GPU-native gas dynamic solver on octree-based AMR grids

I Menshov\textsuperscript{1}, P Pavlukhin\textsuperscript{1,2}
\textsuperscript{1} Keldysh Institute of Applied Mathematics, Moscow, 125047, Russia \textsuperscript{2} Research and Development Institute Kvant, Moscow, 125438, Russia
E-mail: menshov@kiam.ru, pavelpavlukhin@kiam.ru

Abstract. GPU implementation of the Godunov-type gas dynamic solver on three-dimensional octree-based grids is proposed. In this implementation, all dynamic adaptive mesh refinement (AMR) procedures are performed entirely on GPU as opposed to most of other implementations where grid metadata is transferred to CPU for further updating. Conducted bench tests show that eliminating regular CPU-GPU grid data transferring in the proposed parallel AMR algorithms makes it possible to reduce the grid modification overhead to lower than 2\% on Nvidia Tesla V100. The results concerned with both CUDA and OpenMP implementation performance are also presented.

1. Introduction
Dynamic adaptive mesh refinement (AMR) approach is widely used in computational fluid dynamics (CFD) problems. It permits to increase local mesh resolution only in regions of interest such as steep-gradient fields or surfaces of moving bodies with complex geometry. This approach allows getting more precise solutions comparing to static non-adaptive grids with the same total resolution.

To utilize computational capabilities of modern supercomputers one needs to parallelize not only CFD solvers but also AMR procedures (refinement and coarsening grid cells). The AMR approach can be technically realized for arbitrary unstructured meshes. However, many serious issues come out, if it needs to parallelize the code for clusters with distributive memory model. Particularly, it is rather hard to implement effective load balancing when one needs to redistribute grid data between subdomains assigned to computational nodes. In part, this is because the local grid resolution in these subdomains may significantly vary during iterative solution process, which leads to very poor overall performance. At the same time, AMR implementations on quad-/octree-based grids (Fig. 1), such as P4est \cite{1}, Octor \cite{2}, demonstrate very good performance on thousands supercomputing nodes. One of the reasons for that fact is the space filling curve (SFC) which traverses through recursively refined cells defining global linear ordering (Fig. 1). This ordering preserves memory locality to some degree (compared to arbitrary unstructured grids) and permits to exploit simple load balancing algorithms.

In the last decade, supercomputers come equipped with multiple GPU accelerators which often bring the main contribution to the total system performance. Therefore, developers of CFD codes have become widely exploit such accelerators using CUDA or OpenCL. However, the AMR procedures performing dynamic grid modifications are difficult to realize on GPU,
and most of CFD AMR codes run basic solver procedures on GPU, while the AMR procedures are commonly executed on CPU. For example, Daino [3], a high-level framework for AMR applications, executes computational kernels on GPU but grid tree metadata is managed on CPU. The similar approach is also used in [4, 5]. In doing so, one needs to perform GPU-to-CPU and CPU-to-GPU data transfers after each cycle of grid modification; this leads to decrease in application performance (Fig. 2, a). In the present work we propose an approach where both the CFD kernel and the AMR procedures are running entirely on GPU (Fig. 2, b), and CPU is only used as caller for corresponding CUDA kernels. This approach eliminates regular data transfers between GPU and CPU, which results in better exploiting the GPU resources and releasing CPU for another work.

There are different types of AMR which are applied to different types of grids (structured/unstructured, with different cell topology). For GPU architectures, the most suitable grids are structured ones with quadrilateral (2D) and hexahedron (3D) cell topology. Such grids can be mapped to GPU memory more efficiently comparing to unstructured grids. For this reason, in the present work we use more specific type of structured grids, namely Cartesian grids that consist of rectangles or parallelepipeds. These grids are well combined with immersed boundary methods, e.g., the Free Boundary Method (FBM) [6], which make it possible to solve CFD problems with complex non-conforming (to the grid) geometry.

