FPGA-based Acceleration of the Lattice Boltzmann Method

Kentaro Sano[a*], Oskar Mencer[b] and Wayne Luk[b]

[a] *Department of Computer and Mathematical Sciences*
Graduate School of Information Sciences, Tohoku University
6-6-01 Aramaki Aza Aoba, Sendai, 980-8579, Japan
kentah@caero.mech.tohoku.ac.jp

[b] *Department of Computing*
Imperial College, London
180 Queen's Gate, London SW7 2BZ, United Kingdom
{oskar, wl}@doc.ic.ac.uk

Keywords: Lattice Boltzmann Method, FPGA, Stream Architecture, A Stream Compiler (ASC)

1. Introduction

FPGAs (Field-Programmable Gate Arrays) are becoming more attractive to high-performance scientific computing. FPGAs are high volume, off-the-shelf semiconductor devices containing programmable logic components, embedded arithmetic units, embedded memories and their programmable interconnection network. FPGAs have remarkably increased their potential for high-performance computing by integrating much more programmable hardware resources and increasing their operating frequency, and therefore recent leading-edge FPGAs can have peak floating-point computation performance surpassing that of typical microprocessors [1]. By designing a custom computing machine (CCM) on FPGAs, the properties, e.g., parallelism, regularity and homogeneity, of a specific application can be efficiently exploited by customized data-paths, customized arithmetic units and customized memory systems.

Although FPGAs are programmable, programming FPGAs requires designing hardware. Therefore, it is very difficult for software programmers to implement CCMs for specific applications on FPGAs without knowledge of hardware design. Recently, A Stream Compiler (ASC) [2] solves this designing problem for FPGAs. By automating the production of CCMs that process streamed data, ASC allows users to write code with statements similar to the C language [2][3]. ASC also supports floating-point computations with flexible precisions, which are very suitable for efficient resource utilization on FPGAs.

We consider which application is suitable for FPGAs, and how to make it work. This paper shows that the lattice Boltzmann method (LBM) is suitable for stream processing; an FPGA-based stream accelerator only at 67MHz, implemented with the x1 transfer rate of PCI-Express, achieves 1.15 times faster LBM computation than a 2.2GHz Opteron processor. We estimate the speedup of an FPGA-based stream accelerator with the x8 transfer rate at 7.68. LBM computes fluid flow by tracking fictive particles on a gird. Although relatively large data-sets are necessary to define multiple particle distribution functions on each grid-point, the algorithm for LBM has simplicity and parallelism among grid-points. These properties are appropriate for direct hardware implementation. With a state-of-the-art FPGA, we design and implement a stream accelerator for 2D LBM computation. In the following sections, we describe the stream-based LBM computation and its efficient implementation on an FPGA.

Related work has shown that FPGAs have significant potential for computational fluid dynamics. For instance, a single FPGA implementation of a 3D lattice gas model [4] can run 200 times faster than a software version on a 1.8GHz Athlon processor. It has also been reported that FPGA-based accelerators for computational fluid dynamics [5] promise large improvement in sustained performance at better price-performance ratios with lower overall power consumption than conventional processors.

2. Stream in Lattice Boltzmann Method for the 2D9V Model

The lattice Boltzmann method [6][7] models fluids with fictive particles performing propagation and collision processes over a discrete lattice mesh. Here, we describe the lattice Boltzmann method of the 2D orthogonal 9-

* Corresponding Author.

speed (2D9V) model. In this model, each grid point has a distribution function $f_i(\mathbf{x},t)$ for each of the nine particle speeds $\mathbf{c}_i = (c_{xi}, c_{yi})$ shown in Fig.1.

$$c_{xi} = \frac{\Delta x}{\Delta t}\cos\left(\frac{2\pi(i-1)}{8}\right) \quad \text{and} \quad c_{yi} = \frac{\Delta x}{\Delta t}\sin\left(\frac{2\pi(i-1)}{8}\right). \tag{1}$$

