Computational Science Technical Note **CSTN-151**

Field Programmable Gate Arrays for Computational Acceleration of Lattice-Oriented Simulation Models

A. Gilman and K.A. Hawick

2012

Field Programmable Gate Arrays (FPGAs) have attracted recent attention as accelerators for a range of scientific applications which had previously been only practicable on conventional general purpose programming platforms. We report on the performance scalability and software engineering considerations when FPGAs are used to accelerate performance of lattice-oriented simulation models such as complex systems models. We report on the specifics of an FPGA implementation of cellular automaton models like the Game-of-Life. Typical FPGA toolkits and approaches are well suited to the localised updating of the regular data structures of models like these. We review ideas for more widespread uptake of FPGA technology in complex systems simulation applications.

Keywords: FPGA-based design; simulation; complex systems; implementation development; performance evaluation

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-151,
        author = {A. Gilman and K.A. Hawick},
        title = {Field Programmable Gate Arrays for Computational Acceleration of
                Lattice-Oriented Simulation Models},
        booktitle = {Proc. Int. Conf. on Computer Design (CDES'12)},
        year = {2012},
        pages = {91-97},
        address = {Las Vegas, USA},
        month = {16-19 July},
        publisher = {CSREA},
        institution = {Computer Science, Massey University},
        keywords = {FPGA-based design; simulation; complex systems; implementation development;
                performance evaluation},
        owner = {kahawick},
        timestamp = {2012.05.03}
}
```

Technical Report CSTN-151

Field Programmable Gate Arrays for Computational Acceleration of Lattice-Oriented Simulation Models

A. Gilman and K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
email: { a.gilman, k.a.hawick }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

April 2012

ABSTRACT
Field Programmable Gate Arrays (FPGAs) have attracted recent attention as accelerators for a range of scientific applications which had previously been only practicable on conventional general purpose programming platforms. We report on the performance scalability and software engineering considerations when FPGAs are used to accelerate performance of lattice-oriented simulation models such as complex systems models. We report on the specifics of an FPGA implementation of cellular automaton models like the Game-of-Life. Typical FPGA toolkits and approaches are well suited to the localised updating of the regular data structures of models like these. We review ideas for more widespread uptake of FPGA technology in complex systems simulation applications.

KEY WORDS
FPGA-based design; simulation; complex systems; implementation development; performance evaluation.

1 Introduction

Field Programmable Gate Array technology [1–3] has developed considerably in recent years and commodity priced FPGA development boards now make it feasible to teach their use to students but also to deploy them in applied scientific research [4] as a source of high performance compute resource. FPGAs are finding increasing uses for application development in areas including: accelerating physics calculations [5]; agent-based modelling [6]; bio-informatics data processing [7]; image processing [8–10]; as well as cellular automata simulations [11].

Figure 1: ML605: Xilinx Virtex 6 development board

Engineers have been using FPGAs for a number of years in time-critical applications, such as real-time signal processing. Recent research showing real potential for high performance computing [12], with FPGA implementations of certain problems performing better than CPU/GPU implementations [13]. This is due to inherent flexibility in custom architecture design, allowing for better mapping of some applications to the hardware it is being executed on. However, the time and expertise required for implementation development is considerably higher than for conventional software architectural development. Even though the design entry is performed using high level programming languages, the task is very different to writing a programme for a general purpose computer. We found that a hardware-oriented minds-et is required for efficient implementation and performance optimization.

In this article we explore designing FPGA based computation engines for scientific simulations using a simple cellular automaton [14], using Conway's Game of Life [15], as an example. We used a Xilinx ML605 FPGA development board for this work. This device connects using a conventional PCIe interface slot to a PC, as is seen in Figure 1.

Our paper is structured as follows: In Section 2 we describe the general framework of use for FPGA architectures. In Section 3 we describe our use of FPGA development boards to simulate models like the Game of Life. We present some performance results in Section 4. We discuss their implications, summarise our findings and suggest some areas for further work in Section 5.

2 FPGA Framework

When performing computational tasks we generally use general purpose computers. These are designed to solve any computational task, as long as we can express it as a program. This makes them versatile and relatively easy to use to solve our computational problems. However, this same ability to perform virtually any task we may ask of it limits its computational performance for any one specific task: the general purpose architecture is sub-optimal for any specific task and can be very far from optimal for some tasks.