There exist several different AMR techniques applicable to structured grids. One is the patch-based AMR. In this approach, the numerical solution is projected on multiple grid levels presented as patches over the base grid (Fig. 3 a). New patches are added at higher levels when one needs to increase the local grid resolution. The advantage of this approach is that the solver always runs on each patch which is a simple structured subgrid with good memory locality. However, the grid projection operators that maintain and dynamically update patches for GPU are complex and time-consuming issues even in CPU implementations, not to mention porting all these procedures to GPU entirely. To minimize in some degree regular CPU-to-GPU and GPU-to-CPU data transfers which are immanent in the conventional GPU-accelerated
approach [5], patches with the highest grid resolution can be permanently placed in the GPU memory while the grid modification is entirely performed on CPU [7]. There is a limited number of implementations where most of the AMR procedures are natively executed on GPU, for example [8]. Besides issues mentioned above, the patch-based grids are hardly to exploit on supercomputers with the distributed memory model due to dynamic load balancing issues (which become more complicated when GPU are also utilized).

Another AMR approach exploits block-based grids. In this approach, the initial grid is divided into some fixed blocks which can also be recursively divided into smaller blocks. Refinement and coarsening are performed for whole block even if one needs to modify only one its cell (Fig. 3 b). This may obviously lead to undesirable increase in local grid resolution, for example, in regions with rather smooth solutions. By analogy with the patch-based approach, non-modified blocks can be permanently stored in the GPU memory thus avoiding excessive CPU-to-GPU data transfers in addition to native GPU implementation of most AMR procedures (so that only a smaller part of these procedures is executed on CPU) [9, 10].

Actually, one can avoid the drawback with the excessive grid refinement in the above block-based AMR approach if assumes that each block consists of only one grid cell. This leads to so-called cell-based AMR. In this approach, each grid cell can be recursively divided into subcells, so that the grid is represented by the tree structure (namely, quad- and octree for 2D- and 3D-structured grids), Fig. 1. The cell-based AMR exploits the tree graphs and can be efficiently implemented for parallel executions on thousands of CPU ([1, 2]), offering simple dynamic load balancing due to the space filling curve (SFC) which linearly orders all grid cells. The SFC data representation also preserves memory locality in some degree compared to pure unstructured grids. Therefore, in our opinion, the cell-based AMR with tree structures is the most robust and promising approach for parallel CFD computations on supercomputers nowadays.

For efficiently using computer systems equipped with GPU, one needs to implement all AMR procedures entirely on GPU to avoid CPU-to-GPU grid data transfers. To the author knowledge only a few papers ([11, 12]) address this issue. In these works 2D triangular grids are used. In the present paper we consider the cell-based AMR GPU implementation for 3D Cartesian grids. In [12], the parallel discontinuous Galerkin solver is considered that is executed on GPU along with fully GPU-native cell-based AMR procedures. We actually face the same problems as authors of this paper during developing the parallel AMR algorithms for GPU, but solve these problems in our proper manner. In what follows, we briefly discuss the numerical model to be solved with the cell-based AMR and then present proposed GPU-native parallel AMR algorithms. Efficiency of these algorithms is demonstrated on a test problem in Section 4.
2. Numerical method

We consider the model of compressible fluid flow that is written by the system of conservation laws for mass, momentum, and total energy in the following form:

\[ \frac{∂\vec{q}}{∂t} + \frac{∂\vec{F}_k}{∂x_k} = 0, \]

where \( \vec{q} \) is the conservative vector, and \( \vec{F}_k \) is the flux vector in the direction of the \( x_k \) coordinate, \( k = 1, 2, 3 \). On a given Cartesian grid with the octree-type AMR, the system of equations (1) is discretized by the second-order accurate FVM in space and the second-order accurate predictor-corrector scheme in time. The distribution of the primitive vector (i.e., density, components of the velocity vector, and pressure) within each computational cell is approximated by a linear function that is defined by the value at the barycenter of the cell and the limited gradient,

\[ \vec{z}(\vec{x}) = \vec{z}_i + \nabla \vec{z}_i, (\vec{x} - \vec{x}_i), \vec{x} \in C_i \]

