Computers and Mathematics with Applications 59 (2010) 2380–2392

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

LBM based flow simulation using GPU computing processor

Frédéric Kuznik*, Christian Obrecht, Gilles Rusaouen, Jean-Jacques Roux

Université de Lyon, CNRS, INSA-Lyon, CETHIL, UMR5008, F-69621, Villeurbanne, France

ARTICLE INFO

Keywords:
Lattice Boltzmann method
Graphics Processing Unit
Fluid flow
Multi-threaded architecture

ABSTRACT

Graphics Processing Units (GPUs), originally developed for computer games, now provide computational power for scientific applications. In this paper, we develop a general purpose Lattice Boltzmann code that runs entirely on a single GPU. The results show that: (1) simple precision floating point arithmetic is sufficient for LBM computation in comparison to double precision; (2) the implementation of LBM on GPUs allows us to achieve up to about one billion lattice update per second using single precision floating point; (3) GPUs provide an inexpensive alternative to large clusters for fluid dynamics prediction.

© 2009 Elsevier Ltd. All rights reserved.

1. Introduction

Nowadays, computational methods and related hardware are really inseparable. In fact, the numerical method must fit the hardware architecture to gain benefits from computational possibilities. Of course, the reciprocal is also true: the hardware architecture progress lead the numerical methods that can be used with a reasonable computational cost.

In the last two decades, the Lattice Boltzmann method (LBM) has proved its capability to simulate a large variety of fluid flows [1–5]. . . . However, it has been recognized that the LBM is both computationally expensive and memory demanding [6]. But, because LBM is explicit and generally needs only nearest neighbor information, the method allows a highly efficient parallel implementation using GPU architecture [7,8].

Graphics Processing Unit (GPU) is a massively multi-threaded architecture and then is widely used for graphical and now non-graphical computations [9]. The main advantage of GPUs is their ability to perform significantly more floating point operations (FLOPs) per unit time than a CPU (see Fig. 1).

Fan et al. [10] used a 32 nodes cluster made of nVIDIA GeForce 5800 ultra for LBM computations. They use the GPU vector operations and stacks of 2D textures for 3D computations. With 32 nodes, they found an efficiency of their implementation of 49.2 Million Lattice Update Per Second (MLUPS). Tölke [11] used an nVIDIA 8800 Ultra graphics card to implement a 2D LBM code. He found very good results with a ratio between GPU time and CPU time of about 23 for the same test case.

In this paper we provide an implementation of a general purpose LBM code where all steps of the algorithm are running on the GPU. This implementation is made possible by the use of the nVIDIA CUDA C language programming environment. CUDA provides low level hardware access, avoiding the limitations imposed in fragment shaders. It works on the GT200 processor from nVIDIA, and will be supported on future devices [12]. Algorithms developed for this work will be directly applicable to newer, faster GPUs as they become available.

2. Lattice Boltzmann method

This part is devoted to an overview of the lattice Boltzmann model used for the purpose of this study (LBM).The model is the lattice BGK model (LBGK) from Qian, D'Humières and Lallemand [13]. The main hypothesis of the LBGK are:

- Bhatnagar, Gross and Krook approximation (BGK) $\Rightarrow$ the collision operator is expressed as a single relaxation time to the local equilibrium,

* Corresponding author. Tel.: +33 472 438 461; fax: +33 472 438 522.
E-mail address: frederic.kuznik@insa-lyon.fr (F. Kuznik).

0898-1221/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2009.08.052

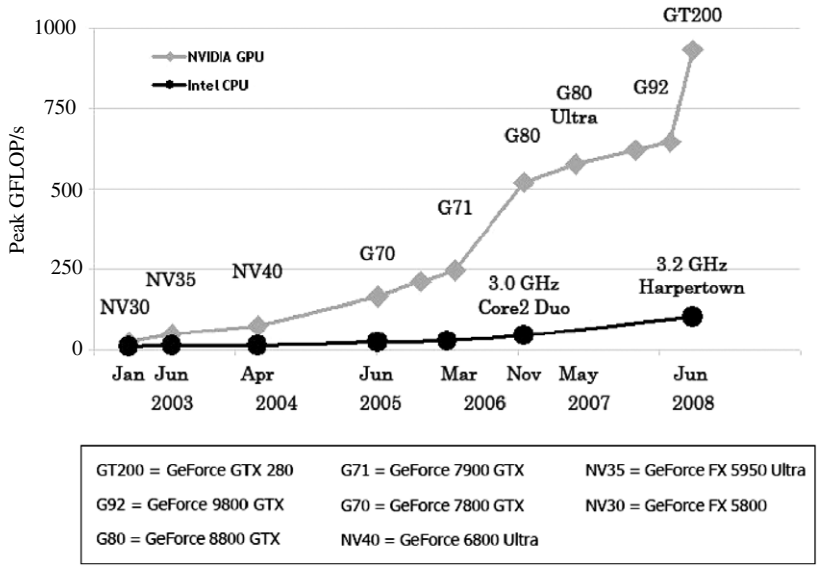

Fig. 1. Performances of CPUs (circle) and GPUs (diamond) over the last few years — extracted from [12].

