Acceleration of Deterministic Boltzmann Solver with Graphics Processing Units

V.V.Aristov[a], A.A.Frolova[a], S.A.Zabelok[a], V.I.Kolobov[b] and R.R.Arslanbekov[b]

[a]*Dorodnicyn Computing Centre of the Russian Academy of Sciences, Ul. Vavilova 40, Moscow, 119333, Russia*
[b]*CFD Research Corporation, 215 Wynn Dr, Huntsville, AL, 35803, USA*

Abstract. GPU-accelerated computing of the Boltzmann collision integral is studied using deterministic method with piecewise approximation of the velocity distribution function and analytical integration over collision impact parameters. The acceleration of 40 times is achieved compared to CPU calculations for a 3D problem of collisional relaxation of bi-Maxwellian velocity distribution.

Keywords: Boltzmann equation, direct Boltzmann solver, Graphics Processing Units, CUDA programming model.
PACS: 05.20.Dd.

INTRODUCTION

Recent advancements in the development of Graphics Processing Units (GPU) make it possible to achieve teraflop performance on a single desktop computer [1]. To achieve peak performance, special numerical algorithms for GPU architecture should be developed to divide the problem into multiple threads executed in parallel. The acceleration of 30-40 times by GPU computations is reported in [2] for the evaluation of the Boltzmann collision integral with semi-regular methods. Frezzotti *et al.* [3, 4] have shown that direct solution of the Boltzmann equation (with both the Bhatnagar-Gross-Krook-Welander (BGKW) model and the hard sphere collision integral) can be easily divided into threads to produce more than 2 orders of magnitude speedup on GPU versus CPU.

In the present paper, the GPU-accelerated computing (with NVIDIA cards and the high level CUDA programming language) is studied for computing the Boltzmann collision integral with deterministic (regular) method using a piecewise approximation for the velocity distribution function and analytical integration over impact parameters (see, e.g. [5]). The work utilizes our previous experience on parallelization in velocity space for the deterministic method described in [6] and the development of different parallel algorithms for direct Boltzmann solvers [7, 8].

We describe special GPU algorithms for solving the Boltzmann kinetic equation by the deterministic (regular) method. This method could be particularly attractive for unsteady problems, problems with low gas velocities (such as flows in MEMS) and flows at small Knudsen numbers. A regular grid in velocity space is used with cubic cells in velocity space. The splitting procedure reduces the problem to advection in physical space, and collisional relaxation in velocity space. This paper focuses on the relaxation part of the problem. The GPU-methods for solving the advection part have already been described in [2-4].

THE BOLTZMANN EQUATION AND THE DETERMINISTIC METHOD

The Boltzmann kinetic equation is written in the usual form using standard notations:

$$\frac{\partial f}{\partial t}+\xi_x\frac{\partial f}{\partial x}=I\left(\xi,x,t\right),\ I\left(\xi,x,t\right)=\int\left[f\left(\xi'\right)f\left(\xi_1'\right)-f\left(\xi\right)f\left(\xi_1\right)\right]gb\,db\,d\varepsilon\,d\xi_1 .$$

Here f is the distribution function, I is the collision integral written for a model of hard spheres (or VHS). The deterministic method [5, 6] for calculation of the collision integral is briefly described below. A uniform rectangular 3D computational mesh is introduced in velocity space with cells $\Delta\xi_x\times\Delta\xi_y\times\Delta\xi_z$. The computational domain is

chosen in such a way that the distribution function is small beyond the domain. The cells are numbered by a 3D index $\mathbf{i} = (i_x, i_y, i_z)$. The location of the center of a cell $X_{\mathbf{i}}$ is defined in the following manner: $\xi_{i\alpha} = \xi_{\alpha\,MIN} + (i_\alpha + 1/2)\Delta\xi_\alpha$, $\alpha = x, y, z$. As the computational domain is finite the 3D index can be easily converted into a 1D index. A piecewise constant approximation of the distribution function is adopted, i.e. the distribution function is assumed to be constant inside of each cell. This approximation of the distribution function can be written as follows:

$$f(\boldsymbol{\xi}) = \sum_{\mathbf{i}} f_{\mathbf{i}} e_{\mathbf{i}}(\boldsymbol{\xi}),\ e_{\mathbf{i}}(\boldsymbol{\xi}) = \begin{cases} 1, \boldsymbol{\xi} \in X_{\mathbf{i}}, \\ 0, \boldsymbol{\xi} \notin X_{\mathbf{i}}. \end{cases} \tag{1}$$

