The Application of High Level Synthesis for Implementation of Lattice Boltzmann Method in ARUZ

Grzegorz Jabłoński, and Joanna Kupis

Abstract—**The paper presents the implementation results of D2Q9 Lattice Boltzmann method on ARUZ, a massively parallel FPGA-based simulator built in Lodz, Poland in 2015, optimized for execution of the Dynamic Lattice Liquid algorithm. The results of tests on a single ARUZ board indicate, that the LBM simulation of 864 $\times$ 384 lattice on 18 panels of ARUZ would reach the performance of $206 \cdot 10^3$ MLUPS (Million Lattice Updates per Second).**

Index Terms—**Distributed System, Reconfigurable System, FPGA, Lattice Boltzmann Method, ARUZ.**

I. Introduction

ARUZ [1] (Analizator Rzeczywistych Układów Złożonych, Analyser of Real Complex Systems) is a massively parallel FPGA-based simulator located at Lodz Technopark. This machine has been designed with a single algorithm (Dynamic Lattice Liquid — DLL [2]) in mind [3] [4]. No attempts so far have been made to extend an applicability of the ARUZ hardware to solve more general computing problems. This paper presents the results of implementation of the Lattice Boltzmann method [5] on ARUZ, including algorithm performance estimation.

II. The ARUZ Architecture

The ARUZ, commissioned in 2015 at Lodz Technopark, comes as a result of close cooperation between the Department of Molecular Physics and the Department of Microelectronics and Computer Science, both from Lodz University of Technology complemented by a professional management expertise of the Ericpol company. This supersimulator weights about 50 tons and is a cylinder 4.5 meters high and 16 meters wide, placed in its own, dedicated, two-floor building. The machine is composed out of 25920 Field Programmable Gate Arrays (FPGAs), interconnected by 70 000 of twisted-pair cables, totalling 100 kilometers in length and 6 tons in weight. Out of 20 machine's panels, 18 are usually engaged in ongoing simulation, whereas remaining two are redundant used as a standby in case of technical problems. Every panel consists of 12 rows, each containing 12 PCBs, called DBoards (Daughter Boards) and is controlled by a dedicated rack-mounted PC. In total, there are 144 DBoards in each panel, giving 2880 for the whole machine. The overall ARUZ architecture is shown in Figure 1.

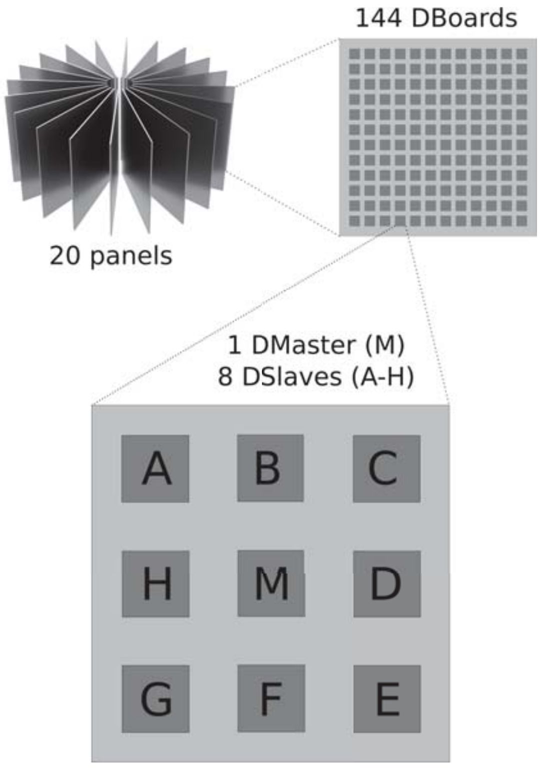


Fig. 1. ARUZ overall structure

Each simulation board carries 9 FPGAs: 8 of them called DSlaves (Artix XC7A200T), constitute the resources for execution of the simulation algorithm and the remaining one called DMaster (Zynq XC7Z015) manages the operation of DSlaves. Each of DSlaves is equipped with the communication interfaces to the 8 closest neighboring FPGAs in a 3D simulation space (local communication) and implements a grid of specialized processing cells, dedicated for performing consecutive steps and complex calculations of a Dynamic Lattice Liquid algorithm.

A. ARUZ – communication structure

The core element of every simulation performed by the ARUZ machine is a communication between its FPGAs. Three different kinds of communication, i.e. local, global and control,

G. Jabłoński and J. Kupis are with the Department of Microelectronics and Computer Science, Lodz University of Technology, Poland (e-mail: gwj@dmcs.p.lodz.pl).

This work was supported by the Polish National Science Centre grant 2015/19/N/ST6/01191.

may be distinguished and every one of them has its own topology, purpose and dedicated protocol implementation.

- Global communication is 1 Gb Ethernet-based and used for the data exchange between DMaster modules in DBoards. It is responsible for configuring DSlaves, determining the current state of the whole system, initializing simulation process and archiving its results. To reduce cost, DMasters in every row of the panel are daisy-chained and only the first board is connected to a PC controlling given panel via an Ethernet switch.
- Control communication is exploited in synchronization of every sequence of steps in DLL cells during the simulation process. This communication is based on two groups of signal lines i.e. command and state lines. State lines help in determining in which stage the simulation process is. Depending on a state information, commands are issued for every DLL cell in the system. The simulation management is organized in a hierarchical way — a single MBoard controls up to 20 CBoards, one per panel, which in turn control individual rows of DBoards.

