Computer Physics Communications 207 (2016) 114–122

Contents lists available at ScienceDirect

Computer Physics Communications

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

GPU accelerated cell-based adaptive mesh refinement on unstructured quadrilateral grid

Xisheng Luo, Luying Wang, Wei Ran, Fenghua Qin *

Advanced Propulsion Laboratory, Department of Modern Mechanics, University of Science and Technology of China, Hefei, 230026, China

ARTICLE INFO

Article history:
Received 27 March 2015
Received in revised form
15 May 2016
Accepted 23 May 2016
Available online 30 May 2016

Keywords:
Cell-based AMR
CUDA GPU computing
Unstructured mesh

ABSTRACT

A GPU accelerated inviscid flow solver is developed on an unstructured quadrilateral grid in the present work. For the first time, the cell-based adaptive mesh refinement (AMR) is fully implemented on GPU for the unstructured quadrilateral grid, which greatly reduces the frequency of data exchange between GPU and CPU. Specifically, the AMR is processed with atomic operations to parallelize list operations, and null memory recycling is realized to improve the efficiency of memory utilization. It is found that results obtained by GPUs agree very well with the exact or experimental results in literature. An acceleration ratio of 4 is obtained between the parallel code running on the old GPU GT9800 and the serial code running on E3-1230 V2. With the optimization of configuring a larger L1 cache and adopting Shared Memory based atomic operations on the newer GPU C2050, an acceleration ratio of 20 is achieved. The parallelized cell-based AMR processes have achieved 2x speedup on GT9800 and 18x on Tesla C2050, which demonstrates that parallel running of the cell-based AMR method on GPU is feasible and efficient. Our results also indicate that the new development of GPU architecture benefits the fluid dynamics computing significantly.

1. Introduction

In recent years, the GPU (Graphics Processing Unit), once used only for graphics processing, has been extended to general purpose computing for its high computing power and bandwidth. Some early researches adopted graphics programming languages such as Cg, OpenGL to accelerate particle algorithms [1–3]. These works showed great potential of using GPU for scientific computing. But coding for scientific computing with these languages was difficult and the application fields were also limited. However, the development of general purpose computing on GPU has never stopped. NVIDIA Corporation released their parallel computing model called CUDA (Compute Unified Device Architecture) for general purpose computing in 2007 which provides an easy-to-use tool for scientific computing and is now widely used in many fields. Many researchers have used the tool in Computational Fluid Dynamics (CFD) and obtained remarkable results of performance increasing. Thibault et al. developed a Navier–Stokes solver for incompressible flow on multi-GPU with a 2nd order accurate central difference scheme and achieved 100× speedup [4]. Bailey and his co-workers used CUDA for accelerating Lattice Boltzmann Method on GPU and obtained remarkable performance enhancement [5]. Frezzotti's group adopted semi-regular methods to solve the Boltzmann equation on GPUs with high efficiency [6]. Ran et al. realized the GPU accelerated CESE method for 1D shock tube problems and achieved high acceleration ratios [7]. Brodtkorb et al. implemented shallow water simulations on GPUs and performed a detailed analysis of it [8]. Lutsyshyn presented a scheme for the parallelization of quantum Monte Carlo method on GPU and the program was benchmarked on several models of NVIDIA GPUs [9].

Implementing CFD method on GPU greatly depends on the mesh type used. Compared with the structured counterpart, methods based on unstructured mesh cannot be efficiently accelerated by GPU because the unstructured configuration leads to the non-coalescent memory accessing on GPU. Some researchers made their efforts to overcome this difficulty. Corrigan et al. implemented an unstructured grid based Euler solver on GPU and obtained a speedup of 33×'s [10]. Kampolis et al. accomplished a GPU accelerated Navier–Stokes solver on unstructured grid in the same year [11] and achieved a remarkable computing performance increasing. Waltz described the performance of CHICOMA, a 3D unstructured mesh compressible flow solver, on GPU and observed speedup of 4–5× over single-CPU performance [12]. Lani et al. provided a GPU-enabled finite volume solver for ideal magnetohydrodynamics on unstructured grids within the COOLFluiD

* Corresponding author. Tel.: +86 551 63600947; fax: +86 551 63601795.
E-mail address: qfh@ustc.edu.cn (F. Qin).

http://dx.doi.org/10.1016/j.cpc.2016.05.018
0010-4655/© 2016 Elsevier B.V. All rights reserved.

platform [13]. Almost all authors employed the renumbering technique to cope the problem of non-coalescent memory accessing, which has been discussed in detail in [14]. As demonstrated in their works, with the renumbering technique, shared memory can be introduced and therefore their codes' performance is efficiently improved.

In fact, the renumbering technique is suitable for static unstructured mesh and may fail for dynamic unstructured mesh. The dynamic unstructured mesh can be generated by adaptive mesh refinement (AMR) which is one of the most important approaches in CFD. In the method, by refining the coarse cells where truncation error is large enough, it takes much less computing resources to solve conservation equations than using fine uniform cells. However, the adaptive mesh is complicated and dynamic, which is not easy to be parallelized, especially on GPU. Wang and his team [15], together with group leading by Hsi-Yu Schive [16], have implemented solvers on a structured mesh with the AMR method. However, porting the mesh adapting part on GPU was avoided in their implementations. In the cell-based AMR method, if mesh adapting is processed on CPU, the data exchanges frequently between CPU and GPU, which will certainly introduce a bottleneck for the code's overall performance. Thus, removing the bottleneck is significant for implementing a parallel algorithm of mesh adapting on GPU, which motivates the current work. We will attempt to implement such a solver with the cell-based AMR on GPU.

The rest of this paper is organized as follows: Section 2 will provide a brief introduction of the numerical method and the cell-based AMR method used in this work. In Section 3, the implementation of the method on GPU is described in detail. The numerical results and the solver performance on GPU will be discussed in Section 4. Finally conclusions are drawn.

2. Numerical method

Consider the two-dimensional Euler equations for an inviscid, compressible flow, given as:

$$\frac{\partial \mathbf{U}}{\partial t}+\frac{\partial \mathbf{F}(\mathbf{U})}{\partial x}+\frac{\partial \mathbf{G}(\mathbf{U})}{\partial y}=0 \tag{1}$$

where **U**, **F** and **G** are defined as:

$$\mathbf{U}=\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ E\end{bmatrix},\qquad \mathbf{F}=\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ u(E+p)\end{bmatrix},\qquad \mathbf{G}=\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ v(E+p)\end{bmatrix}, \tag{2}$$

and

$$E=\rho\left[\frac{1}{2}(u^2+v^2)+e\right],\quad e=\frac{p}{\rho(\gamma-1)}, \tag{3}$$

where ρ, u, v, p, E, e and γ denote the density, x and y velocities, pressure, total energy, internal energy and specific heat, respectively.