As an alternative to using general purpose architecture, we could design a custom architecture, optimized for the specific task at hand in a way as to exploit any spatial or temporal parallelism inherent to the problem. Implementing this custom architecture on completely custom hardware through the manufacture of an application-specific integrated circuit (ASIC) would most likely result in much better performance. Unfortunately, ASIC design and manufacture is an extremely expensive and time-consuming process. Only very few applications can qualify for this approach.

There is, however, an alternative for implementing a custom architecture - reconfigurable hardware or more specifically field-programmable gate arrays. These pre-manufactured chips contain a sea of programmable logic with programmable interconnect fabric running in between. In addition, these devices also contain on-chip memory, optimized digital signal processing (DSP) blocks and a multitude of input/output pins to interconnect with other hardware, like off-chip memory. Transistor count for modern FPGAs is in the billions, allowing these devices to implement some very complex designs.

FPGA chips are designed to be reconfigurable to carry out any digital logic operation. The most basic building block of these chips is a look-up table (LUT) and register pair. 4- to 6- input LUTs are common in modern devices and can implement any boolean logic function of that many inputs. A number of LUT-Register pairs are grouped together into blocks called ”logic slices”. In addition to LUT-Register pairs, logic slices

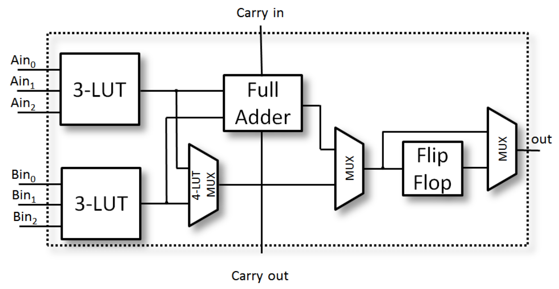

Figure 2: Simplified example of an FPGA logic slice.

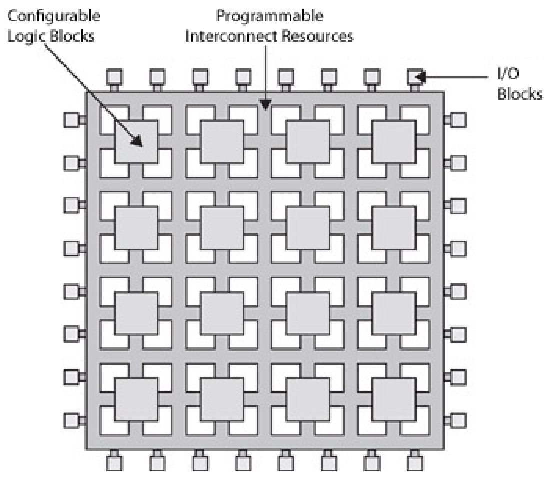
Configurable Logic Blocks
Programmable Interconnect Resources
I/O Blocks

Figure 3: FPGA layout

also generally contain dedicated arithmetic and carry logic and a number of multiplexers that can be used to combine outputs of the slice’s LUTs to implement logic functions with higher number of inputs. A simplified example of a logic slice is shown in figure 2.

A number of logic slices are grouped together to form configurable logic blocks (CLBs). Many thousands CLBs are located on one chip with a complex network of interconnects occupying the fabric between them (see figure 2). Just like the logic slices within CLBs, the interconnect network is programmable and can connect inputs to outputs of various slices as needed to form more complex logic functions. In addition to logic slices, FPGAs commonly contain on-chip RAM resources called block RAM (BRAM). These blocks are generally only a few kilobytes in size, but there are many of them and they can be accessed in parallel or combined into larger memories with fairly large bandwidths.

FPGA design methodology has a number of levels of abstraction ranging from the actual configuration of LUTs all the way up to high-level programming languages, such as Handle-C. It differs from software design in that an actual hardware device that would per-

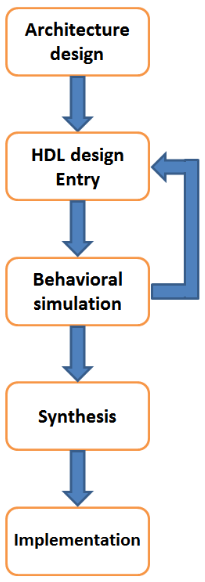

Figure 4: FPGA Design Flow

form the desired function is being designed, rather than just a program to execute on an existing computer architecture. A flowchart of a typical design flow is shown in figure 4. The first step is architecture design, where the computational task at hand is analysed and decomposed into structural blocks with each one described in terms of its functionality and interfaces.