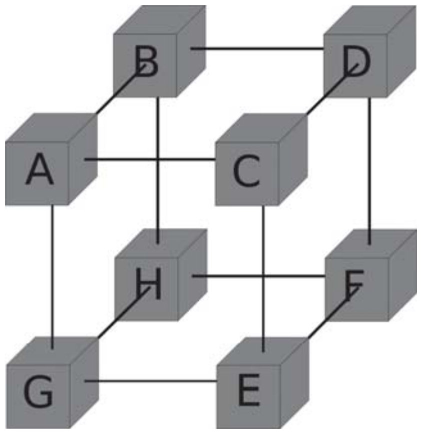


Fig. 2. Cell interconnection structure within a single DBoard

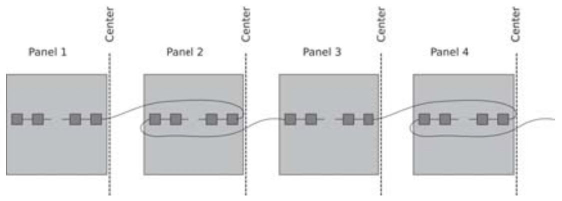
- Local communication enables the dataflow between DSlaves and the data exchange between DMaster and DSlaves within one DBoard in order to initialize simulation cells and read their state during the simulation process. Local communication is bi-directional, LVDS-based and uses 1 GHz source-synchronous clock and a single data line in each direction. It is used for communication between FPGAs placed on the same board, on the same panel or on the neighboring panels. With respect to local communication, DSlaves on every DBoard are topologically organized in a 2x2x2 grid as shown in Figure 2 and connected by differential PCB traces.

  Apart from the connections internal to the board, every DSlave has also serial connections to at most six neighboring boards via Cat6 STP cables. These connections lead to the board above, below, to the left, to the right, at the next panel and at the previous panel. DSlave A is always connected to DSlave A, DSlave B to DSlave B etc. The entire ARUZ contains ca. 70000 such cables.

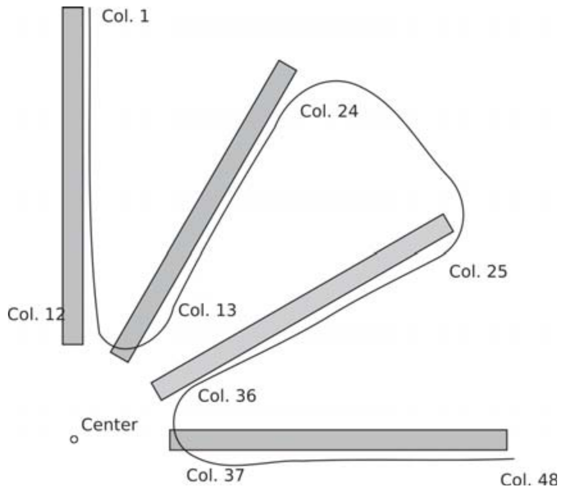
  This interconnection network allows creating of 3D, 2D or 1D simulation grids of different sizes. For 2D simulations, the interconnection structure with an aspect ratio closest to one has the x-axis going along all the panels in an accordion-folded way (see Figure 3(a-b)).

  As a result, we have a matrix of 18 (panels) × 12 (board/panel) = 216 boards in the x axis and 12 boards in the y axis. There are also different configurations giving the same aspect ratio, the one above minimizes the number of least-reliable long connections between panels.

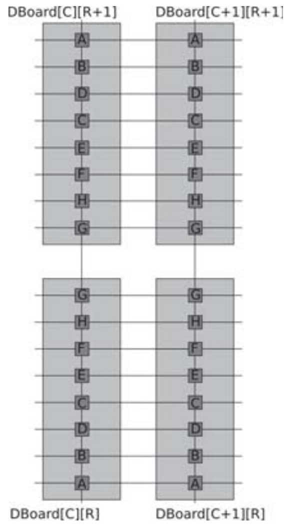
  The DSlaves on a single DBoard can be organized in a few different ways, in a 1 × 8 (Figure 3(c)), 2 × 4 (Figure 3(d)), 4 × 2 or 1 × 8 grid.



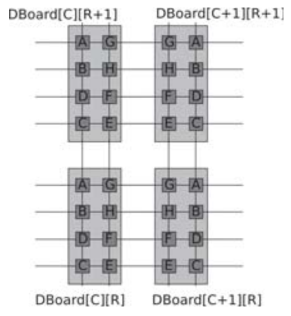
(a) Side view



(b) Top view



(c) Connections on single DBoard and between DBoards - 1 × 8



(d) Connections on single DBoard and between DBoards - 2 × 4

Fig. 3. 2D interconnection matrix on ARUZ

III. The Lattice Boltzmann Method

The lattice Boltzmann method (LBM) was originally proposed in 1988 by McNamara and Zanetti [5]. It is dedicated for solving the Navier-Stokes equations describing fluid dynamics. The method originates from the earlier lattice gas automata model [6], but differs from its predecessor in operating at mesoscopic level, i.e. by using distribution functions instead of velocities of individual molecules.

