Progress in Flight Physics **7** (2015) 247-268
DOI: 10.1051/eucass/201507247

DEVELOPMENT OF ADVANCED COMPUTATIONAL FLUID DYNAMICS TOOLS AND THEIR APPLICATION TO SIMULATION OF INTERNAL TURBULENT FLOWS

V. N. Emelyanov[1], A. G. Karpenko[2], and K. N. Volkov[3]

[1]Faculty of Power Engineering, Baltic State Technical University
1, 1-ya Krasnoarmeyskaya Str., St. Petersburg 190005, Russia
[2]Faculty of Mathematics and Mechanics, St. Petersburg State University
Universitetsky Prosp., Old Petergof, St. Petersburg 198504, Russia
[3]Faculty of Science, Engineering and Computing, Kingston University
Friars Av., Roehampton Vale, London SW15 3DW, United Kingdom

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 internal fluid flows are discussed. The finite volume method is applied to solve three-dimensional (3D) unsteady compressible Euler and Navier–Stokes equations on unstructured meshes. Compute Inified Device Architecture (CUDA) technology is used for programming implementation of parallel computational algorithms. Solution of some fluid dynamics problems on GPUs is presented and approaches to optimization of the CFD code related to the use of different types of memory are discussed. Speedup of solution on GPUs with respect to the solution on central processor unit (CPU) is compared with the use of different meshes and different methods of distribution of input data into blocks. 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.

1 INTRODUCTION

Fluid flow and heat transfer occur in nature and engineering. There are numerous naturally occurring phenomena as well as technical and technological applications in which fluid flows and heat transfer play an important role.

The methods of CFD are extensively applied in design and optimization of industrial devices to get more insight into 3D unsteady flows through fluid or gas passages. Accurate prediction of internal 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 and turbulence phenomena. Investigations of heat transfer, skin friction, secondary flows, flow separation, and reattachment effects demand reliable numerical methods, accurate programming, and robust working practices.

The stagnation in the clock speed of CPUs 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 GPU. These platforms make it possible to achieve speedups of an order of magnitude over a standard CPU in many CFD applications and are growing in popularity [1].

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;

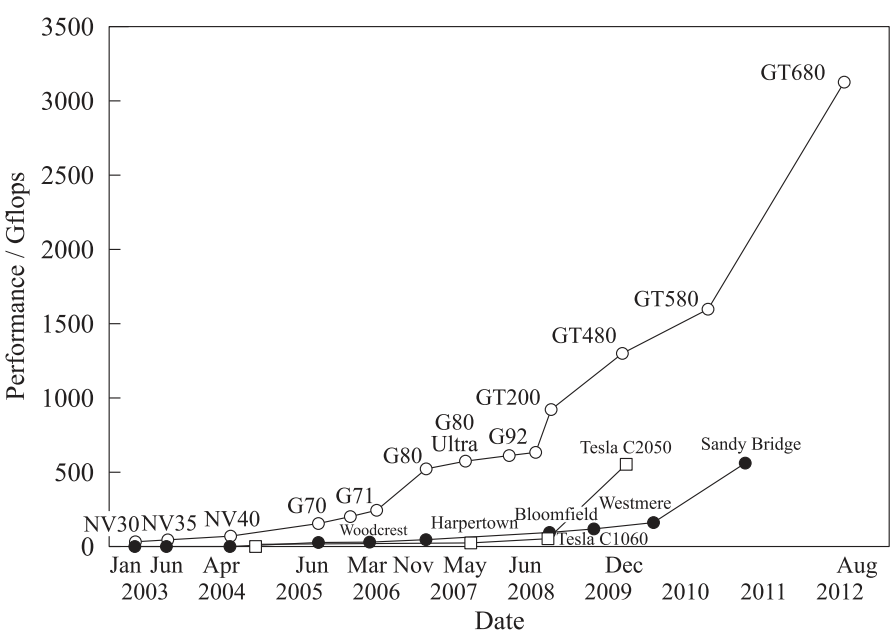

Figure 1 Floating point operations per second (flops) for the CPUs and GPUs: GT680 — GeForce GTX 680; GT580 — GeForce GTX 580; GT480 — GeForce GTX 480; GT200 — GeForce GTX 200; G92 — GeForce 9800 GTX; G80 — GeForce 8800 GTX; G71 — GeForce 7900 GTX; G70 — GeForce 7800 GTX; NV40 — GeForce 6800 Ultra; NV35 — GeForce FX 5950 Ultra; and NV30 — GeForce FX 5800

so, each generation improves on the speed of the previous ones by adding more cores, subject to the limits of space, heat, and cost. Central processing units, 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 the 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 for the time being [2].

Speed and accuracy are the key factors in the evaluation of flow 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 aerospace engineering.

The use of GPUs is a cost-effective way of improving substantially the performance in CFD applications. Taking advantage of any multicore architecture requires programs to be written for parallel execution. For CFD, this has traditionally meant splitting the flow domain into several parts 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. This is also the process adopted for GPUs, but the GPU introduces several additional constraints that make the stream programming paradigm particularly useful.