The numerical method in this study is based on VAS2D developed by Sun and Takayama [17,18]. An adaptive unstructured quadrilateral mesh is adopted in the method and the finite volume method is adopted to discretize Eq. (1) by directly applying them to each control volume Ω:

$$\frac{d}{dt}\int_{\Omega}\mathbf{U}d\Omega+\int_{\partial\Omega}-\mathbf{F}dy+\mathbf{G}dx=0. \tag{4}$$

In the control volume of the ith quadrilateral cell, Eq. (4) can be approximated by evaluating the two integrals to a 2nd order accuracy both in space and time. The variation of physical variables is the flux through the cell's four edges, which yields:

$$\mathbf{U}_i^{n+1}=\mathbf{U}_i^n-\frac{\Delta t}{\Omega_i}\sum_{k=1}^{4}\hat{\mathbf{F}}_{i,k}, \tag{5}$$

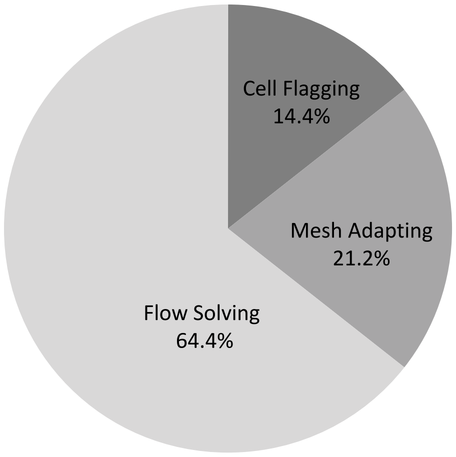


Fig. 1. Averaged percentage of computational time taken by each part for the serial code running on CPU.

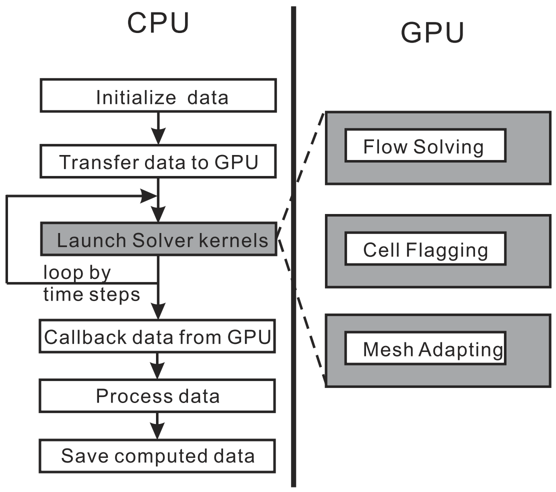


Fig. 2. The main computational procedure for the GPU-enabled flow solver based on unstructured quadrilateral mesh with the AMR.

where $\hat{\mathbf{F}}_{i,k}$ is the numerical flux vector on an edge and it is solved by the status at the midpoint of the edge (left side and right side).

Here, the *MUSCL-Hancock scheme* [19] is adopted for solving the status. At first, the gradients of primitive variables in a cell are evaluated by the *least squares method* with the cell's four neighbor cells. Then, the status can be computed by interpolation with gradients from the centroid to the edge midpoint. With the status at both sides of the edge, the flux can be solved by a Riemann solver. The **AUFS** scheme is adopted as the Riemann solver for its low artificial dissipation in this study, which is developed by Sun in 2003 [20]. When the numerical flux is obtained, following Eq. (5), the flow field is updated by the matching scheme.

As generally, the present computational procedure in every time step can be divided into three major parts: Flow Solving, Cell Flagging and Mesh Adapting. In the Flow Solving part, the primitive variables and their gradients are computed at each cell. In the part of Cell Flagging, based on the variation of physical quantity (usually density), each cell is determined to be adaptive or not and all the cells needed to be coarsened or refined are flagged as **Coarsen** or **Refine**. Then, in the Mesh Adapting part, the flagged cells are compressed or refined and the relative data is updated according to the data structure of mesh.

Fig. 1 sketches the averaged percentages of computational time taken by each part of the serial code when simulating the shock diffraction problem as will be discussed in Section 4. The Flow Solving part takes the major part (64.4%) of the consumed time while the Cell Flagging and the Mesh Adapting take the minor ones. The Flow Solving part and Cell Flagging are processed by looping edges or cells and can be divided into some individual loops. Therefore, they are easy to be implemented in parallel way. However, we have to overcome the difficulty of parallelizing the Mesh Adapting part before the serial code can be accelerated by GPU.

3. GPU implementation

Because the data exchanging between GPU and CPU is expensive, as a golden rule, it should be avoided as far as possible. Thus, to achieve a higher computing performance, the solver is designed as totally running on GPU in this study. The main computational procedure is sketched in Fig. 2. First, in the data initialization, the program allocates a block of large size memory, reads the mesh data, initializes the flow field and carries out the initial adaption. Then the initialized data is sent to GPU. After that, those solver kernels loop by time step and update the flow field. When the computation is finished or a specific time step is reached, the data will be sent back to the Host Memory from the Device Memory for postprocess. Finally, the processed data is saved in files.

As mentioned in the previous section, the Flow Solving part and Cell Flagging are easy to be implemented on GPU. Here, we shall discuss mainly how to implement Mesh Adapting on GPU.

3.1. Adaptive unstructured quadrilateral mesh

The present AMR method which is also developed by Sun and Takayama [18] is a locally adaptive method and the adaptation procedure is suitable for parallelization. Unstructured quadrilateral mesh is used. Cells and their edges are connected via a Cell-Face structure as shown in Fig. 3(a). A cell stores its 4 neighbor edges' indexes in {*Neighbor edge list*}, and correspondingly, an edge saves its 2 neighbor cells' indexes in {*Neighbor cell list*}. A '*List*' is an array for storing cell/edge's indices and all the data related to cell/edge is stored in a plain way in the memory. The present parallelization method is based on these '*Lists*'. During computing, one thread corresponds to one element stored in the '*List*' and threads pick up the addresses of cells/edges from the '*List*', then access data and do the actual process. Information exchanging between cell and edge is performed by reading both lists. In this arrangement, a cell only communicates with its neighbor edges while an edge only communicates with its neighbor cells. As a result, the cells and edges can be processed individually in each loop so that the parallelization can be easily implemented. It should be pointed out that, with the Cell-Face structure, the method can be easily extended to the three-dimensional case.