The collision integral is computed in the center of each cell $X_{\mathbf{i}}$:

$$I_{\mathbf{i}} = \int \left[f(\boldsymbol{\xi}') f(\boldsymbol{\xi}_1') - f(\boldsymbol{\xi}_{\mathbf{i}}) f(\boldsymbol{\xi}_1) \right] g b\, db\, d\varepsilon\, d\boldsymbol{\xi}_1 . \tag{2}$$

The integration over parameter $\boldsymbol{\xi}_1$ in (2) is performed numerically using the centered rectangular quadrature formula on the cells of the computational mesh:

$$I_{\mathbf{i}} = \sum_{\mathbf{j}} \left\{ \int \left[f(\boldsymbol{\xi}')\big|_{\xi=\xi_{\mathbf{i}},\xi_1=\xi_{\mathbf{j}}} f(\boldsymbol{\xi}_1')\big|_{\xi=\xi_{\mathbf{i}},\xi_1=\xi_{\mathbf{j}}} - f_{\mathbf{i}} f_{\mathbf{j}} \right] g_{\mathbf{ij}} b\, db\, d\varepsilon \right\} \Delta\boldsymbol{\xi} . \tag{3}$$

In (3) $\Delta\boldsymbol{\xi} = \Delta\xi_x\, \Delta\xi_y\, \Delta\xi_z$, $g_{\mathbf{ij}} = \left| \boldsymbol{\xi}_{\mathbf{i}} - \boldsymbol{\xi}_{\mathbf{j}} \right|$. Substituting (1) into (3) results in:

$$I_{\mathbf{i}} = \sum_{\mathbf{j}} \left\{ \int \left[\sum_{\mathbf{k},\mathbf{l}} f_{\mathbf{k}} f_{\mathbf{l}} e_{\mathbf{k}}(\boldsymbol{\xi}') e_{\mathbf{l}}(\boldsymbol{\xi}_1')\big|_{\xi=\xi_{\mathbf{i}},\xi_1=\xi_{\mathbf{j}}} - f_{\mathbf{i}} f_{\mathbf{j}} \right] g_{\mathbf{ij}} b\, db\, d\varepsilon \right\} \Delta\boldsymbol{\xi} . \tag{4}$$

Equation (4) can be rewritten in the following manner:

$$I_{\mathbf{i}} = \sum_{\mathbf{k},\mathbf{l}} A_{\mathbf{ikl}} f_{\mathbf{k}} f_{\mathbf{l}} - f_{\mathbf{i}} \sum_{\mathbf{j}} f_{\mathbf{j}} B_{\mathbf{ij}} . \tag{5}$$

The coefficients $A_{\mathbf{ikl}}$ and $B_{\mathbf{ij}}$ depend only on the velocity mesh and have the form:

$$A_{\mathbf{ikl}} = \Delta\boldsymbol{\xi} \sum_{\mathbf{j}} g_{\mathbf{ij}} \int \left[e_{\mathbf{k}}(\boldsymbol{\xi}') e_{\mathbf{l}}(\boldsymbol{\xi}_1') \right]\Big|_{\xi=\xi_{\mathbf{i}},\xi_1=\xi_{\mathbf{j}}} b\, db\, d\varepsilon , \tag{6}$$

$$B_{\mathbf{ij}} = \Delta\boldsymbol{\xi} g_{\mathbf{ij}} \int b\, db\, d\varepsilon . \tag{7}$$

