Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Multi-GPGPU Cellular Automata Simulations using OpenACC

Sebastian Szkoda[a,c], Zbigniew Koza[a], Mateusz Tykierko[b,c]

[a] *sebastian.szkoda@ift.uni.wroc.pl Faculty of Physics and Astronomy, University of Wroclaw, Poland*
[b] *Institute of Computer Engineering, Control and Robotics, Wroclaw University of Technology, Poland*
[c] *Wrocław Centre for Networking and Supercomputing, Wroclaw University of Technology, Poland*

Abstract

The Frisch-Hasslacher-Pomeau (FHP) model is a lattice gas cellular automaton designed to simulate fluid flows using the exact, purely Boolean arithmetic, without any round-off error. Here we investigate the problem of its efficient porting to clusters of Fermi-class graphic processing units. To this end two multi-GPU implementations were developed and examined: one using the NVIDIA CUDA and GPU Direct technologies explicitly and the other one using the CUDA implicitly through the OpenACC compiler directives and the MPICH2 MPI interface for communication. For a single Tesla C2090 GPU device both implementations yield up to a 7-fold acceleration over an algorithmically comparable, highly optimized multi-threaded implementation running on a server-class CPU. The weak scaling for the explicit multi-GPU CUDA implementation is almost linear for up to 8 devices (the maximum number of the devices used in the tests), which suggests that the FHP model can be successfully run on much larger clusters and is a prospective candidate for exascale computational fluid dynamics. The scaling for the OpenACC approach turns out less favorable due to compiler-related technical issues. We found that the multi-GPU approach can bring considerable benefits for this class of problems, and the GPU programming can be significantly simplified through the use of the OpenACC standard, without a significant loss of performance, providing that the compilers supporting OpenACC improve their handling of the communication between GPUs.

1. Introduction

The Frisch-Hasslacher-Pomeau (FHP) model [1] is a lattice gas cellular automaton designed to simulate fluid flows using the exact, purely Boolean arithmetic, without any round-off error. An important advantage of this algorithm is that it is parallelizable both on the data and instruction level, and its execution time is limited only by the memory bandwidth. Although the first implementations of the FHP model, which dated back to 1980s, looked promising, it soon turned out that its real-world applications were very demanding both in terms of the hardware resources (memory) and the time to solution, two barriers that were very hard to pass by the machines available in the 1980s and 1990s [2]. This phenomenon is closely related to the Boolean nature of the algorithm: since one is actually interested in the values of physical parameters like pressure or velocity, which are continuous variables, one needs to average the Boolean parameters of the FHP model over either a relatively large numbers of neighboring nodes, which implies that the lattices used in simulations must be large. However, the performance of computers has grown by several orders of magnitude since then and so the hardware-based argument against the FHP model may have become invalid. Therefore in this paper we examine the problem of an efficient implementation of the FHP model on modern multi-GPU clusters, attempting to examine to what extend the lessons learned from the era of pure vector computers can be exploited in these new technologies.

2. Model

The FHP model of fluid flow is a cellular automaton in which particles forming the liquid are allowed to move only along the bonds of a triangular lattice [1, 4], hopping in discrete time steps Δt to one of the 6 neighboring lattice nodes. The exclusion rule asserts that only one particle is allowed to move along a given bond in a given direction. The particles can meet at lattice nodes and collide, exchanging the momentum and changing their velocities. The evolution from time t to $t + \Delta t$ takes part in two steps: propagation and scattering. In the propagation step each particle moves one node at a time in accordance to its current velocity. In the scattering step particles collide, changing their directions according to the rules imposed by the physical laws of mass and momentum conservation, see Fig. 1. If a particle hits an obstacle, it usually bounces back to mimic the no-slip boundary conditions typical of fluid flow simulations.

There are several versions of the model, each characterized by distinct collision rules. Here we consider the so called FHP I model, in which each lattice node can be occupied by up to six particles, one per direction represented by internode links (bonds). The velocities of the particles shall be denoted by v_i, $i = 0,\ldots 6$.