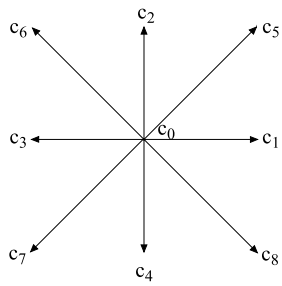

Fig. 2. The square lattice velocities *D2Q*9.

- the Knudsen number is assumed to be a small parameter,
- the flow is incompressible.

The evolution of the density distribution f for a single fluid particle is then given by:

$$\frac{Df}{Dt} = \partial_t f + \left(\vec{\xi}.\nabla\right) f = -\frac{f - f^e}{\tau} \tag{1}$$

$\vec{\xi}$ is the microscopic velocity, τ is the relaxation time and f^e the Maxwell–Boltzmann equilibrium distribution function. The macroscopic variables such as density ρ and velocity $\vec{u}$:

$$\rho\left(\vec{x}, t\right) = \int f\left(\vec{x}, \vec{\xi}, t\right) \mathrm{d}\vec{\xi} \tag{2}$$

$$\rho\left(\vec{x}, t\right) \vec{u}\left(\vec{x}, t\right) = \int \vec{\xi} f\left(\vec{x}, \vec{\xi}, t\right) \mathrm{d}\vec{\xi}. \tag{3}$$

To obtain the Lattice Boltzmann model, the velocity space must be discretized: during $\mathrm{d}t$, the distribution function moves along the lattice link $\mathrm{d}\vec{x}_i = \vec{c}_i \mathrm{d}t$. In our simulations, a 9 velocities 2 dimensional (D2Q9) lattice has been used (Fig. 2).

After discretization, the evolution equation becomes (Fig. 3):

$$f_i\left(\vec{x} + \vec{c}_i \mathrm{d}t, t + \mathrm{d}t\right) + f_i\left(\vec{x}, t\right) = -\frac{1}{\tau}\left(f_i - f_i^e\right). \tag{4}$$

The macroscopic variables such as density ρ and velocity $\vec{u}$ are then given by:

$$\rho = \sum_{i=0}^{i=8} f_i \tag{5}$$

$$\rho\vec{u} = \sum_{i=0}^{i=8} \vec{c}_i f_i. \tag{6}$$

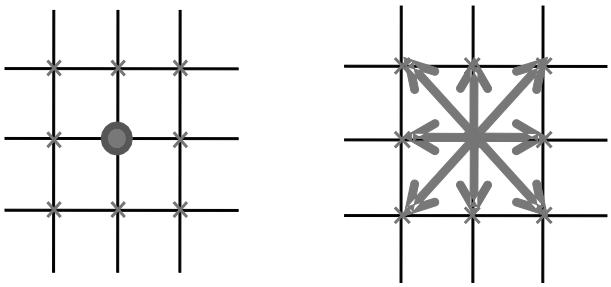

Fig. 3. LBM partition: collision step (left) and propagation step (right).

For the 2D, applying the third-order Gauss–Hermite quadrature leads to the D2Q9 model with the following discrete velocities $\vec{c}_i$ where $i = 1 \ldots 8$. The discrete velocities are $\overrightarrow{c_0} = (0, 0)$, $\overrightarrow{c_1} = -\overrightarrow{c_3} = c(1, 0)$, $\overrightarrow{c_2} = -\overrightarrow{c_4} = c(0, 1)$, $\overrightarrow{c_5} = -\overrightarrow{c_7} = c(1, 1)$, $\overrightarrow{c_6} = -\overrightarrow{c_8} = c(-1, 1)$.

The equilibrium density distribution function is given by:

$$f_i^e = \omega_i \rho \left[1 + 3\frac{\vec{c}_i.\vec{u}}{c^2} + 4.5\frac{(\vec{c}_i.\vec{u})^2}{c^4} - 1.5\frac{\left(u^2 + v^2\right)}{c^2} \right] \tag{7}$$

with $\vec{u} = (u, v)$ and $\omega_0 = 4/9$, $\omega_i = 1/9$ for $i = 1, 2, 3, 4$, $\omega_i = 1/36$ for $i = 5, 6, 7, 8$.

LBM can then be split into collision and propagation steps:

(1) collision:

$$f_i\left(\vec{x}, t^*\right) = -\frac{1}{\tau}\left(f_i - f_i^e\right) \tag{8}$$

(2) propagation:

$$f_i\left(\vec{x} + \vec{c}_i \mathrm{d}t, t + \mathrm{d}t\right) = f_i\left(\vec{x}, t^*\right). \tag{9}$$

Of course, the third step necessary for the implementation of LBM is the determination of the boundary conditions. There are two types of boundary conditions for the case tested in our study: wall boundary condition and imposed velocity boundary condition. For the walls, the classical no-slip boundary condition is imposed by the means of bounce-back rules. A prescribed velocity is easily implemented by constantly refilling the boundary nodes with the equilibrium population corresponding to the desired value of flow speed.

The collision step, which is totally local, required about 70% of the total computational time. The propagation step required 28% of the total computation time. On the whole, 98% of the computational time can easily be parallelized. (these values are extracted from [14]).

