Parallelization of algorithms for solving the Boltzmann equation for GPU-based computations

E.A. Malkov and M.S. Ivanov

Khristianovich Institute of Theoretical and Applied Mechanics SB RAS (Russia, Novosibirsk)

Abstract. The paper describes specific features of parallelization of collision integral computation algorithms, which are conditioned by the CUDA architecture of parallelization on graphic cards [1].

Keywords: Boltzmann equation, collision term, GPU, CUDA.
PACS: 47.45.-n

INTRODUCTION

Graphic accelerators are widely used for various computations in research and engineering because of rapid growth of their performance and low cost. Another favorable factor is the CUDA architecture of general-purpose parallel computations with a convenient application programming interface (API), which was developed by NVIDIA. With this CUDA architecture, writing programs for graphic cards has become a simple routine for application programmers. An avalanche of publications with results of CUDA testing in various fields of science and engineering appeared during the last two years. These activities are aimed at determining the possibilities of parallelization of conventional numerical algorithms within the framework of the CUDA architecture. The difficulties in parallelization on graphic processors are caused by specific features of their architecture. First, the graphic card implements the so-called SIMD – computer. This means that the same operation with different data is performed simultaneously on all multiprocessor cores under control of one controller. Parallelization based on such principles is much less flexible than parallelization on MIMD – computers with independent threads or processes. The second problem in GPU-based parallelization is sharing of the large-size global memory common for all threads (there is also a memory distributed among all groups-blocks, but its size is small). This factor imposes a significant constraint on the number of simultaneously executed threads, because the total volume of read/write data is determined by the data bandwidth of the global memory. If the bandwidth does not provide fast data exchange, then code execution is substantially slowed down. Moreover, the task may be canceled by the operation system (by the watchdog) if the waiting time of some thread exceeds a certain value. Another memory-related drawback is the absence of automatic caching during data writing. Therefore, parallelization should organize threads in a manner to increase the computation time and to reduce the data transfer time, with the use of moderate-size distributed memory for intermediate storage. If it is difficult to do, then the number of simultaneously executed threads has to be limited. Despite these drawbacks, the CUDA architecture offers some useful features for application programmers, for instance, the real number indexing of memory for reading considered below. A bottleneck in the numerical solution of the Boltzmann equation is computing the collision integral because of high multiplicity of integration and the requirement to conserve the phase density, energy, and momentum, which are invariants of the collision operator. Possible methods of parallelization of collision integral computation within the CUDA model, which take into account the specific features of this architecture, are considered in the paper.

TWO APPROACHES TO COMPUTING THE COLLISION INTEGRAL

If the angle θ between the vectors of the relative velocity of molecules before and after the collision is chosen as a collision parameter, then the collision integral is written as

$$
\begin{aligned}
&St(\vec{u}) = G(\vec{u}) - L(\vec{u}),\\
&G(\vec{u}) = \int\limits_{R^3} d^3u_1 \int\limits_{S^2} d^2n f f_1' \left|\vec{V}\right| \sigma\left(V, \cos(\theta)\right),\\
&L(\vec{u}) = \int\limits_{R^3} d^3u_1 \int\limits_{S^2} d^2n f f_1 \left|\vec{V}\right| \sigma\left(V, \cos(\theta)\right),
\end{aligned}
\tag{1}
$$

where

$$
\begin{aligned}
&f = f(t, \vec{r}, \vec{u}), \quad f_1 = f(t, \vec{r}, \vec{u}_1),\\
&f' = f(t, \vec{r}, \vec{u}'), \quad f_1' = f(t, \vec{r}, \vec{u}_1'),\\
&\vec{u}' = \frac{\vec{u} + \vec{u}_1}{2} + \frac{\left|\vec{V}\right|}{2}\vec{n},\\
&\vec{u}_1' = \frac{\vec{u} + \vec{u}_1}{2} - \frac{\left|\vec{V}\right|}{2}\vec{n},
\end{aligned}
\tag{2}
$$