In [5, 6] it is shown that the coefficients $A_{\mathbf{ikl}}$ and $B_{\mathbf{ij}}$ can be computed analytically for the collision model of hard spheres. The practical restriction for the direct application of Eq. (5) in numerical simulations consists of the huge size of the array $A_{\mathbf{ikl}}$. To overcome this limitation, the equation (6) should be used. As the velocities $\boldsymbol{\xi}$, $\boldsymbol{\xi}_1$ and $\boldsymbol{\xi}'$, $\boldsymbol{\xi}_1'$ are linked by the momentum conservation law, the indices **i**, **j**, **k** and **l** are not independent: $\mathbf{i} + \mathbf{j} = \mathbf{k} + \mathbf{l}$, and a new index **n** can be introduced instead of 2 indices **k** and **l** in such a way that all components of 3D indices **k** and **l** to be integer: $\mathbf{k} = (\mathbf{i} + \mathbf{j} + \mathbf{n})/2$, $\mathbf{l} = (\mathbf{i} + \mathbf{j} - \mathbf{n})/2$. Thus, instead of summation over indices **k** and **l** in (5), a summation over indices **j** and **n** can be used:

$$I_{\mathbf{i}} = \sum_{\mathbf{j},\mathbf{n}} A'_{\mathbf{i},\mathbf{j},\mathbf{n}} f_{\frac{\mathbf{i}+\mathbf{j}+\mathbf{n}}{2}} f_{\frac{\mathbf{i}+\mathbf{j}-\mathbf{n}}{2}} - f_{\mathbf{i}} \sum_{\mathbf{j}} f_{\mathbf{j}} B_{\mathbf{ij}} . \tag{8}$$

The coefficients $A'_{\mathbf{i},\mathbf{j},\mathbf{n}}$ are computed as follows:

$$A'_{\mathbf{i},\mathbf{j},\mathbf{n}} = \Delta\boldsymbol{\xi} g_{\mathbf{ij}} \int \left[e_{\frac{\mathbf{i}+\mathbf{j}+\mathbf{n}}{2}}(\boldsymbol{\xi}') e_{\frac{\mathbf{i}+\mathbf{j}-\mathbf{n}}{2}}(\boldsymbol{\xi}_1') \right]\Big|_{\xi=\xi_{\mathbf{i}},\xi_1=\xi_{\mathbf{j}}} b\, db\, d\varepsilon . \tag{9}$$

It can be noted that $A'_{\mathbf{i},\mathbf{j},\mathbf{n}} = A'_{\mathbf{i}+\mathbf{m},\mathbf{j}+\mathbf{m},\mathbf{n}}$ for the uniform computational mesh, and all the velocities in the equation (9) for $A'_{\mathbf{i}+\mathbf{m},\mathbf{j}+\mathbf{m},\mathbf{n}}$ can be transformed to the velocities in the equation for $A'_{\mathbf{i},\mathbf{j},\mathbf{n}}$ by the shift of the coordinate

system on $-\mathbf{m}\,\Delta\boldsymbol{\xi}$. So the coefficients $A'_{\mathbf{i,j,n}}$ depends on **n** and on the difference **i–j**: $A'_{\mathbf{i,j,n}} = \hat{A}_{\mathbf{i-j,n}}$, and the size of the array $A'_{\mathbf{i,j,n}}$ can be reduced the size of velocity mesh times. The size of array $B_{\mathbf{ij}}$ can also be reduced in similar manner: $B_{\mathbf{ij}} = \hat{B}_{\mathbf{i-j}}$. Finally, the collision integral used in our computations has the following form:

$$I_{\mathbf{i}} = \sum_{\mathbf{j,n}} \hat{A}_{\mathbf{i-j,n}} f_{\frac{\mathbf{i+j+n}}{2}} f_{\frac{\mathbf{i+j-n}}{2}} - f_{\mathbf{i}} \sum_{\mathbf{j}} f_{\mathbf{j}} \hat{B}_{\mathbf{i-j}} \,. \tag{10}$$

The fundamental physical property of the collision integral is the existence of collision invariants. These invariants must be conserved in the numerical computation of the collision integral:

$$\sum_{\mathbf{i}} I_{\mathbf{i}} \varphi_\beta\left(\boldsymbol{\xi}_{\mathbf{i}}\right) \Delta\boldsymbol{\xi}_{\mathbf{i}} = 0, \varphi_\beta = 1, \xi_x, \xi_y, \xi_z, \left(\xi_x^2 + \xi_y^2 + \xi_z^2\right). \tag{11}$$

