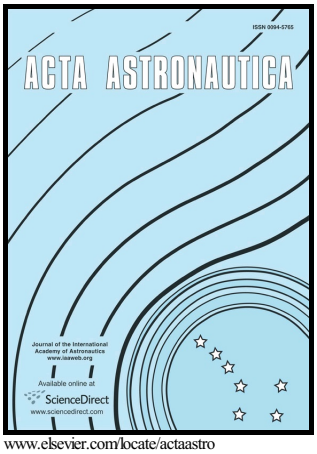
Analysis of impact of general-purpose graphics processor units in supersonic flow modeling

V.N. Emelyanov, A.G. Karpenko, A.S. Kozelkov, I.V. Teterina, K.N. Volkov, A.V. Yalozo



www.elsevier.com/locate/actaastro

PII: S0094-5765(16)30822-0
DOI: http://dx.doi.org/10.1016/j.actaastro.2016.10.039
Reference: AA6061

To appear in: *Acta Astronautica*

Received date: 18 August 2016
Revised date: 22 October 2016
Accepted date: 25 October 2016

Cite this article as: V.N. Emelyanov, A.G. Karpenko, A.S. Kozelkov, I.V. Teterina, K.N. Volkov and A.V. Yalozo, Analysis of impact of general-purpose graphics processor units in supersonic flow modeling, *Acta Astronautica* http://dx.doi.org/10.1016/j.actaastro.2016.10.039

Analysis of impact of general-purpose graphics processor units in supersonic flow modeling

V.N. Emelyanov[1], A.G. Karpenko[2], A.S. Kozelkov[3], I.V. Teterina[1], K.N. Volkov[4], A.V. Yalozo[3]

[1]Faculty of Rocket and Space Engineering, Baltic State Technical University, St. Petersburg, 190005, Russia
[2]Faculty of Mathematics and Mechanics, St Petersburg State University, Old Petergof, St. Petersburg, 198504, Russia
[3]Russian Research Institute of Experimental Physics, Russian Federal Nuclear Center, Sarov, 607188, Russia
[4]Faculty of Science, Engineering and Computing, Kingston University, London, SW15 3DW, United Kingdom

Abstract
Computational methods are widely used in prediction of complex flowfields associated with off-normal situations in aerospace engineering. Modern graphics processing units (GPU) provide architectures and new programming models that enable to harness their large processing power and to design computational fluid dynamics (CFD) simulations at both high performance and low cost. Possibilities of the use of GPUs for the simulation of external and internal flows on unstructured meshes are discussed. The finite volume method is applied to solve three-dimensional unsteady compressible Euler and Navier–Stokes equations on unstructured meshes with high resolution numerical schemes. CUDA technology is used for programming implementation of parallel computational algorithms. Solutions of some benchmark test cases on GPUs are reported, and the results computed are compared with experimental and computational data. Approaches to optimization of the CFD code related to the use of different types of memory are considered. Speedup of solution on GPUs with respect to the solution on central processor unit (CPU) is compared. Performance measurements show that numerical schemes developed achieve 20 to 50 speedup on GPU hardware compared to CPU reference implementation. The results obtained provide promising perspective for designing a GPU-based software framework for applications in CFD.

Keywords
Supersonic flow; Shock tube; Boundary layer; CFD; High-performance computing; Parallel algorithm; Speedup

1 Introduction

Propulsion power engines play an important role in determining space flight safety issues [1]. Modeling of fluid chemically reacting flows and heat transfer in rocket engines is necessary for adequate prediction of the functional efficiency and reliability of rocket engines [2–5] and nozzles [6]. It was demonstrated tha graphic processor units (GPU) could accelerate solution of these problems [7, 8]. New generation of propulsion engines wjould also, definitely, need effective mathematical simulations [9, 10]. The present paper discusses the effectiveness of GPU for fluid dynamics simulations relevant to space flight safety.

The methods of computational fluid dynamics (CFD) are extensively applied in design and

optimization of rocket techniques to get more insight into 3D unsteady flows through fluid or gas passages. Accurate prediction of compressible flows still remains a challenging task despite a lot of work in this area. The quality of CFD calculations of the flows strongly depends on the proper prediction of flow physics (shock waves, rarefaction waves, recirculation regions). Investigations of heat transfer, skin friction, secondary flows, flow separation and re-attachment effects demand reliable numerical methods, accurate programming, and robust working practices.

The stagnation in the clock-speed of central processing units (CPU) has led to significant interest in parallel architectures that offer increasing computational power by using many separate processing units. Modern graphics hardware contains such an architecture in the form of the graphics processing units (GPU). GPU platforms including GPU clusters make it possible to achieve speedups of an order of magnitude over a standard CPU in many CFD applications and are growing in popularity [11].

Figure 1 shows that a recent GPU is significantly more powerful than its CPU contemporary, and that the computing power of GPUs are increasing at a greater rate than that of CPUs. The GPU employs a parallel architecture so each generation improves on the speed of previous ones by adding more cores, subject to the limits of space, heat and cost. CPUs, on the other hand, have traditionally used a serial design with a single core, relying instead on greater clock speeds and shrinking transistors to drive more powerful processors. While this approach has been reliable in the past, it is now showing signs of stagnation as the limit of current manufacturing technology is being reached. Recent CPUs, therefore, tend to feature two or more cores, but GPUs still enjoy a significant advantage in this area [12].

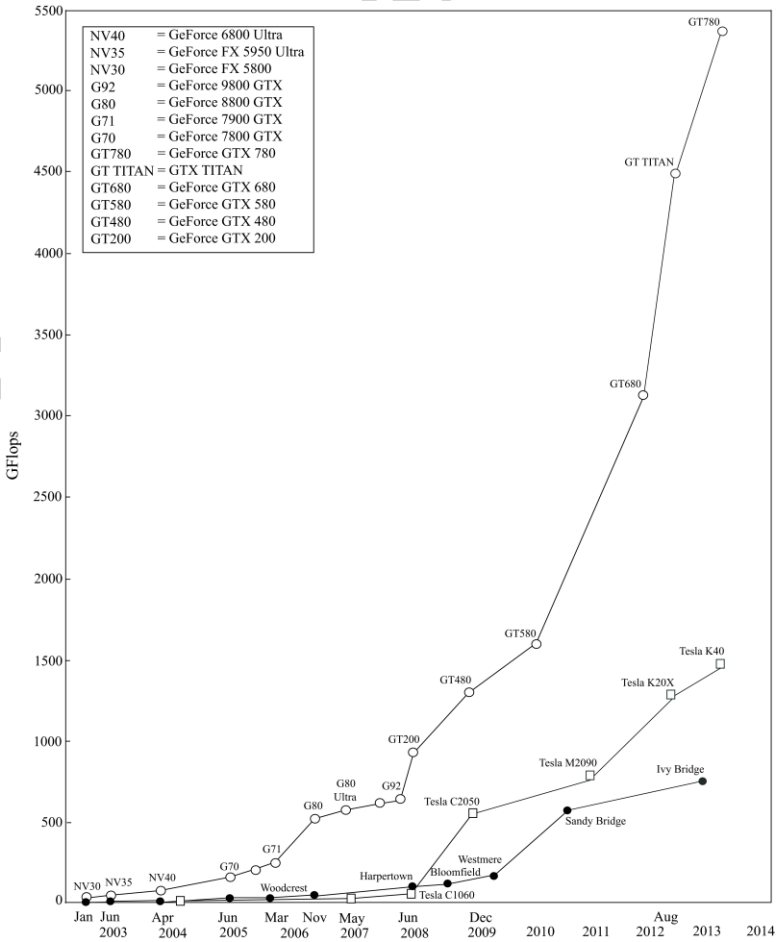


Figure 1. Floating point operations per second for the CPUs and GPUs

Speed and accuracy are key factors in the evaluation of CFD solver performance. In CFD applications, the increasing demands for accuracy and simulation capabilities produce an exponential growth of the required computational resources. High performance computing (HPC) resources are widely used in engineering applications.