Depending on the complexity of the problem to represent and solve, a structured or unstructured mesh is used. Algorithms are more efficiently implemented on structured meshes, and data structures to handle the mesh are easy to implement. However, structured meshes present poor accuracy if the problem to be solved has internal or external boundaries of complex shape (in this case, special treatment of boundary conditions is required on structured meshes). It is difficult to generate structured or block-structured mesh in real geometry. On the other hand, unstructured meshes present more flexibility and higher accuracy to represent problems that have complex geometries and boundaries. However, the data structures to handle it are not easy to implement, and also explicit neighboring information should be stored. In general, unstructured meshes are more utilized because of their flexibility and higher accuracy.

Much of the effort in running CFD codes on GPUs has been directed toward the case of solvers based on structured meshes [3–8]. These solvers are easily to implement on GPUs due to their regular memory access pattern.

Unstructured mesh-based analysis methods on shared memory and distributed memory systems 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.

There has also been interest in running unstructured mesh-based CFD solvers on GPUs. Achieving good performance for such solvers is more difficult due to their data-dependent and irregular memory access patterns [9–12].

Explicit time-marching algorithms are the most convenient ones to be ported on to the GPU. This is because there is no iteration and the new value of a variable depends only on the old time values. Hence, the update of a given variable is done independent of variables being updated on other threads. There is no recursive relation between the variables on the threads, since they are all known at the old time step. However, even for explicit algorithms, a few changes are needed for efficiently implementation of numerical algorithms on the GPU. These relate to the use of shared memory and the layout of data structures. Memory coalescing and block size influence on the speed have been 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 reused 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 is impacted. Implicit algorithms directly ported to a GPU do 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. The present work is undertaken as a part of a larger effort to establish a common CFD code for simulation of internal flows in aerospace and related applications and involves some basic validation studies. The paper describes some of experiences of using GPUs as a paradigm for performing large-scale CFD computations. The governing equations are solved with finite volume code on hybrid meshes. Parallel computations are implemented using the message passing interface (MPI) (clusters) and CUDA technology (GPU). The results obtained are generally in reasonable agreement with the available experimental and computational data. Capabilities and accuracy of various finite difference schemes, acceleration efficiency calculations due to parallelization of computational algorithms are compared and implementation of various approaches to decomposition of computational domain and load balancing are discussed.

2 GOVERNING EQUATIONS

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

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

The equation of state of ideal gas is

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

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

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

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

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

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

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

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; p is the pressure; e is the total energy per unit mass; T is the temperature; and γ is the specific heat ratio.

The governing equations are suitable for both laminar and turbulent flows, and they formally coincide with unsteady Reynolds-averaged Navier–Stokes (RANS) equations. The effective viscosity, μ_e, is calculated as a sum of molecular viscosity, μ, and eddy viscosity, μ_t, and the effective thermal conductivity, λ_e, is expressed in terms of viscosity and Prandtl number:

$$\mu_e = \mu + \mu_t; \quad \lambda_e = c_p \left(\frac{\mu}{\mathrm{Pr}} + \frac{\mu_t}{\mathrm{Pr}_t} \right)$$

where c_p is the specific heat capacity at constant pressure. Molecular and turbulent Prandtl numbers are $\mathrm{Pr} = 0.72$ and $\mathrm{Pr}_t = 0.9$ for air. 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 $\mu_* = 1.68 \cdot 10^{-5}$ kg/(m s), $T_* = 273$ K, and $S_0 = 110.5$ K for air.

3 NUMERICAL METHOD

The equation solved by the CFD code is of the form $dQ/dt = R(Q)$ where Q is the flow variables vector. The flow residual is $R(Q) = H(Q) - L(Q)$ where $L(Q)$ denotes all the spatial differencing terms and $H(Q)$ denotes the terms from boundary conditions and possible source terms.

The Navier–Stokes equations are numerically solved by local time-stepping using an upwind scheme with Godunov's exact or Roe's approximate Riemann solver on an unstructured mesh with different shape of cells. A Runge–Kutta formulation is used for the time integration.

Analysis using an unstructured mesh is implemented as an iterative method where the values of the variables at each solution point are updated until they converge to the solution or reach a number of iterations. The relative L_2 norm of the flow residual is taken as a criterion to test convergence history.

The code user needs to specify physical flow inputs and boundary conditions which typically define solid walls and inflow or outflow regions. Along with these physics-type inputs, there are inputs which choose particular numerical algorithms and specify parameters for them. Some details of the CFD code are provided in [13].

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 precomputed and take account of the geometry of the cell. The nonlinear CFD solver works in an explicit time-marching fashion, based on a Runge–Kutta stepping procedure. The governing equations are solved with upwind finite difference scheme for inviscid fluxes and central difference scheme of the 2nd 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.

4 PROGRAMMING MODEL

Technology CUDA is a parallel computing architecture which introduces a new programming model based on high-level abstraction levels that avoid the former graphics pipeline concepts and ease the porting of a scientific CPU application [2]. 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 of GPUs is unlike traditional CPU programming, because the hardware is dramatically 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 suboptimal 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 that takes months and years to master.

