L, & \text{if } s_* \geqslant 0, \\ \alpha_R, & \text{if } s_* < 0. \end{cases} \tag{13}$$

Now we return to the derivation of the numerical scheme for the fraction evolution Eq. (4). The discrete energy conservation equation is given first

$$(\alpha\rho E)^{n+1}_j = (\alpha\rho E)^n_j - \frac{\Delta t}{\Delta x}\left(F^{(3)}_{j+1/2} - F^{(3)}_{j-1/2}\right) + \Delta t (p_I u_I)_j \Theta. \tag{14}$$

By using the definition $E = e + \frac{1}{2}u^2$ and the discrete mass and momentum conservation Eqs. (10a) and (10b), we can get for the internal energy

$$(\alpha\rho e)^{n+1}_j = (\alpha\rho e)^n_j - \frac{\Delta t}{\Delta x}\left[\left(F^{(3)}_{j+1/2} - u F^{(2)}_{j+1/2} + \frac{1}{2}u^2 F^{(1)}_{j+1/2}\right) - \left(F^{(3)}_{j-1/2} - u F^{(2)}_{j-1/2} + \frac{1}{2}u^2 F^{(1)}_{j-1/2}\right)\right]. \tag{15}$$

Again using the uniformity assumption and doing a bit more algebraic manipulation with the inserted HLLC flux in Eq. (15), it can be simplified as

$$(\alpha\rho e)^{n+1}_j = (\alpha\rho e)^n_j - \frac{\Delta t}{\Delta x} u \left[(\alpha\rho e)^{\mathrm{HLLC}}_{j+1/2} - (\alpha\rho e)^{\mathrm{HLLC}}_{j-1/2}\right], \tag{16}$$

We now make use of the stiffened gas EOS: $\rho e = \frac{p + \gamma B}{\gamma - 1}$. It follows that $(\rho e)^{n+1}_j = (\rho e)^n_j$ under the condition of pressure uniformity. Extracting ρe out of Eq. (16) and noting that $u = u_* = u_I$ as a result of the uniformity assumption and $\alpha^{\mathrm{HLLC}}_{j+1/2} = \phi_{j+1/2}$ according to Eq. (13), a simple upwind scheme for the volume fraction evolution Eq. (4) can be obtained as

$$\alpha^{n+1}_j = \alpha^n_j - \frac{\Delta t}{\Delta x}(u_I)^n_j \left(\phi_{j+1/2} - \phi_{j-1/2}\right), \tag{17}$$

$$\text{with} \quad \begin{cases} \phi_{j+1/2} = \alpha_{j+1/2,L} \\ \phi_{j-1/2} = \alpha_{j-1/2,L} \end{cases} \text{for } (u_I)^n_j \geqslant 0, \quad \begin{cases} \phi_{j+1/2} = \alpha_{j+1/2,R} \\ \phi_{j-1/2} = \alpha_{j-1/2,R} \end{cases} \text{for } (u_I)^n_j < 0.$$

3.1.2. Second-order spatial reconstruction

In complete form, the non-conservative hyperbolic Eqs. 4,5 can be rewritten as

$$\frac{\partial \mathbf{U}}{\partial t} + A(\mathbf{U}) \frac{\partial \mathbf{U}}{\partial x} = 0, \tag{18}$$

where $\mathbf{U} = (\alpha_1, \alpha_1\rho_1, \alpha_1 u_1, \alpha_1 E_1, \alpha_2\rho_2, \alpha_2 u_2, \alpha_2 E_2)^T$, and the matrix

$$A(\mathbf{U}) = \begin{bmatrix} u_I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -\gamma_1 B_1 - p_I & \frac{\gamma_1 - 3}{2}u_1^2 & (3 - \gamma_1)u_1 & \gamma_1 - 1 & 0 & 0 & 0 \\ -\gamma_1 B_1 u_1 - p_I u_I & \frac{\gamma_1 - 1}{2}u_1^3 - u_1 H_1 & H_1 - (\gamma_1 - 1)u_1^2 & \gamma_1 u_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \gamma_2 B_2 + p_I & 0 & 0 & 0 & \frac{\gamma_2 - 3}{2}u_2^2 & (3 - \gamma_2)u_2 & \gamma_2 - 1 \\ \gamma_2 B_2 u_2 + p_I u_I & 0 & 0 & 0 & \frac{\gamma_2 - 1}{2}u_2^3 - u_2 H_2 & H_2 - (\gamma_2 - 1)u_2^2 & \gamma_2 u_2 \end{bmatrix},$$

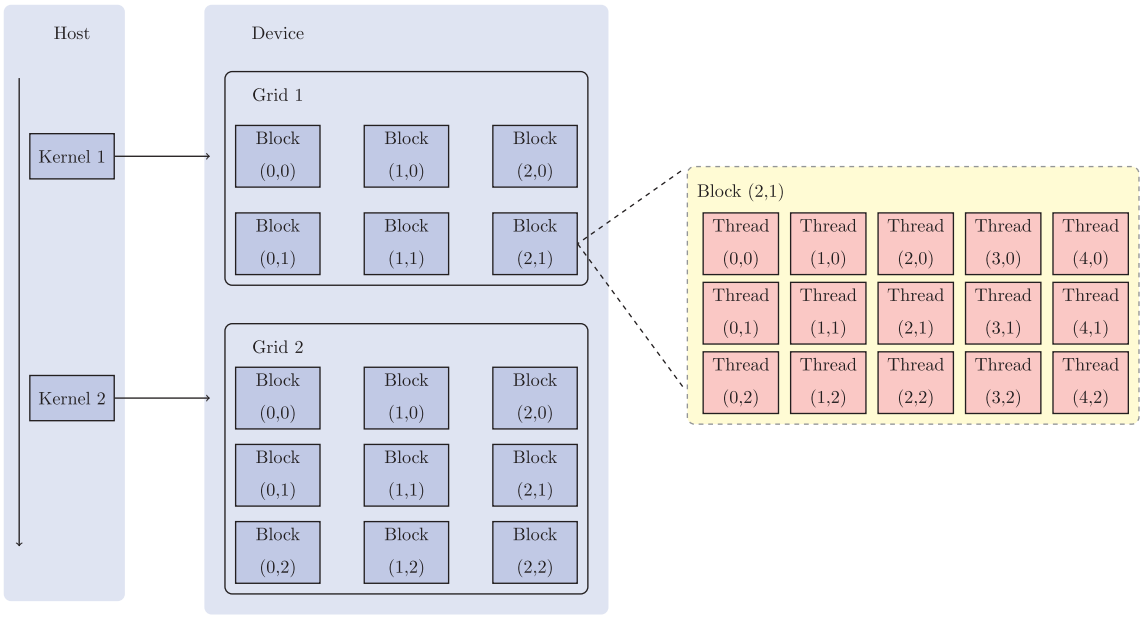


Fig. 1. CUDA programming model.

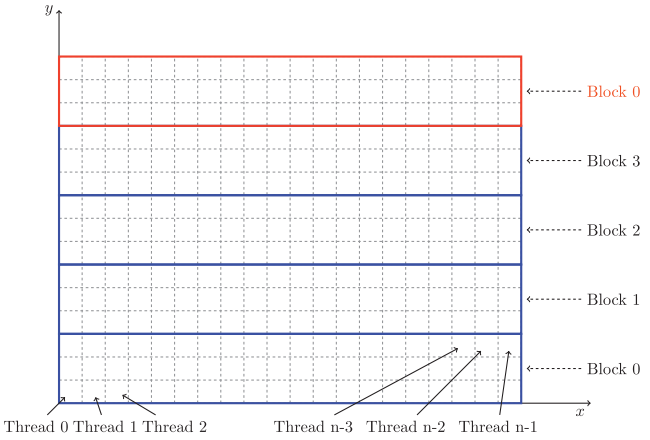


Fig. 2. Correspondence between CUDA threads and computational meshes for a kernel function with gridDim = 4, blockDim = n.

$$H=\frac{E+p}{\rho}=\frac{c^2}{\gamma-1}+\frac{u^2}{2},\quad p=(\gamma-1)\rho e-\gamma B,\quad c^2=\frac{\gamma(p+B)}{\rho}.$$

We adopt TVD-MUSCL reconstruction of Van Leer [23] with the help of Eq. 18

$$\begin{aligned}
\mathbf{U}_{j+1/2,L}&=\mathbf{U}_j+\frac{1}{2}R_j\left(I-\frac{\Delta t}{\Delta x}\Lambda_j\right)\delta\mathbf{W}_j,\\
\mathbf{U}_{j-1/2,R}&=\mathbf{U}_j-\frac{1}{2}R_j\left(I+\frac{\Delta t}{\Delta x}\Lambda_j\right)\delta\mathbf{W}_j,\\
\delta\mathbf{W}_j&=\mathrm{minmod}(L_j\Delta_{j-1/2}\mathbf{U},L_j\Delta_{j+1/2}\mathbf{U}),\\
\Delta_{j+1/2}\mathbf{U}&=\mathbf{U}_{j+1}-\mathbf{U}_j,
\end{aligned}\tag{19}$$

where I is identity matrix, L_j, R_j and Λ_j denote the left and right eigenvector matrixes, and the diagonal matrix of eigenvalues of matrix $A(\mathbf{U}_j)$. They are given in Appendix A.

