DATA STRUCTURES FOR SIMD LOGIC SIMULATION

MOZHGAN KABIRI CHIMEH

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

APRIL 2016

Abstract

Due to the growth of design size and complexity, design verification is an important aspect of the Logic Circuit development process. The purpose of verification is to validate that the design meets the system requirements and specification. This is done by either functional or formal verification.

The most popular approach to functional verification is the use of simulation based techniques. Using models to replicate the behaviour of an actual system is called simulation.

In this thesis, a software/data structure architecture without explicit locks is proposed to accelerate logic gate circuit simulation. We call thus system ZSIM. The ZSIM software architecture simulator targets low cost SIMD multi-core machines. Its performance is evaluated on the Intel Xeon Phi and 2 other machines (Intel Xeon and AMD Opteron).

The aim of these experiments is to:

- Verify that the data structure used allows SIMD acceleration, particularly on machines with gather instructions (section 5.3.1).
- Verify that, on sufficiently large circuits, substantial gains could be made from multi-core parallelism (section 5.3.2).
- Show that a simulator using this approach out-performs an existing commercial simulator on a standard workstation (section 5.3.3).
- Show that the performance on a cheap Xeon Phi card is competitive with results reported elsewhere on much more expensive super-computers (section 5.3.5).

To evaluate the ZSIM, two types of test circuits were used:

1. Circuits from the IWLS benchmark suit [1] which allow direct comparison with other published studies of parallel simulators.

2. Circuits generated by a parametrised circuit synthesizer. The synthesizer used an algorithm that has been shown [2] to generate circuits that are statistically representative of real logic circuits. The synthesizer allowed testing of a range of very large circuits, larger than the ones for which it was possible to obtain open source files.

The experimental results show that with SIMD acceleration and multicore, ZSIM gained a peak parallelisation factor of 300 on Intel Xeon Phi and 11 on Intel Xeon. With only SIMD enabled, ZSIM achieved a maximum parallelistion gain of 10 on Intel Xeon Phi and 4 on Intel Xeon.

Furthermore, it was shown that this software architecture simulator running on a SIMD machine is much faster than, and can handle much bigger circuits than a widely used commercial simulator (Xilinx) running on a workstation.

The performance achieved by ZSIM was also compared with similar pre-existing work on logic simulation targeting GPUs and supercomputers. It was shown that ZSIM simulator running on a Xeon Phi machine gives comparable simulation performance to the IBM Blue Gene supercomputer at very much lower cost. The experimental results have shown that the Xeon Phi is competitive with simulation on GPUs and allows the handling of much larger circuits than have been reported for GPU simulation.

When targeting Xeon Phi architecture, the automatic cache management of the Xeon Phi, handles and manages the on-chip local store without any explicit mention of the local store being made in the architecture of the simulator itself. However, targeting GPUs, explicit cache management in program increases the complexity of the software architecture. Furthermore, one of the strongest points of the ZSIM simulator is its portability. Note that the same code was tested on both AMD and Xeon Phi machines. The same architecture that efficiently performs on Xeon Phi, was ported into a 64 core NUMA AMD Opteron.

To conclude, the two main achievements are restated as following: The primary achievement of this work was proving that the ZSIM architecture was faster than previously published logic simulators on low cost platforms. The secondary achievement was the development of a synthetic testing suite that went beyond the scale range that was previously publicly available, based on prior work that showed the synthesis technique is valid [2].

Acknowledgements

Foremost, I would like to express my special appreciation and thanks to my supervisor Dr. Paul Cockshott, who has been a tremendous mentor for me. I would like to thank him for the continuous support, patience, immense knowledge, encouragement, and for allowing me to grow as a research scientist. His advice on both research as well as on my career have been priceless. It has been an honor to be his student.

I would also like to thank my Viva committee members, Dr. Wim Vanderbauwhede, Dr. Donglai Xu, and Dr. Colin Perkins for their comments and suggestions, and letting the Viva be an special moment.

A special thanks to my family for all their love and encouragement. Words cannot express how grateful I am to my parents for all of the sacrifices they have made on my behalf. Your prayers for me was what sustained me thus far. At the end, I would like express appreciation to my beloved husband Reza who spent sleepless nights with and was always my support in the moments when there was no one to answer my queries. Your faithful support during the final stages of the Ph.D. is so appreciated.

To Reza, Fatemeh, Hosein

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This research aims to use optimization and parallelism to speed up simulation of digital circuits. Accelerating circuit simulation is an important topic as it is part of the verification process in electronic circuit design. The purpose of verification is to validate that the design meets the system requirements and specification. This is done by either functional or formal verification. The most popular approach to functional verification is the use of simulation based techniques. Using models to replicate the behaviour of an actual system is called simulation. Circuit simulation is being widely used as part of a verification testing tool. However, it becomes more time consuming as a VLSI[1] design grows larger.

Given a set of input signal values, a circuit simulator (a program) calculates the values that a circuit will output. For testing, it is also necessary to generate suitable inputs to feed the circuit (or circuit simulator). The amount of information which is generated as part of the output, largely depends upon the abstraction level at which the circuit was described and subsequently simulated.

Circuits may be described and simulated at several levels of abstraction including analogue level simulation, switch level, logic level and functional level simulation [3]. Each considers a different circuit model which abstracts away some properties of electronic circuits. For example, simulating a circuit described in terms of transistors and capacitors will conventionally show how these components interact at the electrical or analogue level, whereas simulating a circuit described in terms of gates, flip-flops and registers will demonstrate the digital interaction amongst the components. Furthermore, at some level of abstraction, some ignore heat dissipation and power consumption, while others ignore analogue behaviour of the the electronic circuits. Although abstraction results in ignoring some characteristic of the circuit in simulation, it also reduces simulation time which is the goal.

This work only considers digital circuits, and analogue behaviour of the components are

[1]Very-large-scale integration

abstracted away. This allows a central concept of this thesis to be more specific and refined to digital logic circuit design and logical correctness. As part of a verification tool, gate level logic simulation plays an important role in VLSI circuit design. It analyses the behaviour of the circuit and correctness of its design. However, this does not ignore the importance of other forms of correctness and validity.

There are several kinds of digital circuits including asynchronous and synchronous, which only the synchronous digital circuits and logic gate level simulation is considered. Logic gate level circuits may contain both primitive components and non primitives, in addition to flip flops. Primitive components could be any of the basic logic gates such as AND, OR, NOT, NOR, NAND, XOR, XNOR. Non primitives, composed of previously mentioned primitive elements, are parts such as FULL ADDER and Multiplexer.

In sequential logic circuits, the output depends not only on the state of current inputs, but also on the condition of earlier inputs. These type of circuits implicitly contain memory elements. There are two types of sequential circuit, synchronous and asynchronous.

In a discrete sequential synchronous model, there is a global clock whith the content of flip flop updated at discrete times with respect to that clock, and circuit outputs are generated. In simulating synchronous circuit models, the aim is to calculate the values that goes into the flip flops at the clock.

Only simulators for clocked synchronous circuit designs are the main concern of this thesis. The efficiency and performance of this sort of simulator depends on the following factors. First, the maximum size of the circuit: numbers of logic gates that can be evaluated. Second, the time in seconds it takes to simulate the circuit for a fixed number of clock cycles. Let n be the number of logic gates in the circuit, and t, the time in seconds to simulate the circuit for one clock cycle. The performance will be $\frac{n}{t}$.

Note that the performance of a logic gate simulator is generally expressed in terms of e, the total number of gate evaluations per unit time. During each clock cycle of the simulated machine, not every transition on logic gate's input leads to a different output. So, usually it is the case that $e < \frac{n}{t}$. The difference between the two quantities of e and $\frac{n}{t}$ is exploited by event based simulation.

On single processor machine, this method is effective. Event based simulation is implemented by a queue of event. When parallelising the queue, multiple threads may try to access to add/remove an event in the queue. This leads to lock contention for the queue. Message passing models of parallelism use queues, for messages. It is built into their basic communication mechanism. Message passing parallelism has been successfully used to accelerate event based simulation.

Recently, massively parallel discrete-event execution on several thousands of processors has been extended to simulating very large numbers of circuits in detailed hardware simulations of microprocessors [4, 5]. On the other hand, in contrast to dynamically observing the the behaviour of the model during the simulation, static scheduling of all the computations would be well suited on massively parallel machines. The technique is called oblivious simulation.

Availability of multicore SIMD processors with gather instructions, allows more efficient high performance logic simulations than alternative software.

1.1 Thesis Statement

This thesis is that it is possible, by the use of simple regular data structures to obtain, with SIMD shared memory multiprocessors, simulation speed, that are as good as or better than other workstation technologies and more cost effective than small super computer cluster for the same task.

1.2 Research Proposal

In this thesis, we are proposing an architecture without explicit locks that allows a contention free parallelism targeted at low cost SIMD multi-core accelerator boards. The Intel Xeon Phi [6] which is a coprocessor with Many Integrated Core architecture developed by Intel (based on the earlier Larrabee [7] many core architecture), is used to accelerate digital logic gate circuit simulation.

There are several existing published works on accelerating digital circuit simulation using GPUs [8, 9, 10, 11, 12] and multicores [4, 13]. However, to the best of my knowledge, none have done digital logic simulation by taking advantage of Xeon phi SIMD architecture.

1.3 Research Contribution

Overall, in this work, the main goal is to accelerate logic gate circuit simulation using a contention free data structure architecture that is targeting low cost SIMD machines. The major contribution of this dissertation are as follow:

1. Reviewing the previous work in the area of parallel logic level circuit simulation
2. Generating larger range of synthetic circuits, above the range of what is reported in the literature.
3. Proposing a lock-free data structure

4. Evaluating the performance of the proposed SIMD simulator (ZSIM)[2] on various architectures

5. Implementing the proposed SIMD simulator in two different programming languages and compare its performance across various compilers

6. Studying the relation between #3 and #4 above

7. Comparing the vectorization performance in ZSIM with SIMD acceleration on Single core and multi-core

8. Comparing the achieved performance to similar existing works on GPU and supercomputers

9. Demonstrating that on larger circuits, the proposed architecture is highly competitive with what reported in the literature including GPUs and supercomputers

1.4 Thesis Outline

The remainder of this thesis is divided into 5 chapters: **Chapter 2** provides background materials including fundamental concepts in circuit simulation. It also discusses the state-of-the-art views on various methods of simulation including event based and oblivious. Moreover, it surveys the most close and relevant research to this work, mainly focusing on parallel logic simulation on GPUs and supercomputers. **Chapter 3** comprises of two sections: SIMD requirements in circuit simulation and the SIMD data structure architecture that is proposed in this thesis. The chapter explores several possible approaches of using SIMD in circuit simulation. Techniques to benefit from SIMD in simulation, including ways of data packing and logic evaluation are discussed. The software simulator architecture that is being used in this research work requires few locks and no queues. It is required to employ a specific form of data structure that can take advantage of SIMD. This chapter's focus is on the SIMD software architecture and data structure that is used in this work. **Chapter 4**, explores various existing circuit test suites, from standard benchmark circuits to synthetic ones. Moreover, the method that was used to generate netlists for the SIMD simulator, is discussed. This chapter also provide details of how the circuits are represented . To evaluate and measure the performance of the proposed simulator, it was tested on several benchmarks on various hardware architectures including the Intel Xeon Phi, an Intel Xeon, and an AMD Opteron NUMA machine. The results were then compared to some of the existing work discussed in the literature. **Chapter 5**, describes the experimental setup and used platforms, followed by

[2]ZSIM is written in C++ and OpenMP by Mozhgan Kabiri Chimeh. Full data is in the university document repository at doi:10.5525/gla.researchdata.342

experimental results and discussions. **Chapter 6**, describes the contributions of this thesis. It restates the objectives, and the conclusions drawn from the experiments with the proposed software/data structure architecture, followed by a section on possible future research works.

1.5 Publications

- Mozhgan Chimeh, Paul Cockshott. ”Optimising Simulation Data Structures for the Xeon Phi”. The 2016 International Conference on High Performance Computing and Simulation.Austria, July 18 22, 2016
- Mozhgan Chimeh, Paul Cockshott, Susanne B. Oehler, Ashkan Tousimojarad and Tian Xu. ”Compiling Vector Pascal to the XeonPhi”. Concurrency and Computation: Practice and Experience. May 2015 [14].
- Mozhgan Chimeh, Cordelia V. Hall, John T. O’Donnell. ”Optimisation and parallelism in synchronous digital circuit simulators”. IEEE International Conference on Computational Science and Engineering. Nicosia, December 57th, 2012 [8].

Chapter 2

Related Work And Preliminaries

This chapter explores some of the background material related to circuit simulation including circuit models and different level of abstractions, simulation types and techniques. The second part of this chapter studies related works on logic simulation. We discuss some of the methods used in the literature and report their results.

2.1 Simulation and Models

Using models to replicate the behaviour of an actual system is called *simulation*. A model is a simpler and abstract version of a desired system. In general, simulation refers to time evolution of a computerized version of a model.

Simulations are used in various areas and fields including the entertainment industry, medicine, engineering, electronics, etc. For example, the purpose in using simulation in flight training devices is to train pilots on the ground in a low risk environment to avoid any possible risk that might happen to pilots or actual aircraft in a case of failure during the process. Other than providing low risk training process, it can also be beneficial economically, when fuel, insurance and maintenance are taken into account [15].

In order to test, experiment, and validate a design, we use models. As explained, simulation itself is a form of modelling. It applies a model over a simulation time to foretell its behaviour. For example, computer simulation helps to predict a behaviour of a model without the need to build the design.