Framework CUDA is a technology for GPU computing from NVIDIA. The GPU is formed by a set of SIMD (single instruction multiple data) 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 mesh 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.

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 scheme. 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 each iteration are divided into three parts:

(1) local cell analysis to obtain a coefficient for each solution point based only on the interaction with the other solution in the same cell;

(2) neighbor cell analysis to compute a coefficient for each solution point based on the interaction with its neighbor solution point; and

(3) 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 neighbor-oriented (edge-oriented) analysis. Although solution point variables and parameters are heavily used in all three main stages, they are accessed with different patterns at each 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, in the second step the pointer to the left and right solution point variables, and the third step involves accessing the two solution points variables. Left and right solution point variables are not physically adjacent, and information is read and used only once, hence, either on a unithreaded or multithreaded 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 unithreaded and multithreaded solutions.

A GPU 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 are 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.

4.3 Redundant Computation

The time derivative computation is parallelized on a per-element basis, with one thread per element [10]. First, each thread reads the element's volume, along with its conserved variables from global memory, from which derived quantities such as the pressure, velocity, total energy, and the flux contribution are computed. The kernel then loops over each of all faces of the control volume in order to accumulate fluxes. The face's normal is read along with the index of the adjacent element, where this index is then used to access the adjacent element's conserved variables. The required derived quantities are computed and then the flux is accumulated into the element's residual.

This approach requires redundant computation of flux contributions, and other quantities derived from the conserved variables. Another possible approach is to first precompute each element's flux contribution, thus avoiding such redundant computation. However, this approach turns out to be slower for two reasons. The fist of which is that reading the flux contributions requires three times the amount of global memory access than just reading the conserved variables. The second is that the redundant computation is performed simultaneously with global memory access, which hides the high latency of accessing global memory.

4.4 Numbering Scheme

In the case of an unstructured mesh, the global memory access required for reading the conserved variables of neighboring elements is at risk of being highly noncoalesced, which results in lower effective memory bandwidth. This is avoided however, if neighboring elements of consecutive elements are nearby in memory. This is achieved here in two steps. The first step is to ensure that elements nearby in space are nearby in memory by using a renumbering scheme [10]. 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 element is then sorted locally, so that the indices of the four neighbors of each tetrahedral element (for triangular mesh) are in increasing order. This ensures that, for example, the second neighbor of consecutive elements are close in memory.

4.5 Shared Memory

Shared memory is an important feature of modern graphics 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 that 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 solvers is data-dependent. With the per-element/thread based connectivity data structure considered here, this information is not provided, and therefore, shared memory is not applicable in this case.

5 PARALLELIZATION TECHNIQUE

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

The computation steps required by the problem considered is classified into two groups: computations associated to 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 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 Fig. 2.

The implementation is split between the CPU and the GPU. All pre- and postprocessing is 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/normals and cell volumes. The initial guess 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 precomputed on the CPU and loaded into GPU texture memory. For a given node, the kernel obtains the indices required

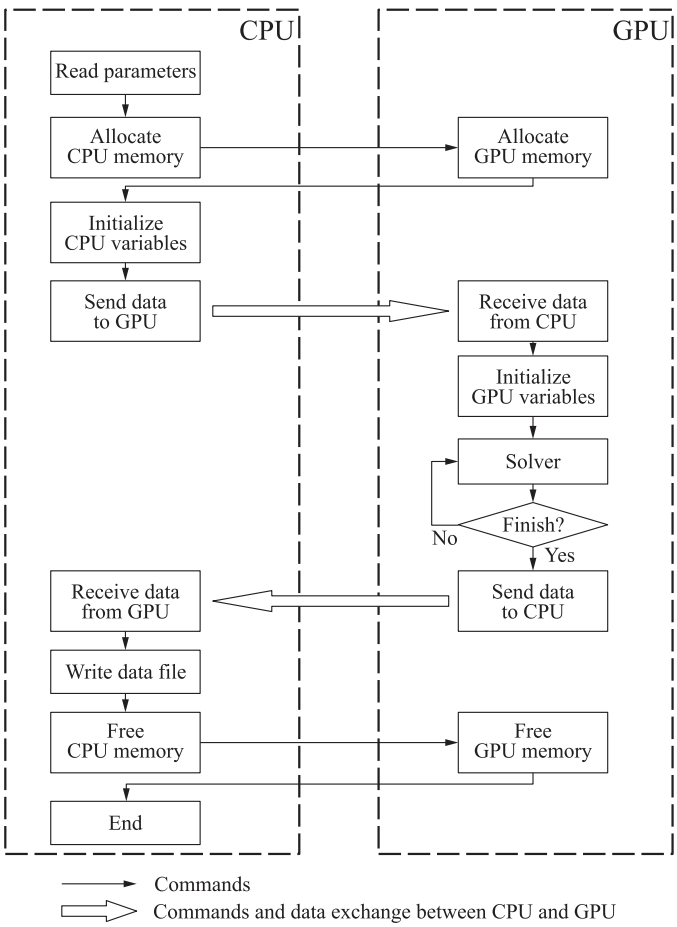

Figure 2 Solution of problem with the use of GPU resources

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 algorithm, obtained from the mathematical description of the numerical scheme, is shown in Fig. 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 the 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