With the cell-based AMR method adopted in this study, each cell is individually processed when the mesh is adapting. In the refining process as shown in Fig. 3(b), an original cell is divided into four smaller cells, while the smaller cell is called 'son' and the original cell is called 'father'. At the same time, the father's each edge splits into two shorter edges, while the shorter edge is called 'daughter' and the original edge is called 'mother'. If a cell is refined, the cell's index is added into the {*Father list*} and its edges' indexes are added into the {*Mother list*}. The son's refinement level is 1 greater than its father's. The original cell and its edges' information are still stored because the cell and edges' information can be used directly in case that the cell needs be coarsened. The coarsening process is opposite to the refining one.

The cell's adaption is determined by its truncation error. Because a 2nd order numerical method is employed in this study, the distribution of physical quantity is linear in each cell. If the distribution of physical quantity is nonlinear, information higher than 2nd order is missed. Therefore, the refinement is adopted for this situation and the 2nd order derivative term is considered in the truncation error estimation. Here the truncation error estimation on an edge ϵ_V is given as:

$$\epsilon_V = \max\left(\frac{|\mathbf{r}_{ji}((\nabla V)_c - (\nabla V)_i)|}{\alpha_f \rho_c + |\mathbf{r}_{ji}(\nabla_r V)_i|}, \frac{|\mathbf{r}_{ji}((\nabla V)_c - (\nabla V)_j)|}{\alpha_f \rho_c + |\mathbf{r}_{ji}(\nabla_r V)_j|}\right), \qquad (6)$$

where V is the physical quantity (usually density) used for estimation, α_f is introduced to prevent a zero denominator, i, j denote the edge's neighbor cells i and j, $\mathbf{r}_{ji}$ represents the space vector between the two cell centers, ρ_c is the average density of cells i and j. The $(\nabla V)_c$ is the gradient directly computed by the primary variables' difference of the edge's neighbor cells divided by $|\mathbf{r}_{ji}|$. $(\nabla_r V)_i$ and $(\nabla_r V)_j$ are the cell's gradients along $\mathbf{r}_{ji}$.

Then the adaption criterion is given as:

$$\begin{cases} \textbf{Refine}, & \epsilon_T > \epsilon_r \\ \textbf{Coarsen}, & \epsilon_T < \epsilon_c, \end{cases} \qquad (7)$$

where ϵ_T is the cell's truncation error estimation, which is the maximum one of its four neighbor edges' ϵ_V. As a default situation in this work, $\epsilon_r = 0.08$, $\epsilon_c = 0.05$ and $\alpha_f = 0.03$.

In addition, the difference of refinement level between two neighbor cells of an edge is restricted to a maximum of 1. Or, even when the cell's truncation error is satisfied with Eq. (7), the cell is not refined or coarsen.

3.2. Mesh adapting on GPU

As stated before, Mesh Adapting is not easy to be implemented on GPU. Both Wang's and Schive's groups have chosen to manipulate this part on CPU [15,16]. On the other hand, Sun's work has shown that the mesh adapting in VAS2D can be partly vectorized with list operations, which provides a very important reference for parallel implementation of the mesh adapting [17, 18]. However, the vectorization of the mesh adapting is insufficient for two reasons. First, the list generation procedures are not vectorized and, without optimization, it will be a bottleneck of the code running on GPU. In addition, the method does not recycle wasted physical variables' memory. It is important to perform the memory recycling in the AMR method because the adapting mesh will consume memory quickly if new memory is constantly allocated for the new cells and edges. In this study, list generation procedures are optimized and the physical variables' memory is recycled. As a result, the method of mesh adapting can be implemented on GPU in a parallel manner.

In the present work, the Mesh Adapting on GPU consists three major procedures, Coarsening flagged father cells, Recycling null cells & edges and Refining flagged cells. First, the cells flagged as **Coarsen** are picked out from the {*Father list*} and added into the {*Temp list*}. The {*Father list*} has to be compressed because the picked-out cells are no longer fathers after coarsened. A father cell in the {*Temp list*} is decomposed as follows. Its 4 inner edges are deleted while the neighbor edges daughters are also deleted if they are no longer needed. The deleting process is executed by flagging these edges' *Flag* as **Null**. The same procedure is performed for the son cells which are not in use. Then, the physical variables in the father cell are updated by interpolating with the 4 sons. The sons are deleted and the father's *Flag* is flagged as **Non** which means that the cell or the edge is in the normal status. Also, the {*Mother list*} should be compressed, for some mothers having lost their daughters during the father cells' decomposition. Algorithm 1 shows the detailed procedure of coarsening cells.

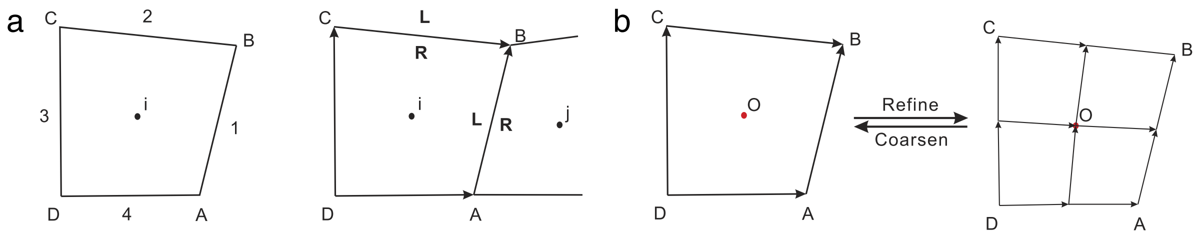


Fig. 3. Adaptive unstructured quadrilateral mesh. (a) Cell and edges. (b) An adapting cell.

Table 1
Key parameters of NVIDIA Geforce GT9800, Tesla C2050 and Intel Xeon E3-1230 V2.

	GT9800	C2050	E3-1230 V2
Type	G92	Fermi	Xeon E3-1200 V2
CUDA/CPU cores	112	448	4
Device memory/DRAM	1 GBytes	3 GBytes	16 GBytes
Shared memory	16 KBytes	48 KBytes	N/A
Register memory	32 KBytes	64 KBytes	N/A
Clock rate	1.50 GHz	1.15 GHz	3.3 GHz
Peak value	504 GFLOPS	1030 GFLOPS	87.8 GFLOPS