Once the device architecture has been designed, a hardware-description language (HDL), such as VHDL or Verilog, is used to formally describe the device. This process can utilise both the behavioural and structural description of the device and its subcomponents.

Behavioural simulation is an important next step that tests the HDL description of the problem against the desired behaviour. At this point any behavioural differences between the HDL code and its desired behaviour are ironed out. A process called synthesis is used next to infer combinatorial and sequential logic from HDL statements. This produces an abstract representation of the desired circuit behaviour called register-transfer level (RTL) logic. This is then decomposed further into an even lower-level (logic gates and transistor level logic) representation called a netlist, generally recorded in a vendor-neutral representation using EDIF format.

The actual implementation process uses a vendor-specific tool to map the netlist representation of the designed device to the actual resources available (LUTs, BRAM, DSPs, etc) on the particular FPGA used for implementation. A process called place and route then proceeds to allocate available resources and performs routing of the signals between them as required. These are complex optimization procedures that are attempting to balance multiple trade-offs, such as chip area uptake reduction, running power consumption minimization, timing performance and implementation process runtime reduction. If one of these goals is more important than others, for example timing performance, it is possible to use a different optimization strategy to put more emphasis on increasing the maximum clock frequency. The final step is the programming file generation, which creates a bit-stream file containing the configuration settings of all the FPGA resources. This file is then simply downloaded onto the FPGA to configure it.

All of the above steps are performed using electronic design automation (EDA) tools. Implementation tools need to be vendor-specific, as they relate to the particular FPGA chip that is used for final implementation. For HDL entry and synthesis, however, 3rd party vendor-independent tools can be used (e.g. Cadence, Synplify) and the resulting netlist can be implemented using any suitable vendor. In addition to these EDA tools, designers can choose to use an even higher-level language to describe the system architecture. Recent research have been aimed at developing techniques for hardware synthesis from high level languages like C/C++ [16, 17]. Some of these are already available on the market (e.g. Handle-C, systemC); although, their uptake is slow and they are nowhere near as popular as Verilog or VHDL.

For this project we have used a Xilinx FPGA. Xilinx is one of two main vendors (Altera being the other one) of FPGA technology. ML605 development board, hosting an xc6vlx240t device from the Virtex-6 family was used. The advantage of this board is the on-board PCIe interface that can be used for exchanging the data between the host PC and the FPGA. This device contains 150,720 LUTs and 301,440 flip-flops in 37,680 logic slices and 416 36Kb block RAMs, totaling 14,976 Kb on-chip memory. We have used Xilinx ISE Design Suite 13.4 for HDL design entry, synthesis and implementation and Xilinx ISIM for behavioural simulation.

3 FPGA Simulation Formulation

We chose to experiment with a simple Game of Life (GoL) [15] simulation for this investigation. The GoL model comprises a 2D array of bit-wise cells representing the states live or dead. At each (synchronous) time step of the model, each cell is updated according to a microscopic rule based on the number of live cells in the Moore neighbourhood of each focal cell. We can

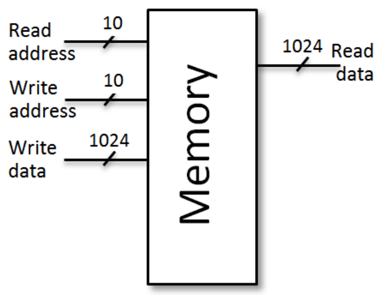

Figure 5: Custom 1024×1024 dual-port RAM.

initialise the cells randomly, but that is the only randomness in the model definition, which is otherwise completely deterministic.

This class of model is interesting since GoL can be generalised to a whole family of similar automata that have exhibit emergent complex systems behaviour that can only be studied effectively using extensive simulations. FPGA technology is particularly well suited to carrying out parallel and effectively synchronous updates on a regular lattice model with localised neighbour communications. The model rules are readily implemented using primitives that can be easily implemented using FPGA component operations.