3.2. Third-order TVD Runge–Kutta scheme

For Eq. (1), Godunov splitting scheme (3) can be regarded as the operator $\mathbf{U}^{n+1}=\mathbf{U}^n+\Delta tL(\mathbf{U}^n)$ in the standard TVD Runge–Kutta scheme. A naive implementation of the third-order TVD Runge–Kutta method to Eq. (1) is like this:

$$\begin{cases}\mathbf{U}^{(1)} &= \mathbf{U}^n+\Delta tL(\mathbf{U}^n)\\ \mathbf{U}^{(2)} &= \frac{3}{4}\mathbf{U}^n+\frac{1}{4}\mathbf{U}^{(1)}+\frac{1}{4}\Delta tL(\mathbf{U}^{(1)})\\ \mathbf{U}^{n+1} &= \frac{1}{3}\mathbf{U}^n+\frac{2}{3}\mathbf{U}^{(2)}+\frac{2}{3}\Delta tL(\mathbf{U}^{(2)})\end{cases}\rightarrow\begin{cases}\mathbf{U}^{(1)} &= L_p^{\Delta t}L_v^{\Delta t}L_H^{\Delta t}\mathbf{U}^n\\ \mathbf{U}^{(2)} &= \frac{3}{4}\mathbf{U}^n+\frac{1}{4}L_p^{\Delta t}L_v^{\Delta t}L_H^{\Delta t}\mathbf{U}^{(1)}\\ \mathbf{U}^{n+1} &= \frac{1}{3}\mathbf{U}^n+\frac{2}{3}L_p^{\Delta t}L_v^{\Delta t}L_H^{\Delta t}\mathbf{U}^{(2)}\end{cases}.\tag{20}$$

However, the problem with Eq. (20) is that it does not guarantee the pressure and velocity are in equilibrium at the end of each sub step. To show this, we denote $\mathbf{U}^{(1*)}=L_p^{\Delta t}L_v^{\Delta t}L_H^{\Delta t}\mathbf{U}^{(1)}$ and substitute it into the mass and momentum equations for $\mathbf{U}^{(2)}$, then we get

$$\begin{aligned}
u_1^{(2)}&=\frac{(3\alpha_1^n\rho_1^n)u_1^n+\left(\alpha_1^{(1*)}\rho_1^{(1*)}\right)u_1^{(1*)}}{3\alpha_1^n\rho_1^n+\alpha_1^{(1*)}\rho_1^{(1*)}},\\
u_2^{(2)}&=\frac{(3\alpha_2^n\rho_2^n)u_2^n+\left(\alpha_2^{(1*)}\rho_2^{(1*)}\right)u_2^{(1*)}}{3\alpha_2^n\rho_2^n+\alpha_2^{(1*)}\rho_2^{(1*)}}.
\end{aligned}$$

It is easy to see that $u_1^{(2)}=u_2^{(2)}$ is not always true because the coefficients may be different even though $u_1^n=u_2^n$ and $u_1^{(1*)}=u_2^{(1*)}$ are satisfied. Similarly, $p_1^{(2)}=p_2^{(2)}$ is not always true either. That is to say, pressure and velocity equilibrium between phases are not always fulfilled for $\mathbf{U}^{(2)}$ and $\mathbf{U}^{n+1}$. This will often lead to computation breakdown according to our practise. To eliminate this problem, we reorder the sequence of operators

$$\mathbf{U}^{(1)}=L_H^{\Delta t}L_p^{\Delta t}L_v^{\Delta t}\mathbf{U}^n,\tag{21}$$

$$\mathbf{U}^{(2)}=\frac{3}{4}\mathbf{U}^n+\frac{1}{4}L_H^{\Delta t}L_p^{\Delta t}L_v^{\Delta t}\mathbf{U}^{(1)},\tag{22}$$

$$\mathbf{U}^{(3)}=\frac{1}{3}\mathbf{U}^n+\frac{2}{3}L_H^{\Delta t}L_p^{\Delta t}L_v^{\Delta t}\mathbf{U}^{(2)},\tag{23}$$

$$\mathbf{U}^{n+1}=L_p^{\Delta t}L_v^{\Delta t}\mathbf{U}^{(3)}.\tag{24}$$

Eqs. (21)–(23) are the three sub-steps of the Runge–Kutta method, and Eq. (24) is an additional step. In each sub-step, we first do velocity and pressure relaxation to ensure equilibrium, then solve the hyperbolic equations. After the third step, we do additional relaxation. Actually, solution $\mathbf{U}^{n+1}$ of Eq. (24) will not be modified by relaxation operators in Eq. (21) in the next time step, so only hyperbolic operator in Eq. (21) is executed (the only exception is at $t = 0$ when initial data are not in equilibrium). As a result, the total computational count is not increased compared with the naive implementation Eqs. (20).

4. Implementation on GPUs

In this section, we first provide a very brief description of the elements of NVIDIA's Compute Unified Device Architecture (CUDA) programming model, then describe techniques used for implementation of the present numerical method for 2D problems on a single GPU, and hybrid MPI/Pthread-CUDA parallelization paradigm for application on multiple GPUs.

4.1. CUDA

Because of their intrinsic highly-parallel microprocessors and a dedicated high-bandwidth memory device, GPUs are ideally suitable for computationally intensive data-parallel applications. A detailed description of GPU hardware and CUDA programming model can be found on the web (e.g., [14]). Fig. 1 illustrates the main idea of CUDA. A CUDA program generally includes two parts, serial and parallel codes. The serial codes that run on the host (CPU) side are responsible for variables declaration, initialization, data transmission, and kernel invocation. The parallel codes (called "kernel function") running on the device (GPU) side are executed in parallel by massive threads (the minimal processing unit) organized to match the GPU hardware feature and to allow for mapping typical data structures (arrays, matrices). A number of threads makes up a block, and many blocks make up a grid, which is the counterpart of a kernel function. The block may be organized into 1D, 2D or 3D arrays, while the grid may be 1D or 2D arrays. Limited by the hardware and for ease of mapping typical data structures to threads, the number of threads in one block and the number of blocks in one grid should be properly selected in the program.

4.2. Optimization for single GPU

The present numerical method (21)–(24) are explicit ones, and in our application, 2D structured Cartesian meshes are used. Therefore, the solution algorithm is well suited to GPU computing. On a single GPU, repeated operations of reconstruction, hyperbolic solver, and pressure–velocity relaxation have to be done for the whole computational mesh cells. These three operations are encapsulated into three kernel functions. In a kernel function, the whole computational mesh cells are divided into many pieces such that each piece fits into the maximum thread number per thread block allowed by a GPU, and the computation of each piece is assigned to a CUDA block. Each mesh cell is mapped to one thread, and one thread may be in charge of several mesh cells belonging to different parts when the total piece number exceeds the maximum block number allowed by a CUDA grid. This is illustrated in Fig. 2. A pseudo code demonstrating how this assignment works is given in Algorithm 1. The optimization techniques used inside this slice of code are introduced below.

Algorithm 1. Pseudo code to demonstrate how an operation L is implemented for the whole computational grid points with $N_x \times N_y$

```
1:  // operation L can be one of L_H, L_p and L_v
2:  __global__ void kernel_OpLForWholeArea (var)
3:  {
4:    //get the mesh cell index in 1D array
5:    int tid_in_grid = blockDim * blockId + threadId;
6:
7:    //deal with one part of cells sized blockDim×gridDim
8:    //at a time within a CUDA grid and then move to the
      next part
9:    for (int i = tid_in_grid; i < Nx * Ny;)
10:   {
11:   //operation L in single mesh cell
12:   OpLForSingleCell (var, i);
13:   //move to the corresponding cell in the next part
14:   i += blockDim * gridDim;
15:   }
16: }
17:
18: __device__ void OpLForSingleCell (var, i)
19: {
20: //load primitive variables from global memory
21: density = var [den_shift + i];
22: velocity = var [vel_shift + i];
23: pressure = var [pre_shift + i];
24: ...
25: }
```

First of all, to obtain a good global memory throughput, the primitive variables are loaded and stored in data organization scheme of Structure of Array (SoA), which is different from that of Array of Structure (AoS) generally adopted in CPU implementation. The latter one has the feature of object oriented and good readability, but code of SoA is easier to be parallelized and enables coalescence and alignment of global memory access of GPU. This is a crucial issue to many applications since global memory access optimization is of dominant importance in achieving satisfactory performance in GPU. For a computational grid of $N_x \times N_y$, primitive variables (say, $\alpha_g, \rho_g, u_g, p_g, \rho_l, u_l, p_l, u_I, p_I$, where index g and l stand for different phases and I for interface) are stored in a global array *var*

$$
\begin{aligned}
var = {} & \underbrace{\alpha_{g1}, \alpha_{g2}, \ldots, \alpha_{gN_x\times N_y}}_{N_x\times N_y}, \underbrace{\rho_{g1}, \rho_{g2}, \ldots, \rho_{gN_x\times N_y}}_{N_x\times N_y}, \\
& \underbrace{u_{g1}, u_{g2}, \ldots, u_{gN_x\times N_y}}_{N_x\times N_y}, \underbrace{p_{g1}, p_{g2}, \ldots, p_{gN_x\times N_y}}_{N_x\times N_y}, \\
& \underbrace{\rho_{l1}, \rho_{l2}, \ldots, \rho_{N_x\times N_y}}_{N_x\times N_y}, \underbrace{u_{l1}, u_{l2}, \ldots, u_{N_x\times N_y}}_{N_x\times N_y}, \\
& \underbrace{p_{l1}, p_{l2}, \ldots, p_{N_x\times N_y}}_{N_x\times N_y}, \underbrace{u_{I1}, u_{I2}, \ldots, u_{IN_x\times N_y}}_{N_x\times N_y}, \\
& \underbrace{p_{I1}, p_{I2}, \ldots, p_{IN_x\times N_y}}_{N_x\times N_y}.
\end{aligned} \tag{25}
$$