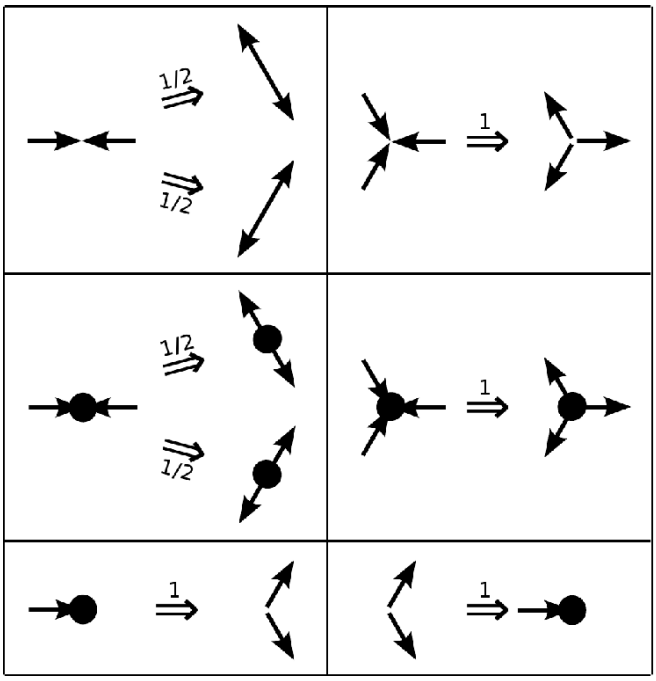

Figure 1 The FHP I and FHP II collision rules. The states before and after the collision are visualized in the first and second column, respectively. The first row shows FHP I collision rules, and the whole table presents the FHP II collision rules.

3. Data structures

3.1. FHP with a look-up table

A convenient representation of a node state is an 8-bit word, or byte, see Fig. 2.

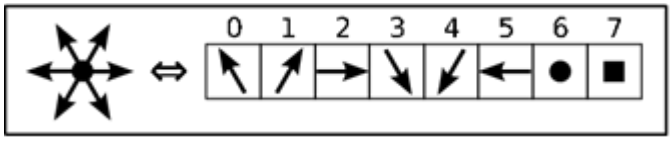

Figure 2 The state of each node is represented by an 8-bit word. Bits 0-5 are mapped onto particles with given nonzero velocities, bit 6 corresponds to a particle at rest, and bit 7 controls whether the node is a boundary node.

For example, a bit pattern 00100100 corresponds to a node with two particles, one moving with velocity v_2 and the other with v_5, located in the area occupied by the fluid. Subsequent nodes represented by 8-bit words are stored in array of length L times H where L and H are the length and height of the two-dimensional system.

One way of implementing the propagation step is to use two two-dimensional arrays holding the system state immediately before and after translation of the particles. During this step, particles from the first array are propagated to the appropriate neighboring nodes in the second array, according to their velocities.

Since the FHP is defined on a triangular lattice, we mapped its nodes into a rectangular lattice, see Fig. 3. This lattice was then mapped into a two-dimensional computer array.

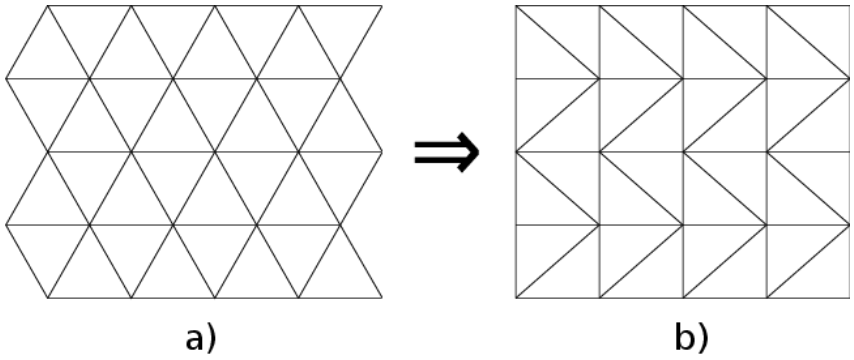

Figure 3 A triangular lattice (a) can be mapped onto a rectangular lattice by shifting every second row by half the lattice constant (b).