Each cell is surrounded by a number of faces \( \sigma \), \( 6 \leq \sigma \leq 24 \), that separate the \( i \)-th cell from its neighbors. The index number of the cell neighbored by the \( i \)-th cell through the face \( \sigma \) is denoted as \( \sigma(i) \). It should be noted that we mean by face the common boundary between two neighbouring cells, i.e., face \( \sigma = \overline{C_i \cap C_{\sigma(i)}} \). Then, the numerical scheme that updates the solution from one time level \( n \) to the next one is written in two predictor-corrector stages:

\[
\begin{align*}
\vec{q}_{i}^{n+1} & = \vec{q}_{i}^{n} + \frac{\Delta t}{V_i} \sum_{\sigma} \vec{F}_k(z_{i}^{n,\sigma}, z_{\sigma(i)}^{n,\sigma}) n_{\sigma,k} S_{\sigma} \\
\end{align*}
\]

where \( n_{\sigma} \) is the outward unit normal to the face \( \sigma \), \( V_i = \text{Vol}(C_i) \), \( S_{\sigma} = \text{Area}(\sigma) \), \( \Delta t \) is the time step, and the face interpolated values are calculated with limited gradient,

\[
\begin{align*}
z_{\sigma}^{n,\sigma} & = \vec{z}_i + \nabla \vec{z}_i, (\vec{x}_\sigma - \vec{x}_i) \\
z_{\sigma(i)}^{n,\sigma} & = \vec{z}_{\sigma(i)} + \nabla \vec{z}_{\sigma(i)}, (\vec{x}_\sigma - \vec{x}_{\sigma(i)}) \\
\end{align*}
\]

Here \( \vec{x}_\sigma \) is the barycenter of the face \( \sigma \).

We use the Godunov method for approximating the numerical flux in (3). Solving the local Riemann problem in the normal direction for each face with the initial data to be equal face interpolated primitive vectors \( z_{\sigma}^{n,\sigma} \) and \( z_{\sigma(i)}^{n,\sigma} \), the numerical flux \( \vec{F}_k \) is then approximated by the differential flux evaluated at the exact solution to the above Riemann problem [13].

All dangling (leaf) cells of the AMR grid are calculated sequentially, independently on which AMR level the cell relates. Therefore, the numerical scheme (3) and (4) updates the conservative vectors in all the cells, and it is stable providing that the timestep \( \Delta t \) meets the CFL condition, \( \Delta t = \min_i(\tau_i) \), \( \tau_i = \frac{(|u_k| + c_k)^{-1}}{h_k} \), where \( h_k \) is the \( i \)-th cell size in the \( k \)-th coordinate direction, \( u_k \) are components of the velocity vector, \( \{c_k\} \) is the velocity of sound.

The limited gradient is calculated with matrix limiting which is the extension of the one-dimensional slope limiting to the multidimensional case.

Let a function \( u(\vec{x}) \) be given by its values \( \{u_i\} \) at cell barycenters of an AMR Cartesian grid. By applying the LSM, first an unlimited gradient is calculated for each \( i \)-th cell from the system of the following equations:

\[ \nabla u_i, (\vec{x}_{\sigma(i)} - \vec{x}_i) = u_{\sigma(i)} - u_i, \forall \sigma \]
By introducing the coordinate matrix \( M = [\Delta \vec{x}_i] \), \( \Delta \vec{x}_i = \vec{x}_{\sigma(i)} - \vec{x}_i \) and the incremental matrix \( \Delta U = [\Delta \sigma u_i] \) this yields the following unlimited gradient:

\[
\nabla u_i = (M^T M)^{-1} M^T \Delta U.
\]

(6)

For the baseline Cartesian grid without AMR, (6) is reduced to the central difference approximation of the partial derivatives of \( u(\vec{x}) \) in each coordinate direction.

The basic idea of the proposed matrix limiter is to apply the 1D slope limiting to each cell-to-neighbor direction. Considering the direction to a \( \sigma(i) \) neighbor, one can introduce the unlimited partial derivative \( \frac{\partial}{\partial \vec{x}} u_i \) and one-sided derivatives \( \frac{\partial}{\partial x^+} u_i \), \( \frac{\partial}{\partial x^-} u_i \) as follows:

\[
\frac{\partial}{\partial x} u_i = \frac{\nabla u_i, \Delta \vec{x}_i}{\Delta \sigma \vec{x}_i}, \frac{\partial}{\partial x^+} u_i = \frac{\Delta \sigma u_i}{\Delta \sigma \vec{x}_i}, \frac{\partial}{\partial x^-} u_i = 2\frac{\partial}{\partial x} u_i - \frac{\Delta \sigma u_i}{\Delta \sigma \vec{x}_i}
\]

(7)

Given these derivatives, we next do slope limiting in the \( \sigma(i) \)-direction and calculate the corresponding limited partial derivative:

\[
\frac{\partial}{\partial x} u_i = \varphi(\theta^+ x_i) \frac{\partial}{\partial x^+} u_i, \theta^+ x_i = \frac{\partial}{\partial x} u_i
\]