2.2 Simulation as Part of Circuit Design Development

Due to the growth of design size and complexity, design verification is an important aspect of the Integrated Circuit (IC) development process. The verification process can take up to

70 percent of the total developmenet time [16]. The purpose of verification is to validate that the design meets the system requirements and specification. The most popular approach to verify the functionality of a circuit design is the use of simulation based techniques.

Logic simulation is an approach to determine validity in IC design. During the simulation, input patterns are applied to the design and the result (output) is compared to the expected behaviour of the system. Due to the growth of circuit complexity, simulation as part of design verification has become a bottleneck.

2.3 Software vs Hardware Simulation

Software simulation is a form of simulation that is executed by a computer system. In order words, we can simulate a software model of the design. Although it is low cost, as the circuit size grows larger, the simulation coverage will become less. Even with very fast simulation, we have to use a sparse test set and hope to find errors with this set.

2.3.1 Software Simulation

A software simulator is a program that simulates an abstract model of a particular system. It takes an input representation of the product or circuit, and processes the hardware description language that describes it to an internal model and compiles it. A system model typically includes processor cores, peripheral devices, memories, interconnection buses, hardware accelerators and I/O interfaces.

Simulation is the basis of much functional verification. It can be done at different level of abstraction and detail from transistor level simulation like SPICE [17] to Register Level Modelling. An overview of different level of abstraction in simulation will be discussed later in this chapter.

Over the period of past two decades, the number of transistors on processor chips are increasing. While the operating frequency increased until 2004, it has levelled off since then (Figure 2.1). These transistors are now used to build multicore processors. However, as the available cores on the computer are not being efficiently utilized, performance of software simulation decreases for larger circuits. Simulation acceleration, emulation, and FPGA prototyping are all solutions to overcome the slow PC simulation speeds for large designs.

Simulation acceleration

By simulation acceleration, we mean use of a hardware description language, such as Verilog or VHDL, along with some special equipment which allows simulation faster than could be done on a contemporary computer.

Some simulation accelerators use hardware such as GPUs (e.g. NVidia Kepler [19]) or FPGAs [20] with embedded processors. Acceleration is done by mapping the synthesizable part of the circuit design into a specially designed hardware platform. Evaluation of HDL constructs in parallel increases the performance. The rest of the design (portions of the simulation that are not mapped into this special hardware), are run in a software simulator on a PC or workstation. Furthermore, the hardware platform works with a software simulator to exchange simulation data. In order to increase the performance, the purpose of acceleration is to remove the events from a software simulator that runs on PC and run them on the hardware platform in parallel.

There are several factors that affect the performance. For instance, the remaining portion of the simulation that is running on the PC software simulator has a great deal of impact on the total simulation time. Due to Amdahls law, a high amount of serial work running on the PC simulator compared to the parallel portion of the work running on the hardware platform, slows down the total simulation. The number of IO signals that are used for communication between the hardware platform and the PC, in addition to the PC bandwidth and communication channel latency, can also reduce the performance. The smaller the bandwidth, the longer it takes to transfer data. For example, on GPU, the PCIe bandwidth can kill the performance. Retaining data on GPU board helps reducing the data transfer time via PCIe between host and device. On GPU, the more data transfers between the GPU and CPU, the more computing time it takes.

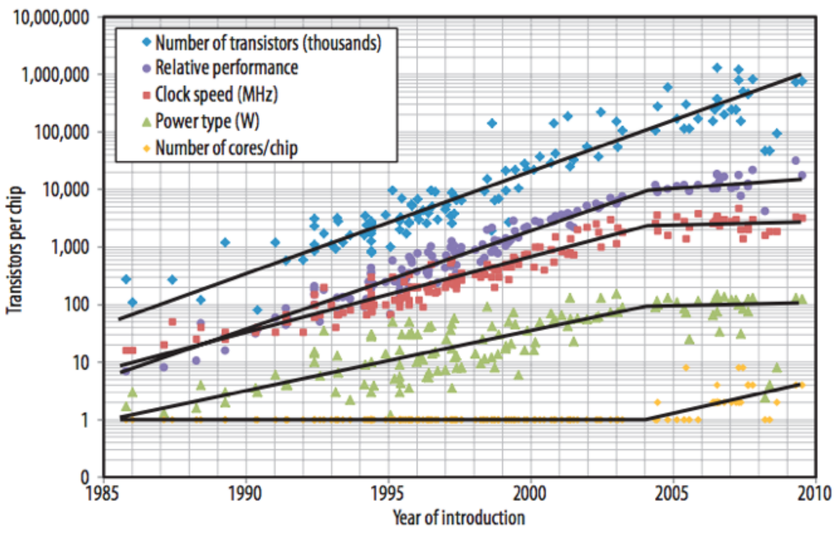


Figure 2.1: Logarithmic chart of technology growth vs. year [18]

2.3.2 FPGA Prototyping and Emulation

As explained earlier, functional verification is done in various way including software simulation, software acceleration, FPGA prototyping, and emulation.

Emulation is done by applying stimuli on a hardware prototype of a design. The word *emulate* was first used by IBM in 1963 [21]. They used microcode hardware to execute programs instead of software simulation. The emulator was used to accelerate simulation. Before 1980, the emulation term only referred to emulation with hardware or microcode [22]. Hardware emulators consist of array of reconfigurable logic computing units that are directly or indirectly interconnected. Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) are examples of programmable logics. FPGAs are used as a primary hardware/computing unit in emulation.

For example, ProtoFlex [23] simulator uses [22]Simics (a full system simulation platform) [24] that runs on a PC workstation as the reference simulator. It integrates a single Xilinx FPGA for acceleration. FPGA-acceleration portion of the Hierarchical ProtoFlex simulator is hosted on a Berkeley Emulation Engine 2 [1] FPGA platform [25].

To emulate an IC, FPGAs are programmed by an external computer to emulate a portion of the circuit. FPGAs may not be easy to use (fitting a design into FPGA is not easy). However, they improve the verification timing by 2 to 4 order of magnitude compared to software simulation. Although emulation is faster than software simulation, it still only validates some of the possible input stimuli. The available number of IO pins in them limits the emulator's performance.

Instead of building a prototype board, the design can be partitioned and mapped to multiple FPGAs and can be reused for other designs. Hardware prototyping using FPGAs is suitable for up to a medium size ASICs that fit into a single FPGA. For larger circuits, as the number of needed FPGAs increases, the partitioning of the design and mapping it to the FPGAs are difficult and error prone.

In order to simulate a design, the design under test (DUT) is modelled to a specific level of abstraction. Modelling a design at a different level of abstraction can affect the simulation time and required memory. Detailed and more accurate models leads to a slower and larger simulation model. For example, abstract models that do not consider detailed timing simulation, are more efficient in terms of simulation runtime; they run faster.

[1]"Berkeley Emulation Engine 2 (BEE2) project is developing a reusable, modular, and scalable framework for designing high-end reconfigurable computers."

2.4 Software-based Circuit Simulation

As explained earlier, simulation is an approach to predict and verify the correctness of the design's behaviour without a physical implementation. As part of design verification, simulation is used at each stage of development and each design stage, to validate the functionally and performance of the design. This is done with respect to the design requirements and the circuit description and specification. A software simulator is a computer program that takes the circuit description in a form of a netlist [2], in addition to the set of input vectors called stimuli, and generate a set of output vectors or responses. When the circuit is verified, the design process proceeds to the next stage. Otherwise, the redesign or modification may be needed if the expected and simulated responses were different.

The history of circuit simulators and the importance of circuit simulation due to growth of integrated circuit marker goes back to 1960-70 [26]. Circuit simulation is necessary to use as building ICs were expensive and difficult to troubleshoot.

The result of simulation very much depends on the level of abstraction the circuit model is described. For example, simulating a circuit model described in terms of transistors will demonstrate the analogue interaction among its components where as simulating a logic circuit model contains logic gates and flip flops, will show the digital behaviour between its components.

2.5 Level of Abstraction Models in Circuit Simulation

For simulation, and testing purposes, circuits are described and modelled at different levels of abstraction including *System Level, Register Transfer Level, Logic Level, and Transistor Level*. Table 2.1, shows different perspectives and levels of abstraction in digital design. In a behavioural perspective, our interest is only on what the circuit does, where as in a structural view, the concern is on the connectivity. In other words, what the circuit is composed of and how the elements are connected. In a physical view, the arrangement of hardware elements and wires on a chip or a circuit board is of interest. Different time units are used for each level of abstraction that a circuit is based on.

Table 2.1: Different levels of abstraction in Digital Design, after [26]

level of abstraction	behavioural view	structural view	physical view	Concept of time
system	input/output relationship	system with input/output	chip, board,or cabinet	sequence,throughput
architecture	bus functional model	organization into subsystems	partitioning,floorplan	partial ordering relationships
register transfer	data transfers and operations	ALUs, muxes, and registers	placement and routing	clock cycles
logic	truth tables, state graph	gates, latches, and flip flops	standard cells or components	events, delays, timing params
transistor	transfer functions (algebraic equation)	transistors, wires, R, L, C	detailed layout, mask polygon	continuous

[2]A netlist is usually generated by a synthesis tool that converts a HDL design by mapping its elements to logic primitives according to specific cell library.

Models at each level of abstraction represent different views of the system. At transistor level, the lowest abstraction level, the component's models are described as algebraic equations. An example of a simulator for this level is SPICE. The analogue simulator should be able to solve differential equations that describes the circuit.

At logic (gate) level, components are described by truth tables, or boolean equations. In contrast to the previous model, time and signal values at this level are discrete values. Instead of current and voltage that is used in transistor level, the connection quantities in this level are 0,1,X, .. values. A simulator that is used could be either event based simulator or cycle based (Section 2.7). Compared to an analog simulator that is time consuming, the simulators that are used for the model (at this level of abstraction), are faster.

Register transfer level is similar to the logic level in terms of time and values, and the used components. However, components in RTL models have higher complexity. The degree of complexity is even greater in system level. The connection between the components in the model is by buses. So, the entities in RTL models are *words* and *bytes* in opposed to *bits* in logic level. In system level, the connection entities are messages.

As the degree of the abstraction goes higher, the model will have less detail. As a result, the simulation of the model would be faster.

2.6 Circuit Models

The simulation is done at various level of abstraction on various circuit models. In the beginning of this chapter, we briefly defined abstraction and modelling. Abstraction simplifies the task of building complex systems. A model is an abstraction of a real physical system which some aspects have been ignored while building it. Some possible models in electronic circuit designs have been mentioned. Circuit models can be described at many level of abstractions.

For example, basic components in analogue circuit model are wires, transistors, diodes, resistors, inductors and capacitors where as in digital circuit model, the two main components are logic gates and wires. Wires in analogue circuit models carries analogue signals which are actually continuous voltages. In contrast, in digital circuit models, signals carry discrete boolean value of 0 or 1. In this section, we explore some of the main circuit models.

2.6.1 Digital vs Analogue

There are two kind of systems and devices namely analogue and digital. In analogue systems, information is represented by using physical quantities; continuous values such as voltage on a wire, magnetic field strength. However, in digital systems, the way of representing the information is using discrete values. For example 0 and 1, low or high, off or on.

In digital circuits, there are two types of logic circuits namely, *combinational* and *sequential* logic circuits. The first type implies to the logic circuits whose output is a function of its inputs, in contrast to sequential logic circuits which their output depends on the history of the input, as well as the present values of the inputs (Figure 2.2). In other words, sequential circuit is a combination of combinational circuit with memory devices.

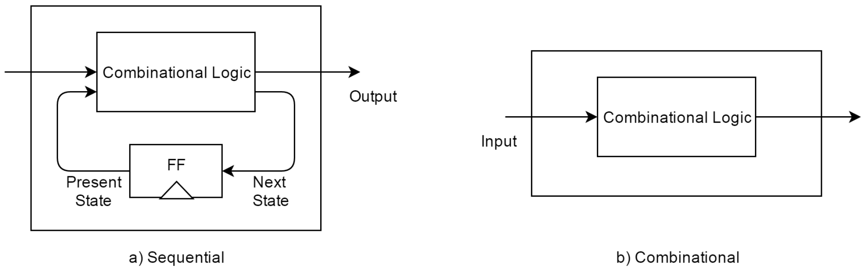


Figure 2.2: Combinational vs Sequential digital circuit model

2.6.2 Synchronous vs Asynchronous

Digital systems are divided in to two categories in terms of their functionality, namely *Synchronous* and *Asynchronous*.

Synchronous Synchronous circuits make up the vast majority of practical applications and are easier to design. In synchronous digital systems there is a global clock which controls the timing of the operations. For example in a synchronous model of a sequential circuit; which is a combination of one or more *flip flops* and logic functions (with no feedback), all the flip flops get the clock signal at the same time. In other words, all the flip flops in the circuit output their signal at the same time.

A Flip flop is a component in digital circuits for storing information. A D type flip flop has one input/output and one clock input. In this work, D flip flop is referred to as a flip flop. The input to the flip flop is determined by a signal which comes from the logic function in the circuit. The logic function includes a set of components which are basically connected to each other with no feed back. In synchronous model of sequential circuits, flip flops change their state at a point in time called the *clock tick*.

Clocks are regular periodic signals that can cause the state of memory element (flip flop) to change. Clock frequency is the number of rising clock edges (clock ticks) in a fixed period of time that determines the speed of a sequential circuit. Clock cycle time (or clock period) is the time between two rising clock edges. Duration of the clock cycle should be long enough so the system has enough time to compute the output and every component in the system gets calculated. In other words, from a rising clock edge, there should be enough time for D flip flops to generate stable output for the state, and for the next state logic to generate the next state. Furthermore, adequate time is also needed for D flip flops to set up after the next state is available. So, in general the clock cycle time should be at least equal to the total of propagation delay of D flip flop and next state logic plus delay the D flip flop setup time. A maximum delay after which all outputs are stable when input changes is called propagation delay.