In order to maintain the collision invariants (11), the polynomial conservative correction proposed by Aristov and Tcheremissine [9] is used:

$$f_{\mathbf{i}}^* = f_{\mathbf{i}}\left[a_0 + a_x \xi_x + a_y \xi_y + a_z \xi_z + a_2\left(\xi_x^2 + \xi_y^2 + \xi_z^2\right)\right]. \tag{12}$$

The function (12) is substituted instead of $f_{\mathbf{i}}$ in front of the second summation in (10):

$$I_{\mathbf{i}} = \sum_{\mathbf{j,n}} \hat{A}_{\mathbf{i-j,n}} f_{\frac{\mathbf{i+j+n}}{2}} f_{\frac{\mathbf{i+j-n}}{2}} - f_{\mathbf{i}}\left[a_0 + a_x \xi_x + a_y \xi_y + a_z \xi_z + a_2\left(\xi_x^2 + \xi_y^2 + \xi_z^2\right)\right] \sum_{\mathbf{j}} f_{\mathbf{j}} \hat{B}_{\mathbf{i-j}} \,. \tag{13}$$

Substituting the collision integral (13) into (11) results in the system of 5 linear equations for 5 unknowns a_0, a_x, a_y, a_z and a_2 which can be easily solved.

The advantage of the considered method is the absence of statistical noise, which arises when semi-regular methods (Monte-Carlo procedures or Korobov's sequences) are used. A serious drawback of the method is its high computational cost (about $N^{8/3}$ operations, where N is the number of cells in velocity space). It is expected that the application of GPU can drastically accelerate these computations.

GPU COMPUTING AND CUDA IMPLEMENTATION

NVIDIA CUDA technology was taken as a tool to implement deterministic Boltzmann solver on GPU. In order to achieve good performance on a GPU the code must be well parallelized in a fashion similar to the traditional multi-CPU computing. Unfortunately, GPU architecture and CUDA technology impose additional requirements to the numerical algorithms making the process of effective code development quite different from the traditional parallel programming and introducing new "degrees of freedom" for tuning specific applications. These requirements are briefly discussed in the following paragraphs.

A CUDA-enabled device has a set of streaming multiprocessors (*e.g.* 30 multiprocessors for NVIDIA GeForce GTX 280). Each multiprocessor can support up to 1024 active threads. The smallest executable unit of parallelism on a device is called *warp* and consists of 32 threads. Within one warp multiple execution paths must be avoided, because branching leads to serialization of the code for a diverging path. Besides warp, a thread block must be considered. A thread block is a set of threads executed on one multiprocessor sharing some resources. So when developing the CUDA algorithm, a programmer must think not only in terms of threads but also in terms of warps and thread blocks.

Any device has access to several types of memory. One must distinguish a slow DRAM memory (global memory) and a fast on-chip memory (shared memory and registers). Global memory has high access latency (more than 400 clock cycles); otherwise shared memory and registers latency is 2 orders lower. The peak performance of operations with global memory is achieved when the memory is accessed in a coalesced manner (the threads of a warp access sequentially aligned memory blocks). The shared memory size is small – 16 Kb per thread block, while global memory size can be up to 1 Gb. One of the challenges in GPU programming is to replace uncoalesced and redundant accesses to global memory by accesses to shared memory when it is possible.

It has been recently shown (see [2]-[4]) that the use of NVIDIA CUDA technology can provide more than 2 orders speedup for Boltzmann solvers. The threads in these papers correspond to points in physical space. This approach suits well for GPU computing of collision integral because the same code is executed for different threads. The advection of the distribution function in physical space is also well processed by GPUs.

In the present paper we consider the possibility to compute the relaxation problem on GPUs. This problem is a fundamental part of any Boltzmann solver, so efficient GPU relaxation code could be a part of more complex solver. Although the relaxation problem seems less complex than solution of spatially inhomogeneous Boltzmann equation, it is not as well parallelized because the physical coordinate cannot be used as a thread ID parameter for obtaining the same code for a warp.