The use of GPUs is a cost effective way of improving substantially the performance in CFD applications [13]. Taking advantage of any multi-core architecture requires programs to be written for parallel execution. For CFD, this has traditionally meant splitting the flow domain into several parts (domain decomposition) that are solved independently on each processor node in a cluster, with the flow properties at boundaries being communicated between the nodes after each time step (processor balancing). This is also the process adopted for GPUs, but the GPU introduces several additional constraints that make the stream programming paradigm particularly useful [14].

Although GPU has attractive characteristics for massively parallel computations, it has not been implemented in CFD for a long time due to the complex programming techniques. Developers must have special knowledge about computer graphics which is unfamiliar for general CFD researchers. But thanks to the CUDA (Compute Unified Device Architecture) library provided by NVIDIA, researchers are free from the restrictions of computer hardware knowledge and need to concentrate on CFD algorithms and CUDA programming language.

Depending on the complexity of the CFD problem to represent and solve, structured or unstructured meshes are used. Computational algorithms are more efficiently implemented on structured meshes, and data structures to handle the mesh are easy to implement [15, 16]. However, structured meshes present poor accuracy if the problem to be solved has complex internal or external boundaries. On the other hand, unstructured meshes present more flexibility and higher accuracy to represent problems that have complex geometries and boundaries [17]. However, the data structures to handle it are not easy to implement, and also explicit neighboring information should be stored.

Much of the efforts in running CFD codes on GPUs has been directed toward the case of CFD solvers based on structured and block-structured meshes [14, 18–23]. These solvers are easily to implement on GPUs due to their regular memory access pattern. There are various examples of implementation of CFD solvers on structured meshes for simulation of flows of viscous incompressible fluid [24–26].

Unstructured mesh based analysis methods on HPC systems with shared memory and distributed memory have been largely studied. However, shared and distributed memory systems are fundamentally different from GPUs. A GPU is a SIMT (Single Instruction Multiple Thread) engine, whereas shared and distributed memory systems are MPMD (Multiple Program Multiple Data) engines. However, the common aspect of these parallel engines is that in both of them the mesh application is limited by memory latency. Achieving good performance for unstructured mesh based CFD solvers on GPUs is more difficult due to their data dependent and irregular memory access patterns [27–29].

Explicit time-marching algorithms are the most convenient ones to be ported on to the GPUs. This is because there is no iteration, and the new value of a variable depends only on the previous time values. Hence, the update of a given variable is done independently on variables being updated on other threads. There is no recursive relation between the variables on the threads, since they are all known at the previous time step. However, even for explicit algorithms, a few changes are needed for efficiently

implementation of numerical algorithms on the GPU [12]. These relate to the use of shared memory and the layout of data structures. Memory coalescing and block size influence the speed achieved. The data should be organized such that adjacent threads access adjacent nodal data. In addition, data should be, where possible, copied to shared memory and re-used as much as possible. Therefore, even explicit algorithm based CFD codes need to be reorganized to take advantage of the GPU architecture.

When an implicit algorithm is used, the efficiency as well as the convergence are impacted. Implicit algorithms directly ported to a GPU are not usually work because of the mixed implicit and explicit updates. It is necessary to remove any recursive updates, so the algorithm could be run on parallel threads.

The most of the work done so far has either been for relatively small codes written from scratch or for a small portion of a large existing code. However, GPU support is available in mathematical packages (MATLAB) and commercial CFD solvers (ANSYS CFX, ANSYS Fluent).

In most cases, time is a precious parameter in space flight safety or post-event technical expertise, engineers having to deliver results with maximum accuracy in a shortest time possible. These performance gains can only be achieved using High Performance Computing (HPC) facilities. This paper aims to highlight the benefits of parallel processing (mainly of GPUs) in the case of space flight modeling.

The present work is undertaken as a part of a larger effort to establish a common CFD code for simulation of flows in aerospace and mechanical applications, and involves some basic validation studies. Up to now, a few researches on fully 3D compressible Navier–Stokes GPU solver for engineering applications have been reported. The motivation of this paper is to assess the in-house compressible CFD code, and to demonstrate successful design of a highly parallel computation system based on GPUs and validate the speedup factor compared with CPU.

The governing equations are solved with finite volume code and high resolution schemes on hybrid meshes. The code is programmed following the standard of CUDA C language. Single precision arithmetic is kept through the entire residual computations with the help of latest GPU hardware and careful design of CFD code. The benchmark test cases include Sod shock tube problems, flat plate boundary layer problem, compressible flow over NACA0012 and RAE2822 airfoils.The results obtained are generally in reasonable agreement with the available experimental and computational data reported in literature. The parallelization methods are studied and speedup factor by GPU cards is measured.

2 Governing equations

In Cartesian coordinates (x, y, z), an unsteady 3D flow is described by the following equation written in conservative form

$$\frac{\partial Q}{\partial t} + \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z} = 0. \tag{1}$$

The pressure is calculated as

$$p = (\gamma - 1)\rho \left[e - \frac{1}{2}\left(v_x^2 + v_y^2 + v_z^2\right) \right].$$

The flow variables vector, Q, and the flux vectors, F_x,F_y and F_z, have the form

$$Q = \begin{pmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho v_z \\ \rho e \end{pmatrix},$$

$$F_x = \begin{pmatrix} \rho v_x \\ \rho v_x v_x + p - \tau_{xx} \\ \rho v_x v_y - \tau_{xy} \\ \rho v_x v_z - \tau_{xz} \\ (\rho e + p) v_x - v_x \tau_{xx} - v_y \tau_{xy} - v_z \tau_{xz} + q_x \end{pmatrix},$$

$$F_y = \begin{pmatrix} \rho v_y \\ \rho v_y v_x - \tau_{yx} \\ \rho v_y v_y + p - \tau_{yy} \\ \rho v_y v_z - \tau_{yz} \\ (\rho e + p) v_y - v_x \tau_{yx} - v_y \tau_{yy} - v_z \tau_{yz} + q_y \end{pmatrix},$$

$$F_z = \begin{pmatrix} \rho v_z \\ \rho v_z v_x - \tau_{zx} \\ \rho v_z v_y - \tau_{zy} \\ \rho v_z v_z + p - \tau_{zz} \\ (\rho e + p) v_z - v_x \tau_{zx} - v_y \tau_{zy} - v_z \tau_{zz} + q_z \end{pmatrix}.$$

The components of viscous stress tensor and components of heat flux vector by conduction are found as

$$\tau_{ij} = \mu \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3} \frac{\partial v_k}{\partial x_k} \delta_{ij} \right), \quad q_i = -\lambda \frac{\partial T}{\partial x_i}.$$

Here, t is the time, ρ is the density, v_x, v_y, and v_z are the velocity components in the coordinate directions x, y, and z respectively, p is the pressure, e is the total energy per unit mass, T is the temperature, and γ is the ratio of specific heat capacities.