The reason for this is that every logic function must complete in one cycle. In an abstracted version, clock cycle should be at least as large as critical path which is the longest path between an input to the circuit and its output with the maximum delay. Note that clock cycle is another form of abstraction; dividing time into discrete intervals. This will be discuss later in Section 2.7.

Figure 2.3(a), shows a synchronous circuit. In this example, data clocked into register *R4* is function *CL4* of the data clocked into *R3* at the previous clock. In other words, $R4 = CL4(R3)$.

Asynchronous The critical path restriction on synchronous circuits causes limitations in clock speed, and for very large circuits this becomes a major problem. So the modern trend is for circuits that have synchronous boxes, and these communicate with each other asynchronously. Asynchronous digital systems do not have a global clock and the output of the system depends on the input value and can be change at any instant of time. Figure 2.3(b), is an equivalent asynchronous circuit in Figure 2.3(a).

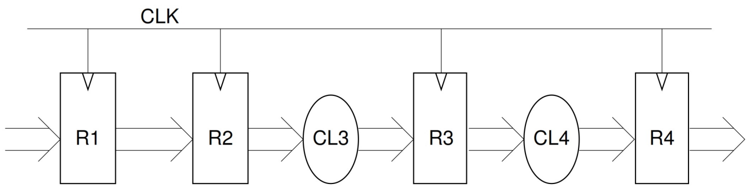
For example, in asynchronous Sequential digital circuits, output depends on its input variable changes and can be affected at any instant of time. Although, these kind of circuits are not controlled by a global clock, there are signals which indicate the completion of operations; every signal, has its own time and value.

2.7 Simulation Techniques

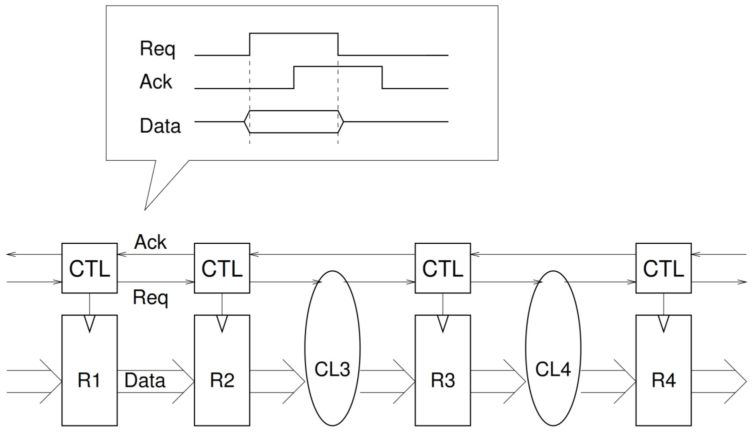
Based on the type of simulation algorithm and the circuit model, simulation technology is classified into different categories.

2.7.1 Cycle Based vs Event Driven Simulation

Cycle based simulation is a way of improving the simulation speed that is based on synchronous design principles. The method is known to be successful in verification projects [28]. Note that designing and verifying an asynchronous design is more difficult than a synchronous design. So, the modern designs are mostly synchronous. The method is used for



(a)



(b)

Figure 2.3: (a) A synchronous circuit,(b) an equivalant asynchronous circuit, taken from [27]

synchronous designs where there is a global clock where all outputs from the state holding elements (latches) are updated at the same time. A cycle is defined as a finest time granularity. During the cycles, all the logic elements are evaluated in a rank order and inputs to the latches are calculated. At the end of each cycle, latches outputs are updated.

The rank ordering method is called *levelisation*. Gates driving inputs to the flipflops are at level n, gates driving gates at level n have level $n - 1$ etc. The levelisation is mainly used to find the order in which circuit elements are simulated.

Event driven simulation is another form of simulation that uses a form scheduler to keep track of any logic evaluation that needs to be done. In other words, it uses a specific data structure (queue) to maintain the list of so called events. The list is in the order that the events must occur. Event occurs when a transition on a logic's inputs leads to a change on it output. The low activity rate of logic elements in the circuit, promotes simulation speed.

Cycle Based Simulation Algorithm

Most often, cycle based simulation implements an algorithm called *oblivious* simulation whose simplicity makes it efficient for further possible optimizations. During each cycle, all the combinational logic elements are simulated. Doing some extra work (unnecessary evaluation of logic elements) in an oblivious algorithm is a trade-off for not having the event scheduler in an event-based algorithm. Using synchronous design to eliminate the timing from functional verification to avoid dealing with hazard and glitches in cycle based simulation, and avoiding multi-valued representation of signals results in a simpler, efficient simulation model, which shall meet the verification quality.

During each clock cycle, logic gate are evaluated in the order of their path depth or level. Then, at the clock tick, the state of the flip flop is updated. At each cycle, the circuit reads in a new set of inputs and at the clock tick flip flops are updated and the outputs are generated. Algorithm 1 is the described cycle based simulation algorithm that is used in this work.

Algorithm 1 Cycle Based Simulation Algorithm

```
1: initialize each flop flop to zero
2: while there is more input do
3:     read inputs
4:     for $pd \leftarrow 0, critical\_path\_depth$ do
5:         simulate each logic function at depth = pd
6:     update flip flops
```

In a levelized circuit, logic gates in the same level are simulated independently. Being able to evaluate logic gates concurrently, makes it suitable for parallelism. Levelisation techniques have been used by Wang et al. [29], Maurer et al. [30], and other authors [31, 5, 32, 33, 34, 35]. The technique has also been used under the term ”ranking” [9]. Parallel logic simulation of independent components at the same time (using a levelisation technique), was used in [36, 37, 38, 39, 12, 8, 40]. In the section 2.9, these will be reviewed in detail.

2.8 Hardware Platforms

In this section, we divide platforms into categories of Supercomputers and Clusters, and Engineering Workstations.

- Clusters and Supercomputers: A set of connected (loosely or tightly coupled) computers is called computer cluster. The computers are connected via network, and each computer has its own instance of the OS. K computer is one of the top clusters manufactured by Fujitsu [41]. It has 705,024 cores and 10,510 TFLOP/s performance. It comprises 88,128 SPARC64 VIIIfx (8-core) processors. The computers annual running cost is around $10M. The same as high ranked supercomputers, this cluster is also very expensive. However, smaller clusters are cheap and available for research purposes, such as Raspberry pi cloud [42].

Note that many supercomputers are clusters, but not all clusters are supercomputers. Supercomputer clusters have high performance interconnect such as Infiniband [43] whereas lower cost clusters tend to use nothing more sophisticated than Ethernet. The current top supercomputer is called Tianhe-2 by NUDT with performance of 33862.7 TFLOPS (teraflops) located in China [41]. It has 16,000 computer nodes, each comprising two Intel Ivy Bridge Xeon E5-2692 and three Xeon Phi 31S1P. There is a total of 3,120,000 cores available. It costs about $390M. The total occupied space is around 720 square meter [44].

The Blue Gene/L supercomputer was released in 2004 and is based on an embedded PowerPC 440 processor, though with significant additions and modifications to the standard PowerPC system. In Section 2.9, an existing work on parallel logic simulation targeting Blue Gene/L supercomputer [4] will be reviewed. Its performance will be compared to ZSIM targeting Xeon Phi. Although, the processor used in Blue Gene/L is old (1999), this existing work is the only other work that targets an architecture other than GPU. Due to the reasons below, the comparisons made are meaningful:

  – The paper is very recent, so with respect to simulation it is the latest state of the art in the context of using super computer clusters.
  – Although the Blue Gene/L machine is old, but that comparisons of the number of cores and the clock speed are still relevant, even if the price comparison is less relevant (see Table 5.18, page 102 in Section 5.3.5).
  – The processor design in the Xeon Phi is an early model Pentium core P54C which was released in 1994. So, the core used in Xeon Phi is even older than the one in Blue Gene/L.

- Engineering Workstations: It is also called a PC. This is a low cost system comparing to above options and can be replicated. Workstations are divided to *CPUs* and *CPUs + Accelerators*. We previously mentioned that the frequency on processor chips levelled off since 2004. Rather than increasing the clock frequencies, CPUs have evolved through the use of parallelism in the system. The innovation of multi core chips and multi sockets chips has its own challenges. Available CPUs in the market are by Intel, AMD, and etc. The Intel Skylake based i7-6700K processor with 4 cores (hyper threaded), 4 GHz clock frequency, and 8MB cache is about $350 [45]. There are also workstations that have CPUs in addition to accelerators such as GPU, Xeon Phi, etc.

CPU + accelerator Workstation

Graphic Processing Units (GPUs) are the example of many core architectures containing massively parallel processors that are being used for general purpose computations rather than only graphical operations. This is due to the availability of this high performance hardware and the general purpose programming models such as Computer Unified Device Architecture (CUDA). For example, a GTX 770 provides 3.2 Tflops with 1536 cores and it costs less than 300. With hundreds of threads available, GPUs are able to achieve a performance of several orders of magnitude in comparison to a traditional CPU. This may not be the case for specific algorithms. NVIDIA introduced Compute Unified Device Architecture (CUDA); a parallel computing platform, that exploits and improve the parallel power of GPU.

At the moment, this research is limited to relatively small, low cost machines that an individual engineer might have. Therefore, we ignore supercomputer architectures or relatively cheaper clusters such as Beowulf clusters. Below is the list of platforms that were available for this work. Table 2.2 is the summary of these machines' specifications.

- A multi-core AMD Opteron 64
- Nvidia GTX 770 with 4 GB memory capacity and 224.3 GB/s bandwidth
- Xeon Phi 5110P with 60 cores (240 threads)
- Parallela: is a credit card size computer with dual core ARM A9 CPU on 64-core Epiphany coprocessor 1G RAM at the starting price of $99.
- Tilera TILEPro 64: is a multicore processor with 64 tiles (cores on chip), connected via a cache coherent 2D mesh network. Each tile is equipped with its own 8KB L1 and 64KB L2 cache. Each core is capable of running a Linux separately. Furthermore, it has four DDR2 controller to reduce DRAM accesses.

Xeon Phi and Tilera have different design strategies. Xeon Phi has ring topology, Tilera uses mesh Topology (mesh topology uses more connections). Note that there are NUMA machines (AMD64) available to use but that is not the key concern as the maximum design in this work would fit into the first bank of memory. Table 2.2, compares the specifications of above platforms. Among these relatively low cost machines, SIMD support, auto cache management, sufficient bandwidth, amount of shared cache and the number of cores are the factors we are looking at.

Parallella does not support SIMD, has 1.3 GB/s bandwidth, has simple slow cores and it is not comparable to Xeon Phi with 60 cores (hyper threaded) that supports SIMD. Tilera has four memory controllers, while Xeon Phi has 16 memory channels and higher bandwidth (320 GB/s) and more cores. After certain number of tiles start accessing memory, the performance of Tilera remains relatively small with more tiles [46]. The performance flattens out as its memory channels saturate faster.

Table 2.2: Specifications of available platforms

Platform	Model	Core	Memory	Speed	SIMD
Nvidia	GTX 770	1536	4 GB	1046 MHz	Yes
Xeon Phi	5110P	60	8 GB	1.053 GHz	Yes
AMD	6366HE	32	256 GB	1.8 GHz	Yes
Tilera	TilePro 64	64	16 GB	700 MHz	No (VLIW)
Parallella	E64G401	66	1 GB	800 MHz	No

Although a GPU has a higher number of cores, we will explain in the literature review (Section 2.9) that due to the low amount of shared memory, gaining performance is only possible for small circuit sizes. In this thesis, the Xeon Phi is used as our main target platform.

2.8.1 MIC architecture (Intel Xeon Phi coprocessors)

Intel introduced an alternative technology to GPUs, based on the Many Integrated Core (MIC) Architecture. The first version is on a 22 nm Knights Corner chip, sold under the name of Xeon Phi.The Intel Xeon Phi coprocessors are Symmetric Multiprocessing (SMPs)[3] systems that plug into the host (Intel Xeon processor) via PCI Express [47]. Xeon Phi cannot operate on its own and needs a host to provide disk and IO. These make the Xeon Phi similar to GPUs.

The Xeon Phi cores are based on the x86 Pentium core architecture. It has 57 to 61 cores clocked at around 1GHz. There are 4 hardware threads per core, which results in roughly 240 logical cores. Every core has 512 bit wide vector registers, in addition to the standard x86 registers.

The interconnection among cores is based on the ring network model. Cores are connected by a high speed bidirectional ring that allows the L2 caches to be accessible by all. In other words, the total of coherent cache over 30MB is available to all the cores. Xeon Phi has its own 6 to 16 GB on board GDDR5 RAM with around 170 GB/s bandwidth. Each core has its own 32KB L1 cache that is only accessible locally [47, 6].

Xeon Phi is equipped with the new set of instructions called Intel Initial Many-Core Instructions (IMCI) that is supported by the Vector Processing Units (VPUs) within each core. Each VPU has 512 bit SIMD (Single Instruction Multiple Data) vectors.

According to Intel, the KNC chip's theoretical peak performance is more than 1 TFLOP/s in double precision. The Xeon Phi has twice the performance per watt of the Xeon.

One of the greatest advantage of using MIC rather than GPUs is that the same code written for the multicore CPU can be used to run on the Xeon Phi coprocessors. On the other hand,

[3]SMP systems are tightly coupled multiprocessors with homogeneous processors running independently while sharing common resources.