$\vec{V} = \vec{u} - \vec{u}_1$, $\vec{n}$ is the unit vector (collision parameter) defining the direction of the relative velocity after the collision, and $\sigma\left(V, \cos(\theta)\right)$ is the differential scattering cross section. The variables t, $\vec{r}$ are included into the expression for the collision integral as parameters of the distribution function and may be omitted in the context of the collision integral description. The integral of direct collisions $L(\vec{u})$ determines the sink at the point $\vec{u}$ of the velocity space, whereas $G(\vec{u})$ determines the inflow to this point. Expressions (1) can be rewritten in symmetric form as [2]

$$
\begin{aligned}
&St(\vec{u}) = G(\vec{u}) - L(\vec{u})\\
&G(\vec{u}) = \frac{1}{2}\int\limits_{R^3} d^3u_1 \int\limits_{R^3} d^3u_2 \int\limits_{S^2} d^2n f(\vec{u}_1) f(\vec{u}_2)\\
&\qquad \times \left(\delta\left(\vec{u}_1' - \vec{u}\right) + \delta\left(\vec{u}_2' - \vec{u}\right)\right)\left|\vec{V}\right| \sigma\left(V, \cos(\theta)\right),\\
&L(\vec{u}) = \frac{1}{2}\int\limits_{R^3} d^3u_1 \int\limits_{R^3} d^3u_2 \int\limits_{S^2} d^2n f(\vec{u}_1) f(\vec{u}_2)\\
&\qquad \times \left(\delta\left(\vec{u}_1 - \vec{u}\right) + \delta\left(\vec{u}_2 - \vec{u}\right)\right)\left|\vec{V}\right| \sigma\left(V, \cos(\theta)\right).
\end{aligned}
\tag{3}
$$

Forms (1) and (3) determine two different approaches to finding the collision integral numerically, which can be briefly described as follows. In the first approach (see, e.g., [3]), five-dimensional integrals $L(\vec{u}_\alpha)$ and $G(\vec{u}_\alpha)$ are estimated for each node point of the computational grid in the velocity space $\vec{u}_\alpha$ (α is the node index) numerically, with the use of statistical or regular methods. The values of the distribution function at points between the nodes are found by means of interpolation. In the second approach (see, e.g., [4]), the value of $\Delta m = f(\vec{u}_\alpha) f(\vec{u}_\beta) \Delta^6 u \Delta^2 n \left|\vec{V}_{\alpha,\beta}\right| \sigma\left(V_{\alpha,\beta}, \cos(\theta_\gamma)\right)$ is determined for each pair of nodes (α, β) of the computational grid and direction $\vec{n}_\gamma$; this value is added to $L(\vec{u}_\alpha)$, $L(\vec{u}_\beta)$, $G(\vec{u}_{\alpha'})$, and $G(\vec{u}_{\beta'})$, where α', β' are the velocity grid node indices corresponding to molecular velocities after the collision with the parameter $\vec{n}_\gamma$. If the post-collision velocities are not in grid nodes, then Δm is distributed between nodes α_i', β_i' located at the shortest distance from (α', β') so that the following relations are valid:

$$\Delta m = \sum_i \Delta m_i,$$
$$(|\vec{u}_{\alpha'}|^2 + |\vec{u}_{\beta'}|^2)\Delta m = \sum_i |\vec{u}_{\alpha'_i}|^2 \Delta m_i + \sum_i |\vec{u}_{\beta'_i}|^2 \Delta m_i, \quad (4)$$
$$(\vec{u}_{\alpha'} + \vec{u}_{\beta'})\Delta m = \sum_i \vec{u}_{\alpha'_i} \Delta m_i + \sum_i \vec{u}_{\beta'_i} \Delta m_i.$$

These algorithmic schemes can be subjected to (different degrees of) parallelization within the CUDA architecture.