The Sutherland's law is used to obtain molecular viscosity as a function of temperature

$$\frac{\mu}{\mu_*} = \left(\frac{T}{T_*} \right)^{3/2} \frac{T_* + S_0}{T + S_0},$$

where $\mu_* = 1.68 \times 10^{-5}$ kg/(m s), $T_* = 273$ K and $S_0 = 110.5$ K for air. The thermal conductivity is expressed in terms of viscosity and Prandtl number as $\lambda = c_p \mu / \mathrm{Pr}$, where c_p is the specific heat capacity at constant pressure, and the molecular Prandtl number is $\mathrm{Pr} = 0.72$ for air.

3 Numerical method

The governing equations solved by the CFD code are of the form

$$\frac{dQ}{dt} = R(Q), \tag{2}$$

where Q is the flow variables vector averaged over the control volume. The flow residual is

$$R(Q) = S(Q) - L(Q),$$

where $L(Q)$ denotes all the spatial differencing terms, and $S(Q)$ denotes terms from boundary conditions and possible source terms.

Equation (2) is written in the form

$$\frac{dQ_i^n}{dt} + L(Q_i^n) = 0, \tag{3}$$

where $L(Q_i^n)$ is the differential operator. The subscript i refers to the control volume, and the superscript n refers to the time layer.

The three-step Runge–Kutta method is used for discretization of the equation (3) in time [30]

$$Q_i^{(1)} = Q_i^{(0)} + \Delta t L(Q_i^{(0)});$$
$$Q_i^{(2)} = \frac{3}{4}Q_i^{(0)} + \frac{1}{4}\left[Q_i^{(1)} + \Delta t L(Q_i^{(1)})\right];$$
$$Q_i^{(n+1)} = \frac{1}{3}Q_i^{(0)} + \frac{2}{3}\left[Q_i^{(2)} + \Delta t L(Q_i^{(2)})\right].$$

Here, $Q_i^0 = Q_i^{(n)}$ and $Q_i^{(3)} = Q_i^{(n+1)}$. An advantage of the Runge–Kutta method is that it ensures positiveness of the difference scheme. If the solution and the operator $L(Q)$ are positive at the time t^n, they also remain positive at the time t^{n+1}.

The inviscid flux is found from the relation

$$F(Q_L, Q_R) = \frac{1}{2}[F(Q_L) + F(Q_R) - |A|(Q_R - Q_L)].$$

where the subscripts L and R refer to cells on the left and on the right edges of the control volume. The matrix A is presented in the form $A = R\Lambda L$, where Λ is the diagonal matrix composed from the Jacobian eigenvalues, and R and L are the matrices composed from its right and left eigenvectors, respectively.

The unstructured CFD code developed uses an edge-based data structure to give the flexibility to run on meshes composed of a variety of cell types. The fluxes through the surface of a cell are calculated on the basis of flow variables at nodes at either end of an edge, and an area associated with that edge (edge weight). The edge weights are pre-computed and take into account geometry of the cell. Some details of the CFD code are provided in [31, 32].

The non-linear CFD solver works in an explicit time-marching fashion, based on a Runge–Kutta stepping procedure. The flux vector is split into the inviscid and viscous components. The governing equations are solved with upwind finite difference scheme for inviscid fluxes, and central difference scheme of the second order for viscous fluxes. For simulation of low-speed flows, convergence to a steady state is accelerated by the use of low-Mach number preconditioning method. The computational procedure involves reconstruction of the solution in each control volume and extrapolation of the unknowns to find the flow variables on the faces of control volume, solution of Riemann problem for each face of the control volume, and evolution of the time step. The Godunov exact Riemann solver and the Roe approximate Riemann solver [33] are used in calculations.

The computational procedure is implemented as a computer code in C/C++ programming language. Parallelization of the computational procedure is performed by a message passing interface (MPI). CUDA technology is used to implement GPU version of the code.

4 Programming model

CUDA is a parallel computing architecture from NVIDIA which introduced a new programming model based on high-level abstraction levels which avoid the former graphics pipeline concepts and

ease the porting of a scientific CPU application [12]. According to the CUDA framework, both the CPU and the GPU maintain their own memory. It is possible to copy data from CPU memory to GPU memory and vice versa.

4.1 Overview

Programming GPUs is unlike traditional CPU programming and massive parallel computers, because the hardware is different. It is often a relatively simple task to get started with GPU programming and get speedups over existing CPU codes, but these first attempts at GPU computing are often sub-optimal, and do not utilize the hardware to a satisfactory degree. Achieving a scalable high-performance code that uses hardware resources efficiently is still a difficult task.

The GPU is formed by a set of multiprocessors, each one having a number of processors depending on specific architecture. At any clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data. A function executed on the GPU is called a kernel. A kernel is executed by many threads which are organized forming a grid of thread blocks that run logically in parallel. All blocks and threads have spatial indices, so that the spatial position of each thread could be identified in the program. Each thread block runs in a single multiprocessor. A warp is the number of threads that run concurrently in a multiprocessor (warp size is 32 threads). Each block is split into warps, and periodically a scheduler switches from one warp to another. This allows to hide the high latency when accessing the GPU memory, since some threads continue their execution while other threads are waiting.

A GPU architecture implements different types of memory for storing data (global memory, constant memory, texture memory, shared memory and registers). This memory structure allows to reduce global memory accesses and collaboration among threads in the same thread block. In terms of latency, global memory access is the slowest whereas registers are the fastest. Since the GPU execution model requires that the information is first placed in global memory and then accessed by the GPU application, it is necessary to optimize global memory access. Global memory access is optimized by achieving peak bandwidth and by reducing the number of accesses.

Although GPU provides large bandwidth for global memory operation, the access pattern of the threads of a warp reduces the achieved bandwidth. To achieve peak bandwidth usage, the GPU coalesces warp memory operations into two or four memory transactions depending on the size of the words accessed. Therefore, warp memory access is organized in such a way that threads access adjacent memory locations. When data is reutilized, it is possible to reduce the number of global memory accesses by storing the data either in registers or in shared memory. Shared memory is common for all the threads in the thread block, which allows collaboration among them. Since shared memory is organized in banks, to avoid bank conflicts threads should access data in different banks.

The performance critical portion of the CFD solver consists of a loop which repeatedly computes the time derivatives of the conserved variables. The conserved variables are then updated using an explicit Runge–Kutta time-stepping procedure. The most expensive computation consists of accumulating flux contributions across each face when computing the time derivatives. Therefore, the performance of the CUDA kernel which implements this computation is crucial in determining whether or not high performance is achieved.

4.2 Memory access

The operations that are carried out in every iteration of the CFD solver are divided into three parts.

- Local cell analysis to obtain a coefficient for each solution point based only on the interaction with the other solution in the same cell.
- Neighbor cell analysis to compute a coefficient for each solution point based on the interaction with its neighbor solution point.
- Update local magnitudes when the local value of the magnitude at the solution point is updated using the two previously computed coefficients.