The architecture for our game of life implementation consists of two parts: the data-path and the control circuitry. The data-path is a collection of modules that store the data and operate on it. The control circuitry implements the logic required to execute the particular set of steps required to fetch the data, perform the computation and store the result. Separating the design into two parts like this makes it easier to design and maintain complex architectures.

Along with the compute logic, the data-path must contain some sort of storage to store the state of each cell in the simulation. In software this would be stored in an array of integers located in the main memory. On the FPGA we can create a dedicated RAM to contain this data on the chip itself, using multiple blocks of BRAM. These blocks are designed to be highly configurable, allowing for a lot of flexibility and can be connected in parallel to create RAMs with any word width (number of bits per stored element) and any depth (number of stored elements). What makes BRAM even more useful is the fact that it is dual-port, meaning that we can read the state of current generation and write the computed next state at the same time.

The FPGA chip that we are using contains a total of 15Mb of storage in BRAMs, which is enough to store the state of one million cells in a 1024×1024 simulation that we want to implement (storing one bit per

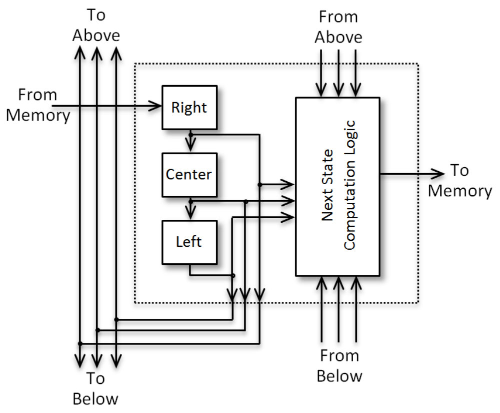

Figure 6: A single processing element contains a 3-bit shift register and combinational logic to compute the next state.

cell). Memory access is a significant bottleneck of modern computers. If we are going to process more than one cell at the same time, we need to have sufficient memory bandwidth to get the required data. We can bit-pack the states of a whole column of cells into a single 1024 bit-wide memory element and create a 1024-deep RAM to store each one of the columns (see figure 5). Doing this allows us to read/write the state of 1024 cells simultaneously.

To process the each cell we employ a processing element (PE) depicted in figure 6. There are 1024 instances of this element, each one computing the next generation state of the cells located in a single row, one element at a time. To avoid having to perform nine memory accesses for each computation (as the state of nine cells is required to compute this) local buffers within each processing element are employed to store the state of three consecutive cells, centered on the index of the cell, which is currently being processed. These stored states are used by the next state-computation logic and also passed to the processing elements directly above and below. The next state-computation logic is shown in figure 7. Because the value of state signals of all the neighboring cells are either one for alive or zero for dead they can be simply added together to compute the number of alive neighbors.

To compute the next state for a column of cells three separate operations need to be performed. A memory read operation, next state computation and a memory write operation. Because next state computation requires fairly simple logic, memory read and write operations are relatively long in comparison. It is possible to pipeline these three operations to increase the overall throughput, as shown in figure 8. If the next computation logic was quite complex, it could be ben-

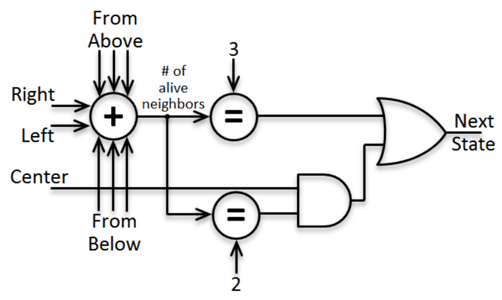

Figure 7: Combinational logic for next state computation.

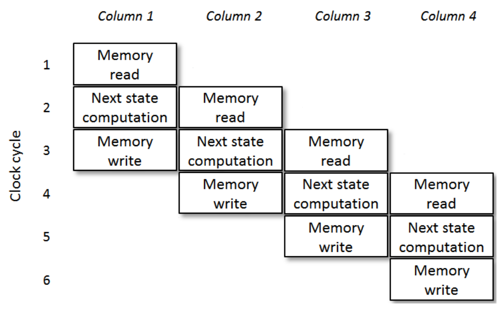

Figure 8: Pipelined memory access: reading, processing and writing each column in 3 consecutive clock cycles increases overall processing throughput.