CPU application code needs to be modified in terms of its algorithm and syntax when built for GPU applications based on the CUDA architecture.

An application written for MIC architecture using the Intel C compiler is reusable not only for the heterogeneous systems containing several Xeon Phi plus CPUs, but also for a system without a coprocessor. In this work, the same applications that were written in both C++ and Pascal, were compiled for Xeon Phi as well as another architecture (AMD Opteron 64). To target a different architecture, only a different flag and compiler were used to generate the executables. Although it was possible to use the heterogeneous model (offload), in this work, this feature was not used. As mentioned in the previous section, the programs were running natively on Xeon Phi.

The second generation of the MIC architecture is based on a 14 nm Knights Landing (KNL) chip. The Knight Landing Xeon Phi features up to 8 billion transistors packed inside a large die. It can be a processor on its own or a coprocessor (connected via PCI device). The Knight Landing was designed to offer higher floating point performance against older generation of accelerators. The next generation will support Intel Advanced Vector Extensions (AVX-512) instead of IMCI with the theoretical performance peak of over 3 TFLOPS/s in double precision [48].

The design on the KNL chip is separated into several tiles. Each tile comprises of 2 cores, each with 2 vector processing units (VPU); 2x AVX512 units. A 1M L2 cache is shared between the two cores. There are 4 threads per core. Tiles are connected via Mesh interconnect. The KNL has upto 72 cores.

KNL has upto 16 GB on-die high bandwidth memory (upto 400 GB/s). Moreover, the chip has 2 DDR4 memory controller that allows the support of up to 384 GB RAM by a complete platform via the 6-channel memory support. Moreover, other than KNL chip, Skylake Xeon also supports 512 bit AVX vector processing unit. Note that Skylake supports vgather in all versions. Gather instructions are used to collect non-contiguous data from memory. It then joins them into a vector register. This makes it more efficient to collect data into vector registers. The AVX512 will only be supported in Xeon server versions of Skylake. Note that the generations that support AVX2 instructions, also have gather instructions.

2.9 Review of Simulators

The initial research work on logic simulation started around 1980, where the concept of oblivious and event based simulation was first addressed [49, 50, 51]. Research on parallel simulation algorithms bloomed around the same time targeting both platforms with distributed memory [52, 53],and multiprocessors with shared memory [54].

Some of the available commercial simulators are based on these concepts. On a single CPU, they use optimizations such as compiled code technique to improve their performance. As previously mentioned, emulation and the use of specialized hardware boards are being used to increase simulation performance. The system comprises of several connected piece of homogeneous (similar) hardware. The emulation is done by partitioning the circuit into the blocks of hardware with optimized computational units [55, 56, 57]. The Yorktown emulation system was the first emulation system developed by IBM [58]. ASICs were used as its primary computing unit. Current emulators achieve around four orders of magnitude speedup comparing to software simulators. Emulators can also handle large designs. However, as previously discussed, they are not cost effective and successful mapping the design to an emulator is time consuming (may take several weeks).

This work is only concerned with software simulators. The remaining of this section explores studies on the related subjects to parallelisaing logic simulation targeting various platforms such as supercomputers and workstations with accelerators such as GPUs.

As previously mentioned, in term of how each logic gates is evaluated by a simulator, logic simulators can be grouped as *oblivious* or *event based* [59]. In oblivious simulators, all logic gates are simulated during each clock cycle, while only logic gates with a change on their inputs are simulated in event based simulators. Event based simulation requires analysis to determine the logic gates that requires scheduling for evaluation. So there will be a dynamic scheduling in contrast to simple static scheduling in oblivious simulation algorithm. Better data locality and static data structure in oblivious simulation, makes it a great option for parallelism targeting SIMD architectures such as GPUs.

Note that, in large designs, only 1 to 10% of the whole circuit is simulated during each cycle. So, despite the complexity of dynamic scheduling process, most commercial simulators are based on event based method to gain performance. Although the sequential event driven logic simulation algorithm is efficient, parallelisation of event based algorithm is challenging [60].

Event based simulation algorithms can be divided into synchronous and asynchronous algorithm. In synchronous event based simulation, a global clock controls the progress of logic processes [4]. For shared accessed, events are stored in a global data structure. The synchronous event based algorithm can be implemented on SIMD architectures [61]. To implement an asynchronous algorithm, each logic process is assigned to a local clock. Asynchronous simulation algorithms are classified into conservative and optimistic categories. In conservative approach, events with smaller time-stamp than the evaluated ones will be pushed forward [62, 63]. In the optimistic approach, events with larger time-stamp than the later events are allowed to be processed [64, 65]. After the related events were processed, the previous evaluation may turn out to be incorrect. In this case, there is a roll back mechanism

[4]Various modules in a simulated system

that restores the computations to before the incorrect evaluation. The optimistic approach may not be efficient on GPUs due to divergence as a result of complexity of flow control.

Although, in this work we only focus on oblivious simulation algorithm, in the literature, we refer to some of the recent works on both form of simulation algorithm.

Parallelising simulation algorithm targeting SIMD (Single Instruction Multiple Data) hardware systems was first done by [38]. The compiled code logic simulation targeting GPUs did not achieve an ideal performance. Due to the communication overhead (data transfer overhead) between CPU and GPU and not optimizing the data transfers between host and device, the CPU outperformed GPU. Moreover, they did not use the general purpose parallel programming model CUDA. Instead, they used Brook [66]. The partitioning algorithm they used was Combinational Fan Out Free Cone by mapping outputs of logic elements to a function of their inputs. The function is an arithmetic equation that can be evaluated using logical operations. The maximum size of the design they used in their tests was around 40K gates.

In contrast to the work in this thesis, Chatterjee et al. [37, 5, 11] uses circuit partitioning algorithms to achieve fast simulation. Chatterjee et al. [37] groups their proposed parallel simulation algorithms into two groups of synchronous and discrete event algorithms. Synchronous algorithms are those where several parallel threads fork and join during each simulation cycle. The join mode happens at a barrier that could be the end of cycle. In event driven algorithms, they use the partitioning algorithm. The circuit is partitioned into non overlapping sub circuits. Then, each is assigned to a thread to simulate. Among the several methods of partitioning a netlist including cone [67], balanced workload [68], random [69] or activity based partitioning [52], Chatterjee et al.'s approach is based on the cone partitioning algorithm and the use of a clustering algorithm. This choice can make a great impact on the communication overhead.

The GCS simulator by Chatterjee et al. uses levelisation as part of its process. The term *level* in Chatterjee et al. has similarities to the term we use in this work. Levelisation determine the sequence of logic gate simulation in the circuit. Chatterjee et al. describes levelisation as a way of organizing the circuit netlist in a way that logic gates in one level are completely independent from each other. So, their inputs do not depend on the output of another logic gate in the same level. In other words, the simulation of logic gates in the same level only depend on the values generated from the previous level. This way, logic gates at the same level can be simulated concurrently in parallel.

Chatterjee et al. proposed an oblivious simulator (GCS) using synchronous parallel simulation algorithm; a compiled code simulator that targets GPU by partitioning the design into clusters [37]. Each cluster is then mapped to a thread block. The clusters are independent and do not communicate with each other during each simulation cycle. To achieve this, a cone partitioning method was used. A circuit netlist can be viewed as a set of logic cones. Each cone contains all the gates contributing to generating the output. Each cluster contains several logic cones. Once a cone is simulated, the output value is written into the output vector. Due to the constraint on the size of the GPU local shared memory, only the frequently accessed data such as the intermediate signal values and the gate type truth table are stored in it. The truth table was used for evaluation of each gate during the simulation. As each thread block only has access to the small local shared memory on GPU, only a few data blocks can be accessible during the simulation. This limits the clustering algorithm performance. Then, the cluster is reshaped by the balancing algorithm, so that the cluster width (number concurrent threads; each thread is responsible for calculating the output of one gate) and height (logic level) will be changed. The purpose is to overcome the limiting factor of gaining speedup by minimizing the number of logic levels.

In Chatterjee's proposed oblivious simulator[37], all the logic gates are simulated during each clock cycle. The simulator was tested against a commercial simulator (event based compiled code simulator) for various number of cycles on a set of verilog designs. The target platforms were 8800 GT GPU (112 cuda cores) with 14 multiprocessors, 512 MB device memory, and 1500 MHz processor clock, in addition to a Pentium 4 workstation with 2 GB memory. It was reported that the GCS outperformed the commercial simulator by a factor of 4 to 60 for designs between 17K and 263K. Although the designed software was simple and statically optimize-able, unfortunately the size of the circuits that can be simulated by this simulator is limited due to the size of the local shared memory as well as number of multiprocessor on the GPU. The GPU used had 16KB shared local memory. So the maximum size of the design that GCS could have been simulated is around 900K (64K*14 = 896K).

Chatterjee et al. have also proposed an event based simulator using macro-gates [5]. The event based simulator has two phases: a compilation, and a simulation phase. The first phase involves circuit netlist transformation and creating the macro-gates. First, the levelised circuit netlist is segmented intro layers. Each layer holds a fixed number of levels, called gap. Gap is in fact the height of a macro-gate. As a logic element could have been assigned to two or more macro-gate, it will be duplicated. So, each macro-gate has a copy of it and can evaluate the logic gate without sharing data. Last, the number of outputs from each macro-gate is variable and called lid. In order to have the same number of gates in each level, the macro-gates are re-structured.

In simulation phase, one or more macro-gates is/are assigned to a multiprocessor. Macro-gates corresponds to a thread block. The number of concurrent thread blocks in a multiprocessor can determine the number of macro-gates that are simulated together.

Each macro-gate consists of several logic gates that are connected to each other. Macro-gates that are required to be simulated due to changes on their input values are tagged and added to the CUDA scheduler for simulation. The simulation of each of these logic blocks is assigned to a CUDA multiprocessor. Thus, there is no communication among macro gates. Concurrent threads within a block simulates logic gates of the same level. After synchronization, the phase moves on to the next level within a macro gate that was assigned to the multiprocessor. During the simulation, the data and the program resides on the device (GPU). The data that is being shared between macro-gates is stored on device memory. Due to the frequent access to the truth table for logic evaluation, it is kept on GPU shared memory, as well as intermediate values. In general, data placement for storing look-up table and intermediate values is similar to the oblivious simulator by Chatterjee et al.

After the simulation of chosen macro gates, their output values will be monitored in order to choose which of the macro gates will should be activated for simulation in the next level. The GPU platform that was used was similar to their previous study [37] and performance of their event-based simulator was compared against the commercial simulator that was running on an Intel Core 2 with 2.4 clock frequency. They reported that their simulator outperformed the commercial simulator by a factor of 4 to 44 for designs between 17K and 1M.

In the segmentation technique, the number of gap and lid are important as small gap value can lead to generation of more outputs (net) per macro-gate and a large gap number can lead to high activation rate of macro-gates. In [5], a fixed value was used for lid so that the number of logic elements in each macro-gate was about the same. The GPU they used allows the concurrent execution of 3 thread block. With 14 multiprocessors available, they only considered lids that would generate the maximum of 42 macro-gates per layer (14*3 = 42).

Note that having equal height layers may result in an imperfect mapping of partitions to CUDA blocks. As a results, in some layers, blocks may stay idle waiting for synchronization. In a later study [11], Chatterjee et al. addressed the issue by introducing an advanced segmentation technique that would have allowed flexible gap value for each layer (but kept the gap value fixed throughout each layer). The value of gap and lid can change the granularity of event based simulation.

Sen et al. [12] also proposed an oblivious logic simulation algorithm similar to Chatterjee et al [37] targeting GPU architecture. The partitioning algorithm that was used by Sen et al. is similar to [70] by Hering et al.. In the cone partitioning algorithm by Hering et al., the fan-in cone of a circuit element includes those part of combinational logic that have influence on that circuit elements' signal values.

Sen et al. used two clustering algorithms. In the first clustering algorithm, they used a threshold value to control the CUDA blocks. Then, they used merging and re-balancing technique to improve the clusters to efficiently use the shared memory on GPU. Chatterjee et al. did not use a threshold value in their clustering algorithm. Instead, they start the clustering from the outputs of a circuit. Furthermore, Sen et al. uses a fixed number of blocks to increase the execution of parallel threads.

Moreover, Sen et al. used the term Cycle Based simulation (CBS), while Chatterjee et al. called it oblivious simulation. Chatterjee et al. used lookup tables for gate evaluation, where as Sen et al. used AIGs (And Inverter Graph) representation where all the logic gates in the circuit were `AND` gates. AIG is a way of representing Boolean function manipulation. Using De-Morgan's rule, a combinational logic comprises of an arbitrary Boolean network can be transformed to an AIG graph.

The technique has been widely used in technology mapping, logic synthetic and verification [71, 72, 73, 74, 75]. Later in Section 3.1.4, this technique will be referred as an alternative to the use of look-up table for logic gate evaluation.

Sen et al. used a different partitioning algorithm to Perinkulam [38] as well as using CUDA programming model which was not used in [38] work.

The simulator by Sen et al. has two phases of compilation and simulation. During the compilation phase, a levelised circuit design is partitioned into clusters. Then, the clusters are balanced and each is mapped to a CUDA block. Each gate at a specific level within a cluster is simulated independently by a thread. In order to control the number of CUDA blocks, a threshold value is used in creating clusters. Each cluster is then assigned to a CUDA block. The threshold value should be equal or greater than the number of logic elements in the a specific level in the circuit. Some designs may have more or less logic gates than the threshold value. So, controlling the CUDA blocks using the threshold value leads to manual effort.

