## High performance reconfigurable computing with cellular automata

Murtaza, S.

**Publication date**
2010

Chapter 5

# Compute Bound CA on FPGA*

After discussing bitwise operation based CA algorithms in the previous chapter, we next move on to the numeric computation based CA algorithms like the lattice Boltzmann method ([92] explains how we can view LBM as a generalised CA). The lattice Boltzmann method is a well known stencil based numerical technique for physical simulations. The floating-point numbers constitute the state of a lattice Boltzmann cell, therefore, its next state computation is based on floating-point operations. Floating-point based CA computations result in a longer cell update and more FPGA resource utilisation per PE implementation. The compute time and the FPGA resource utilisation, per lattice Boltzmann method PE implementation, suits our goal of validating of our performance model for compute bound computations.

Once a CA algorithm is categorised as compute bound using the formulation as presented in Section 3.3 and considering the floating-point computation requirements, formulating a hardware design to ensure the optimal utilisation of FPGA resources and the overall performance gain is a challenge. Hardware design has to ensure that the source and destination memory are not overwhelmed by the computation engine's data streams. Additionally, each PE implementation demands huge FPGA resources, hence pushing more PEs in a design becomes challenging.

In this chapter, using the D2Q9 Lattice Boltzmann Method, we implement and validate our proposed performance model for the compute bound two-dimensional CA algorithm. Hardware algorithms, employed to improve the overall system performance, are also discussed in detail. The chapter concludes with performance results and the possible future extensions.

*This chapter is based on the following publications:

- S. Murtaza and A.G. Hoekstra and P.M.A. Sloot, 'Compute Bound and I/O Bound Cellular Automata Simulations on FPGA logic', *ACM Transactions on Reconfigurable Technology and Systems*, **1**, 1-21 (2009)

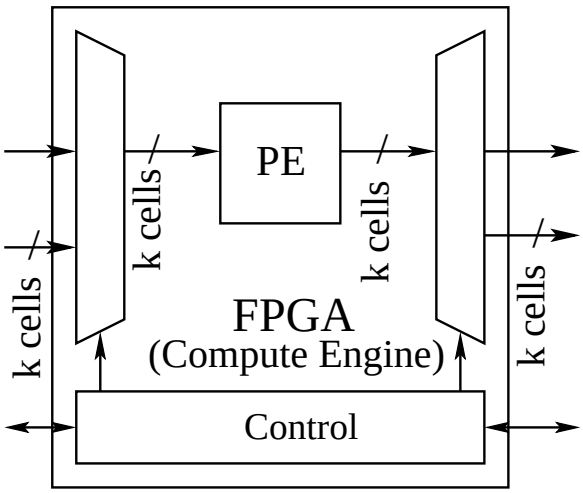


**Figure 5.1:** *A single PE LBM implementation.*

## 5.1 Compute Bound 2D CA on FPGA

Influenced by the computational structure and quite high FPGA resource utilisation per PE, we chose LBM as a test case for our compute bound CA accelerator implementation. And to maximise the overall FPGA resource utilisation, we begin by extending a single PE to a $p$ PE implementation, as shown in Figure 5.1 and 3.2 respectively.

### 5.1.1 Computation Model

The computation method for our compute bound CA accelerator implementation as shown in Figure 3.2 is same as specified in Equation (3.10). We start the computations with an assumption that each of the CA cell needs to compute its collision function followed by the propagation function. So the input data is the state of a cell (nine numbers representing the particle densities in the D2Q9 model) plus memory locations to which the post collision particle densities must be propagated. For the computation of the collision function, each of the cells is self contained, that is, has all the data it requires to compute its collision function. This simplifies the accelerator implementation, with the simplest scenario being a single PE implementation as shown in Figure 3.3 top panel. A single PE implementation layout is as shown in Figure 5.1. PE as explained in next section, is the hardware core that implements the collision function of a compute bound CA algorithm. In order to start the computations for our chosen test bench and the PE implementation, a CE at least needs a single cell's data to set a PE running. To initialise the CE, the specified CA cell's data (that is, nine 32-bit floating-point number, where eight are the particle densities from the cell's eight respective neighbours and the last one representing the cell's own particle at rest) is read from the source memory bank and stored to a PE's register file.