At first sight, selecting threads equivalent to points in velocity space seems to be natural for GPU computations of the collision integral by the deterministic method. Unfortunately, analysis shows that direct application of this approach does not provide performance benefits. The reasons are the unstructured access to arrays of coefficients which are stored in global memory and multiple execution paths.

The algorithm of collision integral computations has been optimized for the GPU computations in the following manner. In order to reduce accesses to global memory thread blocks were organized in such a way that each thread block processed the part of the collision integral with the first components of the 3D indices **i**, **j** and **n** in (10) being constant inside a thread block. This results in that the first components of all 3D indices in (10) being constant inside a thread block. The part of the distribution function with a fixed first component of index is stored in shared memory.

The CUDA code of a kernel for computation of the part of the collision integral, corresponding to collision frequency (the last sum in (10)) is presented in Listing 1. This code is created for the simplest case of a rectangular 3D computational domain where the range of points in each coordinate is constant and independent of other coordinates. The kernel writes the results to a buffer. The buffer is large enough to guarantee that no writing conflict will happen.

```
__global__ void compute_frequency (float * f, float * b, float * frequency_buffer)
{
  short int i_x, i_y, i_z, j_x, j_y, j_z, m_x, m_y, m_z, index;
  __shared__ float f_shared[N_YZ];
  __shared__ float b_shared[N_YZ];
  __shared__ float integral_shared[N_YZ];

  i_x = blockIdx.x; //first components of indices are equivalent to block ID
  j_x = blockIdx.y;

  for (index = threadIdx.x; index < N_YZ; index+= blockDim.x)    // copy the distribution
    f_shared[index] = f[j_x * N_YZ + index];                     // function to shared memory

  m_x = (i_x > j_x) ? i_x - j_x : j_x - i_x; // abs (i - j)

  for (index = threadIdx.x; index < N_YZ; index+= blockDim.x)   // copy the array of coeffi-
    b_shared[index] = b[m_x * N_YZ + index];                    // cients to shared memory

  __syncthreads ();

  for (index = threadIdx.x; index < N_YZ; index+= blockDim.x) { // thread ID corresponds to
    int index1;                                                 // external index i
    integral_shared[index] = 0;
    i_y = y_index (index);
    i_z = z_index (index);
    for (index1 = 0; index1 < N_YZ; index1++) {   // inside a thread summation on j is done
      j_y = y_index (index1);
      j_z = z_index (index1);

      m_y = i_y > j_y ? i_y - j_y : j_y - i_y;
      m_z = i_z > j_z ? i_z - j_z : j_z - i_z;
      int m_index = get_m_index (m_y, m_z);       //get 1 index from 2 components

      integral_shared[index] += b_shared[m_index] * f_shared[index1];
    }
    frequency_buffer[(j_x + i_x * N_X) * N_YZ + index] = //store results to global memory
      integral_shared[index];
  }
}
```

Listing 1. CUDA kernel for computations of the collision frequency.

The kernel for computing the inverse collision integral is presented in Listing 2. It is the main part of the code, because computations of the inverse collision integral by the deterministic method take more than 99% of time. The structure of the code is similar to that of Listing 1. One should mention that the shared memory buffer is updated