The observed performance of this method states that the execution time is linear with respect to the threshold values. The threshold number depends on the circuit structure and design. Sen et al. introduced a second partitioning algorithm to improve their previous approach.

In the optimized version, the clusters are created by starting from circuits primary output signals and latches instead of the given threshold value. This leads to creation of different number of clusters and sizes. Followed by the reshaping algorithm, the number of clusters would be the number of CUDA blocks and the number of gates within the cluster would be less than the total number of gates in all the clusters divided by the number of CUDA blocks. The number of logic gates per level would be up to the number of threads that would be allocated to each thread block. The number of allocated threads to each CUDA block is a multiple 16 threads that is half warp to ensure coalesced memory access. In this approach only the intermediate signal values are stored in the shared memory that requires frequent access. At the end of each cycle, the output data and current state are transferred from GPU to CPU.

Due to the limitation on the size of the shared memory, variables of type unsigned char were used to store the intermediate signal values that are either 0 or 1. This way, there will be more room for larger designs. The use of AIG format was an effective way of reducing the wasting space in shared memory that was used in Sen et al. work .

As logic elements in the circuit are all the same type, it uses the limited amount of low latency memory on GPU. However, the critical path depth of the circuit would be higher when AIG format is used. So, although the use of AIG representation could reduce the local memory size, larger number of gate levels result in execution overhead.

Sen et al. tested their two parallel simulators on several designs. They targeted Quadro FX3800 GPU with 192 CUDA cores (24 streaming multiprocessors and 8 streaming processors), 1 GB device memory, and 1204 MHz processor clock, in addition to an Intel Xeon CPU with 2 multiprocessors at 2.27 GHz and 32 GB memory. They reported that their simulators outperformed the default simulator that is available with AIGER [76] by a factor of 5 and 21 for designs between 2K and 220K. For designed smaller than 1K, their first parallel simulator that used a fixed threshold value for clusters performed very poor comparing to the sequential simulator.

Unfortunately the size of the circuits that can be simulated by this simulator is limited due to the size of the local shared memory as well as number of multiprocessor on the GPU.

Using two blocks for each streaming multiprocessor (SM) on their Quadro FX3800 GPU will leave 8KB local memory for each (the GPU they used has 16KB shared memory). One unsigned char variable was used to represent 8 outputs. It allows to have 64K (8KB * 8) variables for a block. So with the total of 48 blocks, the maximum size of the design that can be simulated is around 2M gates (48 * 64K = 3M variables including inputs, output, latches, and number of `AND` gates). However, Sen et al. did not report any timings for circuits of that size. There is not enough data to backup their theoretical calculation. The maximum circuit size they used was less than 220K.

Yuxuan et al. [9] used an adaptable partition strategy to achieve variable macro-gate [5] partitions based on the characteristics of the circuit, targeting GPU architecture. Macro-gates are scheduled and executed in blocks. Logic gates are then mapped to threads within a block. The gate evaluation is done through a look-up table. Note that the intermediate signals and the look-up table is mapped to the local shared memory.

Having variable macro-gate height instead of equal height [5], would reduce the data communication cost, and saves the amount of synchronization between parallel simulation tasks.

The proposed algorithm was used in both oblivious and event based simulation. The algorithm promotes load balancing among parallel task in order to reduce the synchronization and communication cost among blocks. The smaller height value results in creation of wider macro-gates, and vice versa. They introduced two threshold values to guide the partitioning algorithm: one to control the number of logic gates within each macro to ensure balanced load among parallel tasks, and another to limit the number of output signals from each macro-gates. The second threshold is due to the limited amount of local memory to ensure that all the signals in each macro fits in the low latency shared memory.

They tested their simulators on GTX465 GPU with 352 CUDA cores, 1 GB memory, in addition to an Intel Core Due T2400 with 2 GB memory and 1.83 GHz clock frequency. The parallel versions on GPU was compared to the sequential version on CPU for designs of size 60K to 200K.

Their oblivious simulator achieved a speedup of between 3 to 21, while their event-based simulator gained performance of 2 to 9. The maximum reported speedup was not large enough to be able to generalized the conclusion. Furthermore, similarly to the work by Sen et al [12]. the results were not compared to some existing work and their own sequential simulator was used as a base line to make the comparisons.

The main goal of this work by Yuxuan et al. was to show how other factors such as circuit characteristic can affect the partitioning of the circuit design. Moreover, the size of the partitions and the limited amount of local memory on CUDA platform are related factors that one has to consider in order efficiently utilize the computing resources on the GPU.

Zhu et al. [10] proposed a parallel asynchronous event based simulation algorithm targeting GPU architecture. Different from the work in this thesis, this event based simulation algorithm, does not hold a global clock across logic gates in the circuit. Their algorithm is based on the CMB algorithm [62, 63] (an example of conservative approach). Chandy Misra [62] and Bryant [63] developed the concept of distributed time, event based simulation and classified it as CMB algorithm. The idea in the CMB algorithm is that circuit elements are represented as Logic Processes. Each module in a simulated system is known as a Logic Process (LP). LPs communicate with each other via messages that are composed of a *time stamp* and a *value* that indicates the content of an event. Each LP may have several inputs and outputs and maintain its own local evaluation time. During the simulation, one LP may receive several new events from their inputs and generate several events at it outputs. Although, two or more events may happen as the same time stamp, independent events can be evaluated in parallel.

The CMB algorithm uses a priority queue data structure to store events for every logic gate. Due to the distributed time mechanism, deadlock may occur during the simulation [62].

Zhu et al. emulated a message passing model in their simulation algorithm. They adopted the usages of null messages into their algorithm to avoid deadlocks as a result of distributed time mechanism. After an evaluation of a LP, if it would not generate a new event, a null event would be sent. After another LP receives the null event, it simulation time will be set to null too.

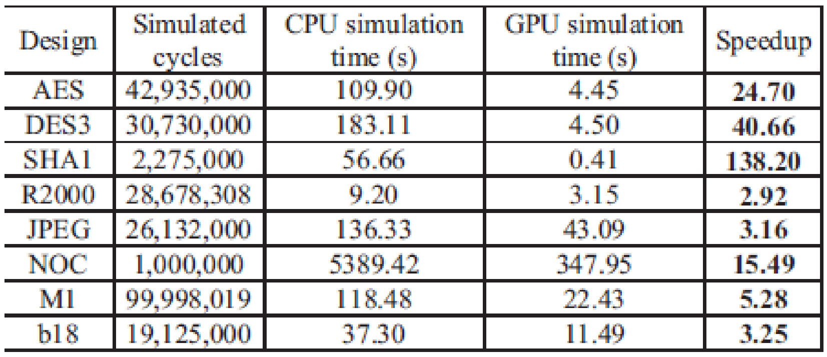
Due to divergent branches as a result of heap operations on the queue, parallelising the CMB algorithm and priority queue data structure on GPU will have poor performance. To avoid time consuming global sorting of the events, Zhu et al, distributed the priority queue of logic gates to their inputs. They used a distributed data structure that stores events related to each gate on its input pin. Each gate is mapped to an LP and each LP is assigned to a thread. Gates are evaluated and new events are added to the data structure. The logic gate evaluation is done via a look-up table (similar to [33]) that is stored in the constant memory. Furthermore, they used gate re-ordering prior to the simulation to avoid the divergence on the GPU. So, threads in a warp would evaluate gates of the same type. To evaluate the performance of the GPU based simulator, they used a typical sequential event based simulator as a baseline. Due to the message passing and local time maintenance overhead, a sequential CMB based simulator is slower that the classic one. They reported that their baseline sequential simulator was 2 times faster than the Synopsys VCS simulator.

They targeted GTX280 GPU with 240 CUDA cores and 1296 MHz processor clock, and a 2.66 GHz Intel Core2 Duo platforms. They tested the simulator on circuit design of size 6K to 117K with two set of input vectors: randomly generated stimuli and the officially released one (deterministic). The GPU based simulator achieved an average speedup of 47 for randomly generated stimuli, and the speedup of up to 59 for the released stimuli. When the activity rate is low, the parallelisation overhead causes insufficient acceleration. Input pattern with higher activity level would perform better with Zhu et al.'s simulator. The size of their circuits are not big enough to draw a conclusion on the scalability of their algorithm.

Chatterjee et al. [5] also proposed an algorithm for event based algorithm, their logic simulator evaluated multiple simultaneous events synchronously. Zhu et al.'s work is CMB based simulator that does not hold a global clock across logic elements in the circuit.

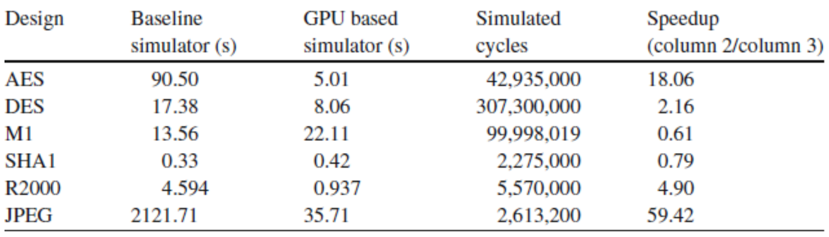
Results from the Previously reported timings by Zhu et al. [77], are inconsistent from these ones [10] (Figure 2.4). On the same circuits, using the same official released stimuli, the reported timings for their CPU baseline event based simulator running on the same system, were inconsistent. For example, the CPU simulation time in seconds for the AES design, takes 109.90 seconds in [77], and takes 90.50 second in [10] (note that results in both papers are for the exact 42M cycles). The CPU simulation time for JPEG design, takes 2121.71 seconds for 2.6M cycles in [10], and takes 136.33 seconds for 26M cycles in [77]. Due to the inconsistency in reported timings for their baseline simulator, their result are not valid and cannot be used for further comparisons. Table 2.3 shows the comparison of these results based on the cycle per microsecond metric for the baseline simulator.

The combination approaches explained in [5, 37] led to the creation of GPU event-based simulator (GCS) by Chatterjee et al. [11]. The GCS is a hybrid simulator that uses event based simulation as a coarse granularity and oblivious simulation within each coarse grain



Design	Simulated cycles	CPU simulation time (s)	GPU simulation time (s)	Speedup
AES	42,935,000	109.90	4.45	**24.70**
DES3	30,730,000	183.11	4.50	**40.66**
SHA1	2,275,000	56.66	0.41	**138.20**
R2000	28,678,308	9.20	3.15	**2.92**
JPEG	26,132,000	136.33	43.09	**3.16**
NOC	1,000,000	5389.42	347.95	**15.49**
M1	99,998,019	118.48	22.43	**5.28**
b18	19,125,000	37.30	11.49	**3.25**

(a) [77]



Design	Baseline simulator (s)	GPU based simulator (s)	Simulated cycles	Speedup (column 2/column 3)
AES	90.50	5.01	42,935,000	18.06
DES	17.38	8.06	307,300,000	2.16
M1	13.56	22.11	99,998,019	0.61
SHA1	0.33	0.42	2,275,000	0.79
R2000	4.594	0.937	5,570,000	4.90
JPEG	2121.71	35.71	2,613,200	59.42

(b) [10, 78]

Figure 2.4: Inconsistent results for the same circuits for baseline sequential event based simulator, taken from [10, 77, 78]

Table 2.3: Comparison of Cycle Per Microsecond from reported results by Zhu et al. [10, 77, 78] (Figure 2.4) All figures are the simulation cycle per microsecond reported for their sequential simulator. Note the huge discrepancy in reported performance between papers.

Design	[10, 78]	[77]
AES	0.474	0.391
M1	7.374	0.844
SHA1	6.894	0.040
R2000	1.212	3.117
JPEG	0.001	0.192

group. GCS uses the approach of partitioning the circuit netlist into macro gates and simulates each macro-gate in an oblivious mode. Logic gates in the same level are simulated in parallel within a block of threads. Furthermore, each individual cluster is scheduled for event based simulation. So, micro-gates are simulated by different thread blocks on different multiprocessors. This way only part of the circuit is simulated at each cycle.

They tested their simulator over a range of designs with 17K-1M logic gates against a commercial simulator for various number of cycles. They targeted the same GPU and CPU that was used in their previous study [5]. In that study, Chatterjee et al. reported that their simulator outperformed the commercial simulator by 4 to 44 times. In this current study, they results showed the advance segmentation method they used led to 15% performance gain.

Limited amount of shared memory on GPU that is shared among threads in a block, puts a constraint on the size of these micro-gates. The number of levels within a micro-gate and number of output signals of each micro-gate are other factors that affect the performance and the scalability of this algorithm.

In one of our previous studies [8], we applied several optimizations techniques to the cycle based simulation targeting GPU architecture and levelised synchronous circuits. We studied the interactions between various simulation optimization techniques and other factors such as circuit structure and the target platform. Instead of using queue as a standard way of implementing event based algorithm, we proposed a lock-free event based simulation technique called marking algorithm. Parallelising event based algorithm that uses queues may cause significant problems. Several threads may require access to the queue for basic operations such as add or removing an event at the same time. This will lead to deadlock.

The marking algorithm is an optimization to the cycle based simulation algorithm (we called 'brute force'). However, we use a Boolean flag for each logic element that indicates whether the logic gate needs evaluation not. During the simulation, all the logic elements in the same level are checked. If the flag was 1, then the logic gate is evaluated.

Further optimizations were done by determining the circuit elements that could save some evaluation time. For example, knowing the value of selector signal in multiplexer in a circuit, can save a lot of computation time as we can simple ignore evaluating all the logic elements that would generate the unwanted input signal to the multiplexer. These optimizations were done to simply avoid the extra work in the cycle based simulation, while maintaining the simple data structure to target parallel GPU architecture.