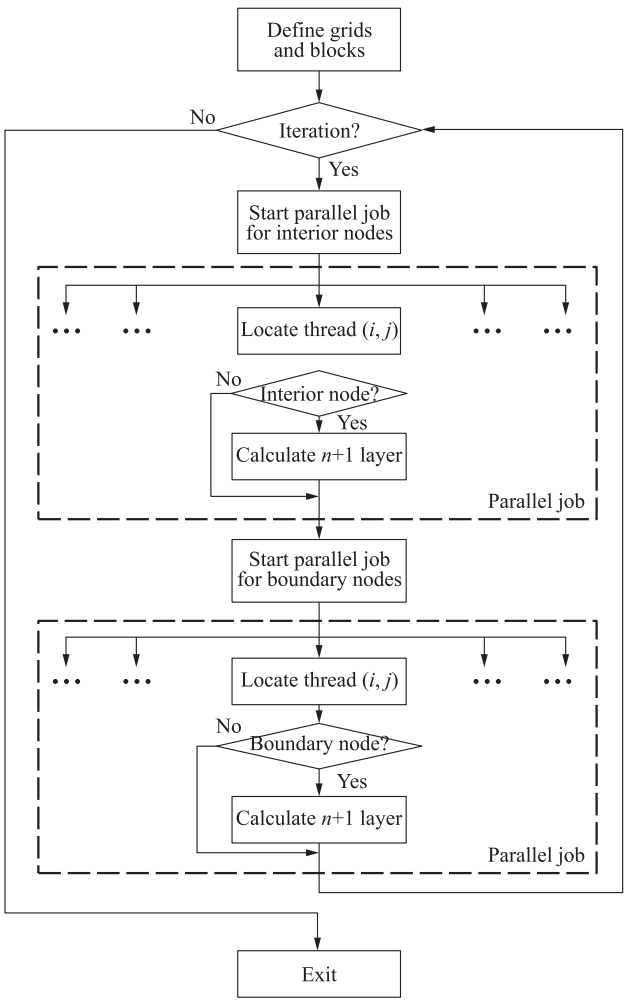

Figure 3 Main calculation stages in the parallel algorithm

associated to 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.

The most costly stage in the algorithm is edge-based calculations involving two calculations for each edge communicating two cells. This contribution is computed independently for each edge 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 volume is approximated from the nth 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 Fig. 4. Each time layer calculation is 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, Fig. 4*a*), and the other one provides flow variables calculation on the next time layer (stage 2, Fig. 4*b*). On the first stage, flow variables in the centers of control volumes are stored in array Q in global memory. One thread is used to calculate the fluxes through the faces of control volume. Each thread uses the flow variables vector in the control volumes i and $i+1$. Fluxes through

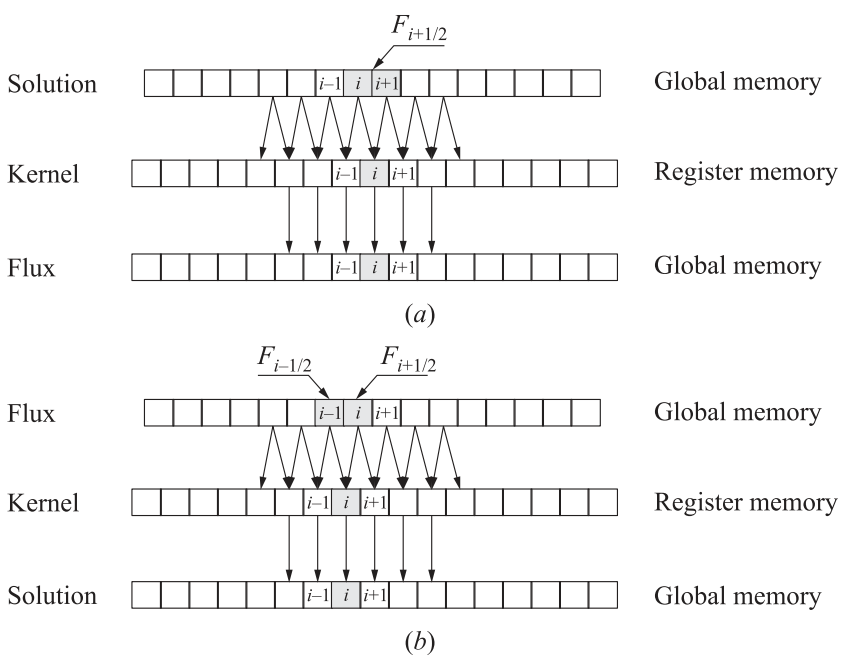

Figure 4 Flux calculation (*a*) and calculation of flow variable vector on a new time layer (*b*)