Algorithm 1: Coarsening flagged father cells.

```
Require: Inputs: $\mathbf{V}$, the primary variables; $Flag$, the flag of cells and edges; $\{Father\ list\}$; $\{Son\ list\}$; $\{Daughter\ list\}$.
Ensure: Outputs: $\mathbf{V}$; $Flag$.
  $Counter_C \leftarrow 0$; $Counter_R \leftarrow 0$.
  for all $father \in \{Father\ list\}$, parallel do
    if $(Flag)_{father}$=Coarsen then
      $P_1 \leftarrow \mathtt{Atomicadd}(Counter_C, 1)$.
      $\{Temp\ list\}_{P_1} \leftarrow father$.
    end if
    if $(Flag)_{father}$=Refined then
      $P_2 \leftarrow \mathtt{Atomicadd}(Counter_R, 1)$.
      $\{Temp\ list\}_{HALF+P_2} \leftarrow father$.
    end if
  end for
  $Maxfather \leftarrow Counter_R$
  for $i = 1$ to $Counter_R$, parallel do
    $\{Father\ list\}_i \leftarrow \{Temp\ list\}_{HALF+i}$
  end for
  for $i = 1$ to $Counter_C, father \leftarrow \{Temp\ list\}_i$, parallel do
    Read $\{Son\ list\}$, find father cell's son cells $s_1 \sim s_4$.
    Read $\{Neighbor\ edge\ list\}$, find inner edges $ie_1 \sim ie_4$.
    $(Flag)_{ie_1 \sim ie_4} \leftarrow$ Null.
    Read $\{Neighbor\ edge\ list\}$, find father's neighbor edges $ne_1 \sim ne_4$.
    Read $\{Neighbor\ cell\ list\}$, find father's neighbor cells $nc_1 \sim nc_4$.

    Read $\{Daughter\ list\}$, find $ne_j$'s daughter $d_1$ and $d_2, j \in [1, 4]$.
    if Any $(Flag)_{nc_j}$ = Refined then
      $\{Neighbor\ cell\ list\}_{d_1,d_2} \leftarrow father$.
    else
      $(Flag)_{d_1,d_2} \leftarrow$ Null; $(Flag)_{ne_j} \leftarrow$ Non.
    end if
    $\mathbf{V}_{father} \leftarrow \mathtt{Interpolate}(\mathbf{V}_{s_1 \sim s_4})$.
    $(Flag)_{s_1 \sim s_4} \leftarrow$ Null; $(Flag)_{father} \leftarrow$ Non.
  end for
```

Because the AMR process deletes and generates cells, the size of mesh usually varies. Therefore, the memory of deleted cells and edges should be recycled and reused for the new ones. The recycling process is quite simple. Those cells and edges flagged as **Null** are first found and their addresses are then added into the {*Null list*} and are ready for reusing by new cells or edges. Detailed information of recycling cells and edges is listed in Algorithm 2.

Algorithm 2: Recycling memory of cells & edges not in use.

```
Require: Input: $Flag$, the flag of cells and edges; $Maxroot_c$, maximum number of the root cells; $Maxroot_e$, maximum number of the root edges.
Ensure: Outputs:$\{Null\ list\}$, the list for null sons and edges' storage; $Counter_{NC}$, counter of null cells; $Counter_{NE}$, counter of null edges.
  $Counter_{NC} \leftarrow 0$; $Counter_{NE} \leftarrow 0$.
  for all $cell$, parallel do
    $i = cell - Maxroot_c$.
    if $(Flag)_{cell}$=Null and $i$ mod $4 = 0$ then
      $P_1 \leftarrow \mathtt{Atomicadd}(Counter_{NC}, 1)$.
      $\{Null\ list\}_{P_1} \leftarrow cell$.
    end if
  end for
  for all $edge$, parallel do
    $j = edge - Maxroot_e$.
    if $(Flag)_{edge}$=Null and $j$ mod $2 = 0$ then
      $P_2 \leftarrow \mathtt{Atomicadd}(Counter_{NE}, 1)$.
      $\{Null\ list\}_{HALF+P_2} \leftarrow edge$.
    end if
  end for
```

Refinement begins with picking out the cells flagged as **Refine** and adding them into the {*Temp list*}. Each cell in the {*Temp list*} becomes a new father cell adding into the {*Father list*}, and is flagged as **Refined**. The new father creates 4 son cells and 4 inner edges. The new cell edges are taking the positions provided by the {*Null list*} preferentially. If the space recorded in the {*Null list*} is not enough, their information will be placed at the tail of arrays. At the same time, the geometrical information is also built for new edges by using the father's information. The new father's neighbor edges are prepared for splitting and flagged as **Refine** if they were not split (the *Flag* is not **Refined**). Also, the neighbor information between cells and edges is updated since 4 new son cells and 4 inner edges are created. These procedures are described particularly in Algorithm 3.

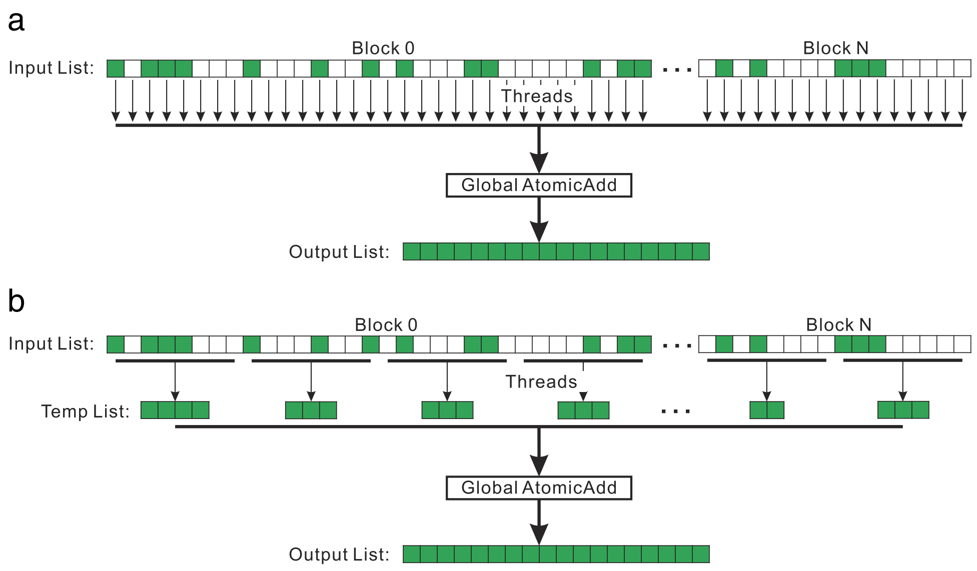


Fig. 4. The list processing with Global atomic operations. (a) Original. (b) Primary (basically optimized).