The propagation and collision processes of fictive particles are given by the following equation:

$$f_i(\mathbf{x}+\mathbf{c}_i\Delta t,\, t+\Delta t) = f_i(\mathbf{x},\, t) - \frac{1}{\tau}\left(f_i(\mathbf{x},\, t) - f_i^{\text{eq}}(\mathbf{x},\, t)\right) \tag{2}$$

where $f_i^{\text{eq}}(\mathbf{x},t)$ is an equilibrium distribution function, τ is a single relaxation time, and Δt is a time step for particles to move to a grid point from the adjacent point. We compute the fluid density ρ and the fluid velocity $\mathbf{V} = (u,v)$ from the distribution functions $f_i(\mathbf{x},t)$ by

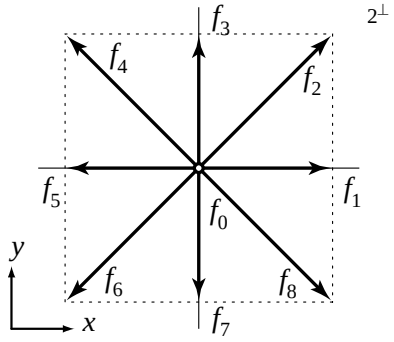


Fig.1 Particle distribution functions for the nine-velocity (2D9V) model.

$$\rho = \sum_i f_i(\mathbf{x},t) \quad \text{and} \quad \rho\mathbf{V} = \sum_i \mathbf{c}_i f_i(\mathbf{x},t). \tag{3}$$

The equilibrium distribution functions $f_i^{\text{eq}}(\mathbf{x},t)$ are given by the following equations:

$$f_0^{\text{eq}} = \rho\left\{\frac{4}{9} - \frac{2}{3}\mathbf{V}^2\right\}, \tag{4}$$

$$f_i^{\text{eq}} = \rho\left\{\frac{1}{9} + \frac{1}{3}(\mathbf{c}_i\cdot\mathbf{V}) + \frac{1}{2}(\mathbf{c}_i\cdot\mathbf{V})^2 - \frac{1}{6}\mathbf{V}^2\right\} \quad \text{for } i = 1,2,3,4, \tag{5}$$

$$f_i^{\text{eq}} = \rho\left\{\frac{1}{36} + \frac{1}{12}(\mathbf{c}_i\cdot\mathbf{V}) + \frac{1}{8}(\mathbf{c}_i\cdot\mathbf{V})^2 - \frac{1}{24}\mathbf{V}^2\right\} \quad \text{for } i = 5,6,7,8. \tag{6}$$

Eq. (2) is solved by the following stages.

1. Translation
2. Macroscopic physical quantity calculation
3. Equilibrium calculation
4. Collision calculation
5. Termination evaluation

In the translation stage, we propagate particles by moving the right-hand side value of Eq. (2) at $\mathbf{x}$ to the adjacent point at $(\mathbf{x}+\mathbf{c}_i\Delta t)$ with its particle speed $\mathbf{c}_i$. In the macroscopic calculation stage, we compute ρ and $\mathbf{V} = (u,v)$ by Eq. (3). In the equilibrium calculation stage, we compute $f_i^{\text{eq}}(\mathbf{x},t)$ by Eqs. (4) to (6). In the collision stage, we compute the right-hand side of Eq. (2). Finally, the computation terminates based on some conditions, e.g., convergence condition, or we go back to the translation stage.

For the stream architecture [2], we focus on Stages 2 to 4, which are completely independent among different grid points. We define the stream architecture of Stages 2 to 4 as follows.

Inputs: $f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8$ for different $\mathbf{x}$

Computation: Macroscopic stage (Eq. (3)), Equilibrium stage (Eq. (4), Eq. (5) and Eq. (6)) and Collision stage (the right-hand side of Eq. (2))

Outputs: $f_0^{\text{new}}, f_1^{\text{new}}, f_2^{\text{new}}, f_3^{\text{new}}, f_4^{\text{new}}, f_5^{\text{new}}, f_6^{\text{new}}, f_7^{\text{new}}, f_8^{\text{new}}$ and ρ

Here, $f_i^{\text{new}}(\mathbf{x},t)$ is the right-hand side of Eq. (2). In the next section, we describe optimized equations and scheduling of their computations for efficient FPGA utilization.