The basis of the LBM is two- or three-dimensional regular grid. Each of the grid nodes has several associated state variables $f_i(\vec{x}, t)$. These variables constitute the density of molecules at place $\vec{x}$ and time t moving in direction $\vec{c_i}$. There are many variants of lattices applied in this model. The most popular one for 2D simulations is the D2Q9 lattice presented in Figure 4.

The molecule density and velocity at the given node can be computed from $f_i(\vec{x}, t)$ using formulas

$$\rho(\vec{x}, t) = \sum_i f_i(\vec{x}, t), \tag{1}$$

$$\rho(\vec{x}, t)\vec{u}(\vec{x}, t) = \sum_i \vec{c_i} f_i(\vec{x}, t) \tag{2}$$

where ρ and $\vec{u}$ denote molecule density and flow velocity, respectively.

There are two main phases of the lattice Boltzmann algorithm

- **Streaming:** the distribution functions f_i are propagated to the neighboring nodes in the direction $\vec{c_i}$.
- **Collision:** The particles incoming from all directions to a given node collide with one another, tending to an equilibrium distribution with the relaxation constant τ.

The collision and propagation can be described by

$$f_i(\vec{x} + \vec{c_i}, t + 1) = (1 - \frac{1}{\tau}) f_i(\vec{x}, t) + \frac{1}{\tau} f_i^{(0)}(\vec{x}, t). \tag{3}$$

The equilibrium distribution of particles $f_i^{(0)}$ can be approximated using [7]

$$f_i^{(0)} = \begin{cases} \frac{4}{9}\rho[1 - \frac{3}{2}\frac{\vec{u}^2}{c^2}] & \text{for } i = 0 \\ \frac{1}{9}\rho[1 + 3\frac{\vec{c_i}\cdot\vec{u}}{c^2} + \frac{9}{2}\frac{(\vec{c_i}\cdot\vec{u})^2}{c^4} - \frac{3}{2}\frac{\vec{u}^2}{c^2}] & \text{for } i = 1,2,3,4 \\ \frac{1}{36}\rho[1 + 3\frac{\vec{c_i}\cdot\vec{u}}{c^2} + \frac{9}{2}\frac{(\vec{c_i}\cdot\vec{u})^2}{c^4} - \frac{3}{2}\frac{\vec{u}^2}{c^2}] & \text{for } i = 5,6,7,8 \end{cases} \tag{4}$$

where c represents a speed of sound ($\frac{1}{\sqrt{3}}$).

The macroscopic quantities ρ and $\vec{u}$ in equation (4) can be computed from the distribution functions f_i using equations (1) and (2).

Special considerations are required for boundary conditions. The simplest one is the non-slip condition with zero velocity at the wall. Frequently a half-way bounce-back condition is applied [8]. Description of the pressure and velocity boundary condition can be found in [9].

The lattice Boltzmann method operates locally in the collision phase and requires sending a single floating point value of a distribution function to each of the neighboring nodes during the streaming phase therefore it fits perfectly to the architecture of ARUZ.

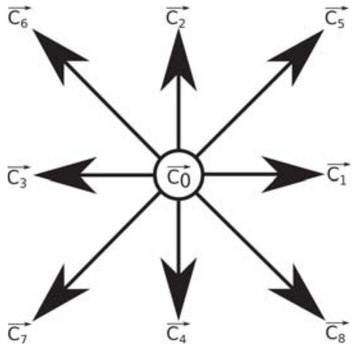


Fig. 4. The D2Q9 lattice

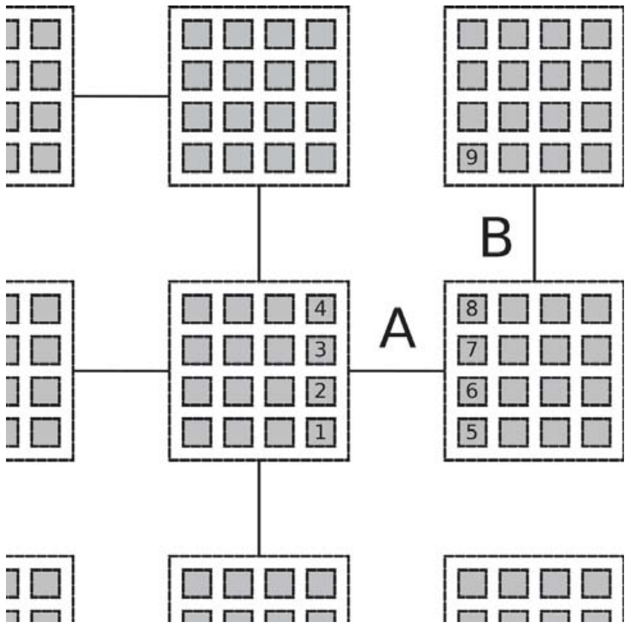


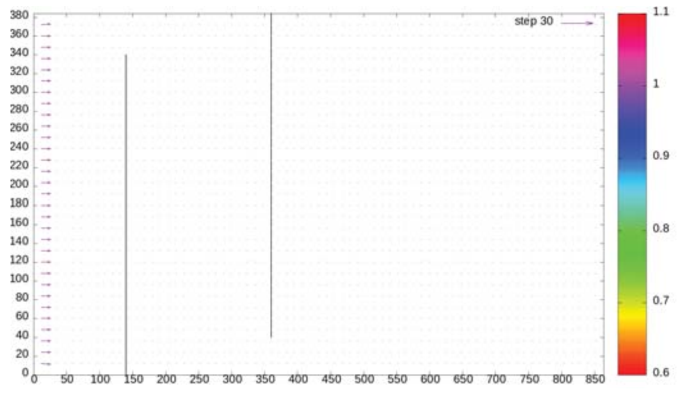
Fig. 5. Data exchange between neighbouring FPGAs