eficial to also pipeline it by dividing it up into smaller computation stages; however, in this case it would not achieve any gains.

We used Verilog HDL to describe the design. Verilog description of the processing element is shown in algorithm 1. It consists of two parts: first part is the `always` statement that describes a synchronous three-element shift register using three non-blocking assignment <= operators. These assignments are executed simultaneously in a single clock cycle, using the value of each variable on the right hand side from the previous clock cycle. The value of this shift register is assigned to the `reg_out` port to be passed to PE above and PE below. The second part consists of computation of the number of alive neighbors by summing their states and the computation of the next state by applying the game rules. This code, along with the description of input and output ports, forms a Verilog module.

Each of the components in the architecture is described in a separate Verilog module, which are instantiated and interconnected using the in/out ports to complete the architecture. Modular design allows for easy reconfiguration. For example, we can change the simulation size, as long as there are enough resources on the device, to any simulation size N simply by changing the number of instances of the PE module in the PE array and BRAM configuration (figure 9). Or we

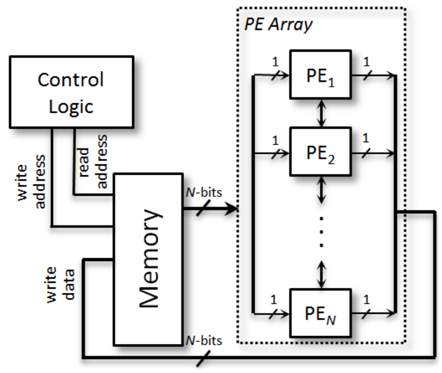

Figure 9: Processing element array contains a N PEs connected in parallel to process N rows of cells simultaneously.

$N =$	**512**	**1024**	**2048**
Flip-flops	1550	3090	6166
LUTs	3793	7559	15113
Logic Slices	1568	3156	6571
Max clock (MHz)	269	238	187
Compilation Time (s)	149	215	375

Table 1: Resource utilization for different size simulation.

can change the internal implementation of any module and as long as the external interface and functionality stays the same the design will still work.

The modules can also be parameterized, with the parameters being set at compile time. This allows for more flexibility and encourages re-usability. The processing elements, for example, can be designed to allow for q states, with q being a compile-time parameter. These can be reused in implementation of other CA, such as Game of Death [18], that require more than two states.

4 Performance Results

Table 1 shows resource utilization (the number of utilized flip-flops and look-up tables and also the total number of used logic slices), maximum clock frequency in megahertz at which each design is able to run and also the compilation time in seconds for three simulation sizes: 512×512, 1024×1024 and 2048×2048. The amount of utilized resources can be seen to go up linearly with the number of instantiated processing elements (512, 1024 and 2048 in each case). The maximum clock frequency goes down with simulation size. The largest simulation size, which also has the largest number of processing elements demonstrates highest throughput of 386 billion individual cells being processed per second versus 244 billion cells/s for second largest and 138 billion cells/s for the smallest. The overall performance of these three designs is 92,000 generation updates per second for the largest, 232,500 generation updates for second largest and 525,000 generation updates for the smallest.

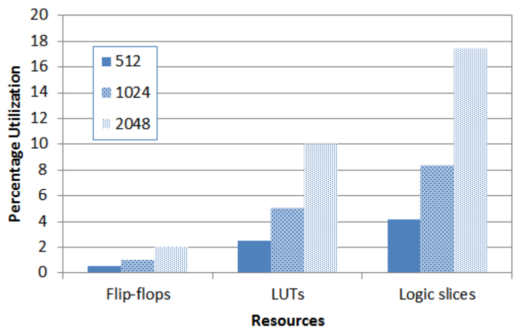

Figure 10: Resource utilization as a percentage of total available resources on xc6vlx240t device.

Figure 10 shows resource utilization figures as a percentage of the total resources available on xc6vlx240t device. It can be seen that even for a simulation as large as 2048 × 2048 elements, logic slice utilization is under 18%.

5 Discussion & Conclusions

The development process of this design followed the design flow shown in figure 4. The overall time was roughly split half and half between HDL entry/behavioural simulation and architecture design/optimization. We found both of these tasks to be highly iterative and the use of good EDA tools can make a big difference to the amount of time/effort taken up by these tasks.