(8)

where \( \varphi(x) \) is the function of limiter. There are several options for choosing this function, which in 1D calculations guarantee the TVD property. Some of them are:

- \( \text{minmod} \): \( \varphi(x) = \min(1, x)^+ \)
- \( \text{MC} \): \( \varphi(x) = \min\left(2, 2x, \frac{1+x}{2}\right)^+ \)
- \( \text{SUPERBEE} \): \( \varphi(x) = \max(\min(2x, 1), \min(x, 2))^+ \)
- \( \text{NOTVD} \): \( \varphi(x) = \min(2x, 2)^+ \)

where the superscript + means the positive part of the function,

\[
f(x)^+ = \begin{cases} f(x), & x : f(x) \geq 0 \\ 0, & x : f(x) < 0 \end{cases}
\]

The limited gradient is constructed with limited partial derivatives \( \frac{\partial}{\partial x} u_i \) from the following conditions:

\[
(\Delta \sigma \vec{x}_i, \nabla u_i) = [\Delta \sigma \vec{x}_i] \frac{\partial}{\partial x} u_i, \forall \sigma.
\]

(9)

This can be solved with the LSM as follows:

\[
\nabla u_i = (M^T M)^{-1} M^T \Phi \Delta U,
\]

(10)

where \( \Phi \) is the diagonal matrix of limiters, \( \Phi = \text{diag}(\varphi(\theta^+)) \).

The AMR procedure is fulfilled with the refinement criterion that is based on the approach proposed in [14]. In this approach, an estimation of the second derivative is used to indicate which computational cell needs for refinement or coarsening. In the one-dimensional case on a uniform mesh this function estimator reads as

\[
\chi(u)_i = \frac{|u_{i+1} - 2u_i + u_{i-1}|}{|u_{i+1} - u_i| + |u_i - u_{i-1}| + 2|u_{i-1} + 2u_i + u_{i+1}|}.
\]

(11)

This criterion has the following desirable properties. It is local, just avoiding any global operations. This indicator is bounded, \( 0 \leq \chi(u)_i \leq 1 \), so one can preset tolerances. And
it is non-dimensional, so that specific physical parameters can be used without problems of dimensioning.

The extension of this indicator to multidimensions and cells of arbitrary topology can be suggested in the following way:

$$\chi(u)_i = \sqrt{\sum_{i,j} (A_{ij})^2 / \sum_{i,j} (B_{ij} + \varepsilon C_{ij})^2}.$$  

(12)

where the summation is taken over the number of space dimension, and

$$A_{ij} = \sum_{\sigma} \left( \frac{\partial u}{\partial x_i} \right)_{\sigma} n_{\sigma,j} S_{\sigma},$$

$$B_{ij} = \sum_{\sigma} \left| \left( \frac{\partial u}{\partial x_i} \right)_{\sigma} n_{\sigma,j} S_{\sigma} \right|,$$

$$C_{ij} = \sum_{\sigma} \left| \left( \frac{\partial |u|}{\partial x_i} \right)_{\sigma} n_{\sigma,j} S_{\sigma} \right|.$$  

(13)

For approximating the derivatives at cell interfaces in (13), one can use a simple interpolation of cell unlimited gradients, for example, by averaging the gradients with cell volumes. Also, to simplify computations, we replace the derivative of the modulus function with the divided finite difference,

$$\left( \frac{\partial u}{\partial x_i} \right)_{\sigma} \to \frac{\Delta u}{\Delta x_i}.$$  

The threshold values for cell refinement and coarsening, $\chi_{rf}$, $\chi_{cr}$, are generally model dependent. However, numerical experiments show that these values can be commonly chosen as $\chi_{rf} \sim [0.2, 0.25]$, $\chi_{cr} \sim [0.05, 0.15]$ [15]. The constant $\varepsilon$ is given a value of 0.01; it switches off unnecessary refinement in regions of small oscillations appearing due to loss of monotonicity.