3. Programming overview

3.1. Hardware architecture

This paragraph is dedicated to a description of the hardware architecture (Fig. 4).The GPU computing processor hardware is the nVIDIA GTX280 which can easily be included in a standard workstation. The processor is composed of ten thread processing clusters (TPCs), with each broken down into three streaming multiprocessors (SMs). Threads are assigned by the thread scheduler, which addresses directly to each streaming multiprocessor through a dedicated instruction unit; the later then assigns tasks to one of the eight thread (or stream) processors (SPs). On the whole, 1 GPU is composed of 240 processors.

The memory bandwidth is 141.7 GigaByte per second and the available amount of memory is 1.0 GigaByte. The GPU can deliver about 1000 GFLOPS (Giga Floating Operations per Second) which corresponds to about 80×86 CPU.

3.2. CUDA overview

CUDA (Nvidia) is a standard C language extension for parallel application development on the GPU, independently of the hardware target. Some definitions are necessary to understand the CUDA programming features:

- the device is the GPU,
- the host is the CPU,
- the kernel is a function that is called from the host and runs on the device,
- a CUDA kernel is executed by an array of threads (see Fig. 5).

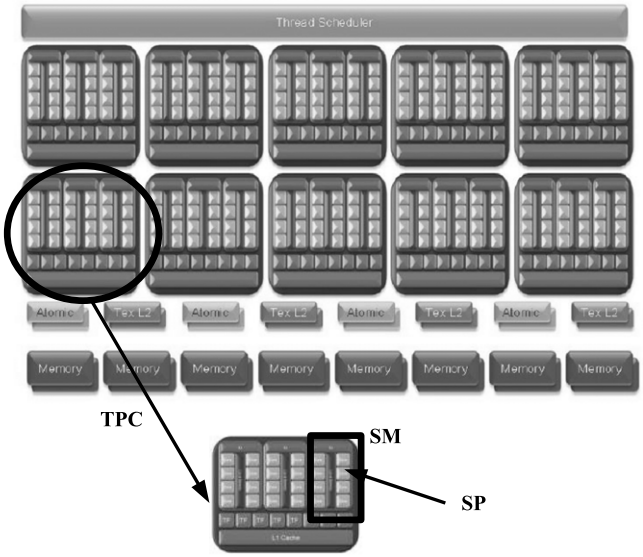

Fig. 4. GPU hardware architecture overview.

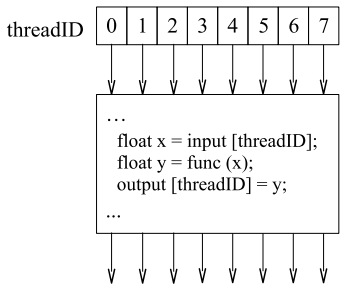

Fig. 5. Array of threads.

In CUDA, these independent threads are organized into blocks which can contain from 32 to 512 threads each (see Fig. 6). A kernel is executed in a grid of thread blocks being indexed by a 2D block id in the form (row,column). Concerning the device, one thread block is executed by one multiprocessor. In a block, each thread is indexed by a thread id in the form (row,column). Threads in a block are executed by processors within a single multiprocessor. One important consequence is that threads from different blocks cannot cooperate.

3.3. Memory access optimization

The memory access of the kernel is an important feature of the implementation performance. A schematic view of the memory access of the device is given Fig. 7. The closer memory is a set of 32-bit register per processor. The shared memory is on-chip memory, the size being 16 kB per multiprocessor. This memory allows data transfer between threads and is really fast as long as the number of concurrent memory accesses is a multiple of 16. This last precaution allows us to avoid bank conflicts.

The global memory, which is the device memory, is large (1.5 GigaByte) but not as fast as shared memory. The host can only read and write the global memory.

In a D2Q9 lattice, each node requires at least $9 \times 4 = 36$ bytes of memory for single precision computations. Therefore the number of lattice nodes that can be concurrently stored into low latency shared memory is limited to approximatively 450 per multiprocessor, which on the GT200 leads to a lattice of at most 80×80. Hence, the use of high latency global memory is unavoidable. To efficiently hide this latency, maximizing the occupancy rate of the multiprocessors is an important issue.

Data layout in global memory can dramatically impact performances. CUDA enabled GPUs are capable of loading or storing memory segments of 32, 64 or 128 bytes in a single memory transaction. Hence, it is possible to reduce the number of global memory accesses as long as two conditions are met:

- coalescence, i.e. neighboring threads should access neighboring data,
- alignment, i.e. addresses should be a multiple of the segment's size.