3. Computational Optimization for FPGA

Since an FPGA has limited resources for user-defined circuits, optimizing computations is very important for the circuit to fit into an FPGA with high throughput and enough floating-point precision. ASC [3] generates the same number of units as the number of operations, i.e., +,-,* and / in ASC code. Therefore, the same computation should appear only once in the code. Moreover, the order of computations affects the data-flow graph generated by ASC. The depth and width of the generated data-path can be controlled by bracketing equations. For these goals, we rewrite Eqs. (3) to (6) as follows. The computations in the macroscopic stage are:

$$\rho = f_0 + f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7 + f_8. \tag{7}$$

$$\rho u = (f_2 - f_6) - (f_4 - f_8) + (f_1 - f_5). \tag{8}$$

$$\rho v = (f_2 - f_6) + (f_4 - f_8) + (f_3 - f_7). \tag{9}$$

$$\rho(u+v) = (f_1 - f_5) + (f_3 - f_7) + 2(f_2 - f_6). \quad (10)$$

$$\rho(u-v) = (f_1 - f_5) + (f_3 - f_7) - 2(f_4 - f_8). \quad (11)$$

We add computations of $\rho(u+v)$ and $\rho(u-v)$ in the macroscopic to be used in the equilibrium stage. The terms $\rho(u+v)$ and $\rho(u-v)$ share the members of Eq. (8) and Eq. (9), and therefore only the three adders/subtractors and the two multipliers are additionally necessary to compute Eq. (10) and Eq. (11).

The computations in the equilibrium stage, Eqs. (4) to (6), are written in the following common form.

$$f_i^{\mathrm{eq}} = \mathrm{A}\rho + \mathrm{B}\begin{pmatrix} \rho u, \\ \rho v, \\ \rho(u+v), \text{or} \\ \rho(u-v) \end{pmatrix} + \frac{1}{\rho}\left\{ \mathrm{C}\begin{pmatrix} \rho u, \\ \rho v, \\ \rho(u+v), \text{or} \\ \rho(u-v) \end{pmatrix}^2 - \mathrm{D}\left[(\rho u)^2 + (\rho v)^2\right] \right\} \quad \text{for } i = 0,1,...,8 \quad (11)$$

where A, B, C and D are constants. In Eq. (11), the second and third terms are just replaced with selections from ρ, ρu, ρv, $\rho(u+v)$ and $\rho(u-v)$, instead of the inner product of $\mathbf{c}_i$ and $\mathbf{V}$. This conversion requires the computation of $1/\rho$ which takes a long time; however $1/\rho$ is pipelined delivering a result in the late pipeline stages. In Fig.2, we show the main computational part of our code written for ASC.

4. Implementation and Results

We implement the stream accelerator for 2D9V LBM by using ASC and a PCI-Express FPGA card, MAX-I, both of which are the products of Maxeler Technologies Inc [8]. Fig. 3 shows the block diagram of MAX-I, where we have a Xilinx Virtex4 FPGA XC4VFX100, two DDR2 RAMs, and a PCI-Express interface with up to x8 transfer rate. This Virtex4 FPGA contains 42,176 slices and 160 DSPs of 18-bit integers. The bidirectional peak data transfer rate between MAX-I and a host PC is theoretically 4GB/sec with the x8 mode of PCI-Express. A PC with an AMD Opteron processor model 248 running at 2.2GHz serves as a host PC.

We implement the prototype of the stream accelerator with the x1 mode of PCI-Express. This circuit operates at 67MHz and uses efficient precision for floating-point numbers; with 10-bit exponent and 18-bit mantissa including a sign bit and a leading bit. This circuit consumes 36,396 slices and 30 DSP blocks of the Vertex4 FPGA. For comparison, we also implement the stream computation in C++ for the Opteron processor. We compile the C++ code for the Opteron processor by using gcc with full optimization options. For both FPGA and software computations, the code of the remaining translation and termination evaluation stages is written in C++ to be executed by the host PC. In the translation stage, boundary computations are also performed. The host PC performs all the floating-point computations in single precision.

We compute a time-dependent 2D flow between two parallel plate surfaces as shown in Fig. 4. There is an obstacle between the two parallel plates. The left and right sides in Fig. 4 are open and referred to as *an inlet* and *an outlet*, respectively. In this computation, the inlet and the outlet have their constant ρ, 1.0 and 0.9 respectively, to cause a flow. The computational grid has 360x180 points, and boundary computations on the upper and lower surfaces are performed as proposed in [7]. Before computation, we initialize the grid points with $f_i^{\mathrm{eq}}(\mathbf{x},t)$ given by Eqs. (4) to (6) with $\mathbf{V} = (0,0)$ and linearly interpolated ρ between the inlet and the outlet. τ was 0.80375 giving RE=200.