The simulation algorithms were tested on GTX 590 GPU with 512 CUDA cores and 1.22 GHz clock frequency. The comparisons were done against the sequential algorithm. The marking algorithm performance varies. Activity rate of the signals, can cause excess overhead as the process of flag checking leads to extra computation. Unfortunately, the reported results in the paper are limited to only one circuit design. The performance of the algorithms were not compared to any other simulators. So, a comparative conclusion cannot be drawn in terms of its performance. Note that depending on the circuit structure (path depth, type of components, and etc), input vectors, and the target architecture. The limited shared memory, the problems with using locks and atomic functions, and so on GPU, are the limited factors that need developers attention.

YAPSIM by Hashiguchi et al. [40] is another parallel logic simulator using GPU. YAPSIM is a levelised simulator that uses fan-out cone partitioning algorithm. They define the fan-out cone as a group of logic elements that determine the value of input to a flip flop or its output. The fan-out cones are re-arranged in descending order of gate count in each cone. Cone groups are divided between blocks. The gate evaluation is done by a look-up table. Although their simulation method is straightforward, enough detailed information was not provided in the paper. Therefore, the netlist and data structure used to represent the signals was not clearly explained in the literature.

They compared their results with its sequential version and a commercial event based simulator C-SIM. They used three GTX480 GPUs with 480 CUDA cores clocked at 1.4 GHz and 1.5 GB memory, and an Intel Core i7-950 CPU clocked at 3.07 GHz with 3 GB memory.

They reported results for circuits of size 2k to 84K. For combinational test circuits, YAPSIM achieved performance of up to 29 against C-SIM and up to 25.3 against the its sequential version. For sequential test circuits, YAPSIM achieved performance of up to 5.6 against C-SIM and up to 7.5 against the its sequential version. YAPSIM was not tested for any large scale circuit. So, scalability of the algorithm is not clear. The limited amount of shared memory (GPU resources) surely affect the performance of YAPSIM as maximum circuit size that can be used here is limited.

Holst et al. proposed an event based timing simulation algorithm targeting GPU(Kepler with 6 GB memory clocked with 980MHz)[79]. Their simulator supports hazards, pulse filtering and pin to pin delays. Signals and wires are treated as a separate delay entities. The simulator by Holst et al. calculates all events on intermediate and output signals for every input transitions. The simulator accepts waveform stimuli for input signals and computes the output value for each logic with multiple inputs. It was tested on a set of circuit designs of under 550K. They reported speedup of two orders of magnitude against the commercial event based simulator running on a single threaded CPU (Intel Xeon with 2.8 GHz and 256 GB memory).

There has been other research works on parallel logic simulation targeting platforms other than GPUs. Gonsiorowski et al. [4] studies parallel logic gate simulation on supercomputers. They mainly focus on parallel simulation of a OpenSPARC T2 crossbar. Gonsiorowski et al. use Parallel Discrete Event Simulation (PDES) simulation kernel ROSS [80] framework that is built on Jefferson's Time Warp [64] and is designed for PDES. Each object is known as LP and messages between two LPs present a signal between gates. Reverse computation [81] method was used to reverse the state of a gate as ROSS does not have a form of state saving. Similar to our work, Gonsiorowski et al. uses gate level netlist with basic Boolean gates and also considers unit-delay model.

The simulation is done at the level of LP. Each LP represents a gate that sends messages to other gates. Some of the data is stored in ROSS and there is only one copy available for the current state of each LP. Messages sent from one gate to another is after half cycle. When a message arrives at a gate, it leads to event update and scheduling for the next half cycle.

The experiments were performed on a 24 core SMP machine clocked at 2.66 GHz, and an IBM Blue Gene/L with 1024 cores, each clocked at 700 MHz. They duplicated a crossbar circuit with 211K gates to produce a larger circuit. Two set of randomly generated input vectors were used: one changing at every 30 cycle, and another every 2 cycles. In general, the optimistic simulation did not perform as well as conservative. One of the issues with an optimistic approach is that it takes up more memory than the conservative approach. The memory is used to store messages in case reverse computation was needed. For a circuit of size 216M gates, they were able to achieve 116M gate transition events per second.

Gonsiorowski et al. [82] did further experiment on the scalability of the algorithm. For this purpose, they used two systems. The first system was a 2 rack 418 teraflop Blue Gene/Q with 32,768 cores and 32TB memory, and the second system was a 96 rack, 20 petaflop Sequoina Blue Gene/Q with 1,572,864 cores and 1.6 PB memory. The second system was joined with a 24 rack Blue Gene/Q.

The scaling experiment was done on PHOLD benchmark. They achieved event rate of 504 billion on 1,966,080 Blue Gene/Q cores (120 rack) and 164 billion at 48 racks to execute 32 trillion event, 250M LP PHOLD configuration.

There are other recent works on parallel simulation [13, 83, 84]. However, as they are not directly related, these are not discussed further.

2.9.1 Final Thoughts

The existing literature is not satisfactory due to some common issues with the reported results; i.e: issue of direct comparison and representative compatibility.

- Using a commercial simulator as a baseline and not disclosing its firm and its performance [5, 37]. At least the commercial simulator should have been compared to a sequential version of the parallel simulator. This way, it would give a better idea of the performance of this unknown commercial simulator.

- Using a wide range of test circuits. So, it is difficult to compare the existing works together due to the very different scales of circuits. There are only a few circuits in common and there is a wide gap between the circuit sizes. For example, the work by Gonsiorowski et al. [4] cannot be compared with Sen et al. [12]. One uses a maximum circuit size of 200M gates and another uses a maximum circuit size of 220K.
- Some of the reported results from the same authors are inconsistent for the baseline simulator timings. The achieved performance are not valid for further comparisons (Table 2.3).

Table 2.4, shows the performance comparison of some of the papers discussed in the literature. Note that the table only shows the timings for the design that was in common in the papers. Unfortunately, the information related to the commercial simulator was not provided in some of the papers.The 'Sequential Simulator' column shows the timings for the sequential implementation of the same simulator. Sen et al. achieved a better speedup amount comparing to all. However, it is not clear how well their commercial simulator performed. Moreover, Yuxuan et al. [9] compared their simulator against their own sequential simulator with running for lower number of cycles than other simulators. Thus, the direct comparison of all of these simulators is not possible.

Table 2.4: Comparisons of Gate level GPU simulators for LDPC design (70K gates)

Paper	Parallel Simulator (sec)	Commercial Simulator (sec)	Sequential Simulator (sec)	Speedup	Cycles
Sen et al. [12]	24.32	382.21	-	15.71	100K
Chatterjee et al. [37]	193	12014	-	62.24	100K
Yapsim et al. [40]	1.77	51.8	44.8	29.26	100K
Yuxuan et al. [9]	10.13	-	91.54	9.03	25001

Unfortunately there is not enough data to draw any conclusion for the scalability of the simulators on the GPU. The maximum circuit size that was used on the GPU was around 1M by Chatterjee et al. [11]. Although some of the simulation performance results on GPU that were presented in the literature are significant, the simulators cannot handle large circuit designs due to a low capacity of memory on GPUs. There are several factors to consider when developing a simulator targeting GPU architectures. Some of these factors that can limit the performance of an application using GPU.

- The limited amount of shared memory
- Local synchronization within a block
- Memory locality within thread blocks
- Regular access pattern to memory for threads within a block

- Data transfers between host and device
- Explicit synchronization between thread blocks

In order to address the above problems, the issue of direct comparison and representative comparable have to be addressed. So, I have to be able to :

- to test designs of sizes from 1K to million gates
- to achieve average speedup of 13 to 14.4 or a maximum speedup of 60 over commercial simulators (comparing to [5, 37])
- to achieve gate transitions per second of at least 116M (comparing to [4])
- to show an improvement over the state of art, ZSIM (my simulator) will be compared to previous works over a wide range of circuits

To achieve this, Chapter 4 shows how I generate circuits of such range, Chapter 3.1 describe how I design an architecture to meet these achievements, and Chapter 5.3 shows the results achieved. The aim of the experiments are to:

- Verify that the data structures used allow SIMD acceleration, particularly on machines with gather instructions (section 5.3.1).
- Verify that, on sufficiently large circuits, substantial gains could be made from multi-core parallelism (section 5.3.2).
- Show that a simulator using this approach out performed an existing commercial simulator on a standard workstation (section 5.3.3).
- Show that the performance on a cheap Xeon Phi card is competitive with results reported elsewhere on much more expensive super-computers (section 5.3.5).

Chapter 3

SIMD Simulation Model

This chapter discusses the ways of using SIMD in logic gate simulation. Two questions are addressed: how to represent signals in the circuit, how to implement logic functions. A combination of possible answers to these questions are explored as approaches to use SIMD in simulation. Furthermore, we explain our SIMD simulation model along with the used data structure.

3.1 SIMD requirements

The vectorization is important because it increases the performance. On Xeon Phi, using vectorization means doing 16 single precision (SP) or 16 integer, or 8 double precision (DP) operation at once. Processors that support Intel SSE instructions would do 4 SP, 4 integer, and 2 DP, while the next generation that would support Intel AVX, could do 8 SP, 8 integer, and 4 DP. Comparing to this, Xeon Phi does 2 times more operations at once.

In order to use SIMD for simulation that allow a single instruction to perform a logical operation on 512 bit worth of data at a time, one may face challenges. It is obvious that one can perform parallel bit operations for logic using either a conventional or a SIMD machine if one packs multiple logical signals into a word. The problem with doing this for logic simulation, the special application of fault simulation aside, is that although one can efficiently handle the logic operations, simulating the interconnect this way is prohibitively expensive. It involves so much work shifting, masking and packing bits that it outweighs any parallelism gain in the logic operations. Note that in this method, all the bits in the 512 bit word must perform the same operation : AND, OR, etc. As a result, the SIMD simulator would have to group logic gates into blocks of ANDs, ORs, etc.

However if one uses one logic value per 32 bit word the Xeon Phi does allow one an effective way of parallelising logic, since 16 words can be operated on at a time, and the costly shifting and masking can be avoided by using the `vgather` instruction that loads a SIMD register `rx` with 16 double-words such that `rx[i]` is loaded from `mem[ry[i]]` where `ry` is another SIMD register, and `i` is in range 0..15.

A single instruction can only operate on 16 logic signals instead of 512, but that is still 15 more than we can do without SIMD. Given that there are 60 cores on the Xeon Phi chip we potentially have a simulation parallelism level of 960.

The challenge is to come up with a data structure that allows both the exploitation of the `vgather` instruction and also ensures good cache locality and allows the whole inner loop of the simulator to operate on multiple logic signals at once. It should be remembered that similar gather instructions are being made available on AVX-512, so the techniques will soon be applicable to machines other than the Xeon Phi.

To efficiently make use of SIMD instructions and do parallel vectorization, there are several factors that one has to consider. The first factor is data layout in memory. If data is not aligned and not laid out well in the memory, more instructions, cache and memory accesses are required to collect the data and organize them into registers, so that the vector operations can perform on them. The extra instructions can reduce the performance. Data locality is another factor that affects the performance including fetching data from the closest cache rather than memory and reusing data. Although data is transferred from the memory, the vector load instruction takes less time if prior to the load, the data has been fetched to the closest cache. On Xeon Phi, this can be done by pre-fetching data from memory to L2, then from L2 to L1. Data that is being reused should be stored closer together, to reduce the number fetches into cache.

When trying to use SIMD for circuit simulation, there are two questions to ask: how to represent logic signals in terms of programming data types, and how to implement a binary logic function?

Consider a simple scalar add operation between two integer numbers. Each occupy 4 bytes in memory. When doing scalar operation, each number is moved from memory to a register and the result is moved back to the memory after the calculation is done. In vector operations, we move multiple data into a register instead of a single datum. The data has to be lined up in the memory, to be able to move one pack of data from memory to a register.

3.1.1 Bit vs Word Data Packing

In this section, we explore the two ways of data packing in memory: 1) bit packing 2) word packing. The way the data is stored in the memory and how it is represented can affect the memory latency. Poor memory access to retrieve data leads to poor performance. In other words, it increases the simulation time.

During simulation, the circuit specification along with the signal values are stored in the memory. However, the information related to the logic elements in the circuit are all read only. On the other hand, the value of the signals are updated at intervals. So, as these current state values are accessed frequently, the way they are represented can affect the memory latency. Poor memory access can affect the performance/simulation time. How well these signals are packed together, can change the simulation time. A high density of packing improves information locality and makes the best use of the buses between memory and cache and between caches and processors. So, the focus is only on ways to represent the state signals.

Figure 3.1 illustrates the `state` array when signals are represented as bits or words. The `state` array is type integer. So, each 32-bit number either represent a signal (word packing) or 32 packed signal (bit packing). The figure also shows the `comp` array that holds the logic gate type. Note that the location of input value signals are stored in `inp0`, and `inp1` arrays.

As mentioned above, one option is to store each signal data as 1 bit in memory. Then, we store 32 signals, packed as a 32 bit number (Figure 3.1-b). For machines with larger words we might pack 64 signals into a word etc. In order to access the data value during the simulation, the single data bit is retrieved via bit-selection operations. This will normally be done by explicit $<<$ or $>>$ operations in C. In Intel assembly language it can be done using the `BT` instruction. In either case access to 32 bits spread out across memory will typically involve a loop to gather them up.

In order to have an efficient access to the signal values, signals values that are packed together a 32 bit number must be the outputs of logic gates of the same level and type. They must be the same type since we can only operate on 32 bits at a time using a CPU logic instruction which performs the same logic operation on all bits in a register. They must be the same level since we will be updating them with a single AND, OR etc instruction, and to avoid race conditions on multiple cores which may be running such instructions we require them to be at the same level.