The second important optimization we made is about the all reduction operation. We design a procedure for the all reduce operation (like *min*, *max*) to be completed within a kernel function, thus CPU operation is not needed. It resolves the problem of synchronization between different blocks in a CUDA grid, making it possible

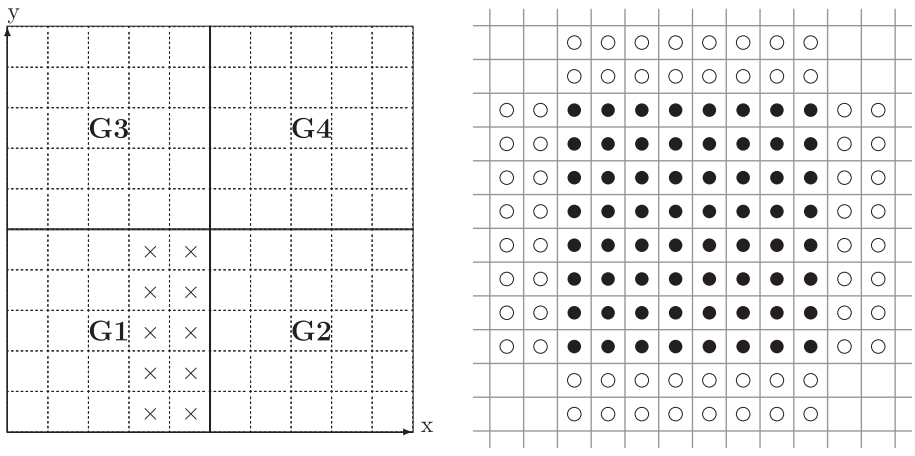
Table 1
Memory usage.

GPU memory	Variable description
Constant memory	CFL, thermodynamic constants, machine zero, maximum iteration number
Register	Intermediate variables, mostly scalars
Shared memory	Intermediate variables, mostly array
Global memory	Primitive variables: α, ρ, u, p

to encapsulate as many as possible calculations within one kernel function, which accelerates the code. The pseudo code in Algorithm 2 describes how to find the maximum value in a list of numbers. Firstly, array A to be compared is divided into many parts each of which corresponds to a thread block, and the maximum of each part is computed and saved at C[0] within a block using shared memory C, which is traditional (lines 6–11). Secondly, C[0] of each block is loaded to global memory B in the order of block ID (lines 13–16), in which atom operator is used to record the number of blocks that have finished the data load (count). Thirdly, all other blocks wait idly while block 0 is computing the maximum of array B (lines 18–24). There are two details deserving attention: 1, we use the criterion "count%gridDim==0" to decide whether the *gridDim* elements have been completely loaded into B (line 20) before block 0 starts the comparison; 2, a global flag refreshed by thread 0 of block 0 (line 23) is utilized to synchronize thread blocks (line 24), and lines 36–43 describe how the synchronization works.

Algorithm 2. Pseudo code of finding the maximum in a list of numbers

```
1:  //B[0] = max_{i=0}^{m} A[i]. Suppose m < blockDim×gridDim, otherwise treat like Algorithm 1
2:  __device__ int flag = 0;
3:  __device__ int count = 0;
4:  __global__ void kernel_AllReduce (double* A, double* B, int m)
5:  {
6:    //load one part of A into shared memory C
7:    __shared__ double C[blockDim];
8:    C[threadId] = A[blockDim * blockId + threadId];
9:    __syncthreads ();
10:   //get maximum value of C
11:   GetMax (C, blockDim, threadId);
12:
13:   // load C[0] =max_{i=0}^{blockDim-1} C[i] to global memory B[blockId]
14:   if (threadId == 0) {
15:     B[blockId] = C[0];
16:     count++;//must use atomicAdd to avoid conflict among multiple writes}
17:
18:   if (blockId == 0) {
19:     //wait until all blocks have finished loading C[0] into B[blockId]
20:     SyncBlocks (threadId, count%gridDim + 1);
21:     //block 0 calculate maximum value of B and store it at B[0]
22:     GetMax (B, gridDim, threadId);
23:     if (threadId == 0) flag = 1;// a signal to free other block's wait}
24:   else {SyncBlocks (threadId, flag);}// other blocks wait idly
25: }
26:
27: //make the comparison within a CUDA block
28: //S[0]= max_{i=0}^{n} S[i]. Suppose n<blockDim
29: void GetMax (double* S, int n, int threadId)
30: {
31:   for (int j = n/2; j>0; j/=2) {
32:     if (threadId < j) S[threadId] = max (S[threadId], S[threadId + j]);
33:     __syncthreads ();}
34: }
35:
36: void SyncBlocks (int threadId, int flag)
37: {
38:   //thread 0 keeps on looping until flag is changed to 1
39:   if (threadId == 0) {
40:     while (1){if (flag == 1) break;}}
41:   //all other threads in current block wait until thread 0 finishes above looping
42:   __syncthreads ();
43: }
```



(a) Domain decomposition

(b) Auxiliary grids

Fig. 3. Grid and mission assignment to four GPUs. (a) Decomposition of the whole domain into non-overlapped sub-domains. A sub-domain is computed by a corresponding GPU. Symbol ×denotes inner cells to be auxiliary cells for neighboring sub-domains; (b) Controlled domain for a GPU. Solid points denote initial cells allocated to the current GPU while hollow points mark auxiliary cells.

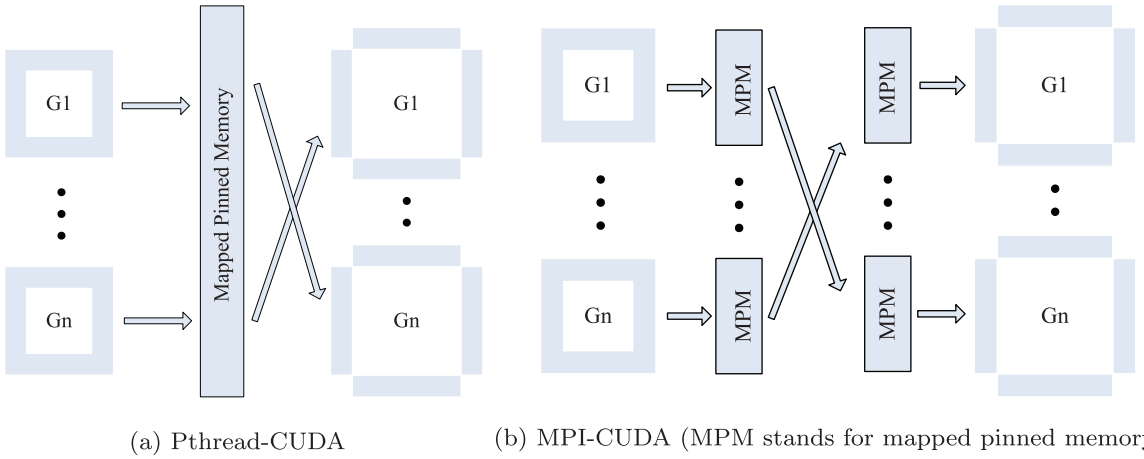


Fig. 4. Data exchange between devices. G stands for grid, and the blue area stands for boundary data that passed between devices. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

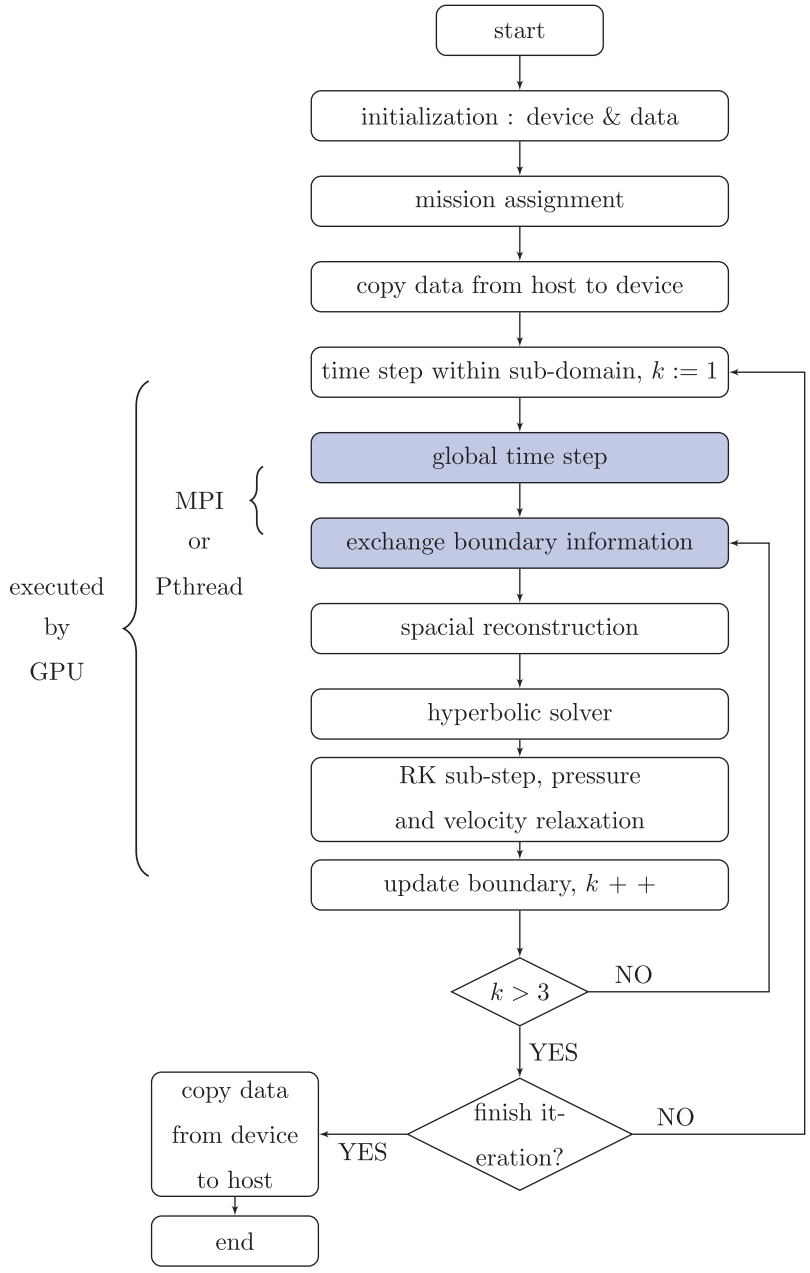


Fig. 5. Computational flow chart for multi-GPU implementation.