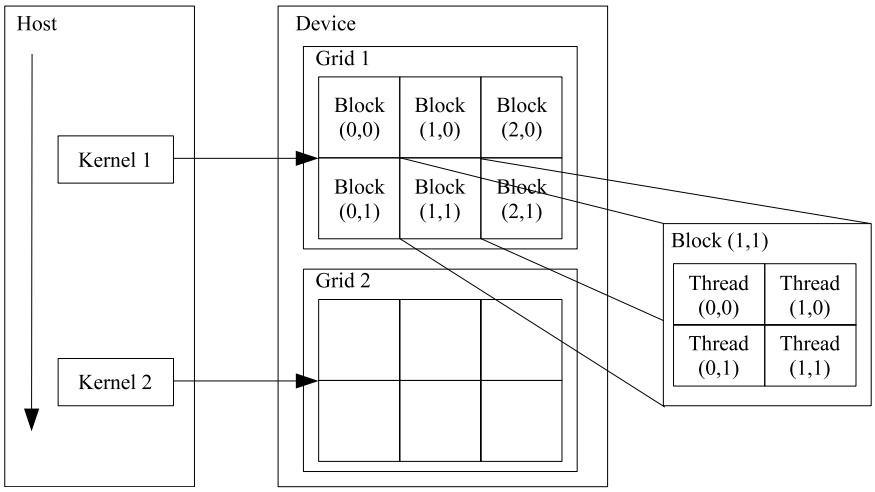

Fig. 6. GPU programming interface.

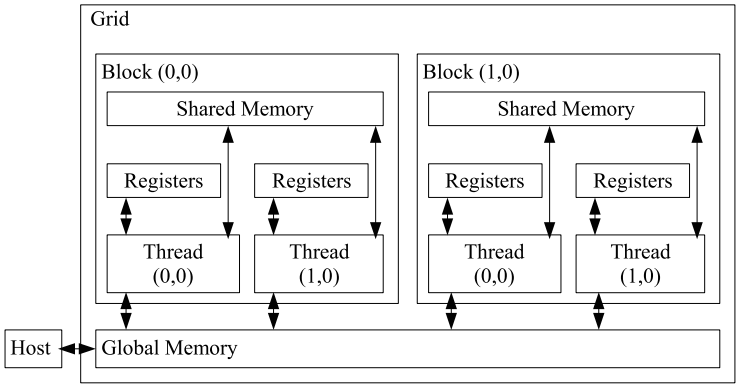

Fig. 7. Kernel memory access.

The former is easily satisfied by using a separate array for each density instead of one array of structures. The later is more problematic because of the propagation step. A careful choice of the lattice's size allows us to avoid misaligned memory stores when propagating along one dimension but the problem remains for the second dimension. A possible solution consists in fetching densities into shared memory. Threads can concurrently access shared memory at no cost as long as there are no bank conflicts. This is readily achieved by using a number of threads per block which is a multiple of 16.

When following the shared memory approach, special care has to be taken of densities crossing block boundaries. Incidentally, this additional step ensures global synchronization across thread blocks.

3.4. Pseudo-code

The first step of the algorithm consists in loading the data from the CPU to the GPU's global memory. This step is computational time consuming because of the CPU's RAM bandwidth.

Once data is loaded in global memory, the data grid is decomposed into threads and thread blocks. One grid point in the physical space is linked to one thread. Each thread is identified by a thread number which depends on the row and column of the thread. Fig. 8 shows the physical grid vs. the computing threads and thread blocks. The number of threads per block, which is limited by available registers and shared memory, is set in order to obtain the maximal number of concurrent threads running on each multiprocessor.

The pseudo-code for the implementation of LBM on GPU is the following:

```
Combine collision and propagation steps:
  for each thread block
    for each thread
      load f_i in shared memory
      compute collision step
      do the propagation step
```

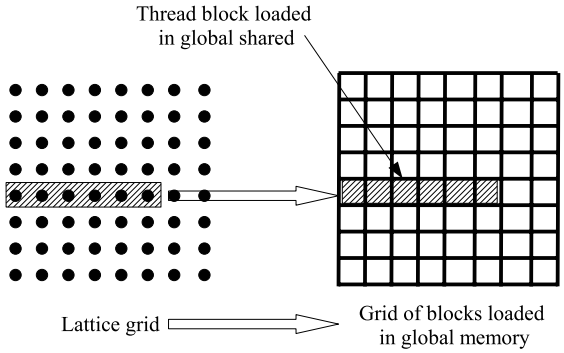

Fig. 8. Physical grid and thread blocks and grids.

end
end
Exchange information across boundaries

Following the pseudo-code description, in the first phase, one thread is used per grid node.

4. Implementation details

This section is devoted to a detailed description of the LBM GPU implementation. The case used in this section concerns the lid driven cavity which is fully described in Section 5.1 of the paper.