It is possible that the number of same type logic elements in the same level may not be a multiple of 32. In this case, we would use zero padding and leave zero for the extra signals. This way, we would ensure all the 32 bit values belong to the output signals of gates at the same level. Figure 3.2) illustrates zero padding in bit packing technique. The `state` array in the figure stores the current state signals. Let's assume that there are `38` signals in level 2 of a circuit. Then, 32 bit signals will be packed together and the next 27 empty locations will be replaced by zero.

Another approach is to represent each signal as a word (Figure 3.1-a). In this case, each signal takes 4 bytes in the memory. Each signal value of 0 or 1 takes 4 bytes of memory. In this method, there is no need to extract a bit, as it is easily retrievable as it is. Obviously the density is not so good in this case, but it may still be faster if we have sufficient cache storage since we avoid shift loops.

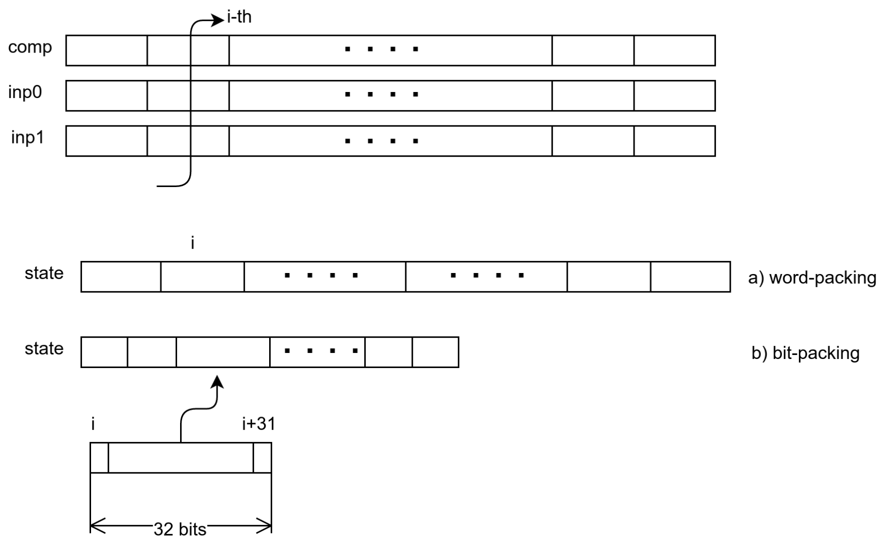


Figure 3.1: Signal Representation using a)word packing b)bit packing Technique.

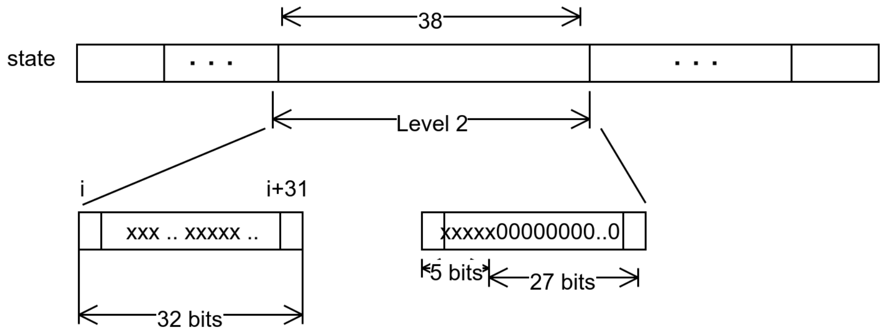


Figure 3.2: Zero padding in Bit Packing Technique

3.1.2 Look-up Table vs Direct Logic

In this section, we explore the two ways of evaluating logic functions: 1) look-up tables 2) direct logic. In gate level circuit simulation, Boolean logic evaluation of two input gate types can be uniformly implemented through an look-up table calculation.At run time this translates to a memory access.

Alternatively, a direct logic evaluation is possible using logical operators. At run time this translates to `AND` or `XOR` instructions, etc.

Look-up Table Look-up tables are a known way of implementing arbitrary binary logic functions such as `AND`.

Finite domain functions can be implemented via look-up tables. This often reduces the amount of computation during their evaluation. The computation of a function now takes a single memory look-up. However, depending on the complexity of the function and the size of the look-up table, it can take a lot of space in the memory. For example, when dealing with complex functions, use of look-up tables has to trade off accuracy against storage space, but they are still widely used for example to store Gaussian kernels in image processing. We demonstrated this use for the Xeon Phi in [14].

For a 2-input logic gate, a look-up table of size 16x4 holds all the possible truth tables. If we use 1 byte per entry, then the table will be permanently resident in the fast cache as it only takes 64 bytes (16*4) of memory space.

In this work, we used a smaller look-up table of size 6x4 to hold the values for 6 primary logic gates of `AND`,`NAND`,`OR`,`NOR`,`XOR`, and `XNOR`. To reduce storage, we use *char* data type to store the values in the look-up table. Our goal was to have logic function evaluations with fast memory access. Because we used a small look-up table that fits in cache memory, look-up table access time will be as fast as the primary cache access plus the time to evaluate two multiply instructions. The need for the multiply instructions is explained in 3.1.3 on page 40.

Use of look-up tables are a simple idea and it has been around for very long. Look-up tables are the main logic components in Field Programmable Gate arrays (FPGA). The look-up table method has been widely used in various studies [33, 37, 77, 11] in that field.

Direct Logic Direct logic is another method for evaluating logic functions. Using bit-wise logical operators are supported by the CPU. Boolean operations are the efficient to determine the value of the logic gate. To simulate an `and` gate, we can simply use bit-wise operation `&`. For example, $c = a\&b$

In larger circuits there are several logic gates of the same type. As mentioned in the previous section, when combining this method with bit packing technique,re-arrangement of logic gates is need prior to the simulation. In previous section, we have mentioned the necessity of this re-arrangement. At each level in the circuit, gates of the same type has to be next to one another. This way, we can pack the value of their output signals together. As a result, less memory is required to store the information in the structure bytes. The method ensures an efficient memory density. Figure 3.3 shows an example of a re-arrangement of logic gates in a circuit in bit packing technique. The figure illustrates the re-arranged logic gates in `comp` array. Logic gates of the same type are stored next to each other. The rest of vector arrays are organized accordingly. The vector array on top is a re-arranged version of array on the bottom of the figure.

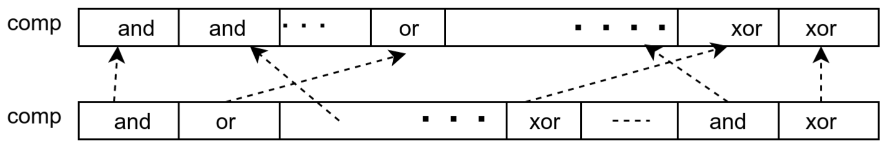


Figure 3.3: Re-arrangement of logic gates in a circuit in Bit packing Technique

3.1.3 Look-up table with Bit Packing

One possible approach is to combine a packed bit representation of data in memory with a look-up table technique for evaluating the logic functions.

In this approach, while sticking to the use of a look-up table to simulate a logic gate, we also store signal values at bits. We pack 32 bit signals into an single word, represented by an `int` in C, and store each in the `state` vector. During the simulation, we retrieve the value of the signals by unpacking.

The method is not efficient and not SIMD parallelisable. Everything is scalar during the simulation. Extracting bit by bit and performing logic operation (either using LUT or direct bit operation), is very inefficient. It is also possible to use the look-up table without pre-organizing the logic gate.

Listing 3.2, is an example showing the implementation of the above technique in C. In the look-up table technique, there is no need to re-arrange the logic gates prior to the simulation.

The circuit specification is stored in flat arrays; `inp0`, `inp1`, and `comp`. The current state values are hold in `state` array. All the array types are integers. During each clock cycle, for each level the simulation function is called once. The logic gates are evaluated and the results are stored in the `state` array. The signals values are stored as bit signals. The `state` array packs 32 bits of signals into one integer values.

For example, in order to simulate a logic gate at location `i`, the location of its input signal values in the `state` array is to be retrieved. To find the locations where the values of input signals are stored, multiple shift instructions are done. The values location `i` in `inp0` and `inp1` array are shifted. In mathematical explanation, we need to divide the integer values in `inp0[i]` and `inp1[i]` by 32. The quotient points to a location in `state` array. Below are the simple steps of retrieving one of the input values of logic gate `i` (the full calculation for `x` and `y` is shown in Listing 3.2 (Lines 10 and 11)):

1. $n = inp0[i] >> 5$: returns an index location in the `state` array. So, `state[n]` is 4 byte integer. When converted, we will have 32 signal values
2. $m = inp0[i] \& 31$: returns the m-th bit of `state[n]`
3. $p = state[n] >> m$: right-shifting m-th bit into the least significant bit position

4. $x = p\&1$: and masking with 1

Listing 3.2 (Line 13) shows the main part of the simulation that uses the look-up table to evaluate the logic gate of type `comp[i]`. The `comp` array contains integers indicating the type of the logic gates. Lets assume that the look-up table array `lut` contains 6x4 values. Listing 3.1 shows a look-up table array containing 24 elements.

```c
char lut[24] = {
    0,0,0,1,
    0,1,1,1,
    1,1,1,0,
    1,0,0,0,
    0,1,1,0,
    1,0,0,1
};
```

Listing 3.1: An example of a look-up table array

Each row in the `lut` array has the truth table values of a 2-input logic gate. Truth table is to show the function of a logic gate. To find the row in `lut` that contains the possible output values for the logic gate at location `i`, we multiply the type of that logic gate by 4. This is in fact a three dimensional array index type that is implemented as a C vector. To calculate the output signal value of `i`-th logic gate, we use a simple calculation as shown in Listing 3.2 (Line 16).

The new signal value is stored in `newstate` variable. Below are the steps for replacing the previous state value with the new one for logic gate `i`:

1. $j = i >> 5$: returns an index location in the `state` array. So, `state[n]` is 4 byte integer. When converted, we will have 32 signal values
2. $k = i\&31$: returns the m-th bit of `state[n]`
3. $mybit = newstate << k$: left-shifting k-th bit and padding to create a 32bit number
4. $p = state[j]$: returns the integer value in location j of `state` array
5. $temp = p\& \sim mybit$: and-ing the previous value in location j of `state` array with negate of `mybit`
6. $s = s|temp$: or-ing `temp` with the previous value `state[j]`

The simulation is done on one logic gate at a time, so is the bit retrieval. The part where signal bits are retrieved by shifting operations is not parallelisable and slows down the simulation. Furthermore, as the the simulation is done on one logic gate at a time, no performance gain is achieved by running in SIMD.

```
void simulation_bits (int *state,int glev_Num, int index) {

 int i;
 int last_index = glev_Num+index;

 int x,y;
 int newstate;

 for (i = index; i<last_index ; i++ ) {
   x = 1 & (state[inp0[i]>>5]  >> inp0[i] & 31);
   y = 1 & ( state[ inp1[i]>>5] >> inp1[i] & 31);

   newstate = lut[comp[i]*4 + x *2 + y];

   mybit = newstate << i & 31;
   state[i>>5] = (state[i>>5] & ~mybit) | mybit;
     }
}
```

Listing 3.2: Pseducocode for simulator program using look-up table with Bit packing technique

3.1.4 Direct Logic with Bit Packing

Another possible approach is to combine a packed bit representation of data in memory with a direct logic technique for evaluating the logic functions.

We can perform circuit simulation using logical operators while using 32 bit packed data type. In this approach, we can either pre-organize the logic gates or substitute them using the De Morgan rule. We previously mentioned the importance of re-arranging logic gates of the same type in the same level prior to simulation. During each clock cycle, logic gates of the same type that belong to the same level are simulated together. For each logic gate type, there is a different simulation function which corresponds to that type. Later, Section 3.1.4, explains the substituting method that can be used instead of logic gate re-arrangement.

Similar to the previous approach, the signal values are presented as single bits. 32 of these signals are packed together as 4byte integer in the `state` array. During the simulation each signal bit is extracted through shift operations during the simulation. The retrieval of signal values from `state` array was explained in the previous section. In the previous method, simulation of logic gates was done one by one. However, in this method, input value signals of 32 logic gates are extracted at the same time, and are evaluated all together. The evaluation is done through performing logical operations.

Listing 3.3, is an example showing the implementation of the above technique in C. In the direct logic technique, it is necessary to re-arrange the logic gates prior to the simulation. Some sort of grouping is done so that logic gates that are in the same level are simulated together.

In the example Listing 3.3 (Lines 9-13), is a `for` loop that packs the extracted 32 bit input signal values for 32 logic gates of the same type at the same level. The steps of retrieving one of the input values of logic gate `i` was explained in detail in previous section. In Listing 3.3 `x` and `y` contains the input values to 32 logic gates of an `and` type. Below are the additional steps that involves packing the extracted 32 bit signal together (the full calculation for `x` and `y` is shown in Listing 3.3 (Lines 11 and 12)):

1. $x = 0$: the variable is 4 byte integer, it's value is zero prior to extraction and packing the 32 signals, eventually contains 32 signal bits ready for logical operation
2. $xbit = state[inp0[j] >> 5] >> (inp0[j]\&31$: returns 1 bit (value of the input signal of logic gate`i`)
3. $x = (x << 1)|xbit$: returns the value of 32 input signals of 32 logic gates packed as integer. $x << 1$ left-shifting by one, the or-ing the result with the extracted bit signal

Listing 3.3 (Line 14), simulates the 32 logic gates by and-ing `x` and `y`; where each `x` and `y` is one 32 bit number containing 32 input signal values as below. The result of the logical operation is a 32 bit number that is replaced by the previous value onto location $i >> 5$. That location in the `state` array hold the output signals of 32 logic gates.