The three main stages perform computations based on information stored in main memory, such as the solution point variables, geometry information, and a set of parameters for cell-oriented or edge-oriented analysis. Although solution point variables and parameters are used in all three main stages, they are accessed with different patterns at every stage. These memory patterns limit data locality between and inside the stages, diminishing efficiency of data caches for reducing memory latency.

In cell-oriented analysis, a set of coefficients for each solution point is computed based on its own information as well as the information of the solution points that belong to the same cell. The solution point information is performed in two steps. The first step involves retrieving the pointer to the beginning of the cell in the array of solution point variables, and the second step involves accessing sequentially all the information in the current cell.

In edge-oriented analysis, a set of coefficients for each solution point is computed based on its own information and the information of its neighbor solution point. Unlike cell-oriented analysis that traverses the mesh at cell level, edge-oriented analysis traverses the mesh at edge level. Accessing the solution point information is done in three steps. The first step involves retrieving the pointer to the solution point, the second step includes retrieving the pointer to the left and right solution point variables, and the third step involves accessing the two solution points variables. In the Riemann solver, left and right solution point variables are not physically adjacent, and information is read and used only once, hence, either on a uni-threaded or multi-threaded solution the cache memories do not help to reduce memory latency.

In the last stage, the solution point variables are updated utilizing only current solution point information and coefficients (read and utilized once). Since coefficients and solution point variables arrays are processed sequentially, cache memories take advantage of spatial locality, and by this way help to reduce memory latency for both uni-threaded and multi-threaded solutions.

4.3 Advanced possibilities

The time derivative computations are parallelized on a per-cell basis, with one thread per cell [28]. First, each thread reads the cell volume, along with its conserved variables from global memory, from which it derives physical quantities such as the pressure, velocity, total energy, and the flux contributions are computed. The kernel then loops over each of all faces of the control volume in order to accumulate fluxes. The face normal is read along with the index of the adjacent cell, where this index is then used to access the adjacent cell's conserved variables. The required derived quantities are computed and then the flux is accumulated into the cell residual.

This approach requires redundant computation of flux contributions, and other quantities derived from the conserved variables. Another possible approach is to first pre-compute each cell's flux

contribution, thus avoiding such redundant computation. However, this approach turns out to be slower because reading the flux contributions requires three times the amount of global memory access than just reading the conserved variables. The redundant computation is performed simultaneously with global memory access, which hides the high latency of accessing global memory.

Shared memory is an important feature of GPU hardware used to avoid redundant global memory access amongst threads within a block. The hardware does not automatically make use of shared memory, and it is up to the software to explicitly specify how shared memory is used. Information is made available which specifies which global memory access is shared by multiple threads within a block. For structured mesh based solvers, this information is known a priori due to the fixed memory access pattern of such solvers. On the other hand, the memory access pattern of unstructured mesh based CFD solvers is data dependent.

In the case of an unstructured mesh, the global memory access required for reading the conserved variables of neighboring control volumes is at risk of being highly non-coalesced, which results in lower effective memory bandwidth. This is avoided, however, if neighboring faces and edges of consecutive cells are nearby in memory. This is achieved in two steps. The first step is to ensure that cells nearby in space are nearby in memory by using a renumbering scheme [28]. The scheme works by overlaying a mesh of bins. Each point in the mesh is assigned to a bin, and then the points are renumbered by assigning numbers while traversing the bins in a fixed order. With such a numbering in place, the connectivity of each cell is then sorted locally on the second step, so that the indices of the four neighbors of each tetrahedral cell (for triangular mesh) are in increasing order. This ensures that, for example, the second neighbor of consecutive cells are close in memory.

5 Parallelization technique

The finite volume mesh is generated from input data with the appropriate setting of initial and boundary conditions. The time stepping is performed by applying a Runge–Kutta TVD method.

The computation steps required by the problem considered are classified into two groups, computations associated to faces and edges, and computations associated to volumes. The numerical scheme exhibits a high degree of data parallelism because the computation at each edge/volume is independent with respect to the computation performed at the rest of edges/volumes. Moreover, the explicit scheme presents a high arithmetic intensity and the computation exhibits a high degree of locality.

Solution scheme with the use of GPU resources is shown in the Figure 2. Single arrows correspond to the commands, and double arrows correspond to commands and data transfers between CPU and GPU (global memory is used).

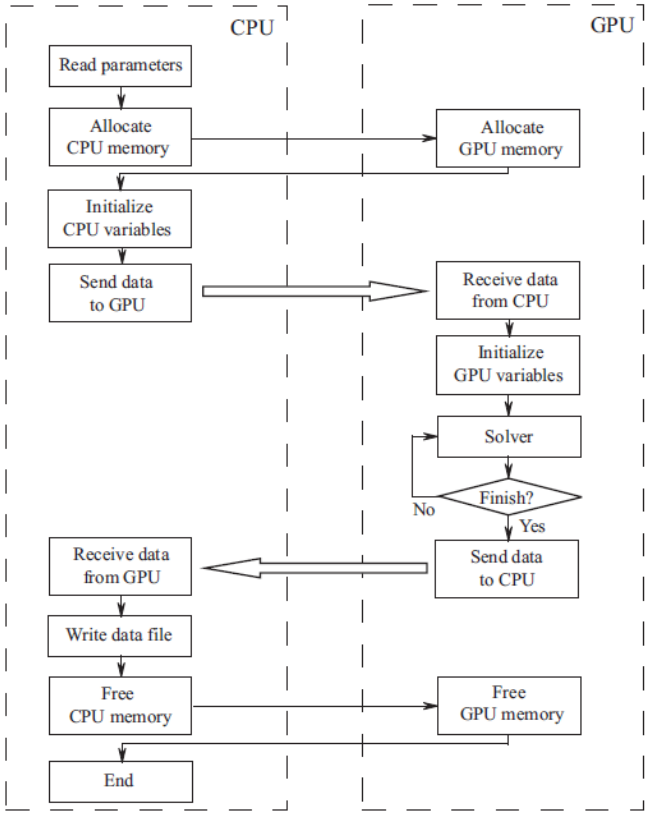


Figure 2. Solution of CFD problem with the use of GPU resources

The implementation is split between CPU and GPU. Pre- and post-processing steps are done on the CPU, leaving only the computation itself to be performed on the GPU. For example, the CPU constructs the mesh and evaluates the face areas, face normals and cell volumes. The initialization of the flowfield is also done on the CPU. Each time step of the computation then involves a series of kernels on the GPU which evaluate the cell face fluxes, sum the fluxes into the cell, calculate the change in properties at each node, smooth the variables and apply boundary conditions. Each kernel operates on all the nodes (no distinction is made between boundary nodes and interior nodes). This causes difficulties if an efficient code is to be obtained. For example, the change in a flow property at a node is formed by averaging the flux sums of the adjacent cells (for mesh with quadrangle cells, four cells surround an interior node, but only two at a boundary node). This problem is overcome using dependent texturing. The indices of the cells required to update a node are pre-computed on the CPU and loaded into GPU texture memory. For a given node, the kernel obtains the indices required and then looks up the relevant flux sums which are stored in a separate GPU texture. This avoids branching within the kernel.