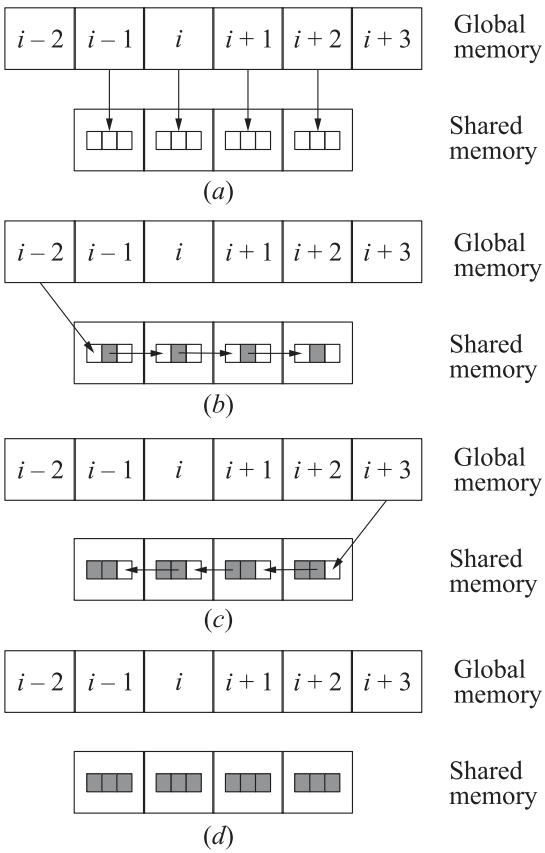

Figure 5 Use of shared memory and flux calculation

faces are stored in array F. On the second stage, a set of threads correspond to the same number of control volumes. 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 computed in the control volume i. The solution is stored in the array Q.

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

Table 1 Updating of flow variables and flux calculation

Step	Thread		
	$i-1$	i	$i+1$
Variables	u_{i-2}, u_{i-1}, u_i	u_{i-1}, u_i, u_{i+1}	u_i, u_{i+1}, u_{i+2}
Reconstruction	u^L_{i-1}, u_{i-1}, u^R_{i-1}	u^L_i, u_i, u^R_i	u^L_{i+1}, u_{i+1}, u^R_{i+1}
Restriction	u^{L*}_{i-1}, u_{i-1}, u^{R*}_{i-1}	u^{L*}_i, u_i, u^{R*}_i	u^{L*}_{i+1}, u_{i+1}, u^{R*}_{i+1}
Evolution	$\widehat{u}^L_{i-1}$, u_{i-1}, $\widehat{u}^R_{i-1}$	$\widehat{u}^L_i$, u_i, $\widehat{u}^R_i$	$\widehat{u}^L_{i+1}$, u_{i+1}, $\widehat{u}^R_{i+1}$
—		Synchronization	
Flux function	$f(\widehat{u}^R_{i-2}, \widehat{u}^L_{i-1})$	$f(\widehat{u}^R_{i-1}, \widehat{u}^L_i)$	$f(\widehat{u}^R_i, \widehat{u}^L_{i+1})$
—		Synchronization	
Flux	$f_{i-2} - f_{i-1}$	$f_{i-1} - f_i$	$f_i - f_{i+1}$

Table 1 shows the order of updating flow variables vector in the shared memory. In the simulation of two-dimensional flows (four flow variables), each thread is processing four flow vectors, and the total number of used shared memory is estimated as $4 * n_v *$ `blockDim.y` $*$ `blockDim.x`.

7 RESULTS AND DISCUSSION

The GPU version of the CFD code is tested for a variety of benchmark 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. All meshes are treated as unstructured meshes. This issue is related to the presentation of the meshes used in the solution of practical problems in the CFD code and not to the visual representation of the meshes.

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 one-dimensional (1D) Euler equations containing simultaneously a shock wave, a contact discontinuity, and an expansion fan [14]. 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 the following: $\rho_L = 1.0$, $u_L = 0.75$, and $p_L = 1.0$ if $0 \leq x \leq 50$, and $\rho_R = 0.125$, $u_R = 0$, and $p_R = 1.0$ if $50 < x \leq 100$.

Table 2 Sod problem. Time and speedup

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

Calculations are performed on different meshes. A number of cells increases from 1024 cells for mesh 1 to 30,720 cells for mesh 2 and to 307,200 cells for mesh 3. The finest mesh, mesh 4, contains about three million cells. The time step is 15.2 μs, and the total calculation time is 7.63 ms. 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.

The time required for calculation of one time step, and speedup of calculations are given in Table 2 (time is given in milliseconds). Option 1 corresponds to Godunov scheme, and option 2 corresponds to Roe scheme. 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 exchange, reducing the GPU performance.

Speedup and memory required are compared based on numerical solutions of different test cases. Numerical accuracy of the results obtained is similar for different meshes used in calculations. For 1D case, Godunov's scheme is less expensive from the computational point of view requiring a small number of communications between processors. In 3D calculations, numerical solution based on Godunov's method is more time consuming as compared to Roe scheme.

7.2 Shock Tube Problem

Three-dimensional model and unstructured mesh are used to solve the shock tube problem. The length of the computational domain is $L = 10$ m (Fig. 6). 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. The time step is 15.2 μs, and the total computational time is 7.63 ms. 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.