Figure 6 presents the results of the example simulation using the D2Q9 variant of the lattice Boltzmann method, with the non-slip walls at the top and bottom and the density boundary condition ($\rho = 0.8$) on the left and right side. There are also two vertical obstacles at grid point 140 and 360. At time 0 the density on the left side changes from 0.8 to 1 to force flow in the horizontal direction. The half-grid boundary conditions are applied at the non-slip walls.

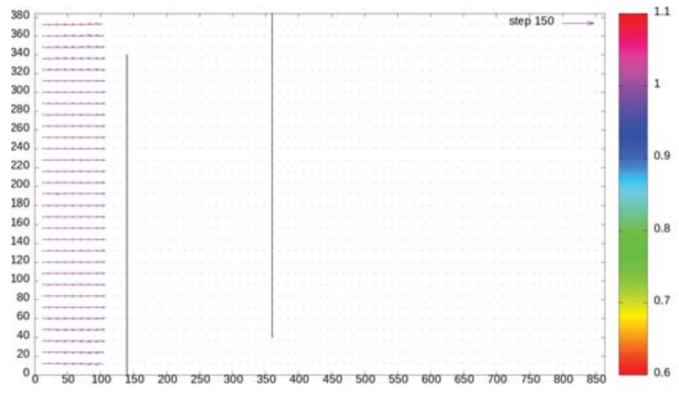
IV. FPGA Implementation of the LBM Algorithm

The FPGA-based implementation of LBM using multiple FPGAs can be found in [10], [11] and [12]. As opposed to ARUZ, this architecture is not massively parallel - it uses at most 4 FPGAs located on the PCIe add-on cards.

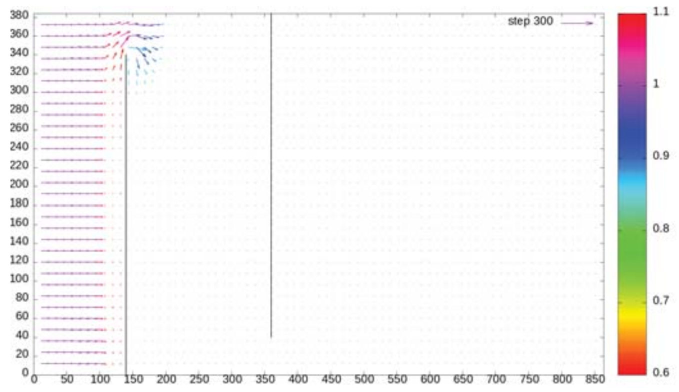
To implement a simulation from the previous chapter seven separate collision operators are needed: one for the interior of the domain, two for the left and right edge, respectively and four for corners. The function implementing the collision operator has been written in C and is presented in Appendix 1. The part of the code for the domain interior has been taken