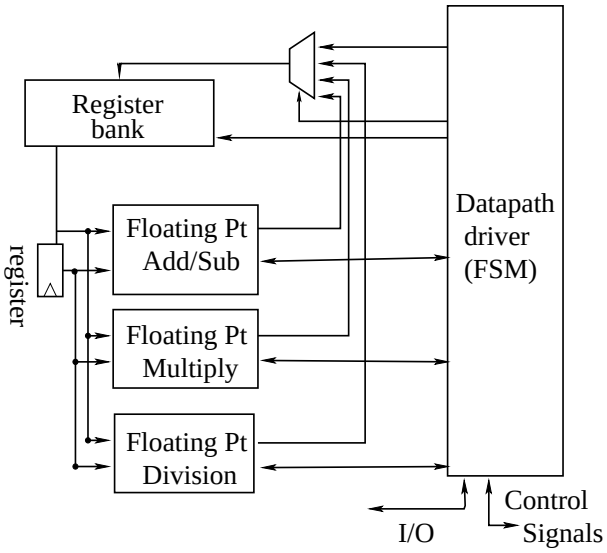


**Figure 5.2:** *PE implementing a LBM transition: Main components of a PE includes a) three floating point cores (each for sums, multiplications and divisions operations respectively) b) 64 register (each 32-bit wide) and control block to drive the engine.*

Once initialised, the PE is busy updating cells (collision computation phase), while in the meantime the CE buffers input data cells from the source memory bank in order to have it processed in the following cycles by the PE. As the PE computes each cell's new state, that is, nine new floating-point numbers (completion of collision phase), the CE writes them out to the specified location (propagation phase) in the destination memory bank.

For us to further improve and maximise the execution time and the FPGA hardware resource utilisation respectively, we implement $p$ PEs in the CE as shown in Figure 3.2 and have them update $p$ cells in parallel. The CE uses memory banks A and B alternatively as source and destination memory to hold the CA lattice. The maximum possible value for $p$ is defined by the FPGA resource utilisation per PE and the chosen D2Q9 LBM test bench's time to compute a cell. In order to appreciate how the chosen test bench's execution time limits $p$, assuming unlimited availability of FPGA resources, $p$ can be increased as long as $p \times \tau_{\mathrm{r}} < \tau_{\mathrm{c}}$ as shown in Figure 3.3 lower panel.

### 5.1.2 PE Design

Processing Engine (PE) implementation for our chosen test bench Lattice Boltzmann method as shown in Figure 5.2, includes a 64 x 32-bit register file, three floating point cores (one for addition, multiplication, and division respectively), and a control block to drive the data path. To compute a specified CA cell's next state, first the PE's register file is initialised with the nine 32-bit floating-point number that represent the cell's current state. With the PE initialised, the control block (a collection of finite state machines)

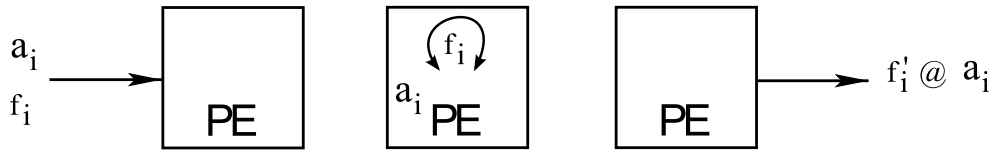


**Figure 5.3:** *Collision and Propagation: (Left) To compute the next state of a LBM cell, each cell's data, that is, particle distribution function represented as $f_i$ (see Chapter 2 for LBM introduction) and the propagation information as $a_i$ is fed to the PE. (Middle) Once loaded, PE stores propagation addresses within the internal buffers and computes the next state using $f_i$. (Right) Finally, the computed next state of the cell, $f_i'$ is written out (which is propagation part) to the specified memory locations $a_i$ in the destination memory bank.*