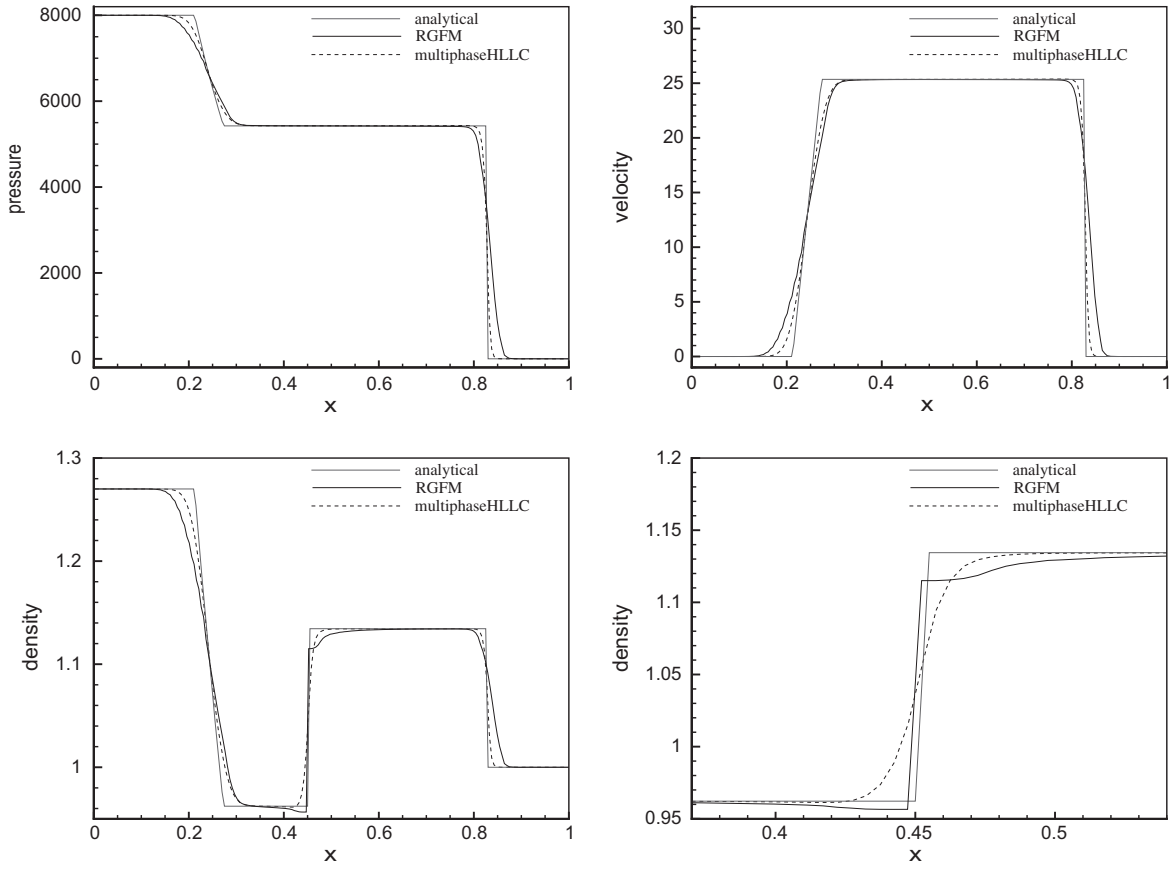


Fig. 6. Flow distributions in the gas–liquid shock tube problem.

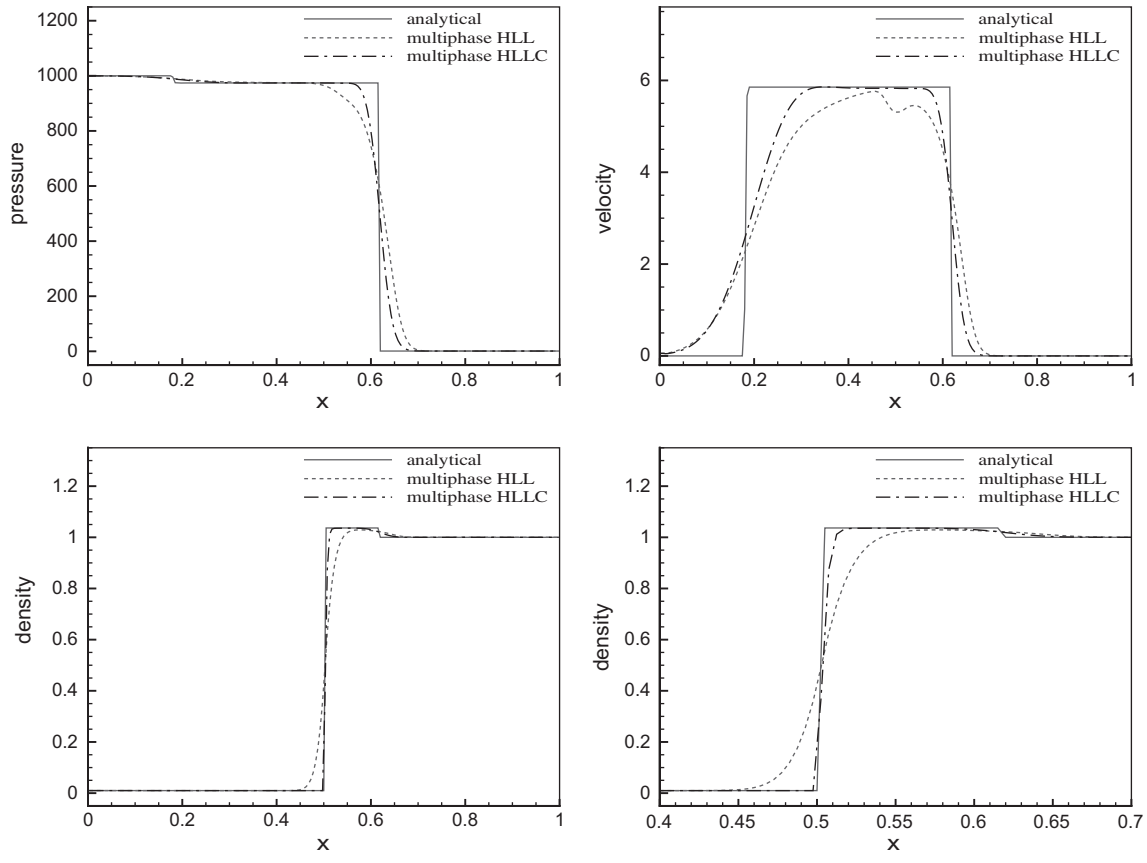


Fig. 7. Flow distributions in the problem of underwater explosion.