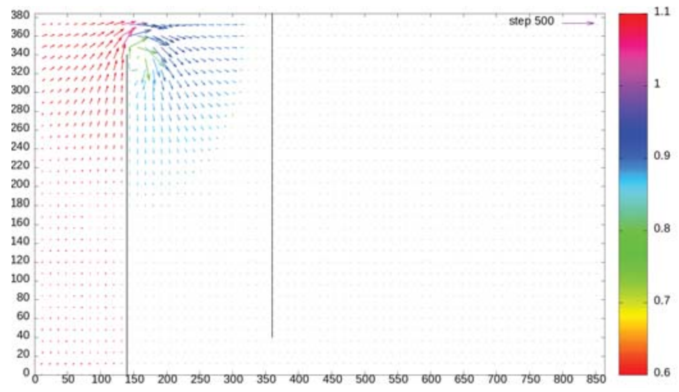
(a) Step 30



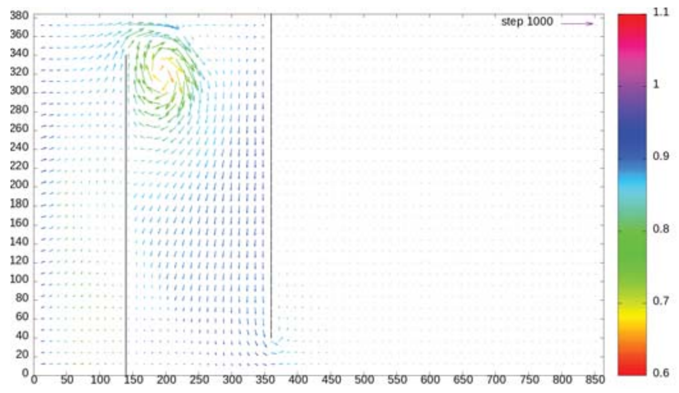
(b) Step 150



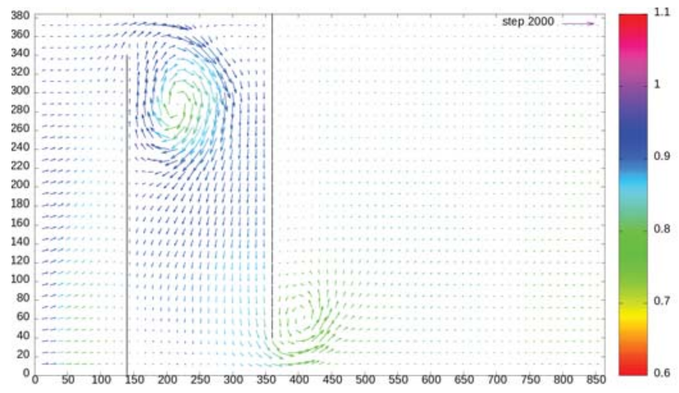
(c) Step 300



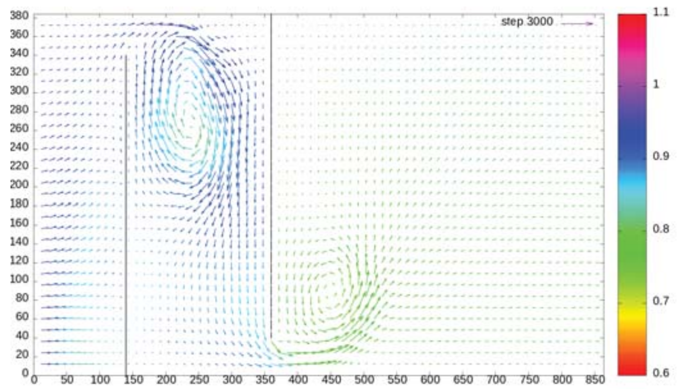
(d) Step 500



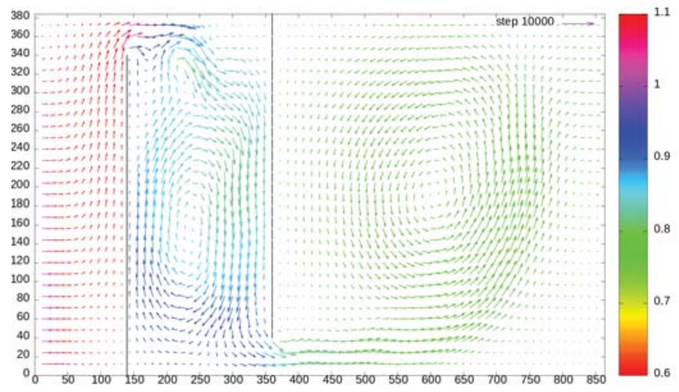
(e) Step 1000



(f) Step 2000



(g) Step 3000



(h) Step 10000

Fig. 6. Example fluid flow simulation on 864 $\times$ 384 lattice at 8 different time steps (performed on a PC). Arrow size and direction - fluid velocity, arrow color - fluid density.

from [10], it has been complemented with the necessary boundary conditions and used for the simulation on a PC to obtain the results presented in Figure 6.

The non-slip boundary conditions should be implemented in the streaming phase and require the collision operator for the interior of the domain.

In order to estimate the simulation performance on ARUZ the LBM collision function for the domain interior and all the boundary conditions has been synthesized in the Vivado HLS 2016.4 environment using 8 ns cycle time as the input constraint. The synthesizer estimations for all the function variants are presented in Table I.

TABLE I
SYNTHESIS RESULTS FOR DIFFERENT LBM COLLISION FUNCTION

	Bulk	Bottom-Left Corner	Top-Left Corner	Top-Right Corner	Bottom-Right Corner	Left Edge	Right Edge
Clock period [ns]	7.49	7.49	7.49	7.49	7.49	8.27	8.27
Latency [cycles]	63	43	43	43	43	84	84
DSP48E Slices	42 (5.7%)	30 (4.1%)	30 (4.1%)	30 (4.1%)	30 (4.1%)	27 (3.7%)	27 (3.7%)
Flip-flops	4615 (1.8%)	3406 (1.3%)	3406 (1.3%)	3406 (1.3%)	3406 (1.3%)	3752 (1.4%)	3752 (1.4%)
LUTs	5247 (4.1%)	3525 (2.8%)	3525 (2.8%)	3525 (2.8%)	3525 (2.8%)	4106 (3.2%)	4042 (3.2%)
Possible instances	17	25	25	25	27	27	27

From these results one can estimate that every FPGA can fit at least 16 nodes. If these nodes are arranged in a matrix 4 nodes wide and 4 nodes high, the 18 panels of ARUZ are able to simulate a domain of (4 (nodes/FPGA) $\times$ 1 (FPGA/board) $\times$ 12 (boards/panel) $\times$ 18 (panels)) $\times$ (4 (nodes/FPGA) $\times$ 8 (FPGAs/board) $\times$ 12 (boards/panel), i.e. 864 $\times$ 384 nodes.

For 16 nodes arranged in a 4 $\times$ 4 matrix (Figure 5), the following cells of the central FPGA must transfer a distribution function value over link A during the LBM streaming phase: cell 1 to 5, 2 to 6, 3 to 7, 4 to 8, 1 to 6, 2 to 7, 3 to 8, 4 to 7, 3 to 6, 2 to 5 and, indirectly, 4 to 9. This gives 11 single-precision floating point numbers, i.e. 352 bits. Subsequently, one floating-point number must be transmitted over link B to cell 9 (32 bits). Similar transmissions occur simultaneously over 3 remaining links in upward, leftward and downward direction.