drives the data path to compute the collision function. The collision function computation includes around 75 floating-point operations that are performed sequentially over the given nine numbers using the required embedded floating-point cores within the PE. During the collision function computations, the register file is also used to store the intermediate results. Finally, the nine new floating-point numbers (completion of collision phase) are written out to specified memory locations in the destination memory bank, that is, the propagation phase. The three phases of a PE (initialisation, collision and propagation) to compute a specified CA cell's next state are as shown in the top, middle and the lower panel of Figure 5.3.

## 5.2 Test Cases

For demonstrating the proposed performance model for the compute bound two-dimensional CA algorithm, we implemented and validated our model for the D2Q9 Lattice Boltzmann Method. Since LBM calculations require the use of 32-bit floating-point numbers throughout the computations, this results in longer cell update and more FPGA resources per PE implementation. For floating-point computations multiple floating-point IP-cores are embedded and employed within each PE implementation. And for each LBM cell update, the PE performs 75 floating-point operations (38 additions/subtractions, 27 multiplications and 10 divisions). The compute time and FPGA resource utilisation per LBM PE implementation suits our goal of validation of our performance model for compute bound computations.

We implemented our LBM system with periodic boundary conditions as shown in Figure 3.2 with number of PEs $p$ equal to two, four, eight, and sixteen respectively. Each of the LBM implementations was tested for four different lattice sizes ($N$ equal to $8 \times 8$, $16 \times 16$, $32 \times 32$ and $64 \times 64$) that were computed for $g = 512$ generations.

The state of each LBM cell is defined by nine floating-point numbers and as a result the whole LBM lattice grid is composed of $9 \times N$ floating-point numbers. Since we start the computations with an assumption that each of the CA cells needs to compute its collision function followed by the propagation function, we also load the source memory bank with the data required for the propagation function. For the propagation function, each cell requires nine address locations, and for the whole LBM lattice this adds up to another $9 \times N$ numbers. Therefore, the source memory is loaded with $2 \times 9 \times N$ (32-bit) words. With source memory loaded, LBM CB starts reading cells from source memory, updates cells (collision function) in PEs, and writes out (propagation function) updated cells to destination memory all in parallel. Once the whole LBM lattice is updated for a single generation and loaded into destination memory accordingly, the computation comes to an end. Other parameters like $\tau_{\mathrm{r}}$, $\tau_{\mathrm{c}}$, and $\tau_{\mathrm{w}}$ depend on the FPGA chip and are discussed further in the results section.

## 5.3 Results

For the LBM implementation we used an ADM-XRC-4FX PCI Mezzanine board from Alpha-data. It is a Xilinx Virtex-4 FX140 based PMC with four independent 256MB DDR2-SDRAM banks. Each DDR2 bank can be accessed independently from the FPGA (user logic) and via PCI interface from the host machine. Detailed FPGA board specifications are presented in Appendix 10. The board comes with a software development kit, including drivers, header and library files, that support a C or C++ program running in the host machine to communicate directly with the board. Additional code is provided for board initialisation and selection, control of the programmable clocks and handling of FPGA configuration files. VHDL was used to describe the behaviour and structure of the algorithms. The VHDL code was compiled and synthesised using Xilinx ISE 9.1 design tools.

Specific to our ADM-XRC board, for the LBM implementation $k = 1/10$ and a maximum of 16 PEs $p$ were implemented. Our LBM PE implementation consumes 674 FPGA core clock-cycles to update one LBM cell. This number was obtained from the VHDL simulations of our LBM PE. Transferring a single word (64-bits) from a source memory bank into the FPGA takes on an average 1.1 FPGA core clock-cycles (1.1 specifies that our system implementation manages 90% bandwidth utilisation since we include the DDR2 memory overheads, that is, latency due to the first read from core, effects like memory page changes and refreshes etc).