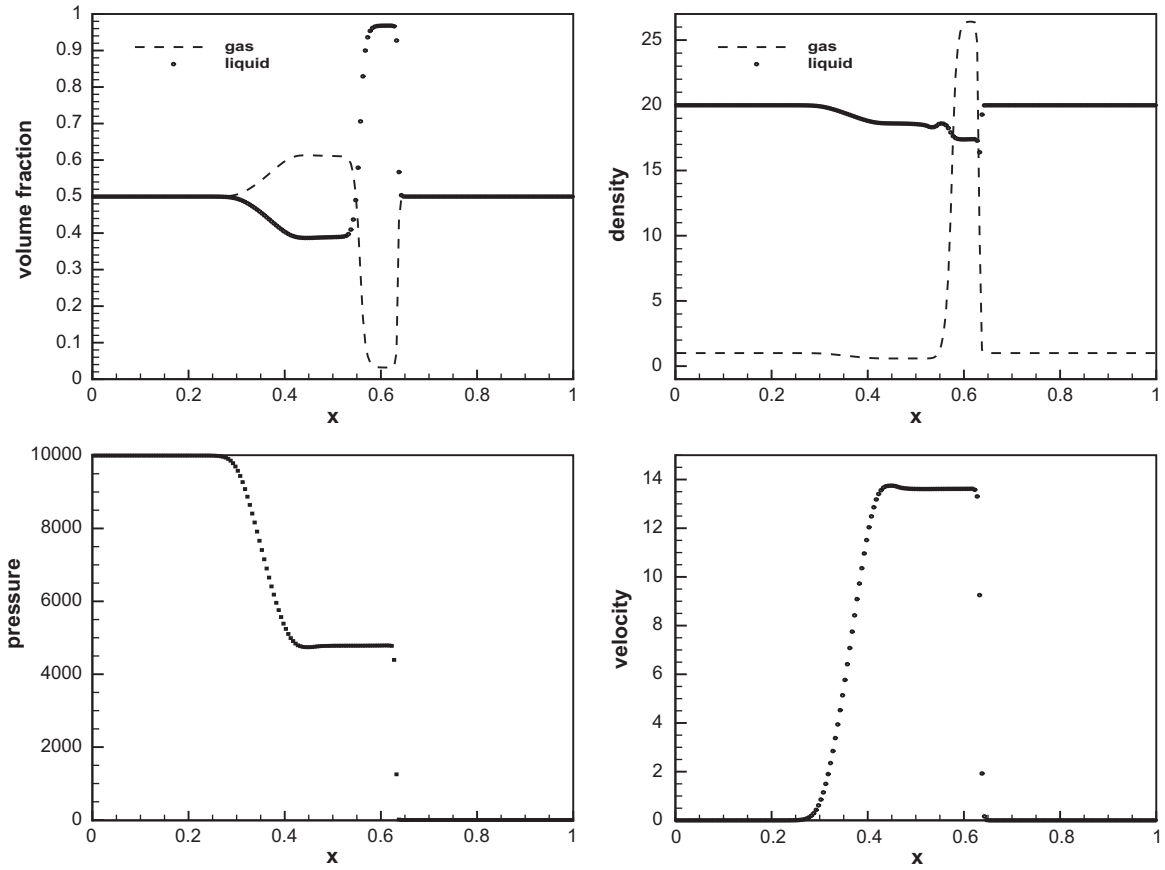


Fig. 8. Flow distributions in the problem of shock tube with mixtures.

Finally, the usage of different types of GPU memory in Table 1 is discussed. The access of global memory is slow, but the on-chip register and shared memories are fast. So only essential variables passing through different kernels to keep the computation going on are stored in global memory. In this work, these variables are only loaded and refreshed once respectively in a kernel, so shared memory is used to store intermediate variables instead of those transfered from global memory to save the data copy time. On the other hand, considering that registers are rare resources in GPU memory, if all intermediate variables are put into registers, it is expected that some of the variables will be implicitly allocated in local memory which is off-chip and not cached. This will dramatically decrease the speed of this kernel since the access to local memory is as slow as that of global memory. So it is necessary to release the pressure on register by allocating shared memory for intermediate variables. Furthermore, the shared memory can also be used to load those non-contiguous accessed data used by numerical schemes from the global memory.

4.3. Implementation on multi-GPUs

Although a single GPU has achieved good performance in many applications, the compute capacity required for simulating large-scale multi-phase flow problems is far beyond what a single GPU can deliver. Consequently, it is worthwhile to develop multi-GPU parallelization techniques.

In our work, domain decomposition method is used. A 2D grid of $N_x \times N_y$ is divided into m and n parts along its two coordinate directions respectively, which makes the total sub-domain number $m \times n$ (Fig. 3(a)). The computation of each sub-domain is assigned to different GPUs. There are two principles to follow: firstly, $m \times n$ equals the total number of GPUs in use; secondly, the shape of each sub-domain should be as square-like as possible to reduce the amount of information exchanged between GPUs. Indeed, flow variables of a sub-domain always reside in the global memory of corresponding GPU from the very beginning to the end of a simulation, and they will be exported to different files when the simulation ends. A five point difference stencil, including the target cell and the four cells that adjacent to it in 2D, is needed to support the MUSCL reconstruction; and the reconstructed values of two adjoining cells are used to evaluate the numerical flux across their common cell face. Therefore, as sketched in Fig. 3(b), two layers of auxiliary cells are defined at every boundary of a sub-domain to couple computation. Take sub-domains G1 and G2 in Fig. 3(a) for example, the left two layers of auxiliary cells (marked by symbol $\times$) for G2 are updated according to the two layers of inner cells near the right boundary of G1.

To facilitate communications, data of the two layers of inner cells near the submodmain boundary are passed between devices by means of MPI or Pthread, which is operated in CPU. Considering that most computation is performed in GPU, to avoid frequent data copy between CPU and GPU, mapped pinned memory is used to store auxiliary boundary data, which has two addresses: one in the host and the other in the device. When boundary computation of a sub-domain is finished, each device uploads the boundary data that others need to mapped pinned memory, and then downloads the auxiliary boundary data for next time step computation. As seen in Fig. 4, there is a little difference in MPI-CUDA and Pthread-CUDA application: in Pthread-CUDA, all CPU threads share a common mapped pinned array as in Fig. 4(a), and no data transfer is performed; while in MPI-CUDA, data is stored in different

Table 2
Speedup (GPU vs. single CPU core).

Grid number	CPU time	GPU time	Speedup
128×32	0.097503	0.002830	34.45
256×64	0.358785	0.011469	31.28
512×128	1.434512	0.042854	33.47
1024×256	4.634705	0.167221	27.71

Table 3
Multi-GPU speedup vs. one GPU for 1024×256 mesh size.

GPU number	Speedup		Efficiency (%)	
	MPI-CUDA	Pthread-CUDA	MPI-CUDA	Pthread-CUDA
1	1	1	100	100
2	1.801	1.836	90.06	91.83
4	3.212	3.266	80.31	81.64
8	5.460	4.935	68.25	61.69

arrays belonging to different MPI processes which is then sent and received between MPI processes as in Fig. 4(b).

The main flowchart is described in Fig. 5. Notice that almost all the field computations are performed on the GPUs, leaving the CPU almost idle during the computation except performing iteration check, data copy and communication.

5. Numerical results

In this section, numerical tests with several 1D and 2D compressible gas–liquid two-fluid problems are provided. For 1D cases, we compared our method with available RGFM [20] as well as HLL method [2] using the same MUSCL reconstruction. RGFM can capture material interface with little diffusion, but it is not a conservative method. HLL is robust, but too diffusive. We use examples with high density and high pressure ratios to compare these numerical methods. Besides, a true two-phase flow problem is simulated to show the capability of our method in dealing with multiphse mixtures. 2D problems are computed using GPUs, and speedups of single- and multi-GPU are presented.

5.1. 1D gas–liquid shock tube with high pressure ratio [24]

We consider a shock tube filled on the left side with high pressure gas and on the right side with liquid. For $x \in [0,1]$, the initial data are

$$(\rho,u,p,\gamma,B) = \begin{cases} (1.27,\ 0,\ 8000,\ 1.4,\ 0), & x \leqslant 0.4, \\ (1.0,\ 0,\ 1.0,\ 7.15,\ 3309), & x > 0.4. \end{cases}$$

In this problem, the pressure ratio at the gas–liquid interface is up to $8000:1$, while density ratio is $1.27:1$. There are 200 uniform cells in [0,1], and the instant time step size is decided by $\Delta t = \mathrm{CFL} \cdot (\Delta x / \max|\lambda|)$, where $\mathrm{CFL} = 0.05$ and λ stands for eigenvalue of matrix A in Eq. (18). Numerical simulation was conducted up to $t = 0.002$. Pressure, velocity, and density distributions near the material interface are shown in Fig. 6. The solid black line is results obtained by RGFM method. It can be seen that there is a density decrease at the interface which might be caused by the non-conservative error of RGFM. Results of present multiphase method (the dashed line) look closer to the analytical solution (the solid grey line).

5.2. 1D underwater explosion [25]

Consider a one-dimensional domain of [0,1], the two flow states are separated at $x = 0.5$ initially:

$$(\rho,u,p,\gamma,B) = \begin{cases} (0.01,\ 0,\ 1000,\ 2,\ 0), & x \leqslant 0.5, \\ (1.0,\ 0,\ 1.0,\ 7.15,\ 3309), & x > 0.5. \end{cases}$$

Rarefaction wave with very high speed will appear on the left side. 200 uniform cells are used in our simulation, and the time step size is computed the same as that in Section 5.1 with

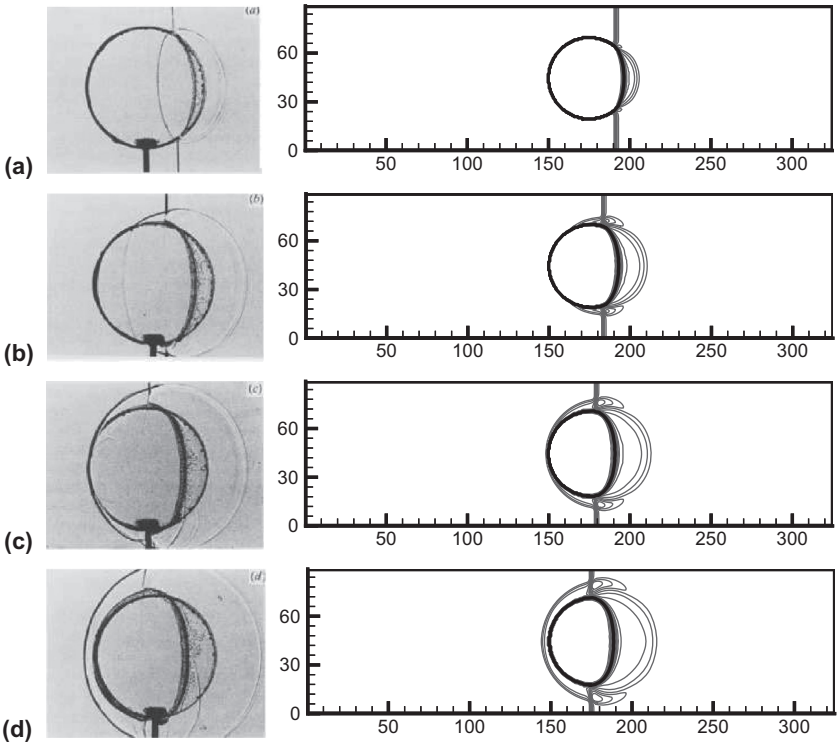


Fig. 9. Comparison of experiment (left) and computed density contours (right) in shock-bubble interaction problem. (a) $t = 32$ μs, (b) $t = 52$ μs, (c) $t = 62$ μs, (d) $t = 72$ μs.