The value assigned to the cell at the coarsening procedure is computed on the base of conservative vectors, $\vec{q}_{cr} = \sum_i \frac{q_{cr(i)}}{8}$ (or /4 for 2D grids), the index $cr(i)$ indicates children cell to be coarsen into the $cr$ cell. In the refinement procedure, the values assigned to the arising children cells are computed on the base of the conservative and the limited gradient of the refined cell:

$$\vec{q}_{cr(i)} = \vec{q}_{cr} + \left( \frac{\partial q}{\partial z} \right)_{cr} (\vec{x}_{cr(i)} - \vec{x}_{cr}).$$  

This ensures the conservative property in both the refinement and coarsening procedure.

3. GPU-native parallel AMR algorithms

General description of our GPU-native AMR algorithms could be found in [16]. Here we discuss some additional AMR algorithms, implementation details, and performance results in the case when AMR routines are combined with the CFD solver into completely a CUDA/OpenMP program.

An initial Cartesian grid called as base grid can be refined by dividing grid cells during CFD solver iterations, so some of the base cells become root cells for the local octree cell structure, Fig. 4 (to simplify description, all schemes are provided for 2D quadtree based grids). The recursive cell refinement is admitted inside each tree. If 8 cells make leaf nodes in the octree with a common parent, they can be coarsen. We maintain 2:1 grid balancing, i.e., each cell is admitted to have no more than 4 neighbors (or 2 neighbors for 2D grids) on any its face. This property must be kept in dynamic grid transformations (coarsening and refinement).

Logically, the AMR grid is represented by three arrays: one for base (marked as “□”), the second for anchored (marked as “○”), and the third for dangling (or leaf, marked as “○”) cells, Fig. 5. Dangling cells are real (physical) grid cells that contain data of the numerical solution; other cells represent metadata for the grid octrees structure with corresponding links between the cells. All cells metadata is stored in a compact form using a small number of
Figure 4. Local quadtrees in the base cells.

Figure 5. Grid arrays before (left) and after (right) coarsening/refinement.

bits; this meets the requirements of reducing memory consumption and decreasing memory bandwidth.

The base cell has the following metadata:

- full/empty flag (1 bit) that indicates whether the base cell is actual or stub one, the stub cell isn’t considered as an element of the real grid;
- link to child (30 bits) that is an array index for the child node which can be a dangling cell, if the base cell is not refined, or an anchored cell, if the base cell has the local octree structure;
- child type (1 bit) that can be the type of dangling or anchored cell;
- order numbers of dangling cells (32 bits) in the structure of the base cell.

The anchored cell has the following metadata:

- full/empty flag (1 bit) that indicates whether the anchored cell is actual or stub one, the stub cell isn’t considered as an element of the real grid;
- link to parent (30 bits) that is an array index for the parent node in the octree structure, which can be an anchored or a base cell;
- parent type (1 bit) that can be the type of anchored or base cell;
- 8 links to children (8 × 30 bits) are arrays of indexes for child octree nodes, which can be dangling or anchored cells;
- 8 corresponding child types (8 × 1 bit) that indicate dangling or anchored the cells are;

The dangling cell has the following metadata:

- full/empty flag (1 bit) that indicates whether the dangling cell is actual or stub one, the stub cell isn’t considered as an element of the real grid;
- link to parent (30 bits) that is an array of indexes for the parent node in the octree, which can be anchored or base cell;
- parent type (1 bit) indicating the cell is anchored or base;
- base index (30 bits) for the root of the local octree structure;
octree coordinates (9 × 3 bits) that define dangling cell address in the octree: i-th 3 bits define the cell or its parent position (of 8 ones) in the i-th tree layer. The current implementation supports up to 9-level octrees although it can be easily extended to any arbitrary value;

- 6 neighbor types for 6 cell faces (6 × 3 bits) define the boundary condition type, if the face is part of the outer boundary of the computational domain, or the relative tree layer of neighbor cells, which are of one upper, the same, or one lower level (2:1 grid balancing doesn’t permit other cases)

- 6 links to neighbors (6 × 30 bits) that are arrays of indexes for neighbor cells over each of 6 current cell faces depending on corresponding neighbor types. If the neighbor is on the same or one upper tree layer, the link is the array of indexes for the dangling neighbor cell. If there are four neighbors for the face of one lower tree layer, the link is the array of indexes for the anchored cell which is the parent for those neighbors;

- coarsening flag (1 bit) to be set by the criterion (12) and indicates that the cell is allowed for coarsening;

- refinement flag (1 bit) to be set by the criterion (12) and indicates that the cell has to be refined;

Additionally, the dangling cell has the physical state vector (density, components of the velocity vector, and pressure) of the 5 doubles size, and the extended vector for storing intermediate solution values (for example, gradients of the state vector) of the 35 doubles size.