Algorithm 3: Refining flagged cells, part 1: Creating son cells.

```
Require: Inputs: $Flag$, the flag of cells and edges; $\{Null\ list\}$;
    $Counter_{NC}$, counter of null cells; $Counter_{NE}$, counter of null edges.
Ensure: Outputs:$Flag$; $\{Father\ list\}$; $\{Son\ list\}$.
  $Counter_{RC} \leftarrow 0$.
  for all $cell$, parallel do
    if $(Flag)_{cell}$=Refine then
      $P \leftarrow \mathtt{Atomicadd}(Counter_{RC}, 1)$.
      $\{Temp\ list\}_P \leftarrow cell$.
    end if
  end for
  for $i = 1$ to $Counter_{RC}$, $cell \leftarrow \{Temp\ list\}_i$, parallel do
    if $i < Counter_{NC}$ then
      $P_c \leftarrow \{Null\ list\}_i$.
    else
      $P_c \leftarrow 4 * (i - Counter_{NC}) + Maxcell$.
    end if
    $\{Son\ list\}_{cell} \leftarrow P_c$; Give index to 4 sons: $s_j \leftarrow P_c + j - 1, j \in [1, 4]$.
    if $2 * i + 1 < Counter_{NE}$ then
      $P_{e1} \leftarrow \{Null\ list\}_{HALF+2*i}$; $P_{e2} \leftarrow P_{e1} + 2$.
    else
      if $2 * i + 1 > Counter_{NE}$ then
        $P_{e1} \leftarrow 4 * (2 * i - Counter_{NE}) + Maxedge$; $P_{e2} \leftarrow P_{e1} + 2$.
      else
        $P_{e1} \leftarrow \{Null\ list\}_{HALF+2*i}$; $P_{e2} \leftarrow Maxedge$.
      end if
    end if
    Create 4 inner edges: $ie_1 \leftarrow P_{e1}, ie_2 \leftarrow P_{e1} + 1, ie_3 \leftarrow P_{e2}, ie_4 \leftarrow P_{e2} + 1$.
    Compute geometrical information of 4 inner edges $ie_{1\sim4}$.
    Create neighboring relationship between $s_{1\sim4}$ and $ie_{1\sim4}$.
    Read $\{Neighbor\ edge\ list\}$, find cell's 4 edges $ne_k, k \in [1, 4]$
    if $(Flag)_{ne_k}$ = Non then
      $(Flag)_{ne_k} \leftarrow$ Refine, $k \in [1, 4]$.
    end if
    $(Flag)_{s_j} \leftarrow$ Non; $(level)_{s_j} \leftarrow (level)_{cell} + 1$; $(Flag)_{cell} \leftarrow$ Refined.
    $\{Father\ list\}_{Maxfather+i} \leftarrow cell$.
  end for
```

The next step is splitting new father cell's neighbor edges, as listed in Algorithm 4. Those edges flagged as **Refine** are added into the {*Temp list*} and prepared for splitting. They will be split into 2 new daughter edges, flagged as **Refined** and added into the {*Mother list*}. If addresses provided by the {*Null list*} are still available, the new daughters occupy them. Otherwise, they are placed after the tail. The new daughters succeed their mothers' information of geometrical and boundary type.

Algorithm 4: Refining flagged cells, part 2: Splitting flagged edges.

```
Require: Inputs: $Flag$, the flag of cells and edges; $\{Temp\ list\}$; $\{Null\ list\}$; $Counter_{RC}$, counter of new fathers; $Counter_{NE}$, counter of
    null edges.
Ensure: Outputs:$\{Mother\ list\}$; $\{daughter\ list\}$.
  $Counter_{RE} \leftarrow 0$
  for all $edge$, parallel do
    if $(Flag)_{edge}$=Refine then
      $P \leftarrow \mathtt{Atomicadd}(Counter_{RE}, 1)$.
      $\{Temp\ list\}_{HALF+P} \leftarrow cell$.
      $(Flag)_{edge} \leftarrow$ Refined.
    end if
  end for
  for $i = 1$ to $Counter_{RE}$, $edge \leftarrow \{Temp\ list\}_{HALF+i}$, parallel do
    if $i < Counter_{NE} - 2 * Counter_{RC}$ then
      $P_e \leftarrow \{Null\ list\}_{HALF+2*Counter_{RC}+i}$.
    else
      if $Counter_{NE} < 2 * Counter_{RC}$ then
        $P_e \leftarrow 4 * (2 * Counter_{RC} - Counter_{NE}) + Maxedge + 2 * i$.
      else
        $P_e \leftarrow 2 * (2 * Counter_{RC} - Counter_{NE} + i) + Maxedge$.
      end if
    end if
    Give index to 2 daughters: $d_1 \leftarrow P_e, d_2 \leftarrow P_e + 1$.
    $\{Daughter\ list\}_{edge} \leftarrow P_e$.
    Read $\{Neighbor\ cell\ list\}$, find edge's neighboring cell $l$ and $r$.
    Add $l$ and $r$ to $\{Neighbor\ cell\ list\}$ of $d_1$ and $d_2$.
    Compute geometrical information of $d_1$ and $d_2$.
    $(Flag)_{d1,d2} \leftarrow$ Non.
    $\{Mother\ list\}_{Maxmother+i} \leftarrow edge$.
  end for
```

The neighbor information between cells and edges should be updated again due to new daughter edges. After the neighbor

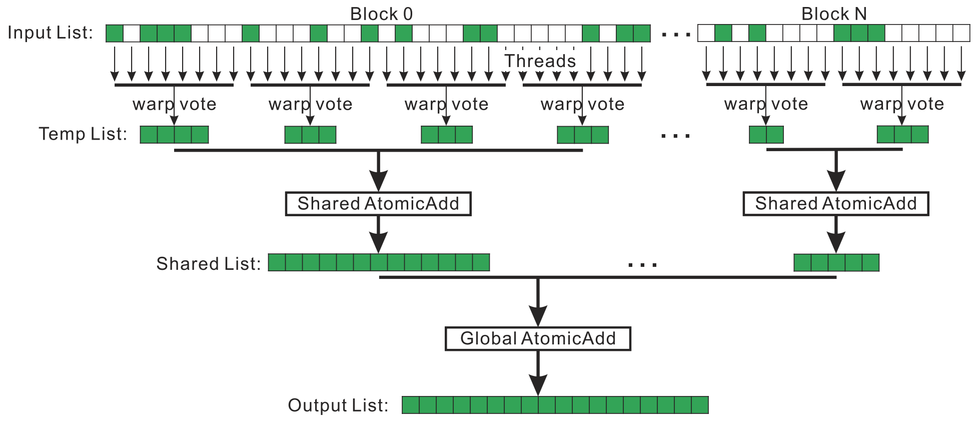


Fig. 5. The list processing optimized with Shared atomic operation. (The warp size is assumed as 8 here.)

relations of new cells and edges are established, the new sons' geometrical information is built by their neighboring edges' information. With the geometrical information, the new sons' physical variables are generated by the interpolation of fathers'.