Verilog HDL is a high-level language; however, we found writing Verilog code to be very different to writing a program in a software language, partly because the Verilog 'program' actually represented a piece of hardware with certain functionality, rather than an algorithm normally described by a program in a software language. The inherent notion of time in the process of describing synchronous logic also made it difficult to get the HDL description right the first time. This is where the behavioural simulation proved to be extremely valuable.

Performance of compute engines such as the one described here can be measured using throughput, a product of how much computation is performed in one clock cycle and the clock frequency. To increase performance, we can either utilize a large number of the processing elements or optimize the processing elements to increase the clock frequency, or both. We have tried a number of ways to simplify/optimize the processing element module and the processing element array, for example by replacing the series of 4-bit additions on lines 8 and 9 in algorithm 1 with an adder tree composed of smaller 2- and 3-bit adders and one 4-bit adder, as illustrated in algorithm 2.

These attempts resulted in simpler RTL representation of the design. This, however, did not necessarily equate to any significant gains in maximum clock frequency or resource utilization. This is partly due to the synthesis tools running various automated logical and physical optimization of the RTL and doing a fairly good job for a relatively simple design such as this one and partly due to the nature of the way digital logic is implemented on FPGAs - using a series of look-up tables.

We found it difficult to manually optimize the design prior to running the synthesis, as it was hard to tell whether what we considered simpler and faster would result in simpler and faster final implementation. This process of manual optimization turned into a series of trial and error steps to find out what worked and what didn't. Unfortunately, with a total compilation time of over 6 minutes for the 2048 × 2048 design this has become a very time consuming process.

The numbers in table 1 indicate that this design scales fairly well with simulation size in terms of resource utilization. Large designs, however, cannot be clocked at the same frequency as the smaller one, even though the main difference between them is the number of instantiated processing elements (all having the same complexity). This is likely to be due to more complex internal signal routing that introduces more delay as resource utilization increases.

Introducing more processing elements, whether it is for a larger simulation size or to increase the throughput even further by processing more than one column at a time, would most likely decrease the maximum clock frequency even further. Introducing more processing elements (if the memory bandwidth allows) would obviously increase throughput, but only up to a certain point, since this increase will have a detrimental effect on the maximum clock frequency.

Figure 10 shows that there is still a large portion of unused resources left on the device. This can either be utilized by more PEs to process multiple columns at the same time as stated above, or it can also be utilized to compute any required measurements, such as statistical metrics, in real-time as the simulation progresses, thus saving time on further analysis. Performing the

Algorithm 1 Verilog code for the Processing Element.

```
always @(posedge CLK)
begin
        right <= Data_in;
        centre <= right;
        left <= centre;
end
assign reg_out = {left,centre,right};
assign numAlive = (fromAbove_in[0] + fromAbove_in[1] + fromAbove_in[2] + left + right
                  + fromBelow_in[0] + fromBelow_in[1] + fromBelow_in[2]);
assign nextGen = (numAlive == 4'd3) || ((numAlive == 4'd2) && centre);
```

Algorithm 2 Verilog code representing an adder tree.

```
// Allocating 2-bit temporary storage:
wire [1:0] temp1,temp2, temp3;
wire [2:0] temp4; // 3-bit temporary storage
assign temp1 = fromAbove_i[0] + fromAbove_i[1]
        + fromAbove_i[2];       // 2-bit adder
assign temp2 = fromBelow_i[0] + fromBelow_i[1]
        + fromBelow_i[2];       // 2-bit adder
assign temp3 = left + right;    // 2-bit adder
assign temp4 = temp1+temp2;     // 3-bit adder
assign numAlive = temp4 + temp3;// 4-bit adder
```

required measurements in parallel to the running of the simulation can also greatly reduce the amount of data that needs to be taken from the device, by discarding the raw data and only reading the required metrics.

In summary, FPGA technology has a promising outlook [19] and we have found it to be well suited to this sort of complex systems simulation, that makes use of regular data structures, localised communications and good use of component primitive operations. FPGA implementations of this class of model will potentially aid the systematic exploration of a model space by providing very fast but cheap simulation platforms.