4.1. CPU program

Listing 1. CPU program

```
int main(int argc, char ** argv)
{
    // Set size
    size_mat = nx * ny;
    mem_size_mat = sizeof(float) * size_mat;

    // CPU memory allocation
    f0 = (float*) malloc(mem_size_mat);
    ..........
    f8 = (float*) malloc(mem_size_mat);

    unsigned int mem_size_mat_char = sizeof(char) * size_mat;
    geo = (char *) malloc(mem_size_mat_char);

    // GPU memory allocation
    float* f0_dev_Old = NULL;
    CUDA_SAFE_CALL(cudaMalloc((void**) &f0_dev_Old, mem_size_mat));
    ..........
    float* f8_dev_Old = NULL;
    CUDA_SAFE_CALL(cudaMalloc((void**) &f8_dev_Old, mem_size_mat));

    float* f0_dev_New = NULL;
    CUDA_SAFE_CALL(cudaMalloc((void**) &f0_dev_New, mem_size_mat));
    ..........
    float* f8_dev_New = NULL;
    CUDA_SAFE_CALL(cudaMalloc((void**) &f8_dev_New, mem_size_mat));

    char* geo_dev = NULL;
    CUDA_SAFE_CALL(cudaMalloc((void**) &geo_dev, mem_size_mat_char));

    // Initialize
```

```
    init();
    init_geo();

    // Copy data from CPU to  GPU
    CUDA_SAFE_CALL(cudaMemcpy(f0_dev_Old, f0, mem_size_mat
    , cudaMemcpyHostToDevice));
    ..........
    CUDA_SAFE_CALL(cudaMemcpy(f8_dev_Old, f8, mem_size_mat
    , cudaMemcpyHostToDevice));

    CUDA_SAFE_CALL(cudaMemcpy(f0_dev_New, f0, mem_size_mat
    , cudaMemcpyHostToDevice));
    ..........
    CUDA_SAFE_CALL(cudaMemcpy(f8_dev_New, f8, mem_size_mat
    , cudaMemcpyHostToDevice));

    CUDA_SAFE_CALL(cudaMemcpy(geo_dev, geo, mem_size_mat_char
    , cudaMemcpyHostToDevice));

    // Define block and grid sizes
    dim3 threads(num_threads, 1, 1);
    dim3 grid1(nx/num_threads, ny);
    dim3 grid2(1, ny/num_threads);

    while(t<t_max)
    {
        // Execute kernel collision_propagation
        collision_propagation<<< grid1, threads >>> (nx, ny,
        num_threads,tau, geo_dev, f0_dev_Old, f1_dev_Old,
        f2_dev_Old, f3_dev_Old, f4_dev_Old, f5_dev_Old,
        f6_dev_Old, f7_dev_Old, f8_dev_Old, f0_dev_New,
        f1_dev_New, f2_dev_New, f3_dev_New, f4_dev_New,
        f5_dev_New, f6_dev_New, f7_dev_New, f8_dev_New);

        // Execute kernel exchange
        exchange<<< grid2, threads >>> (nx, ny, num_threads,
        f1_dev_New, f3_dev_New,
        f5_dev_New, f6_dev_New, f7_dev_New, f8_dev_New);
    }
    // Copy results back to CPU
    CUDA_SAFE_CALL(cudaMemcpy( f0, f0_dev_Old,mem_size_mat
    , cudaMemcpyDeviceToHost));
    ..........
    CUDA_SAFE_CALL(cudaMemcpy( f8, f8_dev_Old,mem_size_mat
    , cudaMemcpyDeviceToHost));
    ..........
}
```

The listing 1 presents a part of the CPU programming stored in a source file (i.e. `D2Q9_LBGK.cu`). The file is then compiled with `nvcc`.

The explicit GPU memory allocation uses the CUDA command `cudaMalloc()` (similarly, the deallocation command is `cudaFree()`). Copy from CPU to GPU is performed using `cudaMemcpy(,,, cudaMemcpyHostToDevice)`; this operation being slow, it must of course be minimized.

The multi-threaded architecture of nVIDIA GPU uses thread blocks and grids. The thread block size and grid size are defined using respectively `dim3 threads(,,)` and `dim3 grid(,)`. In order to optimize the memory access, the thread block is an array which size is a multiple of 16. The grid size is then calculated in order to execute the kernel correctly (Fig. 8 shows the decomposition of the domain).

The kernels are executed using `kernel<<<grid, threads >>>(...)`. In our code, there are two kernels considering first, the collision/propagation step and second, the exchange of information across the boundaries of the grid of threads.