Implementation of the LBM algorithm on ARUZ has not presented any particular difficulties. The ARUZ firmware has been written in such a way, that a lot of VHDL code and supporting software could be reused. The practical implementation required adding a relatively simple finite state machine to the matrix of the synthesized collision function modules and connecting them to the existing communication modules. The firmware of DMasters did not require any changes.

The physical synthesis results in Vivado 2017.2 have shown, that the Vivado HLS estimations were too pessimistic, as the clock rate of 125 MHz could be applied. The VHDL simulation of the 2 $\times$ 2 matrix of DSlaves has indicated, that a single computation cycle takes 1608 ns. The resource utilisation report is presented in Table II. It indicates, that the number of available DSP modules is limiting the number of nodes that can be implemented in a single FPGA.

We obtain exactly the same length of a single computation cycle as previously estimated in [13]. The overestimation of achievable clock cycle has been, by coincidence, exactly compensated by the underestimation of protocol overhead. In effect, the entire machine is able to perform computations with a throughput of (864 $\times$ 384) / 1608 ns = $206 \cdot 10^3$ MLUPS (Million Lattice Updates per Second, [14]). Paper [11] gives 360 $\times$ 180/0.0164 s = 3.95 MLUPS. Thus, the performance of ARUZ would be $52 \cdot 10^3$ times better. The comparison to [12] cannot be done directly, as the authors do not provide such a figure of merit.

TABLE II
RESOURCE UTILISATION REPORT FOR DSLAVE

Site Type	Used	Available	Util%
Slice LUTs	83791	134600	62.25
LUT as Logic	83283	134600	61.87
LUT as Memory	508	46200	1.10
LUT as Distributed RAM	0		
LUT as Shift Register	508		
Slice Registers	77926	269200	28.95
Register as Flip Flop	77926	269200	28.95
Register as Latch	0	269200	0.00
F7 Muxes	254	67300	0.38
F8 Muxes	66	33650	0.20
DSP48E1	564	740	76.22

The simulation cycle length has been confirmed by practical measurements of execution time of 10^9 LBM cycles in a matrix of 2 $\times$ 2 DSlaves on a single DBoard (simulation domain: 8 $\times$ 8 nodes). The algorithm is perfectly scalable, however the entire ARUZ would exhibit slightly longer cycle times due to higher signal delays on 4 m cables (about 20 ns). The computation results obtained on a cluster of 4 FPGAs are identical to the results obtained using a C++ implementation on a PC.

V. CONCLUSION

The measurements on a single DBoard have shown, that ARUZ implementation of the Lattice Boltzmann Method would have the performance of $206 \cdot 10^3$ MLUPS, $52 \cdot 10^3$ times better than presented in [11]. What is important, these computation are done in full IEEE single precision.

The results of automatic synthesis are still not optimal. For example, the HLS tool uses full floating point cores to perform multiplication by 2.0 or 0.5. These operations can be performed much more effectively as they require only an incrementation or decrementation of an exponent, however the Vivado tool is not able to infer it automatically.