There is additional scope to develop some of the model measurement algorithms (such as tracking density of live cells) and mechanisms for collecting statistics that would even further enhance this platform's suitability.

References

[1] Oldfield, J.V., Dorf, R.C.: Field programmable gate arrays - Reconfigurable logic for rapid prototyping and implementation of digital systems. Number ISBN 0-471-55665-3. Wiley (1995)

[2] Chu, P.P.: FPGA Prototyping by VERILOG Examples. Number ISBN 978-0-470-18532-2. Wiley (2008)

[3] Herbordt, M.C., Gu, Y., VanCourt, T., Model, J., Sukhwani, B., Chiu, M.: Computing Models for FPGA-Based Accelerators. Computing in Science & Engineering **10**(6) (November/December 2008) 35–45

[4] Stitt, G., george, A., Lam, H., Readdon, C., Smith, M., Holland, B., Aggarwal, V., Wang, G., Coole, J., Koehler, S.: An End-to-End Tool Flow for FPGA-Accelerated Scientific Computing. IEEE Design and Test of Computers **July/August** (2011) 68–77

[5] Danese, G., Leporati, F., Bera, M., Giachero, M., Nazzicari, N., Spelgatti, A.: An accelerator for physics simulations. Computing in Science and Engineering **9**(5) (September 2007) 16–25

[6] Chen, E., Lesau, V.G., Sabaz, D., Shannon, L., Gruver, W.A.: Fpga framework for agent systems using dynamic partial reconfiguration. In: Proc. 5th Int. Conf on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS). Number 6867 in LNAI, Toulouse, rance (29-31 August 2011) 94–102

[7] Anan'ko, A.G., Lysakov, K., Shadrin, M.Y., Lavrentiev, M.M.: Development and application of an fpga based special processor for solving bioinformatics problems. Pattern Recognition and Image Analysis **21**(3) (2011) 370–372

[8] Gribbon, K.T., Bailey, D.G., Bainbridge-Smith,

A.: Development issues in using fpgas for image processing. In: Development Issues in Using FP-GAs for Image Processing?, Proceedings of Image and Vision Computing New Zealand 2007, Hamilton, New Zealand (2007) 217–222

[9] Bailey, D.: Design for Embedded Image Processing on FPGAs. Wiley (2011) ISBN 9780470828496.

[10] Huang, Q., Wang, Y., Chang, S.: High-performance fpga implementation of discrete wavelet transform for image processing. In: Proc. 2011 Symposium on Photonics and Optoelectronics (SOPO), Wuhan (16-18 May 2011 2011) 1–4

[11] Olson, A.: Towards fpga hardware in the loop for qca simulation. Master's thesis, Rochester Institute of Technology, New York (May 2011)

[12] El-Ghazawi, T., El-Araby, E., Huang, M., Gaj, K., Kindratenko, V., Buell, D.: The promise of high-performance reconfigurable computing. Computer **41**(2) (feb. 2008) 69 –76

[13] Che, S., Li, J., Sheaffer, J., Skadron, K., Lach, J.: Accelerating compute-intensive applications with gpus and fpgas. In: Application Specific Processors, 2008. SASP 2008. Symposium on. (june 2008) 101 –107

[14] Murtaza, S., Hoekstra, A.G., Sloot, P.M.A.: Cellular automata simulations on a fpga cluster. High Performance Computing Applications **25, Online**(2) (October 2010) 193–204

[15] Gardner, M.: Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". Scientific American **223** (October 1970) 120–123

[16] Edwards, S.A.: The challenges of hardware synthesis from c-like languages. In: Proceedings of the 2005 Design, Automation and Test in Europe. (March 2005) 66 –67

[17] Edwards, S.: The challenges of synthesizing hardware from c-like languages. Design & Test of Computers, IEEE **23**(5) (May 2006) 375 –386

[18] Hawick, K., Scogings, C.: Cycles, transients, and complexity in the game of death spatial automaton. In: Proc. International Conference on Scientific Computing (CSC'11). Number CSC4040, Las Vegas, USA (July 2011)

[19] Constantinides, G.A., Nicolici, N.: Surveying the landscape of fpga accelerator research. In: IEEE Design and Test of Computers. Volume July/August. (2011) 6–7