Figs. 5 and 6 show the computational results at time step 1200 and 4400, respectively. The software and the FPGA show similar results in the early time steps; however the difference between the software and the FPGA increases as a time-step proceeds because of the FPGA's low precision computation in the current implementation with the 18-bit mantissa. Accumulation of slight difference in computation causes quite a different flow field. However, the precision problem is resolved by further optimizing resource utilization so that 32-bit or more precision is available on an FPGA.

We measure the time of the stream computation in each time step by using the gettimeofday() function. In the case of software computation, the Opteron processor running at 2.2GHz takes 0.0190 seconds on average, while an Opteron processor running at 2.8GHz takes 0.0137 seconds on average. On the other hand, the FPGA-based computation takes 0.0164 seconds. This means that with the x1 mode, we achieve 1.15 and 0.833 times faster computation by FPGA than those by the AMD Opteron processors running at 2.2GHz and 2.8GHz, respectively. Since we have the input of 9 values and the output of 10 values for each of 360x180 grid-points in each time step, the measured I/O bandwidth of the current implementation with the x1 mode is $360 \times 180 \times (9+10) \times 4 / 0.0164 = 286.4$MB/sec. The peak bidirectional bandwidth of the x1 PCI-Express is 500MB/sec, and therefore 57% of the total bandwidth is practically utilized. The FPGA card that we use for implementation has the x4 and x8 modes of PCI-Express. Moreover, we expect that the circuit on the FPGA has enough throughput for the x8 transfer rate. Since the x4 mode and the x8 mode give the actual transfer rates of

900MB/sec and 1900MB/sec for another application, respectively, the estimated speedups are 3.64 for the x4 mode and 7.68 for the x8 mode in comparison with the Opteron processor running at 2.2GHz. Fig. 7 summarizes the above performance results and estimation.

5. Conclusions

This paper shows the FPGA-based stream accelerator for the 2D9V lattice Boltzmann method. Although the accelerator is currently only implemented in the x1 mode of PCI-Express, the FPGA achieves faster computation than the AMD Opteron processor running at 2.2GHz. Both the precision and the throughput will be improved in the optimized code for the same FPGA with the x8 mode. In addition to the code optimization, we are designing the stream accelerator for all the stages in LBM. By using the embedded RAMs of an FPGA, we can implement the translation stage. For the boundary computations and the termination evaluation, more complicated circuits are necessary. If we implement the advanced stream accelerator for LBM, the stand-alone computation without a host PC achieves much higher performance. In addition, we can also accelerate the 3D case of LBM in the same framework. Design and implementation of the stream accelerator for the 3D LBM is left to future work.