3.3. *Optimization*

The optimization is closely related to the GPU architectures. Two GPUs and one CPU are chosen for discussion in the present study, namely NVIDIA Geforce GT9800, Tesla C2050 and Intel CPU Xeon E3-1230 V2 (Single Core is used). Their key parameters are given in Table 1.

3.3.1. *Parallel list operations*

The original implementation of parallel list operations is sketched in Fig. 4(a), in which the flagged cells (or edges) in the {*Input list*} are first picked out and added into the {*Output list*}. Each flagged cell (or edge) obtains its position in the {*Output list*} by atomic add instruction, which means that only one thread can access the position's variable because the atomic operation locks the variable when processing it. As a result, most threads are in the waiting queue for obtaining their positions and, obviously, the efficiency is very poor.

For the old GPU architectures such as G92, the list processing with atomic operations can be optimized. In the primary (basically optimized) implementation of parallel list operations, as shown in Fig. 4(b), each thread processes multiple cells (or edges) in a loop and picks out the flagged ones to the {*Temp list*}. Then the threads obtain the positions for the flagged ones in the {*Output list*} by atomic add and write them in. With this method, the frequency of using Global AtomicAdd is rapidly reduced. Thus, the list processing's performance can be enhanced. The primary implementation of parallel list operations brings out the primary GPU code which can be run on the old GPU GT9800 (GT9800-primary) and the newer GPU C2050 (C2050-primary) in the present study.

For the newer GPU architecture such as Fermi, more powerful functions are provided and therefore more optimizations can be realized. The optimized implementation of parallel list operations in the present study as shown in Fig. 5 uses warp vote functions in a warp to obtain the number of flagged cells (or edges) and their positions in the {*Temp list*} first. The new GPU code (C2050-optimized) is quite simple as:

```
Predicate=Flag;
Warpstat=__ballot(Predicate);
Position=__popc(Warpstat&((1<<(Threadid%Warpsize))−1));
Countofflag=__popc(Warpstat);
```

where the function __ **ballot**(*Predicate*) evaluates *Predicate* for all active threads of the warp and returns an integer whose Nth bit is set. Function __ **popc**(*Warpstatus*) returns the number of bits equal 1 in *Warpstat*. More information is provided in NVIDIA's CUDA C Programming Guide [21]. After the flagged cells (or edges) are picked out in a warp, one thread of a warp obtains a position in the {*Shared list*} by Shared AtomicAdd and appends the {*Temp list*} to the {*Shared list*}. Finally, one thread of a Block obtains the Shared list's position in the {*Output list*} with Global AtomicAdd and adds the {*Shared list*} to the {*Output list*}. With this method, the frequency of using Global AtomicAdd can be minimized. Because the Shared AtomicAdd does not hamper other Blocks' processing, the degree of parallelism can be much higher than that of the method shown in Fig. 4.

3.3.2. *Device memory accessing*

For the Flow Solving part, these operations are completely parallel processed by GPU. But there is a serious problem of coalescent accessing memory in the current study because the neighbor cells or edges are normally not neighbors in the lists in an unstructured mesh. As a result, when the cell or edge accesses data from its neighbor cells or edges by the neighbor lists, the data cannot be accessed in a coalescent way. This usually drags performance seriously. Some researchers have improved performance by adopting renumbering scheme for the problem [10,11]. However, it is infeasible for the mesh generated by the AMR method. Since the mesh is always varying, using such a scheme means that once the mesh is adapted, the memory storing physical variables, gradients and lists should be re-arranged accordingly. It is very likely that the scheme consumes more time than the non-coalescent accessing does, which has been addressed as well in [15].

Although the coalescent Device Memory accessing is not available, the L1 and L2 caches available in the newer Fermi GPU can increase the performance of Device Memory accessing. On the other hand, the L1 cache is built with Shared Memory on the chip. They have two configurations: Shared Memory/L1 48 KB/16 KB and 16 KB/48 KB. Obviously, a larger L1 cache can perform better in this study, which is also adopted in the C2050-optimized code.

4. Results and analysis

In this section, the simulation results of shock diffraction problem are presented to verify the two GPU codes (the primary code running on GT9800 and C2050, and the optimized code on C2050) and analyze their performances. All the codes are based on CUDA C and CUDA 5 and the simulation results are computed with the single-precision floating-point format.

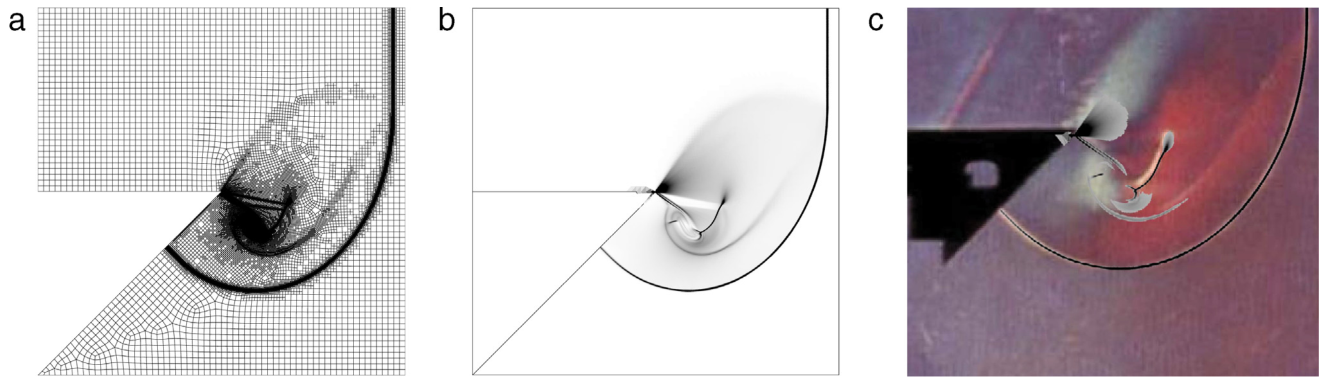


Fig. 6. The shock diffraction problem: incident shock wave with $Ms = 2.43$. GPU result is given with the CFL number 0.7, 4-level refinement at a dimensionless time of 0.125. (a) The refined mesh. (b) The numerical schlieren illustrating the wave configuration. (c) Experimental result for comparison.
Source: Reproduced courtesy of Professor Beric Skews, University of the Witwatersrand, published in [22].