As soon as all particles have been propagated, the arrays are swapped and the collision step is performed. Collision in each node is a local operation that does not involve neighboring nodes. It can be implemented using a look-up table, exploiting the fact that an 8-bit word can be regarded as an integer in the range of 0,…,255 collisions with the boundaries are defined through the same look-up table as used for inter-particle collisions [4]. We imposed reflecting (no-slip) boundary conditions on the top and bottom of the system and periodic boundary condition in the horizontal direction. Periodic boundary condition were implemented by extending the system by two columns, one on each side, each mirroring the column on the opposite edge of the system, see Fig. 4.

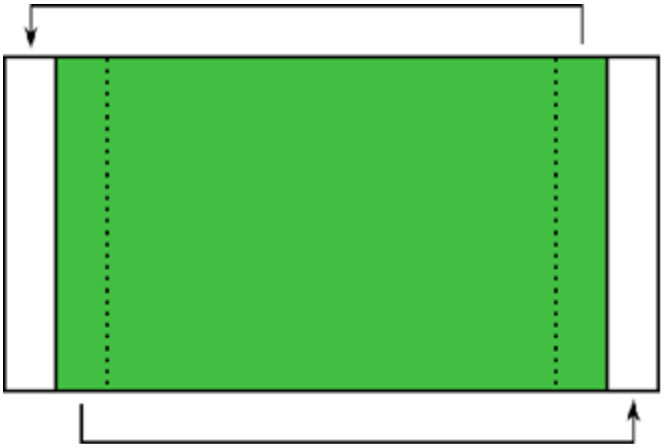

Figure 4 A schematic of periodic boundary conditions. Each edge column is mirrored in an extra column added on the other side of the grid.

In each step the first column is copied to the additional column on the right side, and similarly the last one is occupied to the extra column on the left.

To enforce fluid flow, an external constant force acting horizontally on the particles can be imposed. To this end, whenever the bits in a node fitted the pattern (..1..0..), where "." denotes 0 or 1, with some probability P we exchanged the bits on positions 2 and 5 to form a pattern (..0..1..). This corresponds to reversing the velocity of some of particles moving horizontally against the external driving force and results in momentum transfer, which, in turn, is equivalent to the action of an external driving force [4].

3.2. FHP with a multispin coding technique

While a look-up table appears to be a straightforward implementation of the collision step on scalar machines, vector computers exhibit a better performance if the multispin coding technique was chosen for the data structures [5]. In this approach particle velocities at each lattice node can be represented by single bits and stored in independent arrays X_i, i = 1,…,6. Each array stores information about the presence (bit set to 1) or absence (bit set to 0) of particle in a particular direction. To avoid conditional statements needed to distinguish the neighborhood of even and odd lattice rows (cf. Fig 3), the lattice is mapped into a parallelogram rather than into a rectangle, cf. Fig 5.

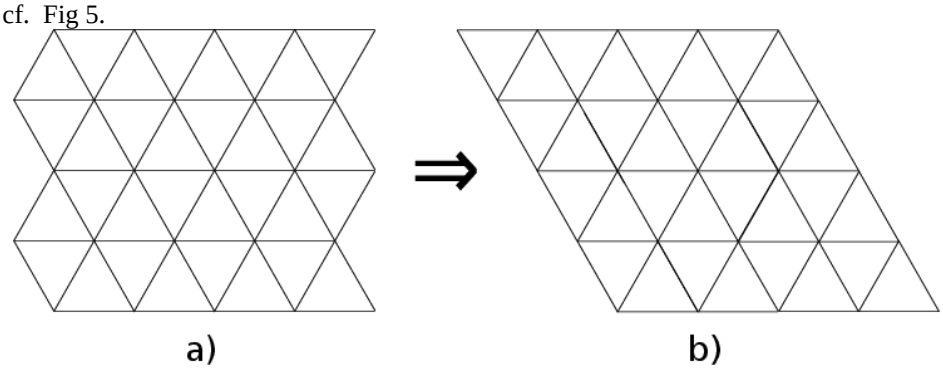

Figure 5 A triangular lattice (a) mapped onto parallelogram (b).

For effective scattering, instruction-level and data-level parallelism of the FHP model can be exploited. Such an approach involves dividing the data into separate bits and distributing them appropriately in computer words. Then it is possible, through logical and arithmetic bitwise operations, to execute more than one operation per processor clock. For the propagation step to be performed efficiently, a special ordering of bits representing particles in lattice nodes in arrays X_i must be applied. The first N/B nodes are assigned in first bits of subsequent words, next N/B nodes are assigned to the second bits of consecutive words, and so on B times, where B is the maximum computer word length. Most often B equals 64, being the length of the long long integer type. The aim of such a data distribution is to perform the particle movement through a simple assignment of words. This is a significant improvement over the look-up table method, in which the propagation step involves detecting the values of individual bits and propagating them—as individual bits—to the neighboring lattice nodes.