twice during the internal loop. This is due to the fact that the components of the index **n** could be both positive and negative. The results are stored in a global memory buffer of a size large enough to avoid memory writing conflicts.

```
__global__ void compute_inverse_collision_integral (float * f, float * a,
                                                    float * inverse_integral_buf)
{
  short int i_x, i_y, i_z, j_x, j_y, j_z, n_x, n_y, n_z, m_x, m_y, m_z, index;
  short int k_x, k_y, k_z, l_x, l_y, l_z;
  __shared__ float f1_shared[N_YZ];
  __shared__ float f2_shared[N_YZ];
  __shared__ float a_shared[N_YZ];
  __shared__ float integral_shared[N_YZ];

  j_x = blockIdx.x % N_X; //first components of indices are equivalent computed from block ID
  i_x = blockIdx.y;
  n_x = (blockIdx.x / N_X) << 1 | ((i_x + j_x) & 1);
  k_x = (i_x + j_x + n_x) >> 1;  l_x = (i_x + j_x - n_x) >> 1;

  for (index = threadIdx.x; index < N_YZ; index+= blockDim.x)    // copy the distribution
    f1_shared[index] = f[k_x * N_YZ + index];                    // function to shared memory
  for (index = threadIdx.x; index < N_YZ; index+= blockDim.x)
    f2_shared[index] = f[l_x * N_YZ + index];

  for (index = threadIdx.x; index < N_YZ; index+= blockDim.x) // initialization of shared
    integral_shared[index] = 0.0f;    // memory buffer for computing the collision integral

  m_x = (i_x > j_x) ? i_x - j_x : j_x - i_x; // abs (i - j)

  __syncthreads ();

  for (index = 0; index < N_YZ; index++) { // summation on n loop
    int index1, i_index, k_index, l_index;
    n_y = ny_index (index);
    n_z = nz_index (index);

    for (index1 = threadIdx.x; index1 < N_YZ; index1 += blockDim.x) // copy array a to shared
      a_shared[threadIdx.x] = a[get_index_in_a (m_x, n_x, index, threadIdx.x)]; // memory

    for (i_index = threadIdx.x; i_index < N_YZ; i_index += blockDim.x) { // thread ID is
      i_y = y_index (i_index); // defines the point i in which collision integral is computed
      i_z = z_index (i_index);

      for (j_y = (n_y | i_y) & 1; j_y < N_Y; j_y += 2) //the sum n_ + i_ + j_ must be even
       for (j_z = (n_z | i_z) & 1; j_z < N_Z; j_z += 2) { // summation on j loop
        m_y = i_y > j_y ? i_y - j_y : j_y - i_y;
        m_z = i_z > j_z ? i_z - j_z : j_z - i_z;
        int m_index = get_m_index (m_y, m_z);       //get 1 index from 2 components

        k_y = (i_y + j_y + n_y) >> 1; // the indices
        l_y = (i_y + j_y - n_y) >> 1; // of post-collision
        k_z = (i_z + j_z + n_z) >> 1; // velocities
        l_z = (i_z + j_z - n_z) >> 1;

        k_index = get_1D_index_from_y_and_z (k_y, k_z);
        l_index = get_1D_index_from_y_and_z (l_y, l_z);
        integral_shared[i_index] += a_shared[m_index] *
          (f1_shared[k_index] * f2_shared[l_index] + // accumulating the collision integral
           f1_shared[l_index] * f2_shared[k_index]); //to shared memory buffer

        k_index = get_1D_index_from_y_and_z (l_y, k_z);
        l_index = get_1D_index_from_y_and_z (k_y, l_z);
        integral_shared[i_index] += a_shared[m_index] *
          (f1_shared[k_index] * f2_shared[l_index] + // accumulating the collision integral
           f1_shared[l_index] * f2_shared[k_index]); //to shared memory buffer
      }
    }
  }
  for (index = threadIdx.x; index < N_YZ; index+= blockDim.x)
    inverse_integral_buf[(i_x + blockIdx.x * N_X) * N_YZ + index] =
      integral_shared[index];  // copy the results from shared memory to global memory
}
```

Listing 2. CUDA kernel for computing the inverse collision integral.

After the kernels have completed their work the data from buffers are collected using a simple kernel which sums the parts of the buffer corresponding to the same velocity points. Then the conservative correction procedure (11)-(13) is executed. The moments of the collision integral, which are necessary for the conservative correction are computed by the kernel. The linear system for the 5 unknowns a_0, a_x, a_y, a_z, and a_2 is solved in a serial manner. The final value of the collision integral is obtained with the formula (13) by a kernel, which corrects the collision integral.

The results of computations with the developed code with the use of NVIDIA GeForce GTX 295 GPU show the 40 times speedup for the computational mesh 20x20x20 with respect to the CPU version of the program executed on Intel Core 2 3GHz processor. This shows that the speedup of the proposed method is not as great as in the papers [2]-[4] but nevertheless the GPU in this case are several times more efficient than the same price CPU computing.