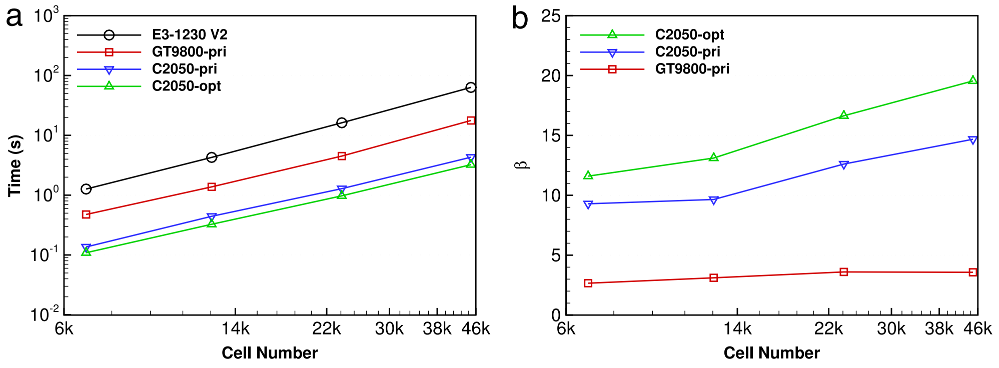


Fig. 7. Performance analysis of the shock diffraction problem: (a) execution time and (b) speedups β.

4.1. Verification of GPU's numerical results

In the shock diffraction problem, an incident shock passes around a convex corner and a part of it moves along the wall. The shock bends due to the reflected rarefaction wave. A slipstream and a contact surface are generated after the shock diffraction. A vortex is formed since the slipstream and the contract surface interact each other. If the incident shock is strong enough, a second shock will appear. This problem has been studied experimentally and numerically [22,23]. The simulation configuration in the present work is set as the same as one of the experiments by Professor Beric Skews [22]. The computational area is 0.64×0.64 and the convex corners' angle is 135°. The incident shock has a Mach number of $Ms = 2.43$ and $CFL = 0.7$ is used. Initially, the unstructured mesh contains 3640 cells and has a mesh size of $\Delta x \approx \Delta y \approx 0.01$. The maximum refinement level is set as 4. The dimensionless initial conditions of primary variables are given as:

$$(\rho, u, v, p) = \begin{cases} (3.249, 1.990, 0, 6.722), & x < 0.26, y > 0.32 \\ (1.0, 0, 0, 1.0), & \text{otherwise.} \end{cases}$$

With no surprises, the two GPU codes provide the same results. The refined mesh with 4-level refinement is illustrated in Fig. 6(a), which contains 44,689 cells inside. Numerical schlieren is shown in Fig. 6(b), in which the wave configuration of the incident shock, the diffracted shock, the second shock, the slipstream, the contact surface and the vortex can be identified clearly. As shown in Fig. 6(c), the positions of the shock waves agree with the experimental results very well [22].

4.2. Acceleration performance

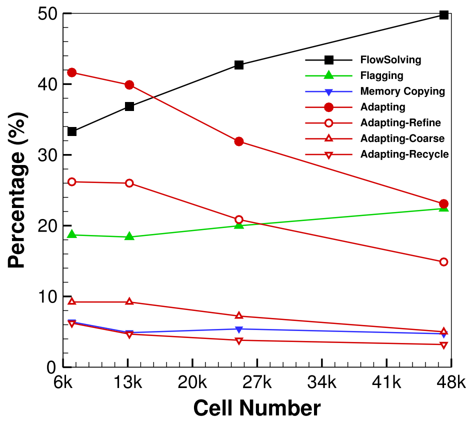
To analyze the code's performance, the running time of the parallel code on GPU is compared with that of the serial one on CPU. The speedup β is defined as:

$$\beta = \frac{t_{CPU}}{t_{GPU}}, \tag{8}$$

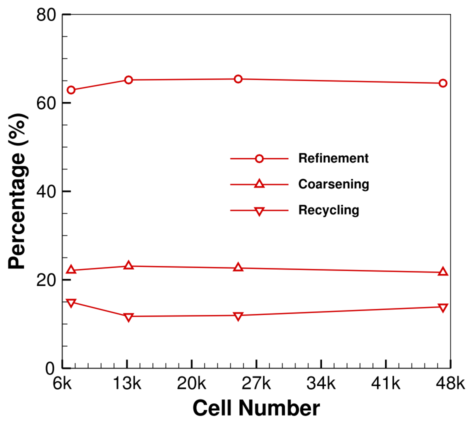
where the total executing time on CPU, t_{CPU}, comprises only the time of the main loop executed while the total processing time on GPU, t_{GPU}, which includes the additional time of transferring data between Host and Device for fairness. Single-precision floating-point format is adopted in both GPU and CPU. Key parameters of chosen GPU and CPU are listed in Table 1. Two benchmark problems are analyzed here.

The computing times and speedups of the shock diffraction problem are shown in Fig. 7. The x-coordinate shows the final number of cells for different refinement levels (from 1 to 4) in a logarithmic scale. In Fig. 7(a), the total executing times of GPUs and CPU rise in a nearly linear way as the cell number increases under the logarithmic scale. Fig. 7(b) illustrates the speedups β of primary parallel code (basically optimized) running on GT9800, C2050 and the optimized code running on C2050. The speedup β of GT9800-primary has a little increase from refinement level 1 to 4, which is about 3.5 in average. The speedups β of code on C2050 rise obviously as the cell number increases. When the refinement level is 4, $\beta = 15$ and 20 for the C2050-primary case and for the C2050-optimized one, respectively, are obtained.

Generally, the speedup of the parallel code running on GT9800 is limited mainly by the non-coalescent accessing of Device Memory. As described before, the unstructured mesh with AMR makes it very difficult to coalescent accessing of Device Memory. It is an extremely serious problem for such an old GPU with strict conditions of coalescent accessing. At the same time, running the same code on Fermi GPU C2050 is about 3 times faster than that on GT9800. As shown in Table 1, the C2050 has a higher



(a) Whole computation.



(b) Adapting process.

Fig. 8. The average percentages of time consuming taken by different parts of the optimized code running on C2050 against cell number.

Table 2
The execution time and speedups of codes' different parts for shock diffraction problem at refinement level 4. GPUs' time consuming data is obtained by the NVIDIA Visual Profiler.

	Flow solving	Cell flagging	Mesh adapting
E3-1230 V2	40.71 s	9.09 s	13.40 s
GT9800-pri (β)	6.92 s (5.9)	3.03 s (3.0)	6.95 s (1.9)
C2050-pri (β)	1.90 s (21.4)	0.93 s (9.8)	1.38 s (9.7)
C2050-opt (β)	1.61 s (25.3)	0.73 s (12.5)	0.75 s (18.0)

theoretical peak value of floating-point computing (about 2 times of GT9800). Besides, the Fermi architecture has less restriction on coalescent accessing and the L1 & L2 caches are introduced which make it have better performance of Device Memory accessing. The considerable performance increasing of the same code running on the newer device has showed that the new development of GPU architecture brings with significant improvements for fluid dynamics computing.