When some (dangling) cells are coarsened or refined, theirs position is marked as “empty“ (marked as “×“) and newly created cells are placed in the end (yet non-filled part) of the arrays, Fig. 5. Since multiple CUDA/OpenMP threads may perform these operations simultaneously, the atomic addition instruction is used to insert new cells in the arrays of anchored and dangling cells.

By doing so, we continuously generate “holes“ in these arrays. This may lead to exhaustion of the available space for new cells, and therefore one needs to implement some kind of memory defragmentation procedure. Let’s consider the parallel defragmentation algorithm for dangling cells (Fig. 6); for anchored cells it will be the same.

Figure 6. Defragmentation algorithm.

A n-th OpenMP thread or CUDA warp is associated with n-th base cell and its local octree. The n-th base cell contains $S_n$ – the number of its dangling subcells, which is used in the prefix scan:

$$B_n = \sum_{i=1}^{n} S_i,$$
Obtained $B_n$ is then used as the start index in a new array for copying all dangling cells of the local octree in the SFC order from the current fragmented array. The prefix scan is naively implemented via consequential thread/warp synchronization, i.e., this procedure is performed with the inter-thread serialization process. Once $n$-th thread has computed $B_n$, its $S_n$ dangling cells are copied to the new array starting at $B_n$ index by traversing the local octree of the base cell in the SFC order. This part is potentially performed simultaneously in different threads without any synchronizations. Using the shared memory in GPU and the optimized prefix scan algorithm is the subject of the future work.

One can see that the proposed defragmentation algorithm requires additional memory for storing ordered and compacted dangling cells. However, new arrays do not actually allocated. Each dangling cell has the physical state vector (5 doubles) and the extended vector (35 doubles). Since the only needed for updating data is the physical state vectors, the space of extended vectors is used in the defragmentation procedure as a new array in which dangling cells data is copied. After that the physical state vector space is included in the extended vector space, i.e., swapping between the physical state vector and the part of extended vector memories is performed. Consequently, the defragmentation algorithm does not use any excessive memory and exploits only one allocated for the solver. The proposed defragmentation algorithm not only compacts grid data but also increases memory locality since cells are ordered by SFC (by local SFC in each octree and global one over base cells). The impactation of this property on the solver performance is the subject for future study.

In [12] another approach is implemented: new coarsen cells are added in place where initial yet non-coarsen ones were; new refined cells are added in free leftover (after coarsening) place or in the end (yet non-filled part) of cell arrays. This approach reduces number of “holes“ and leads to more compact and dense grid data storing, but it requires some additional effort for tracking and reusing free places in cell arrays. Furthermore, such grid modification leads to poor memory locality when references to neighbor cells are performed (although, it is mentioned in [12] that it does not matter). Nevertheless, in the case when the number of cells to be coarsen in solver iterations exceeds the number of cells to be refined, authors of [12] have to implement a kind of defragmentation procedure based on the original in-place compaction algorithm which also doesn’t require additional memory space.

The solver functions use data of neighboring (over faces) cells, therefore there should be an effective parallel algorithm for searching neighbors in the AMR grid. Each dangling (physical) cell has octree coordinates, so the neighbor cells coordinates are defined directly. The neighbor is then searched by simple traversing over the octree according to these coordinates. Each thread searches neighbors for its assigned dangling cells independently of other threads.

Since the solver to be used is based on the explicit time integration scheme, the corresponding parallel implementation is quite simple. Each thread performs computations for its assigned
dangling cell. The only drawback for such parallelization is double flux calculation for neighboring cells having common faces. This will be excluded in the future implementation that is assumed to support also implicit time integration schemes. The workflow for our CUDA AMR code is presented in Fig. 7. There is also OpenMP implementation with all identical multi-threaded AMR and solver procedures running on CPU.

If an iteration is that of the grid update (for example, each \( n \)-th iteration), then after having calculated the Refinement criterion for all dangling cells, one needs to modify coarsening/refinement cell flags before actual grid modification in order to guarantee of the 2:1 grid balancing. The corresponding parallel Tune_refine_flags and Tune_coarse_flags procedures representing the 2:1 grid balancing algorithm are described in what follows.