References

[1] K. Underwood, “FPGAs vs. CPUs: Trends in peak floating-point performance,” Proceedings of the International Symposium on Field-Programmable Gate Arrays, pp. 171–180, 2004.
[2] Oskar Mencer, “ASC: a stream compiler for computing with FPGAs,” IEEE Trans. on CAD of Integrated Circuits and Systems, vol.25, no. 9, pp.1603-1617, 2006.
[3] Evgeny Fiksman, Yitzhak Birk and Oskar Mencer, “ASC-Based Acceleration in an FPGA with a Processor Core Using Software-Only Skills,” Procs. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM2006), pp.271-272, 2006.
[4] Tomoyoshi Kobori and Tsutomu Maruyama, “A High Speed Computation System for 3D FCHC Lattice Gas Model with FPGA,” Procs. of International Conference on Field-Programmable Logic and Applications (FPL2003), pp.755-765, 2003.
[5] William D. Smith and Austars R. Schnore, “Towards an RCC-Based Accelerator for Computational Fluid Dynamics Applications,” The Journal of Supercomputing, vol. 30, no. 3, pp.239-261, 2004.
[6] P. A. Skordos, “Initial and boundary conditions for the lattice Boltzmann method,” Physical Review E, vol. 48, no. 6, pp.4823-4842, 1993.
[7] Takaji Inamuro, Masato Yoshino and Fumimaru Ogino, “A non-slip boundary condition for lattice Boltzmann simulations,” Physics of Fluids, vol. 7, no. 12, pp.2928-2930, 1995.
[8] http://www.maxeler.com

```
/*************** MACRO calculation stage ***************/
rho     = ((f2 + f6) + (f4 + f8)) + ((f1 + f5) + (f3 + f7)) + f0;
MC_tmp1 = f2 - f6;
MC_tmp2 = f4 - f8;
MC_tmp3 = f1 - f5;
MC_tmp4 = f3 - f7;
rho_u       = MC_tmp1 - MC_tmp2 + MC_tmp3;
rho_v       = MC_tmp1 + MC_tmp2 + MC_tmp4;
MC_tmp5 = MC_tmp3 + MC_tmp4;
MC_tmp6 = MC_tmp3 - MC_tmp4;
rho_uPv    = MC_tmp5 + 2.0 * MC_tmp1;
rho_uMv    = MC_tmp6 - 2.0 * MC_tmp2;

/************** Equilibrium calculation stage **************/
rho_u_sqd     = rho_u * rho_u;
rho_v_sqd     = rho_v * rho_v;
the3rdTerm    = rho_u_sqd + rho_v_sqd;
feq0 = (4.0/9.0) * rho - ((2.0/3.0) * the3rdTerm) * one_rho;

EQ_tmp1    = (1.0/9.0) * rho;
EQ_tmp2    = (1.0/3.0) * rho_u;
EQ_tmp3    = (1.0/3.0) * rho_v;
EQ_tmp4    = (1.0/2.0) * rho_u_sqd;
EQ_tmp5    = (1.0/2.0) * rho_v_sqd;
EQ_tmp6    = (1.0/6.0) * the3rdTerm;
EQ_tmp7    = (EQ_tmp4 - EQ_tmp6) * one_rho;
EQ_tmp8    = (EQ_tmp5 - EQ_tmp6) * one_rho;
EQ_tmp9    = EQ_tmp1 + EQ_tmp7;
EQ_tmp10 = EQ_tmp1 + EQ_tmp8;
feq1 = EQ_tmp9    + EQ_tmp2;
feq5 = EQ_tmp9    - EQ_tmp2;
feq3 = EQ_tmp10 + EQ_tmp3;
feq7 = EQ_tmp10 - EQ_tmp3;

EQ_tmp11 = (1.0/36.0) * rho;
EQ_tmp12 = (1.0/12.0) * rho_uPv;
EQ_tmp13 = (1.0/12.0) * rho_uMv;
EQ_tmp14 = (1.0/8.0) * (rho_uPv*rho_uPv);
EQ_tmp15 = (1.0/8.0) * (rho_uMv*rho_uMv);
EQ_tmp16 = (1.0/24.0) * the3rdTerm;
EQ_tmp17 = (EQ_tmp14 - EQ_tmp16) * one_rho;
EQ_tmp18 = (EQ_tmp15 - EQ_tmp16) * one_rho;
EQ_tmp19 = EQ_tmp11 + EQ_tmp17;
EQ_tmp20 = EQ_tmp11 + EQ_tmp18;
feq2 = EQ_tmp19 + EQ_tmp12;
feq6 = EQ_tmp19 - EQ_tmp12;
feq4 = EQ_tmp20 - EQ_tmp13;
feq8 = EQ_tmp20 + EQ_tmp13;
```

Fig.2 The main computational part of the code written for ASC.

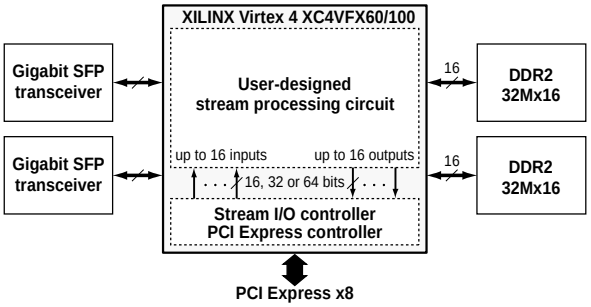


Fig.3 The block diagram of the FPGA card, MAX-I.