4.2. Collision/propagation kernel

Listing 2. Collision/propagation kernel

```
__global__ void collision_propagation(int nx, int ny, int num_threads,
float tau, char* geoD,...)
{
  // Setup indexing
  int tx = threadIdx.x;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int xStart = tx + bx*num_threads;
  int yStart = by;
  int k = nx*yStart+xStart;

  // Allocate shared memory
  __shared__ float F1_OUT[NT];
  ........
  __shared__ float F8_OUT[NT];

  __shared__ float F0_IN=f0_Old[k];
  ........
  __shared__ float F8_IN=f8_Old[k];

  // Check if it is a fluid or boundary node
  if(geoD[k] == FLUID)
  // Collision
  {
    rho=F0_IN+F1_IN+F2_IN+F3_IN+F4_IN+F5_IN+F6_IN
        +F7_IN+F8_IN;
    vx=(F1_IN-F3_IN+F5_IN+F8_IN-F6_IN-F7_IN)/rho;
    vy=(F2_IN-F4_IN+F5_IN+F6_IN-F7_IN-F8_IN)/rho;

    square =1.5*(vx * vx +vy *vy );
    f_eq0 =4./9.*rho*(1. - square);

    rho*=0.1111111111111111111111;
    f_eq1=rho*(1. + 3.0*vx + 4.5 *vx*vx - square);
    f_eq3=f_eq1-6.0*vx*rho;
    f_eq2=rho*(1. + 3.0*vy + 4.5 *vy*vy - square);
    f_eq4=f_eq2-6.0*vy*rho;
    ........
      F0_IN+=(f_eq0-F0_IN)*tau_inv;
      ........
    F8_IN+=(f_eq8-F8_IN)*tau_inv;
  }

  else if(geoD[k] == SET_U)
  // Velocity boundary condition
  {
    ........
  }

  else if(geoD[k] == WALL)
  // Wall boundary condition
  {
    ........
  }

  // Write to shared memory and Propagation
  if(tx==0)
```

```
  {
    F1_OUT [tx+1]=F1_IN;
    F3_OUT [num_threads-1]=F3_IN;
    F5_OUT[tx+1]=F5_IN;
    F6_OUT[num_threads-1]=F6_IN;
    F7_OUT[num_threads-1]=F7_IN;
    F8_OUT[tx+1]=F8_IN;
  }
  else if(tx==num_threads-1)
  {
    F1_OUT [0]=F1_IN;
    F3_OUT[tx-1]=F3_IN;
    F5_OUT[0]=F5_IN;
    F6_OUT[tx-1]=F6_IN;
    F7_OUT[tx-1]=F7_IN;
    F8_OUT[0]=F8_IN;
  }
  else
  {
    F1_OUT[tx+1]=F1_IN;
    F3_OUT[tx-1]=F3_IN;
    F5_OUT[tx+1]=F5_IN;
    F6_OUT[tx-1]=F6_IN;
    F7_OUT[tx-1]=F7_IN;
    F8_OUT[tx+1]=F8_IN;
  }

  // Synchronize
  __syncthreads();

  // Write to global memory
  f0_New[k]=F0_IN;
  f1_New[k]=F1_OUT[tx];
  f3_New[k]=F3_OUT[tx];

  if (by < ny-1)
  {
    k = nx*(yStart+1) + xStart;
    f2_New[k]=F2_IN;
    f5_New[k]=F5_OUT[tx];
    f6_New[k]=F6_OUT[tx];
  }

  if (by > 0)
  {
    k = nx*(yStart-1) + xStart;
    f4_New[k]=F4_IN;
    f7_New[k]=F7_OUT[tx];
    f8_New[k]=F8_OUT[tx];
  }
}
```

The listing 2 presents a part of the GPU kernel concerning the collision and propagation step inside a thread block. The programming is stored in the source file `collision_propagation_kernel.cu`.

The `k` parameter is the index array of the data and is calculated depending on gridID and threadID. The main idea of the kernel is to use the shared memory in order to calculate the new density distributions stored in an array `Fi_OUT[NT]` which is in the shared memory. This allows us to execute the propagation during the data copy to `Fi_OUT[NT]`. Of course, there is no propagation along the block boundaries. Then, the unknown densities at these boundaries are used to store the known ones that must pass through the block boundaries. The next kernel is used to pass information across the blocks boundaries.

The `__syncthreads()` function is necessary for the multiprocessor to wait for the execution of all threads before transferring the results from the array in shared memory to the global memory.

4.3. Exchange kernel

Listing 3. Exchange kernel

```
__global__ void exchange(int nx, int ny, int num_threads
                ,........)
{
  // Setup indexing
  int nbx=nx / num_threads;
  int num_threads1 = blockDim.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;

  int bx;
  int xStart, yStart;
  int xStartW, xTargetW;
  int xStartE, xTargetE;
  int kStartW, kTargetW;
  int kStartE, kTargetE;

  // Exchange across boundaries
  for(bx=0; bx<nbx-1 ;bx++)
  {
    xStart = bx*num_threads;
    xStartW = xStart+2*num_threads-1;
    xTargetW = xStartW-num_threads;
    yStart = (by)*num_threads1 + tx;
    kStartW = nx*yStart+xStartW;
    kTargetW = nx*yStart+xTargetW;
    f3_New[kTargetW] = f3_New[kStartW];
    f6_New[kTargetW] = f6_New[kStartW];
    f7_New[kTargetW] = f7_New[kStartW];
  }

  for(bx=nbx-2; bx>=0 ;bx--)
  {
    xStart = bx*num_threads;
    xStartE = xStart;
    xTargetE = xStartE+num_threads;
    yStart = (by)*num_threads1 + tx;
    kStartE = nx*yStart+xStartE;
    kTargetE = nx*yStart+xTargetE;
    f1_New[kTargetE] = f1_New[kStartE];
    f5_New[kTargetE] = f5_New[kStartE];
    f8_New[kTargetE] = f8_New[kStartE];
  }
}
```

The listing 3 presents a part of the GPU kernel concerning the exchange of information across thread block boundaries. The programming is stored in the source file `exchange_kernel.cu`.

5. Performance measurements

5.1. Presentation of the test case

In order to test the implementation of the LBM model on the GPU, the lid driven cavity case is used. This case has been chosen because it has been extensively studied in the literature. Fig. 9 present the lid driven cavity problem with the boundary conditions.