Appendix 1: The collision function source code

```
void lbm_node(
  float f0,
  float f1,
  float f2,
  float f3,
  float f4,
  float f5,
  float f6,
  float f7,
  float f8,

  float *f0_out,
  float *f1_out,
  float *f2_out,
  float *f3_out,
  float *f4_out,
  float *f5_out,
  float *f6_out,
  float *f7_out,
  float *f8_out,
  float omega,
  float* energy,
  float rho
) {

  float rhoux;

  float MC_tmp1 = f5 - f7;
  float MC_tmp2 = f6 - f8;
  float MC_tmp3 = f1 - f3;
  float MC_tmp4 = f2 - f4;

  float MC_tmp1mMC_tmp2 = MC_tmp1 - MC_tmp2;
  float MC_tmp1pMC_tmp2 = MC_tmp1 + MC_tmp2;

#if BCTYPE==BC_BULK
      rho = ((f0 + f1) + (f2 + f3)) +
        ((f4 + f5) + (f6 + f7)) + f8;
#endif

#if BCTYPE==BC_BOTTOMLEFT
      f1 = f3;
      MC_tmp3 = 0;
      f2 = f4;
      MC_tmp4 = 0;
      f5 = f7;
      MC_tmp1 = 0;
      f6 = f8 = 0.5f *
        (rho - (((f0 + f1) + (f2 + f3))
        + ((f4 + f5) + f7)));
      MC_tmp2 = 0;
      MC_tmp1mMC_tmp2 = 0;
      MC_tmp1pMC_tmp2 = 0;
#endif

#if BCTYPE==BC_TOPLEFT
      f4 = f2;
      MC_tmp4 = 0;
      f1 = f3;
      MC_tmp3 = 0;
      f8 = f6;
      MC_tmp2 = 0;
      f5 = f7 = 0.5f *
        (rho - (((f0 + f1) + (f2 + f3))
        + ((f4 + f6) + f8)));
      MC_tmp1 = 0;
      MC_tmp1mMC_tmp2 = 0;
      MC_tmp1pMC_tmp2 = 0;
#endif

#if BCTYPE==BC_TOPRIGHT
      f3 = f1;
      MC_tmp3 =0;
      f4 = f2;
      MC_tmp4 = 0;
      f7 = f5;
      MC_tmp1 = 0;
      f6 = f8 = 0.5f *
        (rho - (((f0 + f1) + (f2 + f3))
        + ((f4 + f5) + f7)));
      MC_tmp2 = 0;
      MC_tmp1mMC_tmp2 = 0;
      MC_tmp1pMC_tmp2 = 0;
#endif

#if BCTYPE==BC_BOTTOMRIGHT
      f2 = f4;
      MC_tmp4 = 0;
      f3 = f1;
      MC_tmp3 = 0;
      f6 = f8;
      MC_tmp2 = 0;
      f7 = f5 = 0.5f *
        (rho-(((f0 + f1) + (f2 + f3))
        + ((f4 + f6) + f8) ));
      MC_tmp1 = 0;
      MC_tmp1mMC_tmp2 = 0;
      MC_tmp1pMC_tmp2 = 0;
#endif

#if BCTYPE==BC_LEFTDENSITY
      rhoux = rho - ((((f0 + f4) + f2) +
        ((f3  + f3) + (f6 + f6)))
        + (f7 + f7));
      MC_tmp3 = 2.0f / 3.0f * rhoux;
      float MC_tmp7 = 1.0f / 6.0f * rhoux;
      float MC_tmp8 = 1.0f / 2.0f * MC_tmp4;
      f1 = f3 + MC_tmp3;
      MC_tmp2 =  -MC_tmp7 - MC_tmp8;
      f8 = f6 - MC_tmp2;
      MC_tmp1 = MC_tmp7 - MC_tmp8;
      f5 = f7 + MC_tmp1;

      MC_tmp1mMC_tmp2 = 0.5f * MC_tmp3;
      MC_tmp1pMC_tmp2 = -MC_tmp4;
#endif

#if BCTYPE==BC_RIGHTDENSITY
      rhoux = rho - ((((f0 + f4) + f2) +
        ((f1 + f1) + (f8 + f8)))
        + (f5 + f5));
      MC_tmp3 = -2.0f / 3.0f * rhoux;
      float MC_tmp7 = 1.0f / 6.0f * rhoux;
      float MC_tmp8 = 1.0f / 2.0f * MC_tmp4;
      f3 = f1 - MC_tmp3;
      MC_tmp2 = MC_tmp7 - MC_tmp8;
      f6 = f8 + MC_tmp2;
      MC_tmp1 = -MC_tmp7 - MC_tmp8;
      f7 = f5 - MC_tmp1;
      MC_tmp1mMC_tmp2 = 0.5f * MC_tmp3;
      MC_tmp1pMC_tmp2 = -MC_tmp4;
#endif

  float one_rho = 1.0f / rho;

  float rho_u = MC_tmp1mMC_tmp2 + MC_tmp3;
  float rho_v = MC_tmp1pMC_tmp2 + MC_tmp4;
  float MC_tmp5 = MC_tmp3 + MC_tmp4;
  float MC_tmp6 = MC_tmp3 - MC_tmp4;
  float rho_uPv = MC_tmp5 +
    (MC_tmp1 +  MC_tmp1);
  float rho_uMv = MC_tmp6 -
    (MC_tmp2 + MC_tmp2);
  float rho_u_sqd = rho_u * rho_u;
  float rho_v_sqd = rho_v * rho_v;
  float the3rdTerm = rho_u_sqd + rho_v_sqd;
  *energy = the3rdTerm;
  float feq0 = 4.0f/9.0f * rho -
    2.0f/3.0f * the3rdTerm) * one_rho;

  float EQ_tmp1 = 1.0f/9.0f * rho;
  float EQ_tmp2 = 1.0f/3.0f * rho_u;
  float EQ_tmp3 = 1.0f/3.0f * rho_v;
  float EQ_tmp4 = 1.0f/2.0f * rho_u_sqd;
  float EQ_tmp5 = 1.0f/2.0f * rho_v_sqd;
  float EQ_tmp6 = 1.0f/6.0f * the3rdTerm;
  float EQ_tmp7 = (EQ_tmp4 - EQ_tmp6)
    * one_rho;
  float EQ_tmp8 = (EQ_tmp5 - EQ_tmp6)
    * one_rho;
  float EQ_tmp9 = EQ_tmp1 + EQ_tmp7;
  float EQ_tmp10 = EQ_tmp1 + EQ_tmp8;
  float feq1 = EQ_tmp9 + EQ_tmp2;
  float feq3 = EQ_tmp9 - EQ_tmp2;
  float feq2 = EQ_tmp10 + EQ_tmp3;
  float feq4 = EQ_tmp10 - EQ_tmp3;

  float EQ_tmp11 = 1.0f/36.0f * rho;
  float EQ_tmp12 = 1.0f/12.0f * rho_uPv;
  float EQ_tmp13 = 1.0f/12.0f * rho_uMv;
  float EQ_tmp14 = 1.0f/8.0f *
    (rho_uPv * rho_uPv);
  float EQ_tmp15 = 1.0f/8.0f *
    (rho_uMv * rho_uMv);
  float EQ_tmp16 = 1.0f/24.0f * the3rdTerm;
  float EQ_tmp17 = (EQ_tmp14 - EQ_tmp16)
    * one_rho;
  float EQ_tmp18 = (EQ_tmp15 - EQ_tmp16)
    * one_rho;
  float EQ_tmp19 = EQ_tmp11 + EQ_tmp17;
  float EQ_tmp20 = EQ_tmp11 + EQ_tmp18;

  float feq5 = EQ_tmp19 + EQ_tmp12;
  float feq7 = EQ_tmp19 - EQ_tmp12;
  float feq6 = EQ_tmp20 - EQ_tmp13;
  float feq8 = EQ_tmp20 + EQ_tmp13;

  float deltaf0 = (omega)*(f0 - feq0);
  float deltaf1 = (omega)*(f1 - feq1);
  float deltaf2 = (omega)*(f2 - feq2);
  float deltaf3 = (omega)*(f3 - feq3);
  float deltaf4 = (omega)*(f4 - feq4);
  float deltaf5 = (omega)*(f5 - feq5);
  float deltaf6 = (omega)*(f6 - feq6);
  float deltaf7 = (omega)*(f7 - feq7);
  float deltaf8 = (omega)*(f8 - feq8);

  *f0_out = f0 - deltaf0;
  *f1_out = f1 - deltaf1;
  *f2_out = f2 - deltaf2;
  *f3_out = f3 - deltaf3;
  *f4_out = f4 - deltaf4;
  *f5_out = f5 - deltaf5;
  *f6_out = f6 - deltaf6;
  *f7_out = f7 - deltaf7;
  *f8_out = f8 - deltaf8;
}
```