Besides the six velocity arrays, one still needs an array holding the placement of obstacles and an array of random bits used in the scattering step to mimic the stochastic choice of the scattering pattern for the cases where a given collision can result in two collisions with probability 1/2, cf.Fig 1.

4. Implementations

We developed and examined three implementations of the FHP I model: a reference CPU implementation, a CUDA[6] implementation and an OpenACC-based[7] implementation. They all use the multispin coding technique.

4.1. Reference CPU implementation

Our CPU C++ implementation of the FHP algorithm is shown in Listings 1, 2, and 3. Arrays x_1, x_2, x_3, x_4, x_5, x_6 of 64-bit-long type unsigned long long int and size *ivb = Width*Height/B* store the bits representing the states of the particles in a particular lattice node. Arrays *ob* and *nob* indicate whether a given lattice node is occupied by an obstacle; any of them can be obtained from the other by the bitwise negation. The collision step, see Listing 1, is implemented as a loop over all words. The collision rules (Fig. 1) are implemented through Boolean operations reflecting the Boolean formulation of the FHP microdynamics. Variable *ncol* stores the information on the particles not undergoing collisions. Variables *cang* and *caang* help determine the result of non-deterministic collisions in a way that conserves the angular momentum. Array *ang* holds the information on the direction of rotation of non-deterministic FHP collisions for each lattice node; after each collision its value at a given node is negated so that on average the number of clockwise and counter-clockwise rotations is the same both globally and locally. Since the collisions tend to occur stochastically, this deterministic algorithm mimics the generation of random numbers. Macro *NEXTY* implements the collision rules, taking advantage of the 60-degree rotational symmetry of the system. Finally, arrays y_1, y_2, y_3, y_4, y_5, y_6 hold the system state after all collisions have been completed.

The propagation step involves simple assignments form arrays x_i to y_i. It consists of two independent loops to account for the boundary conditions.

The implementation of periodic boundary conditions in the horizontal direction (Listing 3) requires the use of the circular bit shift operation. To this end we used a software implementation, as shown in Listing 4. To exchange particles between the two vertical system boundaries, the words representing the system state at the boundaries are circular-shifted one bit in an appropriate direction and assigned to the words representing the other boundary.

4.2. Nvidia CUDA implementation

CUDA is a minimal extension to the standard C++ language that enables to run and synchronize thousands of GPU threads in parallel. In this programming model, a GPU device is a coprocessor running kernels asynchronously with the host CPU. The main task of the CPU is to transfer data to the GPU, launch GPU computational kernels in a given order, and read the results from the GPU.

In the CUDA implementation of the FHP model the CPU subroutines shown in Listings 1, 2, and 3. were reimplemented as CUDA computing kernels. In the collision kernel the shared memory was used to optimize the multiple memory reads of arrays x_i, *ob* and *nob*. In the remaining kernels the use of the shared memory wouldn't be as profitable, as they use only simple memory assignments.

The multi-GPU communication was realized through the Cuda Streams and Unified Virtual Addressing (UVA) technologies. By using Cuda Streams one can distribute and asynchronously execute multiple computing kernels working on separate workloads. Before the simulation begins, the system is divided into N parts where N is the number of GPUs, with the first *H/N* rows assigned to the first GPU, the next *H/B* rows to second GPU and so on. The communication is provided by the *cudaMemcpyPeerAsync* function which, using the UVA, can directly copy data between different devices bypassing the CPU.

4.3. OpenACC implementation

OpenACC is a pragma-based programming standard similar to the well known OpenMP. It aims to extend OpenMP so as to generate parallel code for various compute accelerators and is currently supported by three compilers, of which the Portland Group compiler, pgi, was chosen for the tests.