SOLUTIONS OF TEST PROBLEMS

Collisional relaxation of a bi-Maxwellian distribution was studied to explore the GPU acceleration process. The initial distribution is sum of 2 Maxwellians with different mean velocities: f=exp{-(ξ-u_1)^2}+exp{-(ξ-u_2)^2}, u_1.=(1.75, 0.25, 0), u_2.=(-1.25, 0.25, 0). Nondimensional Boltzmann equation is considered with the unit factor in front of the collision integral and b=1. The temporal evolution of the velocity distribution function during the relaxation is presented in Fig. 1. The time step is selected automatically equal to a half of inverse collision frequency. The left plot shows the evolution of the distribution function in the cross-section ξ_y=0.25, ξ_z=0, the right plot corresponds to the cross-section ξ_x=0.25, ξ_z=0. The numerical error of collision integral approximation is checked for Maxwell distribution functions for inverse and direct collision integrals separately. The difference of numerical and analytical values is equal to 0.5%. As the relaxation time is about 10 the error of the solution is about 5% during relaxation.

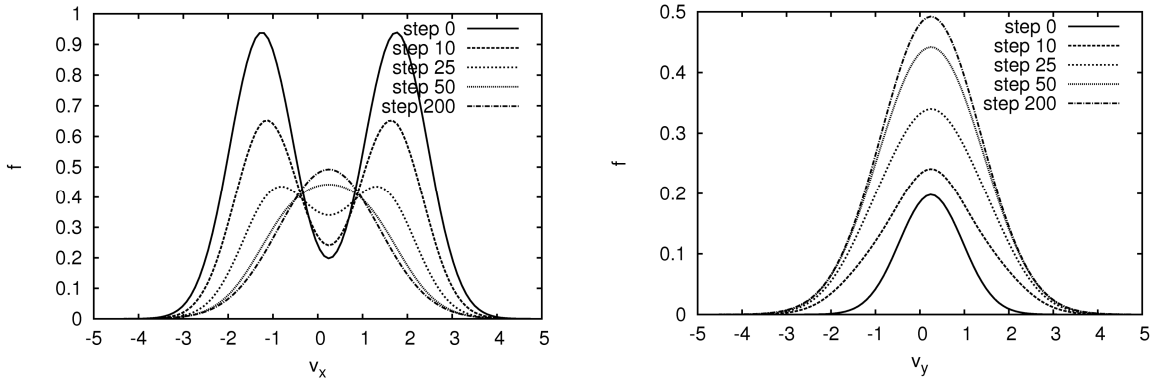

FIGURE 1. Relaxation problem. The evolution of the distribution function.

ACKNOWLEDGEMENTS

This work was supported by the Program No 2 of the Presidium of the Russian Academy of Sciences.

REFERENCES

1. P. G. Howard, "GPU Computing: More exciting times for HPC!", available at http://www.microway.com/pdfs/microway_**GPU**_whitepaper_2009-05.pdf.
2. Yu. Yu. Kloss, F. G. Tcheremissine and P.V. Shuvalov, *Numer. Meth. And Programming* **11**, 144-152 (2010) (in Russian).
3. A. Frezzotti, G. P. Ghiroldi and L. Gibelli, "Solving Kinetic Equations on GPUs I: Model Kinetic Equations", arXiv:0903.4044v1.
4. A. Frezzotti, G. P. Ghiroldi and L. Gibelli, "Solving the Boltzmnn Equation on GPU", arXiv:1005.5405v1.
5. V. V. Aristov, Direct methods for solving the Boltzmann equation and study nonequilibrium flows, Kluwer Academic Publishers, Dordrecht, 2001.
6. V. V. Aristov and S. A. Zabelok, *Comp. Math. Math. Phys.*, **42**, 406-418 (2002).
7. V. V. Aristov and S. A. Zabelok. *Math. Model.*, **14**, 5-9 (2002).
8. V. V. Kolobov, R. R. Arslanbekov, V. V. Aristov, A. A. Frolova and S. A. Zabelok, *J. Comp. Phys.*, **223**, 589-608 (2007).
9. V. V. Aristov and F. G. Tcheremissine, *USSR Comput. Math. Phys.*, **20**, 208-225 (1980).