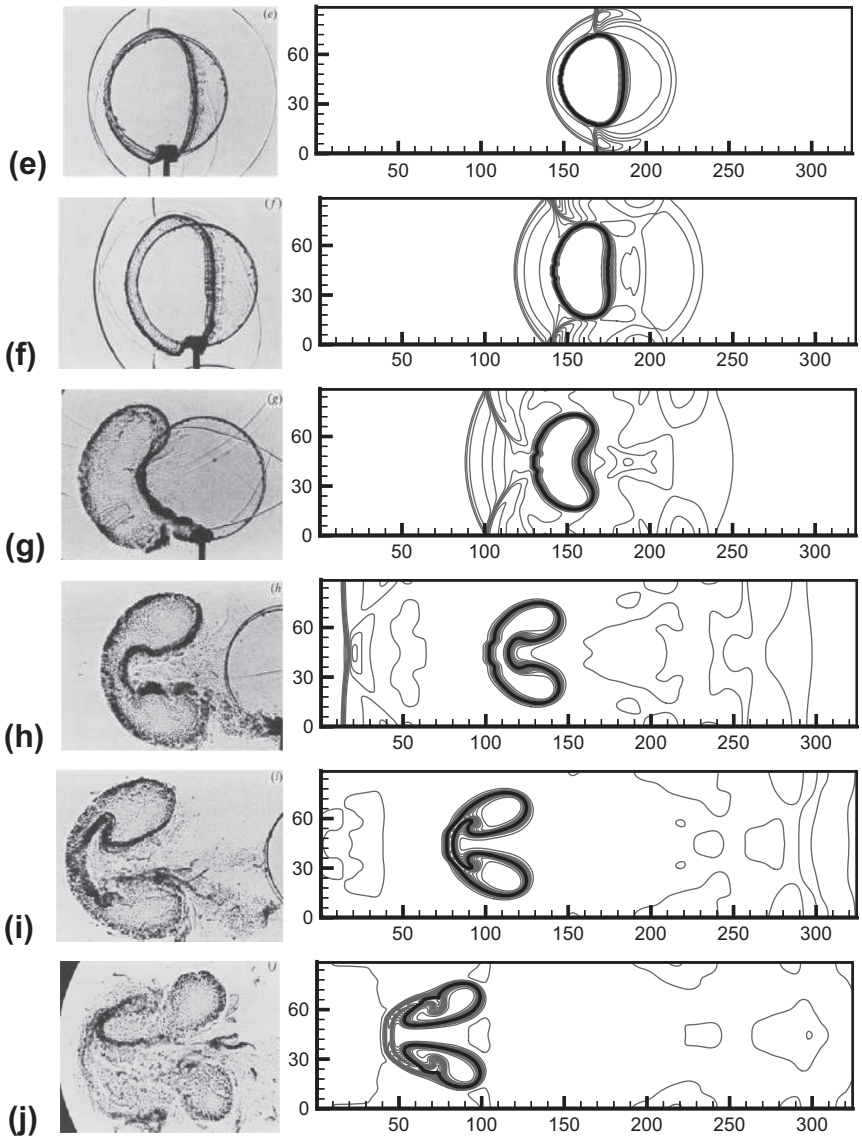


Fig. 10. Comparison of experiment (left) and computed density contours (right) in shock-bubble interaction problem. (e) $t = 82$ μs, (f) $t = 102$ μs, (g) $t = 245$ μs, (h) $t = 427$ μs, (i) $t = 674$ μs, (j) $t = 983$ μs.

CFL = 0.1. At a very short time $t = 0.000718$, corresponding pressure, velocity and density distributions are shown in Fig. 7. Results of HLL scheme [2] and present HLLC method are compared. HLL scheme uses only two waves instead of three as in HLLC, and contact discontinuity is severely smeared out. This might also explain why the velocity distribution is abnormal near the interface. Results computed by HLLC scheme are more accurate and have less numerical diffusion.

5.3. 1D shock tube with fluid mixtures [2]

This gas–liquid shock tube problem is a little different from that in Section 5.1. In this case, uniform volume fraction initial condition ($\alpha = 0.5$) is used, which makes it a true two-phase mixture problem. The gas and liquid densities are 1 and 20, respectively, in the entire domain of [0,1]. A diaphram located at $x = 0.5$ separates the mixtures. The initial pressure in the left chamber is 10^4 and 1 in the right chamber. 200 uniform cells are used and CFL = 0.05. The results at $t = 0.004$ are shown in Fig. 8. The volume fraction varies across the rarefaction and shock waves, which is in agreement with results of HLL scheme [2].

5.4. 2D shock–bubble interaction

This is a classical problem [26–28]. A 2D bubble filled with helium initially in equilibrium is surrounded by air. Initially the bubble is located at $x = 175$ and $y = 44.5$; its radius is equal to 25. A shock initially at $x = 225$ from the right of the bubble is characterized by Mach number $M_s = 1.22$. The data are

$$(\rho,u,v,p,\gamma) = \begin{cases} (1.3764,\ -0.394,\ 0,\ 1.5698,\ 1.4), & \text{postshock},\\ (1,\ 0,\ 0,\ 1,\ 1.4), & \text{preshock},\\ (0.138,\ 0,\ 0,\ 1.0,\ 1.67), & \text{inside bubble}. \end{cases}$$

The computational domain is $[0,325] \times [0,89]$. Characteristic boundary conditions are used for the left and right boundaries, while reflective conditions are used for the upper and bottom boundaries. The simulation is done with GPUs and several mesh resolutions ranging from 128×32 to 1024×256 are used. Using a single GPU, we observe 27× to 34× speedup relative to a single-core CPU computation as seen from Table 2. On the 1024×256 resolution, linear speedup can be achieved by multi-GPU parallel computing although there might be a little decrease