For cases running on C2050, the optimized code can have a larger L1 cache and can adopt atomic operations on Shared Memory. As given in Table 2, with a larger L1 cache, the performance of the Flow Solving part is enhanced about 15.3% and the Cell Flagging part about 21.5%. Meanwhile, with the optimized list processing, the Mesh Adapting part's performance is improved by 45.7%. These results imply that the optimized method is effective and successful, especially for the Mesh Adapting process.

The average percentages of time consuming taken by each part of optimized code running on C2050 against cell number are illustrated in Fig. 8. Comparing with the results of serial code running on CPU (see Fig. 1), the Flow Solving part does not take up the most percentage since the other two parts consume much time. Especially, if less cell is used in the computation, more percentage of cell has to be refined or coarsened and the Adapting process takes up more time. So, it is reasonable that the percentages of Flow Solving and Cell Flagging increase and percentage of Adapting decreases when the cell number increases, as shown in Fig. 8(a). On the other hand, if the CFL number is lower, time steps increase and subsequently the executing time also increases. But the Adaption Processing will be executed less frequently. As an inference, despite the executing time increases, the speedup will increase, too. Thus, it is significant for the cases requiring a low CFL number. It should be noted that the changing tendency of Refinement, Coarsening and Recycling with the cell number is the same as that of Mesh adapting. The percentages of time taken by the three kernels over the Adapting time change hardly with the cell number, as shown in Fig. 8(b). Therefore, a different method is needed to further improve the adaption performance on GPU.

5. Conclusions

The cell-based AMR on unstructured quadrilateral mesh is realized on GPU in this study. Specifically, we implemented and optimized the well-validated numerical method-VAS2D on GPU: Null memory recycling is added to improve the utilization efficiency of memory; List processing is parallelized on GPU with low frequency atomic operations. In this way, we have made one step further to realize the AMR on GPU. Our work is, to the best of our knowledge, the first unstructured cell-based algorithm that has been fully implemented on GPU.

The shock diffraction problem is simulated with the solver running on CPU (Intel E3-1230 V2) and on GPUs (Geforce GT9800 and Tesla C2050) for comparison. The simulation results are consistent with the experimental result, which validates the method implemented on GPU. The non-coalescent memory accessing is a serious problem which drags the performance of the GPU code and is nearly impossible to be solved in the cell-based AMR. However, 4×'s speedup on GT9800 and 15× on C2050 are still achieved by the GPU code to the series code on the CPU E3-1230. With the optimization of configuring a larger L1 cache and adopting Shared Memory based atomic operations, the optimized code gains a 20×'s speedup on the C2050. In the Mesh Adapting part, 2×'s speedup on GT9800 and 18× on Tesla C2050 are obtained by the parallelized algorithms, respectively. As a whole, the considerable speedups show our implementation is successful, and it has proved that running cell-based AMR method on GPU, including the mesh adapting processes, can be practicable and high-efficiency. Our results also indicate that the new development of GPU architecture benefits the fluid dynamics computing significantly.

In the future work, we will extend our code to three-dimensional for more practical usages. Implementing it on GPU cluster is also considered. However, for multiple nodes' parallel computing, the AMR algorithm used in present research maybe not efficient and the dynamic load balancing issues would be a big challenge. The mesh partitioning algorithm based on space filling curves (SFC) [24] which is a proximity preserving linear mapping of any multi-dimensional space cells maybe is very useful and suitable for our future work on multiple GPUs.

Acknowledgment

This research was carried out with the support of the National Natural Science Foundation of China under Grants 11172292 and 11272310.

References

[1] W. Li, X. Wei, A. Kaufman, Vis. Comput. 19 (7–8) (2003) 444–456.
[2] T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, K. Chihara, in: ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH, 2004.
[3] J. Yang, Y. Wang, Y. Chen, J. Comput. Phys. 221 (2) (2007) 799–804.
[4] J.C. Thibault, I. Senocak, in: Proceedings of the 47th AIAA Aerospace Sciences Meeting, 2009.
[5] P. Bailey, J. Myre, S.D. Walsh, D.J. Lilja, M.O. Saar, Parallel Processing, 2009. ICPP'09. International Conference on, IEEE, 2009, pp. 550–557.
[6] A. Frezzotti, G.P. Ghiroldi, L. Gibelli, Comput. Phys. Comm. 182 (12) (2011) 2445–2453.
[7] W. Ran, W. Cheng, F. Qin, X. Luo, J. Comput. Phys. 230 (24) (2011) 8797–8812.
[8] A.R. Brodtkorb, M.L. Sætra, M. Altinakar, Comput. & Fluids 55 (2012) 1–12.
[9] Y. Lutsyshyn, Comput. Phys. Comm. 187 (2015) 162–174.
[10] A. Corrigan, F. Camelli, R. Löner, J. Wallin, 19th AIAA Computational Fluid Dynamics, American Institute of Aeronautics and Astronautics, Inc., San Antonio, Texas, USA, 2009.
[11] I. Kampolis, X. Trompoukis, V. Asouti, K. Giannakoglou, Comput. Methods Appl. Mech. Engrg. 199 (9–12) (2010) 712–722.
[12] J. Waltz, Internat. J. Numer. Methods Fluids 72 (2) (2013) 259–268.
[13] A. Lani, M.S. Yalim, S. Poedts, Comput. Phys. Comm. 185 (10) (2014) 2538–2557.
[14] R. Löhner, Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, Wiley, 2008.
[15] P. Wang, T. Abel, R. Kaehler, New Astron. 15 (7) (2010) 581–589.
[16] H.Y. Schive, Y.C. Tsai, T. Chiueh, Astrophys. J. Suppl. Ser. 186 (2) (2010) 457.
[17] M. Sun, Numerical and experimental studies of shock wave interaction with bodies (Ph.D. thesis), Tohoku University, 1998.
[18] M. Sun, K. Takayama, J. Comput. Phys. 150 (1) (1999) 143–180.
[19] Doyle D. Knight, Elements of Numerical Methods for Compressible Flows, Cambridge University Press, New York, 2006.
[20] M. Sun, K. Takayama, J. Comput. Phys. 189 (1) (2003) 305–329.
[21] NVIDIA, CUDA, CUDA C Programming Guide 5.5, Tech. Report, NVIDIA Corporation, 2013.
[22] B.W. Skews, J. Fluid Mech. 29 (1967) 705–719.
[23] M. Sun, K. Takayama, J. Fluid Mech. 478 (2003) 237–256.
[24] G.V. Nivarti, M.M. Salehi, W.K. Bushe, J. Comput. Phys. 281 (2015) 352–364.