We used OpenACC to quickly generate NVIDIA CUDA C code, an alternative to developing the CUDA code manually, as described above. OpenACC provides of a set of compiler directives and runtime library routines. The FHP algorithm is well suited for automatic parallelization, so it was enough to take the CPU code and put the *#pragma acc parallel loop* compiler directive in front of each for loop, see Listing 5. To reduce redundant PCI-E transfers, the data should be copied to the accelerator once, before the usage of the *#pragma acc data copy* directive, which automatically copies data back to the host at the end of the scope, see Listing 5. Functions, like those presented in Listing 4, have to be rewritten either as preprocessor directives or as inline functions before to be executed on the device.

A standard approach to the multi-GPU programming requires is based on the MPI standard. A the time we developed the OpenACC code, the Portland Group OpenACC compiler did not support any of the GPU-aware MPI implementations, so the standard MPICH2 implementation of the MPI was chosen. Since it is unaware of the UVA, the communication between different GPUs requires the first and last row of the arrays x_1 and x_4 to be sent to the next GPU, and x_1, x_6 to be send to the previous one, via the host. Data updates from the host to the device and vice versa were implemented by *#pragma acc update host(array[0:size])* and *#pragma acc update device(array[0:size])*. Unfortunately, OpenACC does not support partial updates of the data, thus extra arrays have to be used for interprocess communication, which involves an additional overhead.

5. Results

Using a single high-end consumer-class GPU accelerator, we achieved the performance of $\approx$ 40 Gups (Giga lattice uploads per second), see Table 1. Using professional HPC computing nodes with eight Nvidia M2090 units we could go up to 220 Gups. These results are much higher than 2 Gups reported [2] for Cray-YMP, the state of the art vector supercomputer from early 1990s .

The results for the weak and strong scaling for up to 8 GPUs are shown in Figs. 6 and 7, respectively. The weak scaling is almost linear for both CUDA-based implementations, indicating that OpenACC is a very good alternative to writing CUDA kernels manually. It also suggests that the FHP model can be successfully run on much larger clusters and is a prospective candidate for exascale computational fluid dynamics. The strong scaling is less favorable, especially for OpenACC. This reflects a relatively high latency and communication overhead if the workloads per GPU device are getting smaller relative to the communication.

Comparing power consumption of Cray-YMP, which was about 200 kW, to modern 8-GPU nodes, which consume less than 4 kW, we can conclude that the power consumption per 1 Gups has decreased by four orders of magnitude.

Table 1 The performance of our implementations on various CUDA-capable devices, in GUPS (lattice node updates per second[5,8]).

Implementation	GTX480 [GUPS]	M2090 [GUPS]	K20M [GUPS]
OpenACC	36.8	30.1	N/A
CUDA	41.1	32.3	57.0

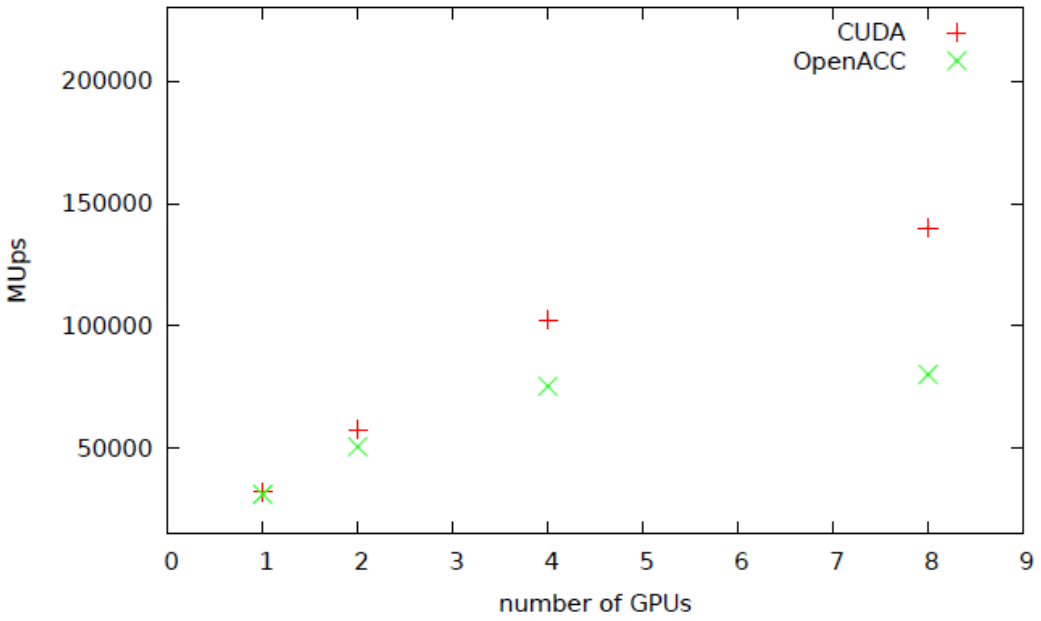

Figure 6 The strong scaling on a node with 8 NVIDIA M2090 cards.