Time of calculation of 1000 time steps and speedup of calculations are presented in Table 3 (time is given in seconds). Three indices are used to specify

computational option. The first index corresponds to the solution of Euler equations (option 1) or to the solution of Navier–Stokes equations (option 2). The second index corresponds to the time-marching scheme used in calculations based on one-step (option A) or two-step (option B) Runge–Kutta scheme. The third index corresponds to the exact (indices 1 and 3) or approximate (indices 2 and 4) Riemann solver of the first (indices 1 and 2) or second (indices 3 and 4) order. The calculations based on the finest mesh containing about 10 million of cells (mesh 4) with Godunov scheme give speedup of 42 (option 1A2). For the solution of viscous problem with the scheme of the second order, the speedup drops to 22 (option 2A2).

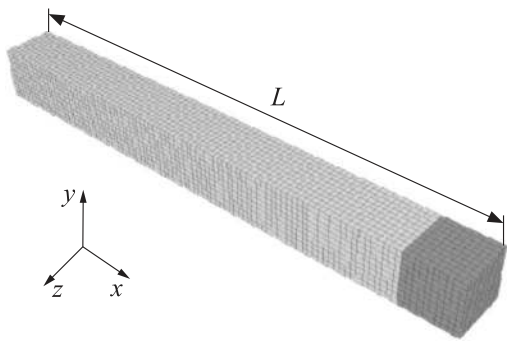

Figure 6 Shock tube problem

The time required for calculation of 1000 time steps on the mesh with 10^7 cells and memory usage are given in Table 4. Option 1 corresponds to GPU calculations based on Godunov scheme, option 2 corresponds to CPU calculations based on Godunov scheme, and option 3 corresponds to calculations based on commercial package Ansys Fluent with 8 nodes.

Table 3 Shock tube problem. Time and speedup

No.	Mesh 1			Mesh 2			Mesh 3			Mesh 4		
	CPU	GPU	S	CPU	GPU	S	CPU	GPU	S	CPU	GPU	S
1A1	6.63	0.47	13.96	69.99	2.64	26.50	671.22	22.67	29.61	6329.61	198.63	31.87
1A2	14.72	0.62	23.68	121.83	3.73	32.68	934.05	31.60	29.56	8787.27	207.82	42.28
1A3	12.03	1.29	9.30	130.33	10.41	12.52	1262.17	94.27	13.39	11018.60	826.18	13.34
1A4	20.66	1.55	13.33	193.47	12.81	15.11	1630.47	115.92	14.06	13934.40	872.42	15.97
2A1	18.79	1.36	13.83	198.96	10.98	18.12	1918.32	99.60	19.26	17485.10	874.99	19.98
2A2	27.79	1.60	17.35	261.63	13.31	19.65	2285.81	120.88	18.91	20542.40	914.83	22.46
1B1	12.63	0.91	13.95	133.88	5.07	26.39	1283.90	44.05	29.14	12163.30	382.90	31.77
1B2	29.14	1.20	24.30	242.48	7.29	33.26	1909.59	61.83	30.89	16989.20	406.65	41.78
1B3	23.47	2.50	9.37	254.60	20.39	12.49	1630.47	186.18	8.76	22484.20	1636.73	13.74
1B4	41.57	3.04	13.67	380.87	25.60	14.88	3227.67	230.86	13.98	27978.30	1734.85	16.127
2B1	36.83	2.63	13.98	388.74	21.25	18.29	3776.11	197.58	19.11	35257.70	1736.47	20.30
2B2	56.28	3.14	17.90	515.97	26.28	19.63	4672.10	240.95	19.39	40595.40	1822.97	22.27

Table 4 Shock tube problem. Time and memory

No.	Memory, MB	S/M	Time, s	S/M
1	2582.28	—	305.29	—
2	2696.72	1.04	14916.60	48.86
3	8210.16	3.18	7662.00	25.10

Graphics processing unit implementation of the CFD code is preferable against parallel implementation of similar algorithms in commercial CFD package (for the given computational resources).

7.3 Lid-Driven Cavity Problem

Large-eddy simulation of lid-driven cavity flow is considered at $\mathrm{Re} = 10^3$. The Smagorinsky model is used [15] as a subgrid scale model and a projection method is applied to discretization of the Navier–Stokes equations. The mesh contains 128^3 nodes.

Time costs are presented in Fig. 7 for various parts of the computational procedure.

Poisson equation is solved with the red/black Gauss–Seidel method (routines red_kernel and black_kernel), taking about two third of the total computation time.