The purpose of these procedures is to reset coarsening/refinement flags for cells where the grid modification in accordance with the refinement criterion violates 2:1 balancing. Tune_refine_flags resets the refinement flag, if the cell has a neighbor of the upper octree layer (a bigger cell). Tune_coarse_flags resets the coarsening flag, if the cell has a neighbor of the same octree layer with the flag set for refinement or a neighbor is of the lower octree layer. Parallelization of these procedures is simple: each thread treats its own cells and updates flags without any inter-thread communication. After these procedures, the grid Coarsening and Refinement procedures are performed in accordance with the modified flags; this will preserve 2:1 balancing. Obviously, this approach potentially reduces the number of cells to be refined or coarsen due to the criterion (12), but the parallel algorithms proposed are fast, so the grid modification can be implemented after each timestep; this permits to track shock waves with small overhead as shown in the following section. Since link references to neighboring cells may become not valid after Coarsening, Refinement and Defragmentation procedures one has to perform Neighbors searching over all dangling cells.

4. Numerical Results

We used the three-dimensional Sedov blast wave [17] test case to verify our CFD solver combined with the proposed AMR procedures for ability to capture the propagation of strong shock waves. In a \( 2.4 \times 2.4 \times 2.4 \) cube with a \( 20 \times 20 \times 20 \) base Cartesian grid, constant density \( \rho_0 \) is set to be equal 1 at initial time moment \( t = 0 \). Inside a sphere with a radius of 0.1 in the center of the cube the internal energy \( E_0 \) at \( t = 0 \) is chosen so that at \( t = 1 \) the shock wave position would be at a radius of 1. In the rest part of the cube pressure equals to \( 10^{-5} \) at \( t = 0 \). Initially the grid was refined up to 5 levels near the sphere of energy \( E_0 \), Fig. 8. The solver is executed up to \( t = 1 \) with CFL = 0.9. Coarsening and refinement procedures are performed each timestep, whereas the defragmentation is performed after every 40 timesteps. Two solutions are obtained with maximum 4 and 5 refinement levels, respectively. Coarsening and refinement thresholds \( \chi_{cr} \), \( \chi_{rf} \) equal to 0.02 and 0.1, respectively.

We use the following hardware and software: CUDA version is executed on Nvidia Tesla V100 (CUDA 10.0), OpenMP version is executed on dual-socket node equipped with two 16-cores Intel Xeon Gold 6142 v4 (Intel C++ compiler 18.0). Grid and pressure distribution are shown in Fig. 9, the comparison with analytical solution is presented in Fig. 10. As one can see, the criterion (12) slightly under-estimates refinement requirements on the diagonal directions but nevertheless, the shock wave is fairly well captured on the highest grid refinement level. As expected, the 5-level grid refinement solution is closer to the analytic solution comparing with the 4-level solution.

Execution times for OpenMP and CUDA versions are presented in Fig. 11. All Solver, Refinement criterion, AMR (includes Tune_refine_flags, Tune_coarse_flags, Coarsening, Refinement, Neighbors searching) and Defragmentation OpenMP procedures are performed with the same number of threads. One can see that the CPU version of the defragmentation algorithm doesn’t scale when the number of threads is increased (that was also mentioned in our previous
Figure 8. The initial grid and pressure distribution, $t = 0$.

Figure 9. The grid and pressure distribution, $t = 1$.

Figure 10. Pressure distribution at $t = 1$ for 4- and 5-level AMR.

Figure 11. Times for solver and AMR procedures on CPU and GPU (normalized per $10^6$ cells).

while other procedures improve their performance and even benefit from Hyper-threading capability (64 threads case). Tesla V100 is about an order faster than the dual-socket node equipped with Intel Xeon Gold 6142 v4 CPU. One could notice that Refinement criterion computing is a time-consuming process in all test cases and several times slower than other AMR procedures, so there is the reason to exploit different error estimation methods more suitable for GPU in future implementations. Although the defragmentation takes quite a long time, one needs to take into account that it is performed only every 40 iterations. Therefore this procedure consumes lower than 2% of the total runtime on Tesla V100. As the result, whole overhead for AMR support (including refinement, coarsening, 2:1 balancing, neighbor searching and defragmentation procedures) and refinement criterion takes 2% and 0.3% of the total runtime, respectively, on Tesla V100. One needs to mention that these results were obtained for the “worst” case when the grid modification was performed at each timestep/iteration.