In order to check the convergence of the simulation, the norm used $\|.\|$ is:

$$\|x\| = \max_{grid} \left| x^n - x^{n'} \right| \tag{10}$$

with x^n the value of x at the iteration n and $n' = n - 2000$.

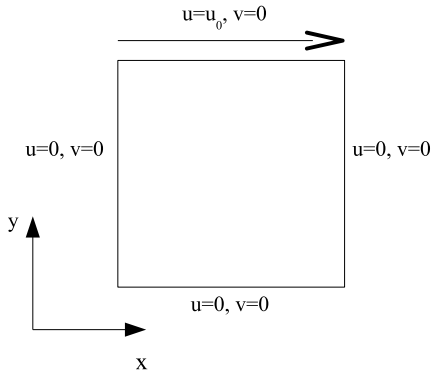

Fig. 9. Lid driven cavity problem.

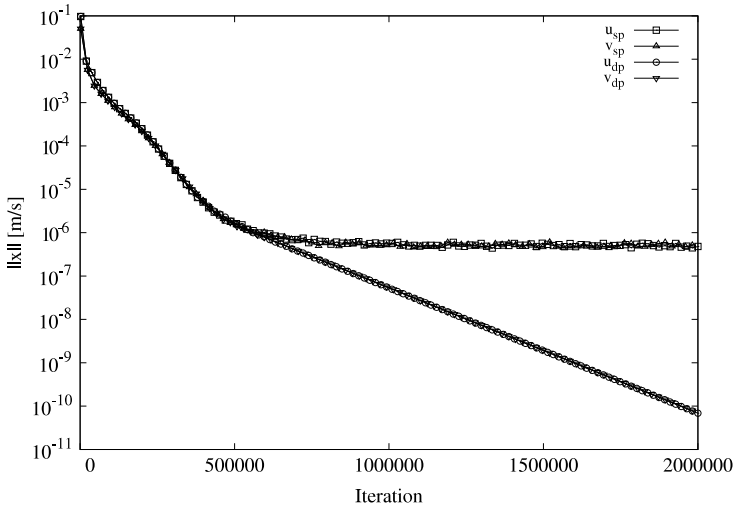

Fig. 10. Single versus double precision floating point arithmetic: *sp* for single precision and *dp* for double precision.

5.2. Comparison of single and double precision floating points—Numerical test

The GT200 GPU supports both single precision floating point and double precision floating point. Then, this section deals with the use of single precision floating point for Lattice Boltzmann simulation of fluid flow instead of classical double precision floating point. The main idea is to evaluate the difference between these two floating point formats for a LBM use.

Both double and single precision floating point calculations have been carried out using the GT200 GPU. The case tested is the lid driven cavity problem at $R_e = 1000$.

Fig. 10 shows the norm evolution of the horizontal and vertical components of the velocity. The calculation has been performed for 2×10^6 iterations. The horizontal asymptote of the velocity norm is close to the limit that can be obtained with single precision floating point (i.e. 1.68×10^{-7}). Of course, the double precision floating point allows us to obtain a lower convergence criteria than for the floating point. The single precision calculation exhibits oscillations of the order of $1/2^{24} = 6 \times 10^{-8}$. However, the convergence is assured because of the numerical scheme stability.

From a numerical point of view, the maximum velocity magnitude difference between the two calculations is about 10^{-3} m/s i.e. a maximum relative difference of 10^{-2} which validates the use of single precision arithmetic with GPUs for LBM calculations.

5.3. Numerical results

GPUs that offer support for single precision floating point arithmetic do not meet all the operations of the IEEE 754 standards [16]. It may be argued that the precision of results obtained via LBM simulation using GPUs are thus suspect. To demonstrate that this is not the case, simulations of LBM lid driven cavity are compared with the results of Ghia et al. [15].

Figs. 11 and 12 show the comparisons between the LBM with GPU computations and the results from [15] and for $R_e = 1000$. The LBM with GPU is precise enough to predict correctly the flow in the lid driven cavity.

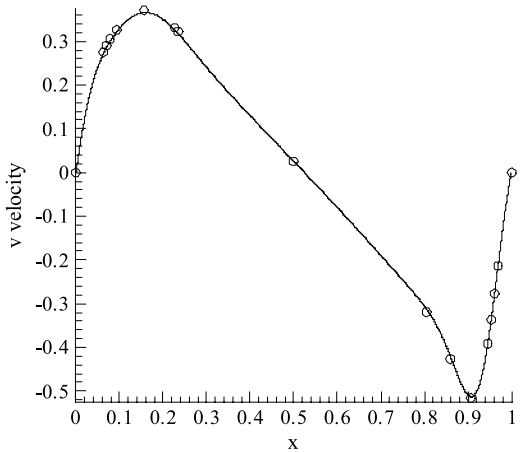

Fig. 11. Vertical velocity at $y = 0.5$ for $R_e = 1000$ — circles are data from [15] and line is LBM.