References

[1] R. Kiełbik, G. Jabłoński, P. Amrozik, Z. Mudza, and J. Kupis, "Aruz - the unique massively parallel fpga-based system," in *Dedicated parallel machines – a breakthrough in computation ARUZ-Workshop 2016, Lodz, Poland, 1-3 December 2016*, 2016, pp. 8–9.

[2] T. Pakuła and J. Teichmann, "Model for relaxation in supercooled liquids and polymer melts," in *Materials Research Society Symposium – Proceedings, Volume 455*, 1996, p. 211.

[3] P. Polanowski, J. Jung, and R. Kielbik, "Special purpose parallel computer for modelling supramolecular systems based on the dynamic lattice liquid model," *Computational Methods in Science and Technology*, vol. 16, no. 2, pp. 147–153, 2010.

[4] K. Hałagan, P. Polanowski, J. Jung, and M. Kozanecki, "Modelling of complex liquids with cooperative dynamics using aruz," in *Dedicated parallel machines – a breakthrough in computation ARUZ-Workshop 2016, Lodz, Poland, 1-3 December 2016*, 2016, pp. 10–11.

[5] G. R. McNamara and G. Zanetti, "Use of the boltzmann equation to simulate lattice-gas automata," *Phys. Rev. Lett.*, vol. 61, pp. 2332–2335, Nov 1988.

[6] U. Frisch, B. Hasslacher, and Y. Pomeau, "Lattice-gas automata for the navier-stokes equation," *Phys. Rev. Lett.*, vol. 56, pp. 1505–1508, Apr 1986.

[7] D. A. Wolf-Gladrow, *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*. Springer, 2004.

[8] S. Succi, *The lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press, 2001.

[9] Q. Zou and X. He, "On pressure and velocity boundary conditions for the lattice boltzmann bgk model," *Physics of fluids*, vol. 9, no. 6, pp. 1591–1598, 1997.

[10] K. Sano, O. Mencer, and W. Luk, "Fpga-based acceleration of the lattice boltzmann method," in *Proceedings of the International Conference on Parallel Computational Fluid Dynamics (ParCFD2007) CDROM (paper-041)*, 2007.

[11] K. Sano, O. Pell, W. Luk, and S. Yamamoto, "Fpga-based streaming computation for lattice boltzmann method," in *2007 International Conference on Field-Programmable Technology, ICFPT 2007, Kitakyushu, Japan, December 12-14, 2007*, 2007, pp. 233–236.

[12] K. Sano, Y. Kono, H. Suzuki, R. Chiba, R. Ito, T. Ueno, K. Koizumi, and S. Yamamoto, "Efficient custom computing of fully-streamed lattice boltzmann method on tightly-coupled fpga cluster," *SIGARCH Comput. Archit. News*, vol. 41, no. 5, pp. 47–52, Jun. 2014.

[13] G. Jabloński and J. Kupis, "Performance estimation of lattice boltzmann method implementation in aruz," in *2017 MIXDES - 24th International Conference "Mixed Design of Integrated Circuits and Systems*, June 2017, pp. 308–313.

[14] A. G. Shet, S. H. Sorathiya, S. Krithivasan, A. M. Deshpande, B. Kaul, S. D. Sherlekar, and S. Ansumali, "Data structure and movement for lattice-based simulations," *Phys. Rev. E*, vol. 88, p. 013314, Jul 2013.

Grzegorz Jabłoński was born in 1970. He received MSc and PhD degrees in electrical engineering from Lodz University of Technology in 1994 and 1999 respectively. He is currently an Assistant Professor in the Department of Microelectronics and Computer Science Lodz University of Technology. His research interests include compiler construction, microelectronics, simulation of electronic circuits and semiconductor devices, thermal problems in electronics, digital electronics, embedded systems and programmable devices.

Joanna Kupis is a PhD student in the Department of Microelectronics and Computer Science, Lodz University of Technology.