A graphical description of the parallel computational algorithm, obtained from the mathematical description of the numerical scheme, is shown in the Figure 3. The main calculation stages are identified and the main sources of data parallelism are represented indicating that the calculation affected by it are performed simultaneously for each data item of a set (the data items represent the volumes or faces/edges of the finite volume mesh). Time stepping process is repeated until the final simulation time is reached. At the $(n+1)$-th time step, the residual is evaluated to update the state of each cell. In order to add the contributions associated with each edge, two variables are used in the algorithm for each volume. The first variable is used to store the contributions to the local time step size of the volume, and the second variable is used to store the sum of the contributions to the state of

cell.

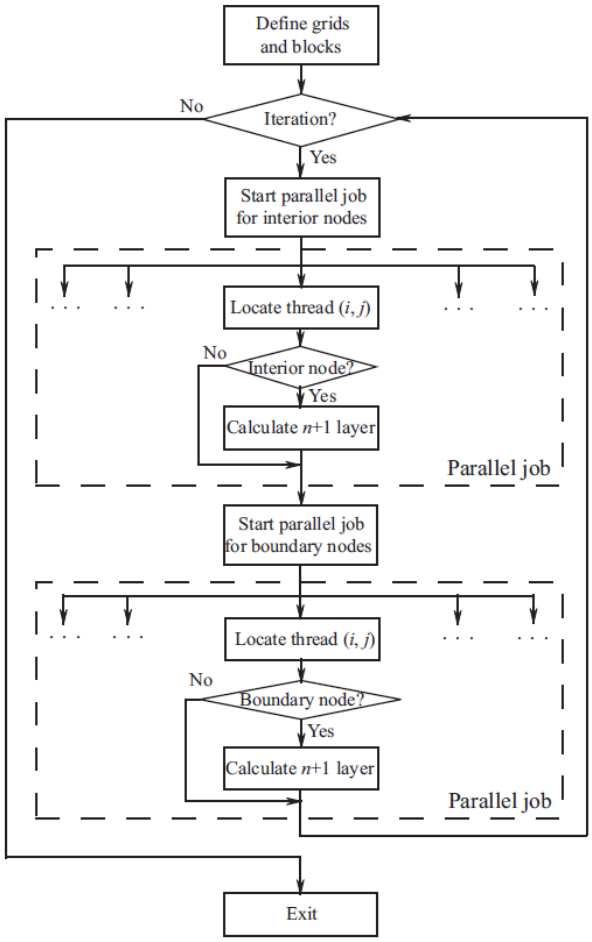


Figure 3. Main calculation stages in the parallel algorithm

The most costly stage in the algorithm is edge-based calculations involving two calculations for each face communicating two cells. This contribution is computed independently for each face and is added to the partial sums associated to each cell. For each control volume, the local time step is computed. The computation for each volume does not depend on the computation for the rest of volumes and therefore this stage is performed in parallel. The minimum of all the local time steps previously obtained for each volume is computed. The $(n+1)$-th state of each control volume is approximated from the n-th state using the data computed in the previous phases. This stage is also completed in parallel.

6 Flux calculations

The implementation of the finite volume method using a global memory and register file is illustrated in the Figure 4. Each time layer calculations are performed in two stages. Two kernels are used for the parallel implementation of the finite volume method on GPU, one of which calculates the flow through the faces of control volumes (stage 1), and the other one provides flow variable calculations on the next time layer (stage 2). On the first stage, flow variables in the centers of control volumes are stored in global memory (array Q). One thread is used to calculate the fluxes through the faces of control volume. Each thread uses the flow variables vector in adjacent control volumes, i and $i+1$. Fluxes through cell faces are stored in array F. On the second stage, a set of threads corresponding to

the same number of control volumes is launched to calculate the flow variables vector on a new time level. The fluxes through the faces $i-1/2$ and $i+1/2$ are used, and the solution is computed in the control volume i. The solution is then stored in the array Q.

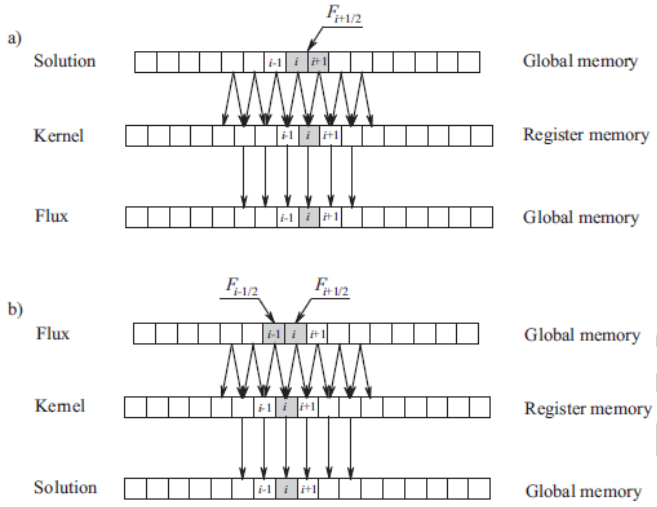


Figure 4. Flux calculation (*a*) and calculation of flow variables vector on a new time layer (*b*)

The use of shared memory in the calculation of flow variables vector is presented in the Figure 5, which shows how to copy the data from global memory to shared memory. For example, the implementation of upwind numerical scheme requires the use of three control volumes to calculate fluxes and limiters. On step 1, flow variables vector corresponding to the centered location is copied (fragment *a*), and on steps 2 and 3 flow variables vectors corresponding to the left and right locations are copied (fragments *b* and *c*). Each thread makes treatment of the three flow variables vectors stored in the shared memory (fragment *d*).

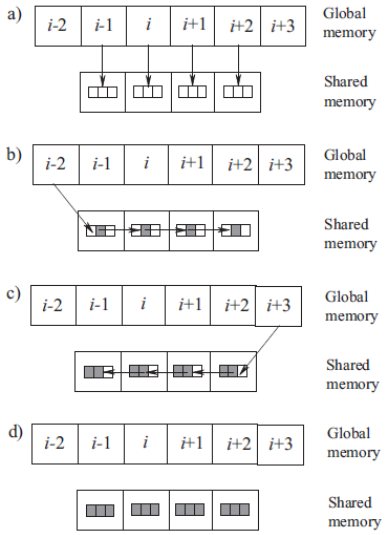


Figure 5. Use of shared memory in flux calculations

7 Results and discussion

The GPU version of the CFD code is used and validated for a variety of benchmark test cases. Numerical calculations are performed with unstructured in-house finite volume CFD code. An

equivalent solver is made in C++ to be run in a CPU for benchmarking purposes.

7.1 Sod problem

The Sod problem constitutes a particularly interesting and difficult test case, since it presents an exact solution to the full system of 1D Euler equations containing simultaneously a shock wave, a contact discontinuity, and an expansion fan [33]. The analogous 2D steady expansion wave and its interaction is discussed in [34, 35]. This problem is chosen to validate the numerical schemes and assess the temporal accuracy of the numerical solution obtained by the present method, since an analytical solution exists. The initial conditions in the present computation are as follows: $\rho_L = 1$, $u_L = 0$, $p_L = 1$ if $0 \le x \le 0.5$ (left state), and $\rho_R = 0.125$, $u_R = 0$, $p_R = 1$ if $0.5 < x \le 1$ (right state).