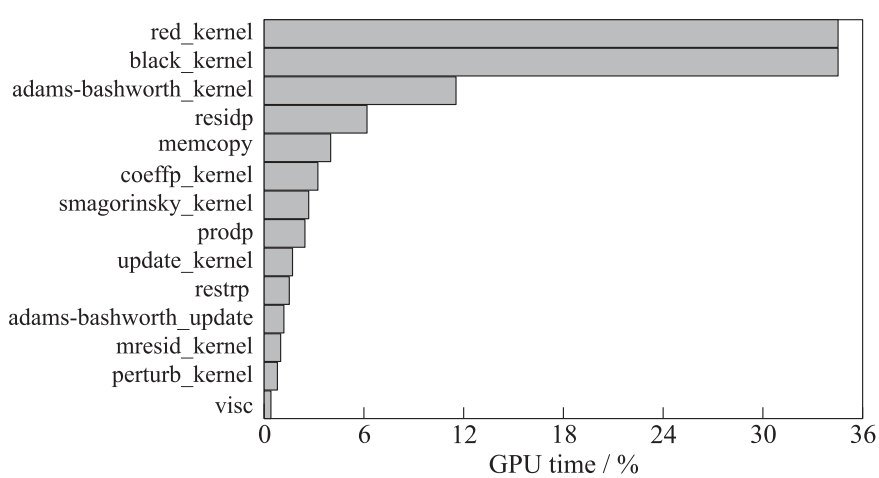

Figure 7 Time consuming for different steps

Table 5 Lid-driven cavity problem. Time and speedup

Mesh	$n_x = n_y = n_z = 4$			$n_x = 32$, $n_y = 1$, $n_z = 8$		
	CPU, s	GPU, s	S	CPU, s	GPU, s	S
16^3	0.34	0.39	0.86	0.34	0.31	1.11
32^3	3.08	1.24	2.50	3.08	0.66	4.73
64^3	31.14	6.49	4.81	31.14	2.71	11.51
128^3	291.05	50.92	5.72	291.05	18.35	15.88

The next expensive routine in terms of computational time is a routine adams_bashworth implementing time evolution.

The routine residp calculates the residuals of the Poisson equation.

Copy and memory allocation for flow variables is produced by the routine memcopy.

Routines coeffp_kernel and smagorinsky_kernel produce calculations of the coefficients related to the discrete Poisson equation and calculations based on the Smagorinsky model.

The routines update_kernel and adams_bashworth_kernel are used to go to the next time step.

Contributions of other routines such as prodp, restrp, mresid_kernel, perturb_kernel, and visc, supporting some nonimportant operations, are relatively small.

Speedup of calculations on different meshes with different partitioning techniques into blocks of size $n_x \times n_y \times n_z$ are presented in Table 5 (computational time is given in seconds for 100 time steps). On the mesh containing 128^3 nodes, speedup changes in 2.8 times for different partitioning techniques.

The most time consuming part of the computational algorithm is the solution of Poisson equation for pressure. The developed CFD solver uses a simple iterative scheme to solve Poisson equation. More advanced methods (Krylov subspace methods, multigrid methods) could be applied in the future. Preliminary results obtained show their advantages against iterative methods in terms of computational time and overall speedup (however, computational work per time step increases).

7.4 Channel Flow Problem

Large-eddy simulation of turbulent flow in a channel with square cross-sectional shape is carried out at the Reynolds number $\mathrm{Re}_\tau = 360$ on the mesh with $256 \times 64 \times 64$ nodes. The ratio of the channel length to its width is $L/H = 4$. The mesh is uniform along the x coordinate and the mesh is stretched near the walls; so, $y^+ \sim z^+ \sim 1$ on the channel walls.

Table 6 Channel flow problem. Time and speedup

Mesh	$n_x = n_y = n_z = 4$			$n_x = 32$, $n_y = 1$, $n_z = 8$		
	CPU	GPU	S	CPU	GPU	S
$256 \times 64 \times 64$	208.09	34.56	6.02	208.09	14.32	14.50
$512 \times 64 \times 64$	448.47	64.29	6.98	—	—	—

The calculations are based on the Smagorinsky subgrid scale model [15]. The nonlinear CFD solver works in an explicit time marching fashion, based on a three-step Runge–Kutta stepping procedure and piecewise parabolic method. The governing equations are solved with Chakravarthy–Osher scheme for inviscid fluxes and the central difference scheme of the 2nd order for viscous fluxes. Calculations are performed on the CPU Intel Core 2 Duo with 3 GHz and GPU card GeForce GTX 480.

Speedup of calculations on different meshes is shown in Table 6 based on different partitioning techniques. The computational time is given in seconds for the 100 time steps. On the mesh with $256 \times 64 \times 64$ nodes, changing a partitioning technique into blocks leads to speedup increasing by 2.4 times.

7.5 Flat Plate Flow

The turbulent flow over a smooth flat plate is well-known CFD benchmark solution [16], and it is used for verification and validation of other CFD codes.

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 ($U = 10$ m/s), static pressure ($p = 101\,325$ Pa), and static temperature ($T = 300$ K) are fixed on the inlet boundary. No-slip and no-penetration boundary conditions are used on the plate. The plate is adiabatic. Weak boundary conditions are applied to the outlet boundary. Slip boundary conditions are used on the far-stream boundaries.