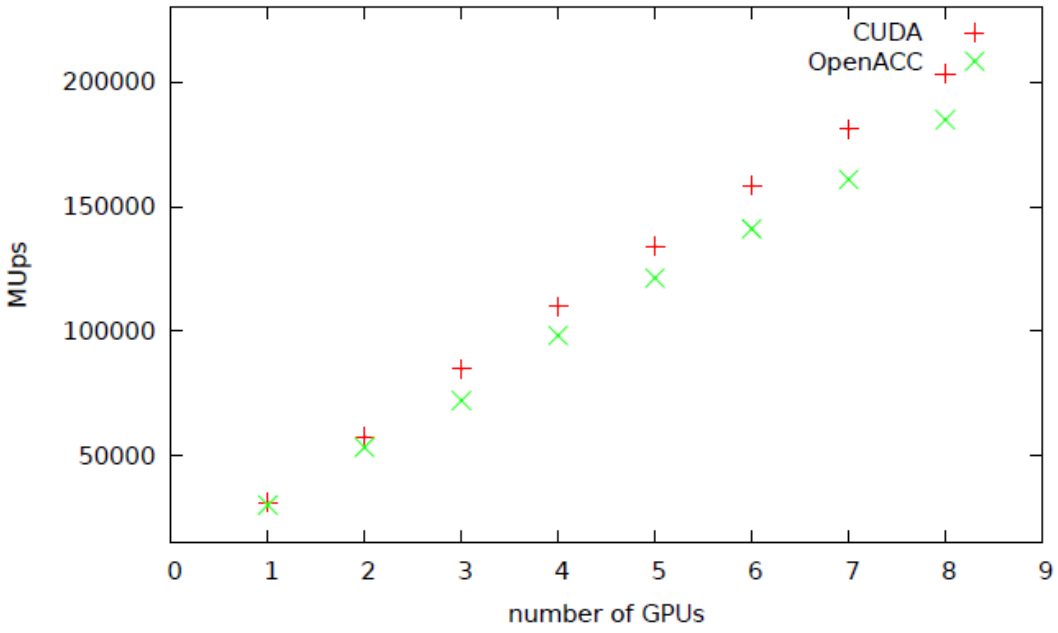

Figure 7 The weak scaling on a node with 8 NVIDIA M2090 cards.

The choice between technologies depends on both the expected performance and the programming costs. On the one hand NVIDIA CUDA gives us the full control over the program and, assuming a programmer's deep understanding of the underlying accelerator architecture, should outperform automatic the tools. On the other, the results show that by using a handful of compiler pragmas we can port a program from the CPU to the GPU in just a few minutes, without a significant performance loss.

The Portland Group OpenACC implementation is still under development. As soon as it can support a CUDA-aware MPI implementation, the development of efficient Multi-GPU applications will be much easier, cheaper and popular. It is likely that manual CUDA programming will be used only in the most crucial parts of the code where every millisecond matters.

6. Listings

```
#define TYPE unsigned long long int
#define NEXTY(cw,ccw,same,oop)\
    ((( cw )  & ( cang ))+(( ccw ) & ( caang ))+\
    (( same ) & ( ncol ))+(( oop ) & ( ob[i] )))
TYPE col,ncol,cang,caang;
for(i=0; i<ivb; i++)
{
  ncol =(\
        (x1[i]^x4[i])|(x2[i]^x5[i])|\
        (x3[i]^x6[i])\
        )&\(\
        (x1[i]^x3[i])|(x3[i]^x5[i])|\
        (x2[i]^x4[i])|(x4[i]^x6[i])\
        )&(nob[i]);

  col     = (~ncol) & ( nob[i]);
  cang    = (  col) & ( ang[i]);
  caang   = (  col) & (~ang[i]);
  ang[i]  = (  col) ^ ( ang[i]);

  y1[i] = NEXTY(x2[i],x6[i],x1[i],x4[i]);
  y2[i] = NEXTY(x3[i],x1[i],x2[i],x5[i]);
  y3[i] = NEXTY(x4[i],x2[i],x3[i],x6[i]);
  y4[i] = NEXTY(x5[i],x3[i],x4[i],x1[i]);
  y5[i] = NEXTY(x6[i],x4[i],x5[i],x2[i]);
  y6[i] = NEXTY(x1[i],x5[i],x6[i],x3[i]);
}
```