First, to demonstrate our proposed performance model for the compute bound two-dimensional CA algorithm, we implemented and validated our model for the D2Q9 Lattice

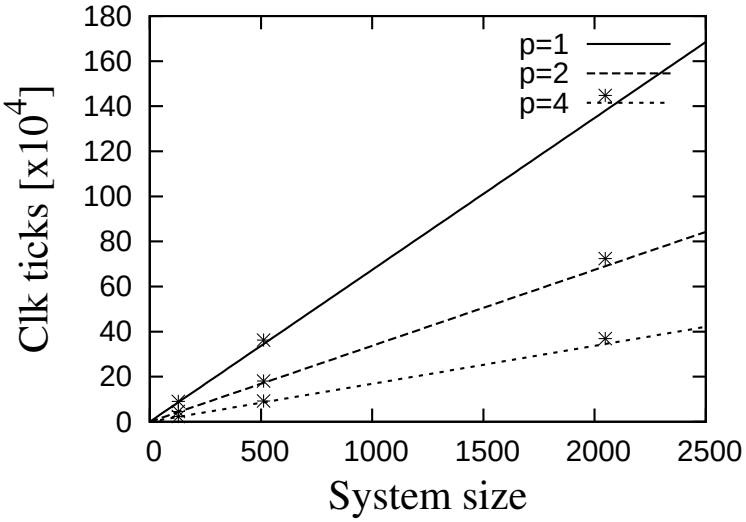


**Figure 5.4:** *Execution time to compute a single generation for the LBM model. Lines represent the performance model as specified by Equation (3.10) and the points represent the measured execution times using ADMXRC Virtex4-FX140 board.*

Boltzmann Method with number of PEs $p$ equal to two, four and eight respectively. With this setup as shown in Figure 5.4, varying simulation sizes defined $N$ equal to $8 \times 16$, $16 \times 32$ and $32 \times 64$ respectively. Plugging these numbers in Equation (3.10) we get the theoretical number for our LBM implementation. For the hardware side measurements we used a counter implemented in the FPGA core. The counter measures the number of FPGA core clock-cycles for the entire duration of a single generation of the LBM lattice, starting from the initialisation of the CB in FPGA to the flushing out of the data at the end of the computation. Figure 5.4 shows measurements together with theoretical results where the lines represent our model and points are the measured execution times. The LBM implementation takes longer than the theoretical results due to usage of DDR2-SDRAM banks, and some redundant finite state machine cycles within our LBM CB implementations. However, the model predictions are accurate within 7%.

Further, we used a Dell PC with an Intel P4 2.99-GHz and 3.25-GB RAM to measure the microprocessor-based performance for our D2Q9 Lattice Boltzmann implementation in Fortran, and compared this with our FPGA-based implementations. For this setup we extended our FPGA-based LBM implementation to a maximum of $p = 16$ PEs implementation. Each of the LBM implementations with number of PEs $p$ equal to two, four, eight, and sixteen respectively were tested for four different lattice sizes ($N$ equal to $8 \times 8$, $16 \times 16$, $32 \times 32$ and $64 \times 64$) that were computed for $g = 512$ generations. The resulting execution times are shown in Figure 5.5. For the overall execution times, we measured the wall-clock time using software running on the host machine. Execution times included the host machine's pre and post CA processing and the time taken by FPGA engine to