The flat plate boundary layer problem is solved on different meshes. Laminar flow calculations are performed on one core of AMD Phenom 2.3 GHz and one

Table 7 Flat plate problem. Time and speedup

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

module of Tesla S1070 platform consisting of 240 cores with 1.44 GHz. The mesh consists of 111,670 nodes. Computational time of one iteration is 462.6 s on CPU and 11.5 s on GPU and speedup is 40.2. The turbulent flow calculations are based on CPU Xeon X5670 2.93 GHz and one module of Tesla S2050 platform. The computational time and speedup of calculations are shown in Table 7 for one iteration. Increasing a number of nodes from 10^5 to 10^7, speedup increases on 10%.

8 CONCLUDING REMARKS

Graphics processing units have evolved as a new paradigm for scientific computations. They are essentially multicore machines with a large number of computational units sharing a common memory. They are viewed as SIMD computers. Their 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 algorithm.

Possibilities of the use of GPUs for CFD calculations are discussed. The finite volume method is applied to solve full Euler and Navier–Stokes equations on unstructured mesh. Technology CUDA is used for programming implementation of parallel computational algorithms. Solution of some CFD problems on GPUs is presented and approaches to optimization of the CFD code related to the use of different types of memory are discussed. Speedup of solution on GPUs with respect to solution on CPU is compared with the use of different meshes and different methods of distribution of input data into blocks. Speedup of CFD calculations changes from 10 to 50 depending on the problem, computational procedure, and computational resources. This makes GPUs very attractive for computing industrial fluid flows.

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

Further work is focused on parallel implementation of implicit schemes and convergence acceleration techniques.

ACKNOWLEDGMENTS

This work was supported by the Russian Foundation for Basic Research (grant 13-07-12079). The authors wish to thank colleagues from the Russian Federal Nuclear Center (Sarov, Russia) for the access to the computational resources and discussion of the computational results.

REFERENCES

1. Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Krűger, A. E. Lefohn, and T. J. Purcell. 2007. A survey of general-purpose computation on graphics hardware. *Comput. Graph. Forum* 26(1):80–113.
2. Sanders, J., and E. Kandrot. 2011. *CUDA by example: An introduction to general-purpose GPU programming*. Boston: Pearson Education. 312 p.
3. Scheidegger, C. E., J. L. D. Comba, and R. D. da Cunha. 2005. Practical CFD simulations on programmable graphics hardware using SMAC. *Comput. Graph. Forum* 24(4):715–728.
4. Hagen, T. R., K.-A. Lie, and J. R. Natvig. 2006. Solving the Euler equations on graphics processing units. *Computational science*. Eds. V. N. Alexandrov, G. Dick van Albada, P. M. A. Sloot, and J. Dongarra. Lecture notes in computer science ser. Berlin–Heidelberg: Springer Verlag. 3994:220–227.
5. Brandvik, T., and G. Pullan. 2009. An accelerated 3D Navier–Stokes solver for flows in turbomachines. ASME Paper. No. GT2009-60052.
6. Thibault, J. C., and I. Senocak. 2009. CUDA implementation of a Navier–Stokes solver on multi-GPU desktop platforms for incompressible flows. AIAA Paper No. 2009-758.
7. Shinn, A. F., S. P. Vanka, and W. W. Hwu. 2010. Direct numerical simulation of turbulent flow in a square duct using a graphics processing unit (GPU). AIAA Paper No. 2010-5029.
8. Kuo, F.-A., M. R. Smith, C.-W. Hsieh, C.-Y. Chou, and J.-S. Wu. 2011. GPU acceleration for general conservation equations and its application to several engineering problems. *Comput. Fluids* 45(1):147–154.
9. Kampolis, I. C., X. S. Trompoukis, V. G. Asouti, and K. C. Giannakoglou. 2010. CFD-based analysis and two-level aerodynamic optimization on graphics processing units. *Comput. Method. Appl. Mech. Eng.* 199(9–12):712–722.
10. Corrigan, A., F. Camelli, R. Lőhner, and F. Mut. 2011. Semi-automatic porting of a large-scale Fortran CFD code to GPUs. *Int. J. Numer. Meth. Fl.* 69(2):314–331.
11. Volkov, K. N., V. N. Emelyanov, A. G. Karpenko, I. V. Kurova, A. E. Serov, and P. G. Smirnov. 2013. Numerical solution of fluid mechanics problems on general-purpose graphics processor units. *Numer. Meth. Programming* 14(1):82–90.
12. Volkov, K. N., V. N. Emelyanov, A. G. Karpenko, P. G. Smirnov, and I. V. Teterina. 2013. Implementation of a finite volume method and calculation of flows of a viscous compressible gas on graphics processor units. *Numer. Meth. Programming* 14(1):183–194.
13. Volkov, K. N. 2010. Large-eddy simulation of free shear and wall-bounded turbulent flows. In: *Atmospheric turbulence, meteorological modelling and aerodynamics*. USA: Nova Science. 505–574.
14. Sod, G. A. 1978. A survey of several finite difference methods of systems of nonlinear hyperbolic conservation laws. *J. Comput. Phys.* 27(1):1–31.
15. Smagorinsky, J. 1963. General circulation experiments with the primitive equations. *Mon. Weather Rev.* 91(3):99–164.
16. Schlichting, H., and K. Gersten. 2000. *Boundary layer theory*. Berlin: Springer Verlag. 799 p.