ADVANTAGES OF TEXTURE MEMORY

In applying the first approach to compute the collision integral, one should use the so-called texture memory to store the values of the distribution function in the computational grid nodes; in the CUDA architecture, this is the cached memory for reading. In addition to caching, the texture memory possesses one more useful property beginning from CUDA 2.0, which is three-dimensional the real number indexing [5]. This means that interpolation in internodal points of the velocity grid with the use of the texture memory is performed by hardware. Unfortunately, this refers only to trilinear and stepwise interpolation. Some code fragments with comments are given below to illustrate the basic principles of working with the texture memory in the context of computing the collision integral.

```
/**********************Operation with texture*************************************/
texture<float,3, cudaReadModeElementType> df_tex;
__constant__  float4 Om_c[N_DIRECT];

__global__   void  stoss(…){
……..……
V=sqrtf( (vx-v1x)*(vx-v1x)+(vy-v1y)*(vy-v1y)+(vz-v1z)*(vz-v1z) );

for( int iOm=0; iOm<N_DIRECT;iOm++){

  vpx=vx+v1x - V*Om_c[iOm].x;  vpx/=2;
  vpy=vy+v1y - V*Om_c[iOm].y;  vpy/=2;
  vpz=vz+v1z - V*Om_c[iOm].z;  vpz/=2;

  v1px=vx+v1x + V*Om_c[iOm].x;  v1px/=2;
  v1py=vy+v1y + V*Om_c[iOm].y;  v1py/=2;
  v1pz=vz+v1z + V*Om_c[iOm].z;  v1pz/=2;
/********Using real indexing during texture memory reading **********/
  s+=( tex3D(df_tex, (vpx-vmin)/(vmax-vmin),  (vpy-vmin)/(vmax-vmin), (vpz-vmin)/(vmax-vmin))*
        tex3D(df_tex, (v1px-vmin)/(vmax-vmin), (v1py-vmin)/(vmax-vmin),(v1pz-vmin)/(vmax-vmin)))
*V*Om_c[iOm].w;
  }
……
}

void Stoss(double* df_h, …){
……
/**************************************Creation of a CUDA array************************/
const cudaExtent volumeSize = make_cudaExtent(n, n, n);

/**************************************Creation of a CUDA array************************/
cudaArray* df_Array=0;
cudaChannelFormatDesc  channelDesc=cudaCreateChannelDesc<float>();
cudaMalloc3DArray(&df_Array, &channelDesc, volumeSize);
/*************************************************************************************/
```

```
/*****************Copying data to the CUDA array*****************************/
 cudaMemcpy3DParms  cpyParams={0};
 cpyParams.srcPtr=make_cudaPitchedPtr( (void*)df_h,
 volumeSize.width*sizeof(float),  volumeSize.width,  volumeSize.height);
 cpyParams.dstArray=df_Array;
 cpyParams.extent=volumeSize;
 cpyParams.kind=cudaMemcpyHostToDevice;
 cudaMemcpy3D(&cpyParams);
/*****************************************************************************/
/*******************Tuning the texture parameters*****************************/
 df_tex.normalized=true;
 df_tex.filterMode=cudaFilterModeLinear;  //trilinear real interpolation
 df_tex.addressMode[0]=cudaAddressModeClamp;
 df_tex.addressMode[1]=cudaAddressModeClamp;
 df_tex.addressMode[2]=cudaAddressModeClamp;
/*****************************************************************************/
/*********************Binding of the CUDA array to the texture*****************/
  cudaBindTextureToArray(df_tex, df_Array, channelDesc);
/*****************************************************************************/

……
  cudaUnbindTexture(df_tex);
  cudaFreeArray(df_Array);
}
```

USING CONSTANT MEMORY

In both approaches to computing the collision integral, the data that are not changed from on iteration to another should also be stored in the cached read-only memory, i.e., constant memory. A code fragment with comments, which illustrates working with the constant memory by an example of storage of the field of scattering directions, is given below.

```
__constant__  float4 Om_c[N_DIRECT];

__global__ void stoss(…){……}

/****************Array of directions on the host side*********************************/
  float4 Om_h[N_DIRECT];

/******** Procedure of forming a four-dimensional array containing the direction vectors and the unit sphere area elements in the case of an anisotropic field of directions*****************/
  setDirections(Om_h);

/*****Copying the array of directions to the constant memory of the graphic unit**/
  cudaMemcpyToSymbol(Om_c, Om_h, N_DIRECT*sizeof(float4));
/*****************************************************************************/
```