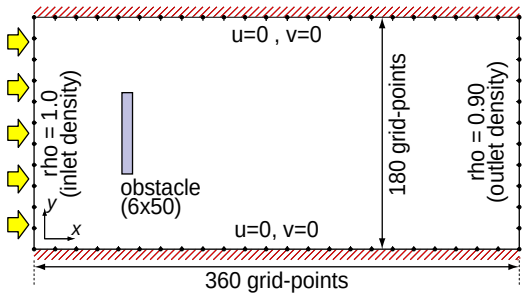
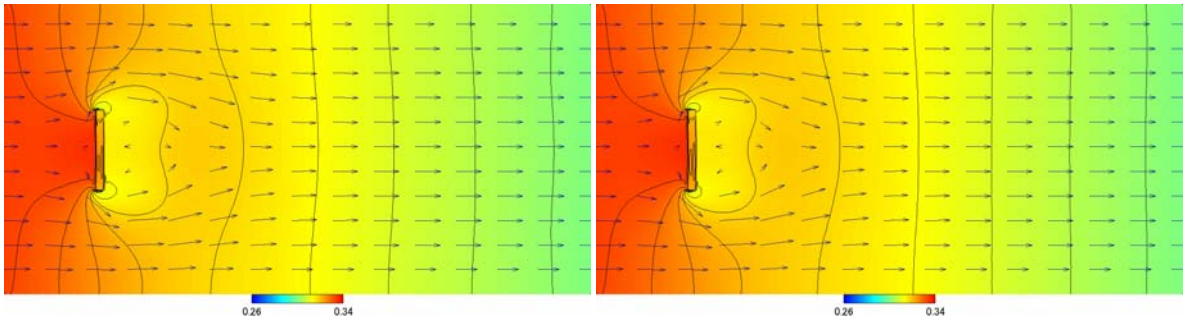
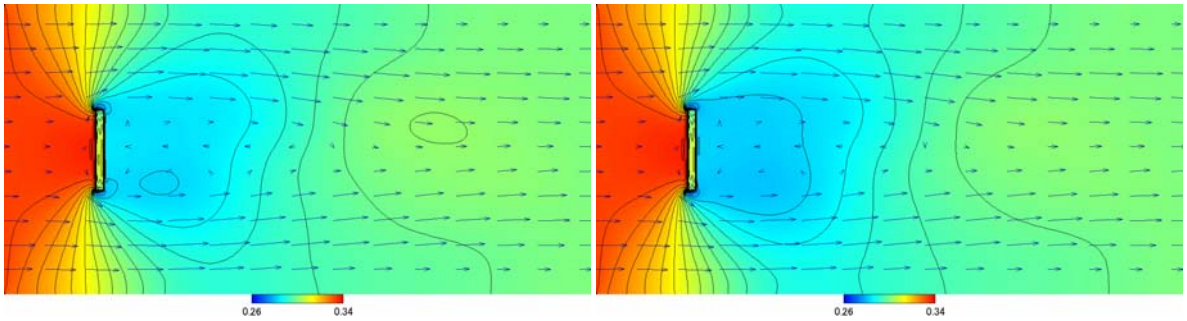


Fig.4 Conditions for the benchmark computation.



a. software b. FPGA

Fig.5 Computational results at time-step 1200.



a. software b. FPGA

Fig.6 Computational results at time-step 4400.

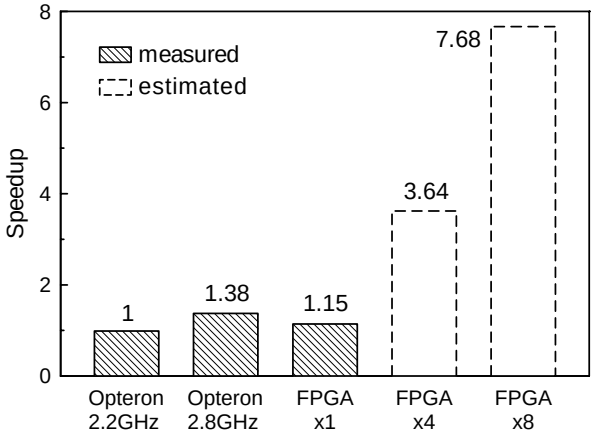


Fig.7 The measured and estimated performance comparison. FPGA acceleration has the x1, x4 or x8 transfer mode of PCI-Express.