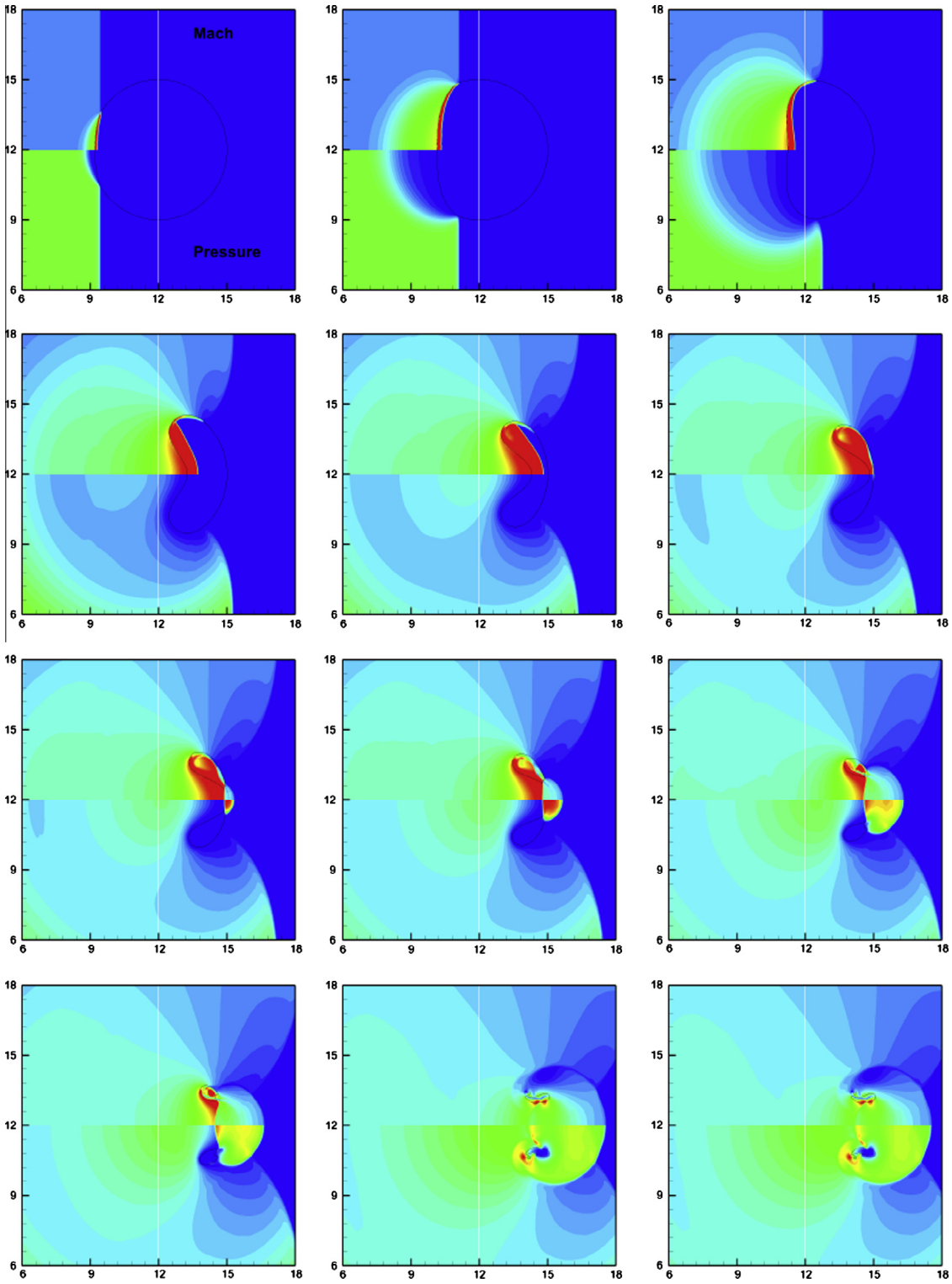


Fig. 11. Dynamics of shock-induced bubble collapse in water: 1.0 μs, 1.6 μs, 2.2 μs, 3.1 μs, 3.5 μs, 3.7 μs, 3.8 μs, 3.9 μs, 4.1 μs. For each instant, the upper half is Mach number contours and the lower half is pressure contours. The solid line is the bubble delineated by volume fraction $\alpha = 0.5$.

in per-GPU performance when more GPUs are used, as shown in Table 3. It also shows that the parallel efficiency of MPI-CUDA is slightly lower than that of Pthread-CUDA when using fewer than 4 GPUs, but is evidently higher when using 8 GPUs.

To verify the computed results on multiple GPUs, Figs. 9 and 10 show the time history of computed density contours using 512×128 meshes in comparison with experimental Schlieren images. The time instants are counted after the shock first touches the bubble, and are identical for both simulation and experiment: 32 μs, 52 μs, 62 μs, 72 μs, 82 μs, 102 μs, 245 μs, 427 μs, 674 μs, 983 μs. The bubble outline is displayed in bold line by volume fraction contour $\alpha_1 = 0.5$. It is seen that the numerical results are comparable to the bubble-shock interaction experiment of [26].

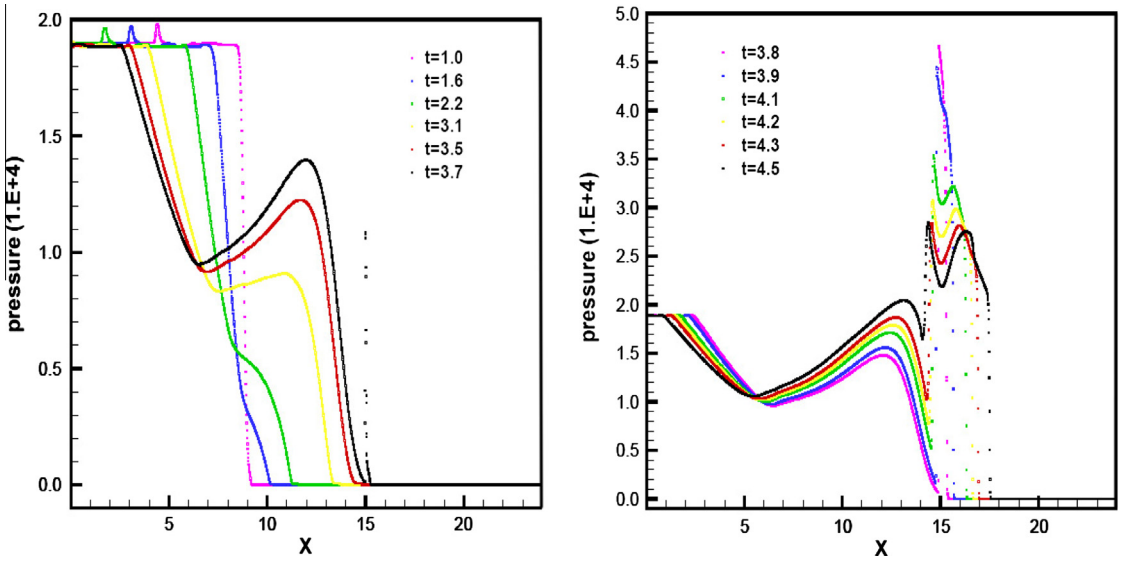


Fig. 12. Pressure distribution at $y = 12$.

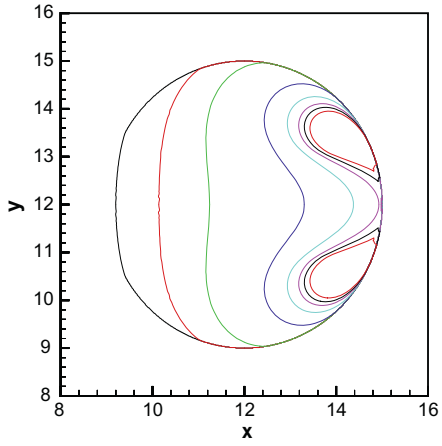


Fig. 13. Evolution of bubble shapes ($\alpha = 0.5$) from $t = 0.01$ on, $\Delta t = 0.006$.

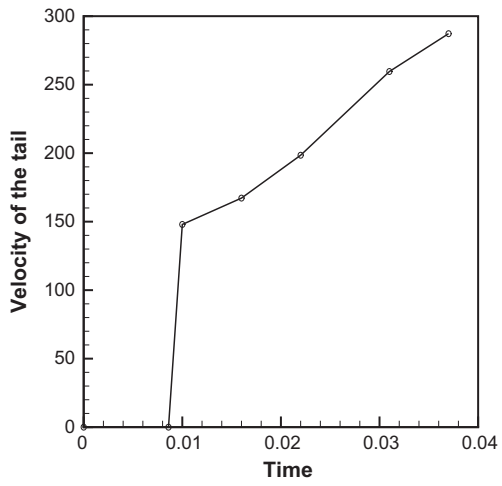


Fig. 14. Velocity of the tail of the bubble.

5.5. 2D shock-induced bubble collapse in water

We now consider an area filled with air in the middle, and water in the rest region. The computational domain is $[0, 24] \times [0, 24]$, and the diameter of air bubble is $d = 6$. A strong shock, which is characterized by Mach number $M_s = 1.72$, locates at $x = 6.6$ initially and moves rightward. The experiment results were given by [29], and many numerical results could be found [30–35]. We choose the same initial data as [34], and the non-dimensionalized flow distribution is as follows.

$$(\rho,u,v,p,\gamma,B) = \begin{cases} (1.32365,\ 68.158,\ 0,\ 19000,\ 4.4,\ 6000), & \text{postshock},\\ (1,\ 0,\ 0,\ 1,\ 4.4,\ 6000), & \text{preshock},\\ (0.001,\ 0,\ 0,\ 1,\ 1.4,\ 0), & \text{inside bubble}. \end{cases}$$

Because of the great difference in flow quantity, when the incident shock reaches the material interface, strong reflected rarefaction wave is generated at the air–liquid interface moving towards the contrary direction. Inside the bubble, the pressure ratio on two sides of the transmitted shock is 33.28, about 1/570 of that of the incident shock, whose ratio is 19000. In order to capture these complex wave structures, CFL = 0.1 and 1024×1024 grids are used in present computation on 8 GPUs. Variations of Mach number and pressure with time are shown in Fig. 11, and the evolution of pressure at location $y = 12$ is shown in Fig. 12. There is a small overshoot at $x = 5$ due to the initial discontinuity decomposition by Godunov type methods which is beyond the scope of this paper. Around time $t = 3.7$ μs, the air bubble is highly compressed at $x = 15$, which explains the sudden pressure increasing phenomenon. As the bubble breaks up the pressure at this point begins to decrease. Fig. 13 shows snapshots of the bubble shape evolution. It is seen that as time evolves, the air bubble becomes involuted with a distinct water jet formed at the centerline. Fig. 14 describes development of the velocity at the bubble tail with time. The velocity is 287.219 (2.87219 km/s in dimensional unit) when the water jet hits the other side of the bubble cutting the bubble into two half. The computed hit velocity is close to the published results: 2.82 km/s [35] and 2.85 km/s [33,34].

6. Conclusions

A simple HLLC-type Riemann solver-based numerical method has been presented for solving the compressible seven-equation

two-phase model. The conventional HLLC flux is utilized for the conservative fluxes, and the corresponding discrete schemes for the non-conservative terms and the volume fraction evolution equation are constructed, which leads to a simple upwind scheme. Moreover, the third-order TVD Runge–Kutta method is implemented in conjunction with the operator splitting to obtain a robust procedure by virtue of reordering the sequence of operators. The resulting numerical method is implemented by using multi-GPU parallel computing techniques, and optimization strategies are undertaken to speedup the code significantly. Numerical tests with several 1D and 2D compressible gas–liquid two-fluid flow problems with high density and high pressure ratios demonstrate that the present numerical method is more accurate and robust than HLL and RGFM methods, and the resulting multi-GPU code has effectively reduced the solution time required for the expensive seven-equation compressible two-phase model.

Acknowledgments

This work is supported by Natural Science Foundation of China (11261160486, 11321061), State Key Program for Developing Basic Sciences (2010CB731505) and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing.