5. Conclusions
Fully GPU-native AMR CFD solver implementation has been proposed. One of the main features of this implementation is that all dynamic grid modification procedures are performed entirely on GPU along with the solver itself thus avoiding pci-e grid metadata transfers between
CPU and GPU; this leads to better utilization of the GPU resources. The cell-level refinement on octree-based grids permits to improve local grid resolution only in regions of interest, more precisely compared to block- or patch-based grids. The tests performed with the proposed implementation have shown that the AMR procedures overhead takes only 2% of the total runtime on Nvidia Tesla V100. This result confirms the perspectivity of the proposed parallel algorithms and implementations. These algorithms are supposed to use in multi-GPU AMR solvers to be developed in our future work.

Acknowledgments
This research was supported by the Grant No 17-71-30014 from the Russian Science Foundation.

References
[1] Burstedde C, Wilcox L C and Ghattas O 2011 P4est: scalable algorithms for parallel adaptive mesh refinement on forests of octrees {SIAM J. Sci. Comput.} 33 3 1103-33
[2] Tiankai T, O’Hallaron D R and Ghattas O 2005 Scalable parallel octree meshing for terascale applications {SC ’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing} pp 4-4
[3] Wahib M, Maruyama N and Aoki T 2016 Daino: a high-level framework for parallel and efficient AMR on GPUs {SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis} 621-32
[4] Schive H, ZuHone J A, Goldbaum N J, Turk M J, Gaspari M and Cheng C 2018 Gamer-2: a GPU-accelerated adaptive mesh refinement code – accuracy, performance, and scalability {Monthly Notices of the Royal Astronomical Society} 481 4 4815-40
[5] Ji H, Lien F and Zhang F 2015 A GPU-accelerated adaptive mesh refinement for immersed boundary methods {Computers & Fluids} 118 131-47
[6] Menshov I and Kornev M 2014 Free-boundary method for the numerical solution of gas-dynamic equations in domains with varying geometry {Math Models Comput Simul} 6 612-21
[7] Guzik S M and Riley J 2018 Adaptive mesh refinement on parallel heterogeneous (CPU/GPU) architectures {AIAA Aerospace Sciences Meeting} doi:10.2514/6.2018-1831
[8] Asuncion M and Castro M J 2017 Simulation of tsunamis generated by landslides using adaptive mesh refinement on GPU {Journal of Computational Physics} 345 91-110
[9] Beckingsale D, Gaudin W, Herdman A and Jarvis S. 2015 Resident Block-Structured Adaptive Mesh Refinement on Thousands of Graphics Processing Units {44th International Conference on Parallel Processing} doi: 10.1109/ICPP.2015.15 61-70
[10] Saetra M L, Brodtkorb A R and Lie K 2015 Efficient GPU-implementation of adaptive mesh refinement for the shallow-water equations {J Sci Comput} 63 23-48
[11] Alladef A, Leon S E, Celes W. and Paulino G H 2016 Massively parallel adaptive mesh refinement and coarsening for dynamic fracture simulations {Engineering with Computers} 32 533-52
[12] Giuliani A and Krivodonova L 2019 Adaptive mesh refinement on graphics processing units for applications in gas dynamics {Journal of Computational Physics} 381 67-90
[13] Godunov S K, Zabrodin A V, Ivanov M V, Kraiko A N and Prokopov G P 1976 {Numerical solution of multidimensional problems of gas dynamics} (Moscow: Nauka)
[14] Lohner R 1987 An adaptive finite element scheme for transient problems in CFD {Computer methods in applied mechanics and engineering} 61 3 323-38
[15] Dumbser M, Zanotti O, Hidalgo A and Balsara D S 2013 ADER-WENO finite volume schemes with space-time adaptive mesh refinement {Journal of Computational Physics} 248 257-86
[16] Pavlukhin P and Menshov I 2019 GPU-Aware AMR on Octree-Based Grids {Parallel Computing Technologies. PaCT 2019. Lecture Notes in Computer Science} vol 11657, ed Malysikin V. (Cham: Springer)
[17] Sedov L 1959 {Similarity and dimensional methods in mechanics, acod} (NewYork: Press)