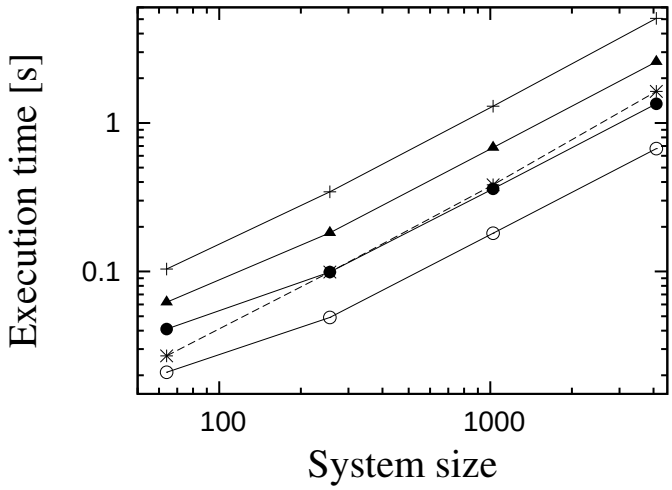


**Figure 5.5:** *Execution times for computing* 512 *generations of the LBM D2Q9 grid of size N using Fortran and a single FPGA-enabled PC implementations. Solid lines represent FPGA-based results and the dotted line represents Fortran code results. A plus represents a FPGA with 2PEs, a triangle for a FPGA with 4PEs, a filled circle for a FPGA with 8PEs, and an open circle for an FPGA with 16PEs execution times respectively.*

compute the required number of generations. As shown in Figure 5.6, with the increase in the number of PEs the overall execution time decreases proportionally as expected from Equation (3.10). When using all the logic available on the FPGA, that is, for 16 PE's, we achieved a speed-up of 2.3 as compared to the Fortran implementation.

## 5.4 Conclusion and Future Work

This chapter presented a detailed discussion on a LBM D2Q9 FPGA-based implementation. Based on the computational structure of the chosen test bench its FPGA-based implementation was classified as a compute bound computation algorithm. Several test runs were performed using FPGA-based implementations, followed by comparing its wall-clock measurements to our Fortran implementations. For a single FPGA-based implementation, an overall speed-up of 2.3 as compared to a Fortran implementation was achieved.

The performance model for the single FPGA-enabled D2Q9 LBM implementation implies it to be compute bound as long as the implementation has $p \leq 61$. Therefore, with the current-generation FPGA devices, for example, Virtex-5 FPGA chip would improve the speed-up simply by including more PEs per chip and higher FPGA clock frequency.

Possible future extensions are, to have a design with programmable number of PEs embedded within the implementation. Having programmable number of PE implementations would enable smooth experiments and measurements. A new addition could be

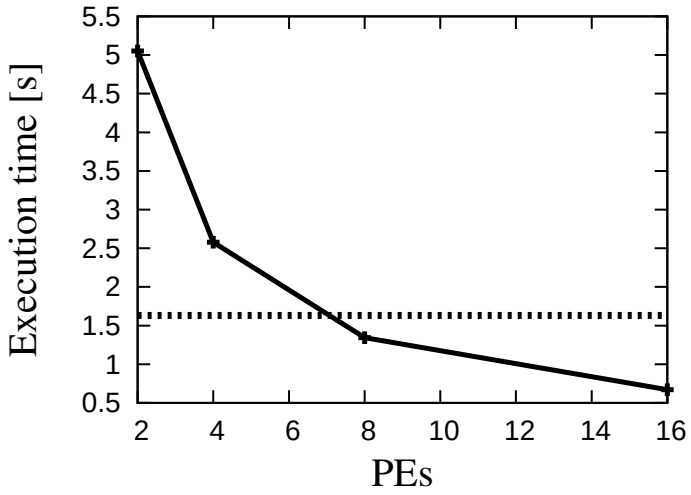


**Figure 5.6:** *Execution times for computing* 512 *generations of the* $64 \times 64$ *LBM D2Q9 grid on a single FPGA-enabled PC implementations. FPGA-based execution times are represented by stars and the dotted line represents the Fortran implementation.*

to share the resources among the PEs. Since each PE internally is a sequential circuit with its own set of floating-point cores for addition, multiplication and division respectively, the sharing of floating-point cores among the PEs is a possibility though not trivial. Compared to 2D LBM model, a 3D LBM implementation would greatly benefit with the given hardware design. Therefore, extending the 2D to 3D LBM implementation is highly recommended.