Calculations are performed on various meshes. A number of cells increases from 1024 cells for mesh 1 to 30720 cells for mesh 2, and to 307200 cells for mesh 3. The finest mesh, mesh 4, contains about three million cells. The time step is 1.52×10^{-5} s, and the total calculation time is 7.63×10^{-3} s. Courant number is equal to 0.85. Calculations are performed on one module of Tesla S1070 platform with 1.44 GHz (number of cores is 256), and on a single core of CPU AMD Phenom 2 with 3 GHz.

Distributions of flow quantities are presented in the Figure 6 ($t = 0.2$). Solid line corresponds to the exact solution of the Sod problem, and symbols ∘ correspond to the numerical solution.

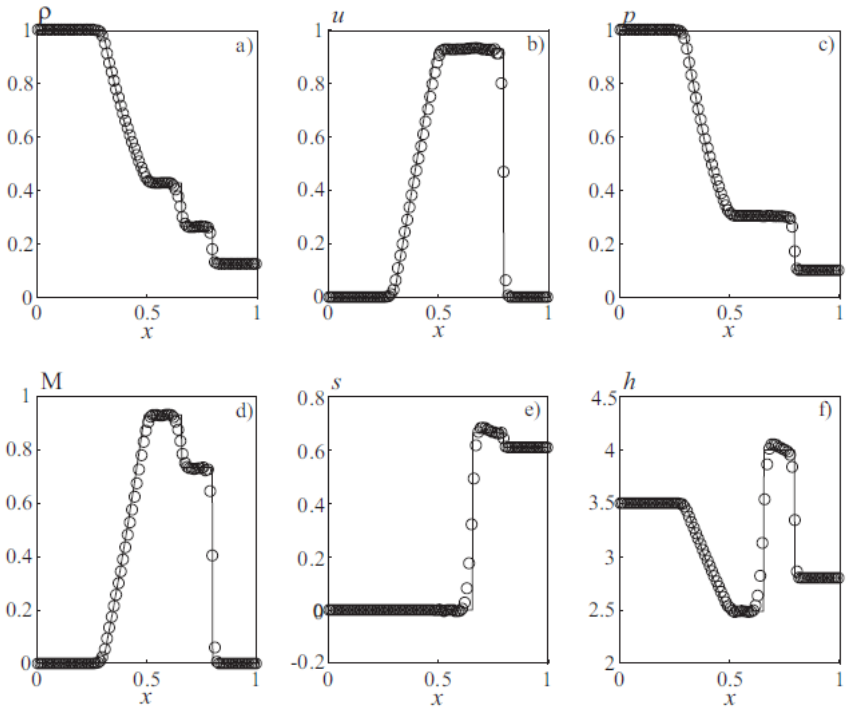


Figure 6. Solution of Sod problem: density (*a*), velocity (*b*), pressure (*c*), Mach number (*d*), entropy (*e*), enthalpy (*f*)

The time required for calculation of one time step, and speedup of calculations are given in the Table 1 (time is given in milliseconds). Option 1 corresponds to Godunov scheme involving exact solution of Riemann problem, and option 2 corresponds to Roe scheme involving approximate solution of Riemann problem. For both options, a good growth of speedup, S, is observed. However, Godunov method is not ideal from the parallelization point of view, since the exact solution of the Riemann

problem involves a large number of data transfers, reducing the GPU performance. Convergence speed of Newton iterative solver varies from one control volume to another one.

Table 1. Time (in ms) and speedup for Sod problem

No	Mesh 1			Mesh 2		
	CPU	GPU	S	CPU	GPU	S
1	1.63	0.13	12.43	47.70	0.20	245.25
2	0.14	0.07	1.87	5.51	0.17	33.17
No	Mesh 3			Mesh 4		
	CPU	GPU	S	CPU	GPU	S
1	460.64	0.92	502.50	4627.61	8.06	574.39
2	43.58	0.57	76.00	436.09	5.22	83.48

7.2 Shock tube problem

The shock tube test case considers a long tube containing a gas separated by a thin membrane. The gas is assumed to be at rest on both sides of the membrane, but it has different constant pressures and densities on each side. At time $t = 0$, the membrane is ruptured, and the problem is to determine ensuing motion of the gas. The solution of this problem consists of a shock wave moving into the low pressure region, a rarefaction wave that expands into the high pressure region, and a contact discontinuity which represents the interface.

Unstructured tetrahedral mesh is used to solve 3D shock tube problem. The length of the computational domain is $L = 10$ m. Initial states correspond to the Sod problem (the membrane is located at $x/L = 0.4$). Calculations are based on different meshes. The coarsest mesh contains about 10^4 cells (mesh 1), and the finest mesh contains about 10^7 cells (mesh 4). The intermediate meshes contain 10^5 cells (mesh 2) and 10^6 (mesh 3) cells. Typical mesh is shown in the Figure 7 (cross section). The time step is 1.52×10^{-5} s, and the total computational time is 7.63×10^{-3} s. Courant number is equal to 0.85. The calculations are performed on one module of Tesla S1070 platform with 1.44 GHz (a number of cores is 256), and one core of CPU AMD Phenom 2 with 3 GHz.

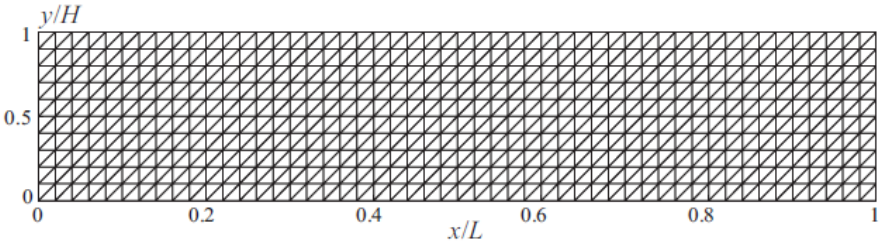


Figure 7. Unstructured mesh extruded in spanwise direction

The numerical results, shown in the Figure 8, indicate higher resolved solutions for a given time step and given mesh size than the numerical results reported in [36]. The results computed have no spurious oscillations at any shock or contact discontinuities.

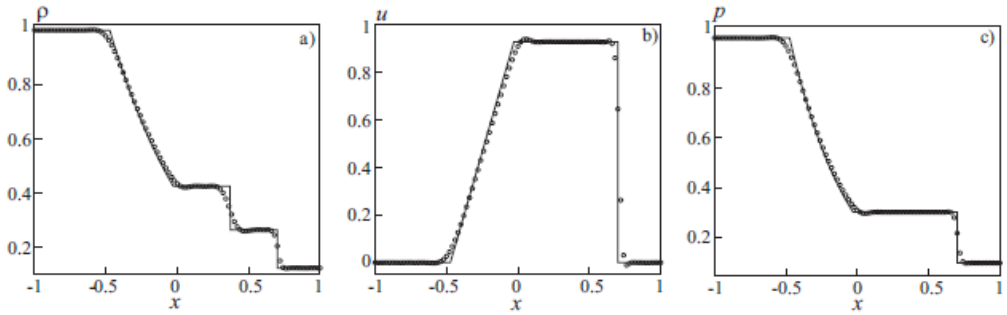


Figure 8. Solution of shock tube problem: density (*a*), velocity (*b*), pressure (*c*)

Speedup of calculations are presented in the Figure 9 (time of calculation of 1000 time steps was measured). Three indices are used to specify computational option. The first index corresponds to the solution of Euler equations (option 1, inviscid flow) or to the solution of Navier–Stokes equations (option 2, viscous flow). The second index corresponds to the time-marching scheme used in calculations based on one-step (option A) or three-step (option B) Runge–Kutta time-stepping procedure. The third index corresponds to the exact Godunov (option 1) or approximate Roe option 2) Riemann solvers. The calculations based on the finest mesh containing about 10 millions of cells (mesh 4) with Godunov scheme give speedup of 42. For the solution of viscous problem with the scheme of the second order, the speedup drops to 22.