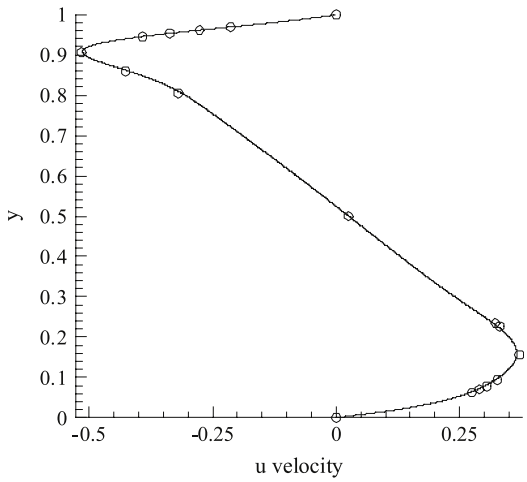

Fig. 12. Horizontal velocity at $x = 0.5$ for $R_e = 1000$ — circles are data from [15] and line is LBM.

Table 1
Performance of implementation in MLUPS — simple precision floating point.

Mesh grid size	Number of threads				
	16	32	64	128	256
256^2	183	341	555	712	783
512^2	188	370	617	840	935
1024^2	169	329	571	819	947
2048^2	129	294	508	781	909
3072^2	147	299	524	786	915

5.4. Performances of LBM with GPU

Table 1 presents the performance of the implementation using single precision floating point, measured in MLUPS (Million Lattice site Update Per Second), function of the LBM mesh grid size and the number of threads per block. From this table, two main conclusions can be done. First, the number of threads must be at least 128 to have good calculation efficiency. Second, the size of the mesh grid must be at least 512^2: this is due to the streaming multiprocessors. They all must be used for calculation to obtain a good GPU productivity.

Table 2 presents the performance of the implementation using GPU double precision floating point. Of course, the number of lattice sites updated per second is quite lower than for simple precision. The mean multiplication factor between the two arithmetics is about 3.8.

Table 2
Performance of implementation in MLUPS — double precision floating point.

Mesh grid size	Number of threads				
	16	32	64	128	256
256^2	52	89	144	190	209
512^2	52	89	146	205	234
1024^2	50	86	144	205	239
2048^2	46	82	138	202	239

6. Conclusions

We have presented a general purpose LBM simulation fully implemented on a single GPU. We have tested our GPU implementation using GT200 processing unit. The case tested was the well-known lid driven cavity problem.

With the use of single precision floating point numbers instead of double precision floating point numbers, the accuracy of the results remains satisfactory. Moreover, the computational time is 3.8 times less with simple precision !

From a computational point of view, simple precision floating point calculations on GPUs give good results compared to the literature data. This enables its use for computational fluid dynamics prediction.

Smaller, less power hungry, easier to maintain, and inexpensive compared to a CPU cluster, GPUs offer a compelling alternative. And this is only the beginning, as shows Fig. 1.

References

[1] S. Succi, The Lattice Boltzmann—For Fluid Dynamics and Beyond, Oxford University Press, 2001, p. 288.
[2] O. Filippova, D. Hanel, A novel BGK approach for low mach number combustion, J. Comput. Phys. 158 (2000) 139–160.
[3] R. Mei, W. Shyy, D. Yu, L.S. Luo, Lattice Boltzmann method for 3-D flows with curved boundary, J. Comput. Phys. 161 (2000) 680–699.
[4] Z. Guo, T.S. Zao, A lattice Boltzmann model for convective heat transfer in porous media, Numer. Heat Transfer B 47 (2005) 155–177.
[5] W. Shi, W. Shyy, R. Mei, Finite-difference-based lattice Boltzmann method for inviscid compressible flows, Numer. Heat Transfer B 40 (2001) 1–21.
[6] K. Mattila, J. Hyvaluoma, J. Timonen, T. Rossi, Comparison of implementation of the lattice-Boltzmann method, Comput. Math. Appl. 55 (2008) 1514–1524.
[7] W. Li, X. Wei, A. Kaufman, Implementing lattice Boltzmann computation on graphics hardware, Vis. Comput. 19 (7–8) (2003) 444–456.
[8] S. Tomov, M. McGuigan, R. Bennett, G. Smith, J. Spiletic, Benchmarking and implementation of probability-based simulations on programmable graphics cards, Comput. Graph. 29 (2005) 71–80.
[9] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (2008) 5342–5359.
[10] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU cluster for high performance computing, ACM/IEEE Supercomputing Conference 2004, November 06–12, Pittsburgh PA, 2004.
[11] J. Tölke, Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by nVIDIA, Comput. Vis. Sci. (2008).
[12] Cuda programming guide 2.2, 2009. http://www.nvidia.com/object/cuda_home.html.
[13] Y.H. Qian, D. D'Humières, P. Lallemand, Lattice BGK for Navier–Stokes equation, Europhys. Lett. 17 (1992) 479–484.
[14] J. Bernsdorf, How to make my LBcode faster :Software planning, implementation and performance tuning, ICMMES'08 Congress, 2008.
[15] U. Ghia, K.N. Ghia, C.T. Shin, High-Re solutions for incompressible flow using the Navier–Stokes equations and a multigrid method, J. Comput. Phys. 48 (1982) 387–411.
[16] IEEE standard for floating-point arithmetic, Institute of Electrical and Electronics Engineers, 2008.