CONTROLLING THE NUMBER OF THREADS

The core function copied to each thread should be implemented so that the number of simultaneously executed threads is not too large, bearing in mind the constraint imposed by the memory bandwidth. A code fragment organizing nontrivial controlling of the number of threads by means of distributing the load between them is given below.

```
__global__ void stoss(double* shuttle, int n, int flight, int portion) {
   ……
   int idx = blockIdx.x*blockDim.x+threadIdx.x;
   int this_knot=flight*portion+blockIdx.x;

   int i,j,k;
   int i1,j1,k1;

   double vx, vy, vz;
   double v1x, v1y, v1z;
   double V;

   int th=threadIdx.x%blockDim.x;
   int quota=n*n*n/blockDim.x;

   i=this_knot/(n*n);
   j=(this_knot-i*n*n)/n;
   k=this_knot-i*n*n-j*n;

   vx=vmin+(i+0.5)*h; vy=vmin+(j+0.5)*h; vz=vmin+(k+0.5)*h;

   double s=0.0;
   for(int knot_counter=quota*th; knot_counter<quota*(th+1); knot_counter++){
     i1=knot_counter/(n*n);
     j1=(knot_counter-i1*n*n)/n;
     k1=knot_counter-i1*n*n-j1*n;

     v1x=vmin+(i1+0.5F)*h; v1y=vmin+(j1+0.5F)*h; v1z=vmin+(k1+0.5F)*h;

     V=sqrtf( (vx-v1x)*(vx-v1x)+(vy-v1y)*(vy-v1y)+(vz-v1z)*(vz-v1z) );
……
   }
……..
shuttle[idx]=s;
}
void Stoss(double* df_h, …){
……
/*********************Preparing the shuttle*******************************/
    size_t size_of_shuttle=portion*threads_per_knot*sizeof(double);
    double* shuttle = (double *)malloc(size_of_shuttle);
    double* shuttle_device;
    cudaMalloc((void **) &shuttle_device, size_of_shuttle );
    for(int i=0;i<portion*threads_per_knot;i++) shuttle[i]=0.0F;
    cudaMemcpy(shuttle_device,shuttle, size_of_shuttle, cudaMemcpyHostToDevice);
/*****************************Basic cycle**************************/
 for(int flight=0; flight<N/portion;flight++){
     stoss <<< num_of_blocks, threads_per_block>>>(shuttle_device, n, flight,portion);
      cudaThreadSynchronize();
/****************Shuttle withdrawal************************************/
     cudaMemcpy(shuttle, shuttle_device, size_of_shuttle, cudaMemcpyDeviceToHost);
     for (int j=0; j<portion; j++)
         for(int i=threads_per_knot*j;  i<threads_per_knot*(j+1); i++)
                                              Gain[j+flight*portion]+=shuttle[i];
 }
}
```

RESULTS OF TESTING PARALLELIZATION SCHEMES

As an example of using the above-discussed principles of parallelization of collision integral computations in a graphic unit, homogeneous relaxation of a gas of Maxwelling molecules with the initial distribution corresponding to Bobylev's analytical solution [6] was computed. A GeForce GTX 280 video card with a processor frequency of 1.3 GHz was used in the computations. The parallel algorithm turned out to be 80 times more efficient than the non-parallel algorithm executed on an Intel x86 processor with a frequency of 2.4 GHz. This increase in efficiency allows a dense uniform grid to be used in computing the collision integral. Figure 1 shows the normalized moments of the distribution function $m_p = \langle u^p \rangle / (p+1)!!$ computed on a 64x64x64 grid; the solid curves show the theoretically predicted behavior of the moments. No special corrections to ensure satisfaction of conservation laws were applied during the computations. So, energy slow increases during the relaxation time. The "correct" conformity of the moments, normalized by energy, with the theoretical curves is ensured by the high dimension of the computational grid. It should be noted that efficient parallelization of collision integral computations based on the second approach, which would rather simply satisfy the requirement of conservation of the collision operator invariants, has not yet been obtained.

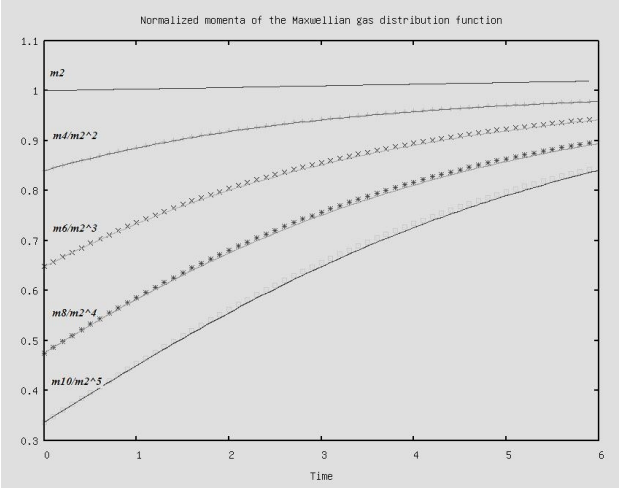

FIGURE 1. Normalized high-order moments (homogeneous relaxation of a gas of Maxwellian molecules)

REFERENCES

1. http://www.nvidia.com/object/cuda_home.html
2. C. Cercignani The Boltzmann equation and its applications. Springer-Verlag, New York, 1988.
3. Ibraghimov I, Rjasanow S. Numerical solution of the Boltzmann equation on the uniform grid. Computing 2002; 69(2):163–186.
4. P.L. Varghese, «Arbitrary post-collision velocities in a discrete velocity scheme for the Boltzmann equation», Rarefied Gas Dynamics: Proceedings of the 25th International symposium, 2007, pp. 227-232.
5. NVIDIA CUDA Programming Guide version 2.0, 2008..
6. Bobylev A.V. Exact solutions of the Boltzmann equation and Maxwell gas relaxation theory // TMF. Vol.60. No.2. P.280-309.