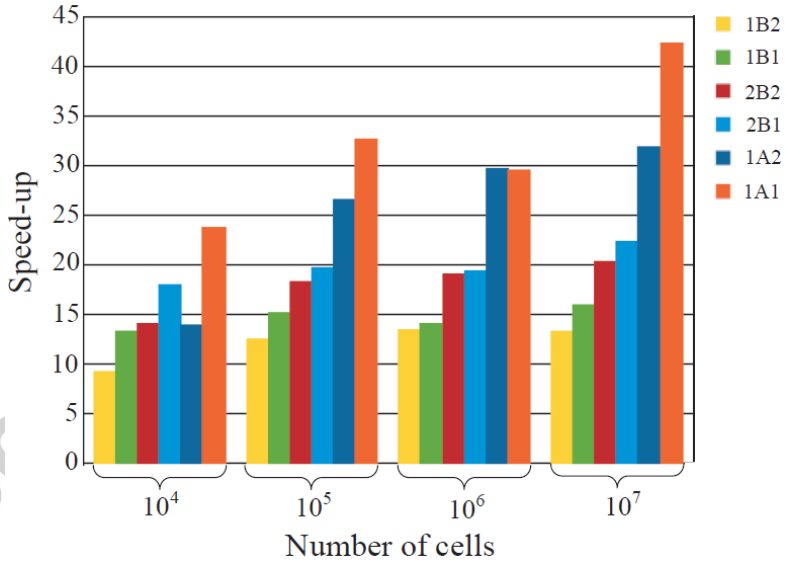


Figure 9. Speedup for shock tube problem

The time required for calculation of 1000 time steps on the mesh with 10^7 cells, and memory usage are given in the Table 2. The option 1 corresponds to GPU parallel calculations based on Godunov scheme, and the option 2 corresponds to CPU calculations based on Godunov scheme.

Table 2. Time and memory for shock tube problem

No	1	2	S/M
Memory, Mb	2582.28	2696.72	1.04
Time, s	305.29	14916.60	48.86

7.3 Flat plate flow

The flow over a smooth flat plate is well-known CFD benchmark solution [38], and it is used for verification and validation of other CFD codes [39].

The length of the computational domain is $30L$ ($10L$ before the plate and $20L$ behind the plate), and the width of the computational domain is $20L$, where L is the length of the plate ($L = 1$ m). Free stream velocity ($U_\infty = 10$ m/s), static pressure ($p_\infty = 101325$ Pa) and static temperature ($T_\infty = 300$ K) are fixed on the inlet boundary. No-slip and no-penetration boundary conditions are used on the plate. The plate surface is adiabatic. Free outflow boundary conditions are applied to the outlet boundary. Slip boundary conditions are used on the far-stream boundary.

The flat plate boundary layer problem is solved on various meshes. The velocity profile in the boundary layer is shown in the Figure 10. The flow calculations are based on CPU Xeon X5670 2.93 GHz and one module of Tesla S2050 platform. The computational time in seconds and speedup of calculations are shown in the Table 3 for one iteration. Increasing a number of nodes from 10^5 to 10^7, speedup increases on 10%.

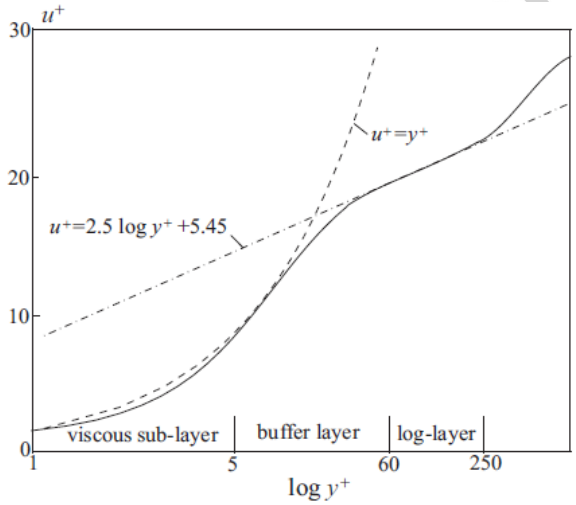


Figure 10. Velocity profile in the boundary layer

Table 3. Time and speedup for flat plate problem

Number of nodes	CPU	GPU	S
1.3×10^5	0.140	0.003	46.67
1.3×10^6	1.406	0.026	54.08
6.6×10^6	7.091	0.126	56.28
1.3×10^7	14.06	0.251	56.02

8 Conclusion

GPUs have evolved as a new paradigm for scientific computations. They are essentially multi-core machines with a large number of computational units sharing a common memory. GPUs cost/performance ratio, and low power consumption make them attractive for high-resolution CFD computations. However, in order to exploit the inherent architecture of the device, the numerical algorithm as well as data structures are carefully tailored to minimize the memory access and any recursive relations in the computational algorithm.

Possibilities of the use of GPUs in CFD calculations were discussed. The finite volume method was applied to solve full Euler and Navier–Stokes equations on unstructured meshes of various topology. CUDA technology was used for programming implementation of parallel computational algorithms. Solutions of some benchmark CFD problems on GPUs were presented, and approaches to optimization of the CFD code related to the use of different types of memory were discussed. Speedup of CFD calculations varied from 10 to 50 depending on the problem to be solved, computational procedures and computational resources. This makes GPUs attractive for computing industrial fluid flows and heat transfer. However, porting legacy codes automatically is not easy. Significant rewrite of the algorithm and the code is necessary. The time investment is worthwhile because multi-core architectures of one form or the other are going to be the necessary trend for high resolution and high performance computing.

The computational procedure was developed as a part of LOGOS multi-functional and multi-purpose CFD package designed in the Institute of Theoretical and Mathematical Physics of the Russian Federal Nuclear Center (Sarov, Russia). LOGOS package was widely used in mechanical engineering and aerospace applications.

Further work is focused on parallel implementation of implicit schemes and convergence acceleration techniques such as multigrid method and low-Mach preconditioning.

Acknowledgements

This work was partially supported by the Russian Foundation for Basic Research (project 16-01-00267 and project 16-38-60142). The author wishes to thank colleagues from the Russian Federal Nuclear Center (Sarov, Russia) for access to high performance computing resources and discussion of the computational results.