Listing 1 Collision step.

```
for(i = ldb; i < ivb - ldb; i++)
{
  x1[i] = y1[ i + ldb - 1 ];
  x6[i] = y6[ i + ldb     ];
  x3[i] = y3[ i - ldb     ];
  x4[i] = y4[ i - ldb + 1 ];
  x2[i] = y2[ i - 1       ];
  x5[i] = y5[ i + 1       ];
}

for(i=0;i<ldb;i++)
{
  x1[i]          = y1[i+ldb-1      ];
  x6[i]          = y6[i+ldb        ];
  x3[i+ivb-ldb] = y3[i+ivb-2*ldb  ];
  x4[i+ivb-ldb] = y4[i+ivb-2*ldb+1];
}
```

Listing 2 Propagation step.

```
for(i = 0; i < (ivb - ldb)/ldb; i++)
{
  x1[i*ldb]          = rotl(y1[i*ldb+2*ldb-1], 1L  );
  x2[i*ldb]          = rotl(y2[i*ldb+ldb - 1], 1L  );
  x4[i*ldb+2*ldb-1] = rotl(y4[i*ldb         ], 63L );
  x5[i*ldb+2*ldb-1] = rotl(y5[i*ldb+ldb     ], 63L );
}
```

Listing 3 Periodic boundary condition.

```
inline TYPE _rotl(TYPE value, TYPE shift)
{
    return (value << shift) | \
           (value >> (sizeof(TYPE)*8 - shift));
}
```

Listing 4 Circular shift.

```
//copy arrays to the device
#pragma acc data copy( x1[0:ivb],x2[0:ivb],... )
{
  //collision step loop

  #pragma acc parallel loop
  for(i = ldb; i < ivb - ldb; i++)
  {
    x1[i] = y1[ i + ldb - 1 ];
    x6[i] = y6[ i + ldb     ];
    x3[i] = y3[ i - ldb     ];
    x4[i] = y4[ i - ldb + 1 ];
    x2[i] = y2[ i - 1       ];
    x5[i] = y5[ i + 1       ];
  }

//further operations as on previous listings

}//end of the scope, data is copied back to host
```

Listing 5 OpenACC example code.

7. Acknowledgments

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763. This work was financially supported by the Polish Ministry of Science and Higher Education (funds for R&D in 2013-2014). This research was supported in part by PL-Grid Infrastructure. Calculations were carried out in Wroclaw Centre for Networking and Supercomputing (http://wcss.pl).

References

[1] Frisch, U., Hasslacher, B., and Pomeau, Y. (1986) Lattice gas automata for the Navier-Stokes equation. Phys. Rev. Lett., 56, 1505.
[2] Stauffer, D. (1991) Computer simulations of cellular automata. 1991 J. Phys. A: Math. Gen. 24 909.
[3] Wolfram, S. (2002) A new kind of science. Wolfram Media, Champaign.
[4] Chopard, B. and Droz, M. (1998) Cellular automata modelling of physical systems. Cambridge Univ. Press, Cambridge.
[5] Kohring, G. (1992) The cellular automata approach to simulating fluid flows in porous media. Physica A, 186, 97–108.
[6] Farber, R. (2011) CUDA Application Design and Development, 1 edition. Morgan Kaufmann.
[7] Portland Group (2010) PGI Fortran & C Accelerator Programming Model new features ver. 1.3.
[8] Johnson, M. G. B., Playne, D. P., and Hawick, K. A. (2010) Data-parallelism and GPUs for lattice gas fluid simulations. Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA10), Las Vegas, USA, 12-15 July, pp. 210–216. CSREA. PDP4521.