Appendix A. Eigenvalues and eigenvectors of the matrix $A(U)$ in Eq. (18)

It is complicated to decompose matrix A directly. Notice that $u_l = u_1 = u_2$ and $p_l = p_1 = p_2$ in the reconstruction stage, the eigen-decomposition of matrix A can be done in the following form temporarily

$$A = R\Lambda L,$$

where Λ is the diagonal matrix of eigenvalues for the matrix A,

$$\Lambda \equiv \mathrm{diag}(u_l, u_1 - c_1, u_1, u_1 + c_1, u_2 - c_2, u_2, u_2 + c_2).$$

L is the matrix composed of left eigenvectors

$$L = (l_1, l_2, l_3, l_4, l_5, l_6, l_7)^T,$$

where

$$l_1 = (1,0,0,0,0,0,0)$$

$$l_2 = \frac{\gamma_1 - 1}{2c_1^2}\left(-\frac{\gamma_1\pi_1 + p_1}{\gamma_1 - 1}, u_1^2 - H_1 + c_1\frac{u_1 + c_1}{\gamma_1 - 1}, -u_1 - \frac{c_1}{\gamma_1 - 1}, 1, 0, 0, 0\right)$$

$$l_3 = \frac{\gamma_1 - 1}{2c_1^2}(0, 2H_1 - 2u_1^2, 2u_1, -2, 0, 0, 0)$$

$$l_4 = \frac{\gamma_1 - 1}{2c_1^2}\left(-\frac{\gamma_1\pi_1 + p_1}{\gamma_1 - 1}, u_1^2 - H_1 - c_1\frac{u_1 - c_1}{\gamma_1 - 1}, -u_1 + \frac{c_1}{\gamma_1 - 1}, 1, 0, 0, 0\right)$$

$$l_5 = \frac{\gamma_2 - 1}{2c_2^2}\left(\frac{\gamma_2\pi_2 + p_2}{\gamma_2 - 1}, 0, 0, 0, u_2^2 - H_2 + c_2\frac{u_2 + c_2}{\gamma_2 - 1}, -u_2 - \frac{c_2}{\gamma_2 - 1}, 1\right)$$

$$l_6 = \frac{\gamma_2 - 1}{2c_2^2}(0, 0, 0, 0, 2H_2 - 2u_2^2, 2u_2, -2)$$

$$l_7 = \frac{\gamma_2 - 1}{2c_2^2}\left(\frac{\gamma_2\pi_2 + p_2}{\gamma_2 - 1}, 0, 0, 0, u_2^2 - H_2 - c_2\frac{u_2 - c_2}{\gamma_2 - 1}, -u_2 + \frac{c_2}{\gamma_2 - 1}, 1\right).$$

R is composed of right eigenvectors

$$R = (r_1, r_2, r_3, r_4, r_5, r_6, r_7),$$

where

$$r_1 = \left(1, \frac{\gamma_1\pi_1 + p_1}{c_1^2}, \frac{\gamma_1\pi_1 + p_1}{c_1^2}u_1, \frac{\gamma_1\pi_1 + p_1}{c_1^2}H_1, \frac{\gamma_2\pi_2 + p_2}{c_2^2}, -\frac{\gamma_2\pi_2 + p_2}{c_2^2}u_2, -\frac{\gamma_2\pi_2 + p_2}{c_2^2}H_2\right)^T$$

$$r_2 = (0, 1, u_1 - c_1, H_1 - u_1c_1, 0, 0, 0)^T$$

$$r_3 = \left(0, 1, u_1, H_1 - \frac{c_1^2}{\gamma_1 - 1}, 0, 0, 0\right)^T$$

$$r_4 = (0, 1, u_1 + c_1, H_1 + u_1c_1, 0, 0, 0)^T$$

$$r_5 = (0, 0, 0, 0, 1, u_2 - c_2, H_2 - u_2c_2)^T$$

$$r_6 = \left(0, 0, 0, 0, 1, u_2, H_2 - \frac{c_2^2}{\gamma_2 - 1}\right)^T$$

$$r_7 = (0, 0, 0, 0, 1, u_2 + c_2, H_2 + u_2c_2)^T.$$

References

[1] Baer M, Nunziato J. A two-phase mixture theory for the deflagration-to-detonation transition (DDT) in reactive granular materials. Int J Multiphase Flow 1986;12:861–89.
[2] Saurel R, Abgrall R. A multiphase Godunov method for compressible multifluid and multiphase flows. J Comput Phys 1999;150:425–67.
[3] Saurel R, Lemetayer O. A multiphase model for compressible flows with interfaces, shocks, detonation waves and cavitation. J Fluid Mech 2001;431:239–71.
[4] Yeom G-S, Chang K-S. A modified HLLC-type Riemann solver for the compressible six-equation two-fluid model. Comput Fluids 2013;76:86–104.
[5] Allaire G, Clerc S, Kokh S. A five-equation model for the simulation of interfaces between compressible fluids. J Comput Phys 2002;181:577–616.
[6] Abgrall R. How to prevent pressure oscillations in multicomponent flow calculations: a quasi conservative approach. J Comput Phys 1996;125:150–60.
[7] Li Q, Feng H, Cai T, Hu C. Difference scheme for two-phase flow. Appl Math Mech 2004;25:536–45.
[8] Zein A, Hantke M, Warnecke G. Modeling phase transition for compressible two-phase flows applied to metastable liquids. J Comput Phys 2010;229:2964–98.
[9] Tokareva S, Toro E. HLLC-type Riemann solver for the Baer-Nunziato equations of compressible two-phase flow. J Comput Phys 2010;229:3573–604.
[10] Tian B, Toro E, Castro C. A path-conservative method for a five-equation model of twophase flow with an HLLC-type Riemann solver. Comput Fluids 2011;46:122–32.
[11] Ambroso A, Chalons C, Raviart P-A. A Godunov-type method for the seven-equation model of compressible two-phase flow. Comput Fluids 2012;54:67–91.
[12] Abgrall R, Saurel R. Discrete equations for physical and numerical compressible multiphase mixtures. J Comput Phys 2003;186:361–96.
[13] NVIDIA Corporation, NVIDIA Compute Unified Device Architecture Programming Guide Version 1.0, 2007.
[14] NVIDIA Corporation, NVIDIA CUDA C Programming Guide Version 4.0, 2011.
[15] T. Brandvik, G. Pullan, Acceleration of a 3D Euler solver using commodity graphics hardware, in: 46th AIAA Aerospace Sciences Meeting and Exihibit, 2008.
[16] Elsen E, LeGresley P, Darve E. Large calculation of the flow over a hypersonic vehicle using a GPU. J Comput Phys 2008;227:10148–61.
[17] J.C. Thibault, I. Senocak, CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows, in: 47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition, 2009, pp. 5–8.
[18] P. Micikevisius, 3D finite difference computation on GPUs using CUDA, 2nd Worksop on General Purpose Processing on Graphics Units, 2009, pp. 79–84.
[19] Khajeh-Saeed A, Perot J. Direct numerical simulation of turbulence using GPU accelerated supercomputers. J Comput Phys 2013;235:241–57.
[20] Wang C, Liu T, Khoo B. A real ghost fluid method for the simulation of multimedium compressible flow. SIAM J Sci Comput 2006;28:278–302.
[21] Toro E, Spruce M, Speares W. Restoration of the contact surface in the HLL-Riemann solver. Shock Waves 1994;4:25–34.
[22] E. Toro, Riemann Solvers and Numerical Methods for Fluid Dynamics, Springer-Verlag 1999.
[23] van Leer B. Towards the ultimate conservative difference scheme, V. a second order sequel to Godunov's method. J Comput Phys 1979;32:101–36.
[24] Liu T, Khoo B, Yeo K. Ghost fluid method for strong shock impacting on material interface. J Comput Phys 2003;190:651–81.

[25] Tang H, Huang D. A second-order accurate capturing scheme for 1D invisid flows of gas and water with vacuum zones. J Comput Phys 1996;128:301–18.
[26] Haas J-F, Sturtevant B. Interaction of weak shock waves with cylindrical and spherical gas inhomogeneities. J Fluid Mech 1987;181:42–76.
[27] Quirk JJ, Karni S. On the dynamics of a shock-bubble interaction. J Fluid Mech 1996;318:129–63.
[28] Fedkiw RP, Aslam T, Merriman B, Osher S. A non-oscillatory Eulerian approach to interfaces in multimaterial flows (the Ghost Fluid Method). J Comput Phys 1999;152:457–92.
[29] Bourne N, Field J. Shock induced collapse of single cavities in liquids. J Fluid Mech 1992;244:225–40.
[30] Grove J, Menikoff R. Anomalous reflection of a shock wave at a fluid interface. J Fluid Mech 1990;219:313–36.
[31] Ball G. Shock induced collapse of a cylindrical air cavity in water: a free Lagrange simulation. Shock Waves 2000;10:265–76.
[32] Hankin R. The Euler equations for multiphase compressible flow in conservation form. J Comput Phys 2001;172:808–26.
[33] Hu XY, Khoo BC. An interface interaction method for compressible multi-fluids. J Comput Phys 2004;198:35–64.
[34] Nourgaliev R, Dinh TN, Theofanous T. Adaptive characteristics-based matching for compressible multi-fluid dynamics. J Comput Phys 2006;213:500–29.
[35] Sambasivana SK, Udaykumar HS. A sharp interface method for high-speed multi-material flows: strong shocks and arbitrary material pairs. Int J Comput Fluid Dynam 2011;25:139–62.