References

1. Betelin V.B., Smirnov N.N., Nikitin V.F. Supercomputer predictive modeling for ensuring space flight safety. ActaAstronautica, 2015, 109, 269–277.
2. Smirnov N.N.,Betelin V.B., Shagaliev R.M., Nikitin V.F., Belyakov I.M., Deryuguin Yu.N., Aksenov S.V., Korchazhkin D.A. Hydrogen fuel rocket engines simulation using LOGOS code.International Journal of Hydrogen Energy, 2014, 39, 10748–10756.
3. Smirnov .N., Nikitin V.F., Stamov L.I., Altoukhov D.I. Supercomputing simulations of detonation of hydrogen-air mixtures. International Journal of Hydrogen Energy, 2015, 40, 11059–11074.
4. Smirnov N.N., Betelin V.B., Nikitin V.F., Stamov L.I., Altoukhov D.I. Accumulation of errors in numerical simulations of chemically reacting gas dynamics. Acta Astronautica, 2015, 117, 338–355.
5. Smirnov N.N., Nikitin V.F. Modeling and simulation of hydrogen combustion in engines. International Journal of Hydrogen Energy, 2014, 39(2), 1122–1136.
6. Silnikov M.V., Chernyshov M.V., Uskov V.N. Two-dimensional over-expanded jet flow parameters in supersonic nozzle lip vicinity. Acta Astronautica, 2014, 97, 38–41.
7. Rybakin B.P., Stamov L.I., Egorova E.V. Accelerated solution of problems of combustion gas dynamics on GPUs. Computers & Fluids, 2014, 90, 164–171.
8. Rybakin B.P. Modeling of 3D problems of gas dynamics on multiprocessing computers and GPU.

Computers & Fluids, 2013, 80, 403–407.
9. Smirnov N.N., Betelin V.B., Nikitin V.F., Phylippov Yu.G., Jaye Koo. Detonation engine fed by acetylene–oxygen mixture. Acta Astronautica, 2014, 104, 134–146.
10. Smirnov N.N., Phylippov Yu.G., Nikitin V.F., Silnikov M.V. Modeling of combustion in engines fed by hydrogen. WSEAS Transactions on Fluid Mechanics, 2014, 9, 154–167.
11. Owens J.D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A.E., Purcell T.J. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 2007, 26(1), 80–113.
12. Sanders J., Kandrot E. CUDA by example: an introduction to general-purpose GPU programming. Boston, Pearson Education, 2011.
13. Jacobsen D.A., Senocak I. Multi-level parallelism for incompressible flow computations on GPU clusters. Parallel Computing, 2013, 39(1), 1–20.
14. Thibault J.C., Senocak I. CUDA implementation of a Navier–Stokes solver on multi-GPU desktop platforms for incompressible flows. AIAA Paper, 2009-758.
15. Fu L., Gao Z., Xu K., Xu F. A multi-block viscous flow solver based on GPU parallel methodology. Computers and Fluids, 2014, 95, 19–39.
16. Tuttafesta M., Colonna G., Pascazio G. Computing unsteady compressible flows using Roe's flux-difference splitting scheme on GPUs. Computer Physics Communications, 2013, 184(6), 1497–1510.
17. Mavriplis D.J. Unstructured mesh discretizations and solvers for computational aerodynamics. AIAA Paper, 2007-3955.
18. Scheidegger C.E., Comba J.L.D., da Cunha R.D. Practical CFD simulations on programmable graphics hardware using SMAC. Computer Graphics Forum, 2005, 24(4), 715–728.
19. Hagen T.R., Lie K.-A., Natvig J.R. Solving the Euler equations on graphics processing units. Lecture Notes in Computer Science, 2006, 3994, 220–227.
20. Brandvik T., Pullan G. An accelerated 3D Navier–Stokes solver for flows in turbomachines. ASME Paper, GT2009-60052.
21. Shinn A.F., Vanka S.P., Hwu W.W. Direct numerical simulation of turbulent flow in a square duct using a graphics processing unit (GPU). AIAA Paper, 2010-5029.
22. Kuo F.-A., Smith M.R., Hsieh C.-W., Chou C.-Y., Wu J.-S. GPU acceleration for general conservation equations and its application to several engineering problems. Computers and Fluids, 2011, 45(1), 147–154.
23. Fu L., Gao Z., Xu K., Xu F. A multi-block viscous flow solver based on GPU parallel methodology. Computers and Fluids, 2014, 95, 19–39.
24. Appleyard J., Drikakis D. Higher-order CFD and interface tracking methods on highly-parallel MPI and GPU systems. Computers and Fluids, 2011, 46(1), 101–105.
25. Meng J., Skadron K. A performance study for iterative stencil loops on GPUs with ghost zone optimizations. International Journal of Parallel Program, 2011, 39(1), 115–142.
26. Krotkiewski M., Dabrowski M. Efficient 3D stencil computations using CUDA. Parallel Computing, 2013, 39(10), 533–548.
27. Kampolis I.C., Trompoukis X.S., Asouti V.G., Giannakoglou K.C. CFD-based analysis and two-level aerodynamic optimization on graphics processing units. Computer Methods in Applied Mechanics and Engineering, 2010, 199(9–12), 712–722.
28. Corrigan A., Camelli F., Löhner R., Mut F. Semi-automatic porting of a large-scale Fortran CFD code to GPUs. International Journal for Numerical Methods in Fluids, 2011, 69(2), 314–331.
29. Emelyanov V.N., Karpenko A.G., Volkov K.N. Development of advanced CFD tools and their application to simulation of internal turbulent flows. Proceedings of the 5th European Conference for Aeronautics and Space Sciences (EUCASS 2013), 1–5 July, Munich, Germany.
30. Osher S., Chakravarthy S. High resolution schemes and the entropy condition. SIAM Journal on

Numerical Analysis, 1984, 21(5), 955–984.
31. Volkov K.N. Large-eddy simulation of free shear and wall-bounded turbulent flows. Atmospheric Turbulence, Meteorological Modelling and Aerodynamics. USA, Nova Science, 2010, 505–574.
32. Betelin V.B., Shagaliev V.B., Aksenov V.B., Belyakov I.M., Deryuguin Yu.N., Korchazhkin D.A., Kozelkov A.S., Nikitin V.F., Sarazov A.V., Zelenskiy D.K. Mathematical simulation of hydrogen-oxygen combustion in rocket engines using LOGOS code. Acta Astronautica, 2014, 96, 53–64.
33. Roe P.L. Approximate Riemann solvers, parameter vectors, and difference schemes. Journal of Computational Physics, 1981, 43(2), 357–372.
34. Sod G.A. A survey of several finite difference methods of systems of nonlinear hyperbolic conservation laws. Journal of Computational Physics, 1978, 27(1), 1–31.
35. Silnikov M.V., Chernyshov M.V. The interaction of Prandtl–Meyer wave and quasione-dimensional flow region. ActaAstronautica,2015, 109, 248–253.
36. Silnikov M.V., Chernyshov M.V., Uskov V.N. Analytical solutions for Prandtl–Meyer wave – oblique shock overtaking interaction. ActaAstronautica,2014, 99, 175–183.
37. Wesseling P. Principles of computational fluid dynamics. Springer, 2000.
38. Schlichting H., Gersten K. Boundary layer theory. Springer Verlag, Berlin, 2000.
39. Volkov K.N., Hills N.J., Chew J.W. Simulation of turbulent flows in turbine blade passages and disc cavities. ASME Paper, GT2008-50672.

Highlights

- The use of graphics processor units for the simulation of flows on unstructured meshes are discussed
- CUDA technology is used for programming implementation of parallel computational algorithms
- Solutions of some benchmark test cases on graphics processor units are reported
- The results obtained provide promising perspective for designing a GPU-based software framework