We unpack 32 bit signals at a time and perform logical operation on them together. Logic gates can be organized before hand, so that those with the same type will be packed together and ready for simulation. In this approach, we can benefit from parallelisim. The simulation is done on 32 logic gates at the same time.

During the simulation the signal bits are extracted and packed together. Then the relevant logical operator (e.g: &) is applied on the 32 bit signals together. The part where we retrieve signal bits by shifting operations is not parallelisable and slows down the simulation. However, the part where we perform the bit-wise operation on 32 elements at the same time in parallel.

Inspection of the assembly code (generated with -S flag at the compile time) showed that `gcc` compiler failed to vectorize the code in Listing 3.3. The C code was compiled using `gcc` with auto vectorization enabled (-O3 flag). Due to the dependency in the inner loops, the Intel compiler cannot safely vectorize this code either.

```
void and_simulation_32bits (int *state, int glev_Num, int index) {
 int i,j;
 int last_index = glev_Num+index;
 int x,y;

 for (i = index; i<last_index ; i+=32 ) {

   x=0;y=0;
   for (j = i; j<i+32 ; j++) {

     x = (x<<1) | (state[inp0[j]>>5] >> (inp0[j] & 31));
     y = (y<<1) | (state[inp1[j]>>5] >> (inp1[j] & 31));
   }
   state[i>>5] = x & y;
 }
}
```

Listing 3.3: Pseducocode for simulator program using Direct logic with Bit packing technique

Implementing Direct Logic by Substitution

As explained, one way of implementing direct logic and bit packing technique is to organize the logic elements prior to the simulation. By doing so, one logic operation is applied on 32 packed signals at the same time. Alternatively, instead of sorting the logic elements prior to simulation, it is possible to substitute logic gates using De Morgan law. Based on the proven De Morgan's theorems, we simply swap logic gates to an AND logic gate. A combinational logic design containing Boolean network can be converted and transform to And Inverter Graph(AIG) by applying De Morgan rules. Sen et al. [12] used AIG in their logic simulator. AIG is used to represent the manipulation of Boolean functions. Note that we are not using AIG format circuit netlist, we only use De Morgan law that is also used in And Inverter Graphs. The transformation and swapping is done as part of the pre-simulation process.

Equations below show substitution for OR and NOR logical operations using AND logic operator based on De Morgan law.

$$A|B = \overline{\overline{A}\&\overline{B}} \tag{3.1}$$

$$\overline{A|B} = \overline{A}\&\overline{B} \tag{3.2}$$

One of the useful properties of logical operators is the ability to toggle a logical value using a control bit (MASK). In below equation, if MASK bit is 1, then the value written in A will be negated. If MASK is 0, then the value of A will remain unchanged.

$$A = A \oplus MASK \tag{3.3}$$

By using three control bit signals `m1,m2,m3`, and above equation, we define the output of an `AND` logic gate as below:

$$C = m3 \oplus ((A \oplus m1) + (B \oplus m2)) \tag{3.4}$$

, where the three bits are equal to 0.

By changing the value of control bit signals `m1,m2,m3`, we can represent one of the `OR`, `NOR,`, or `NAND` logic gates, according to the De Morgan law for substitutions. We can convert these logic gates by replacing the control bit values shown in Table 3.1, in the above equation.

Table 3.1: MASK bits for Logic Substitution

Logic Gate	m1	m2	m3
AND	0	0	0
OR	1	1	1
NAND	0	0	1
NOR	1	1	0

In Listing 3.3, we can replace line 14, with below:

$$state[i >> 5] = m3[i >> 5] \oplus ((x \oplus m1[i >> 5]) \& (y \oplus m2[i >> 5])) \tag{3.5}$$

, where each of `m1, m2, m3` arrays hold 32 packed masking bits.

Figure 3.4, shows the 3 masking arrays holding 0 or 1 values as control bits. Note that the figure shows the masking arrays for 1 level only. Instead of pre-organizing the logic elements in the array, each logic gate is substituted to an `AND` gate and the relative control bits is stored in masking arrays (Table 3.1). The arrays are packed (good memory density). Comparing to the previous method, there is a trade off between more number of logic operations and logic gate's re-arrangement. Note the for `XOR` and `XNOR` logic gates, four masking arrays are needed.

3.1.5 Direct Logic with Word Packing

It is also feasible to combine a packed word representation of data in memory with a direct logic technique for evaluating the logic functions.

As in to the previous approach, we use logical operators instead of look up table, while evaluating logic gates. To implement the direct logic approach, we can also use intrinsics.

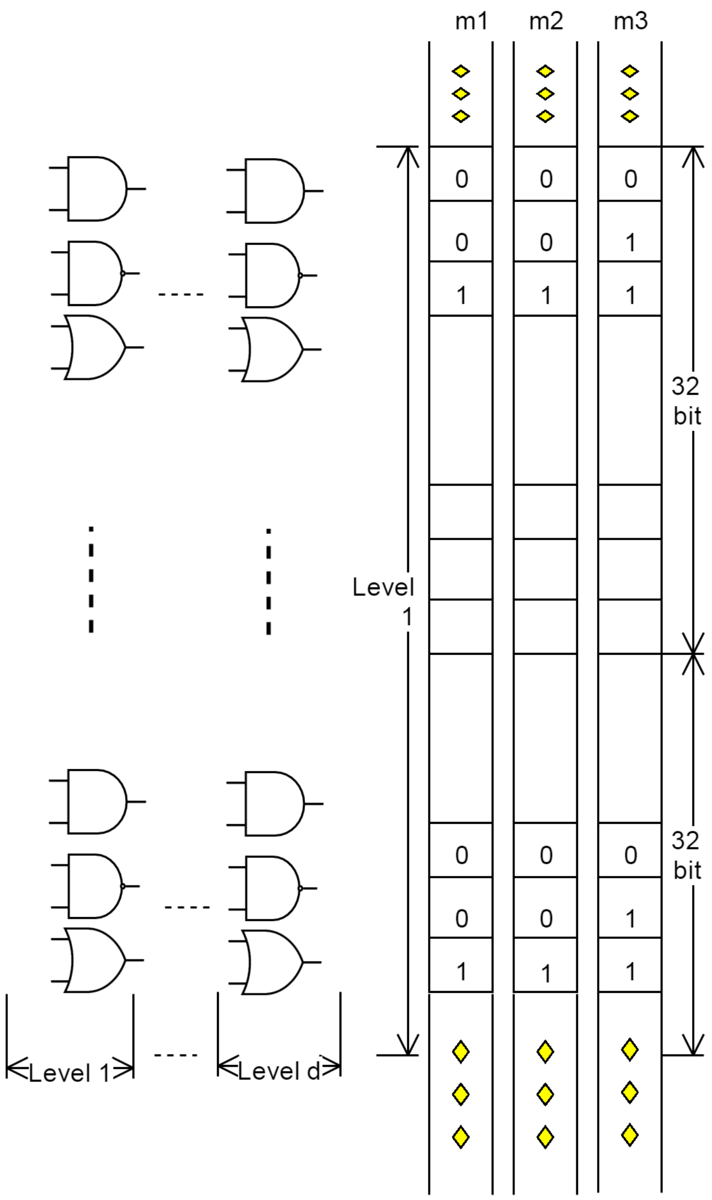


Figure 3.4: Bit packing and direct logic technique using control bits. The `m1, m2, m3` are masking arrays.

For example, one can use Intel intrinsics specifically for MIC. However, when using intrinsics, the programmers have to make sure of using the right instructions to ensure the vectorization. The `vgatter` and `scatter` instruction has to be used. Alternatively, we can let the Intel compiler does the auto vectorization and forget about using intrinsics manually.

Listing 3.4, is an example showing the implementation of the above technique in C. In the direct logic technique, it is necessary to re-arrange the logic gates prior to the simulation. Some sort of grouping is done so that logic gates that are in the same level are simulated together.

In the approach, signals are represented as words instead of bits. Each signal value is stored as a 32 bit number. The data needs more space in memory as it is not as dense as bit packing method. However, one of the advantage of this method is that there is no need to extract any signal value using bit-wise operations.

During the simulation, logics gates of the same type are simulated together.

Listing 3.4 (Line 5) is a `for` loop that loops over the logic gates in the same level and evaluated them. The loop is manually vectorized, so that the inside loop operation is done on 16 elements at the same time. Below are some explanation for syntaxes in Listing 3.4 (Line 7):

1. $temp = inp0[i : i{+}15]$: represent the 16 array element from `inp0[i]` to `inp0[15]`. The vector processor such as Xeon Phi can do 16 operations for a single vector instruction.

    - $inp0[i]$: returns a location in the `state` array that holds the value of the logic gate `i` input signal

2. $x = state[temp]$: returns the value of input signal

3. $state[i : i + 15]$: represent the 16 array element from `state[i]` to `state[15]`. The vector processor such as Xeon Phi can do 16 operations for a single vector instruction.

    - $state[i]$: returns a the value of the logic gate `i` output signal

```
1 void and_simulation_ (int *state,int glev_Num, int index) {
2  int i,j;
3  int last_index = glev_Num+index;
4
5  for (i = index; i<(last_index-last_index%16) ; i+=16 ) {
6
7    state [i:i+15] = state[inp0[i:i+15]] & state[inp1[i:i+15]];
8
```

```
    }
}
```

Listing 3.4: Pseducocode written in Intel Cilk for simulator program using Direct logic with Word packing technique

In the example Listing 3.4, the simulation is done on 16 logic gates at the same time. During the simulation the signal values are retrieved and the logic operator `&` is applied. The `for` loop in the simulation function is parallelisable and achieving good speedup is possible.

Listing 3.6 is the generated assembly code for the Listing 3.5. The C code was compiled using `gcc` with auto vectorization enabled (-O3 flag). The compiler failed to vectorize the code. The code would have been vectorized using Intel MIC. However, we compiled this on the machine that has SSE instructions and it could not vectorize loop.

```
void and_simulation_ (int *state,
   int glev_Num, int index) {
 int i,j;
 int last_index = glev_Num+index;

 for (i = index; i<last_index ; i
   ++ ) {

   state [i] = state[inp0[i]] &
   state[inp1[i]];

 }
}
```

Listing 3.5: Pseducocode written in C for simulator program using Direct logic with Word packing technique

```
.L84:
  movslq  (%r8 ,%rax)   , %rdx
  movslq  (%r9 ,%rax)   , %rcx
  movl      (%rdi,%rdx,4), %edx
  andl    (%rdi,%rcx,4), %edx
  movl    %edx          , (%rsi,%
    rax)
  addq      $4          , %rax
  cmpq      %r10        , %rax
  jne     .L84
```

Listing 3.6: The generated assembly code for the inner loop of simulator written in C using the combination of Word packing and Direct logic technique)

3.1.6 Look-up table with Word Packing

In addition to all the above approaches, another method is the combination of word packing and look-up tables. As in the previous method, the signals are represented in words rather than bits. Moreover, a look-up table is used to evaluate the logic functions.

This approach is highly parallelisable. There is no need to use bit-wise operations such as shift to retrieve signal bits. Furthermore, re-arranging the logic gates prior to the simulation is not needed either. Listing 3.7, is an example showing the implementation of the this technique in C.

```
void simulation (int *state,int glev_Num, int index) {
```

```
2    int i;
3
4    // last index in state that the last gates in this level resides
5    int last_index = glev_Num+index;
6
7    for (i = index; i<last_index ; i++ ) {
8
9      state[i]  = lut[comp[i]*4
10                      + state[inp0[i]]*2
11                      + state[inp1[i]]];
12   }
13 }
```

Listing 3.7: Pseducode for simulator program using Lookup table with Word packing technique

Below is some explanation for the terms used in the Listing 3.7 (Lines 9-11):

1. $comp[i]$: returns a value indicating the logic gate type
2. $inp0[i]$: returns the location of input value signal in the `state` array
3. $state[inp0[i]]$: returns the value of input signal to logic gate of type `comp[i]`

Figure 3.5, shows the location of input 0 signals to logic gates at level `d`. The result of output signals is stored in the `state` array for level `d`. Values in `inp0` array points to a location in the `state` array.

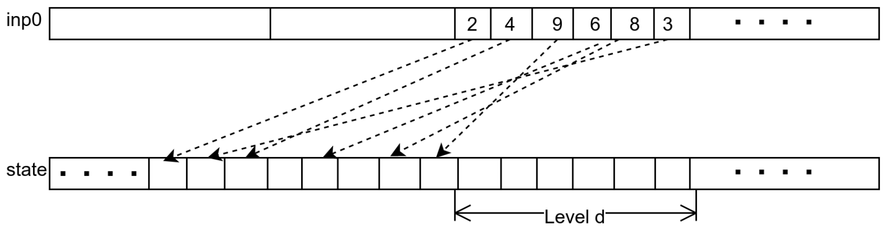


Figure 3.5: Illustration of input value retrieval from the state array

The code is parallelisable and we are able to use vectorization with this technique. Below is the optimization report for inner loop of the code in Listing 3.7. The full program written in C was compiled with Intel C compiler targeting MIC. The optimization report states that the inner loop was vectorized (Listing 3.8). To further prove the point that the simulator using the combination of look-up table and word packing will be vectorized, Listing 3.9 shows the instruction code for one gather construct with IMCI[1].

[1]Intel Initial Many Core Instructions (Intel IMCI are extensions to the existing Intel 64 architecture based vector graphic streaming SIMD instructions [47]

```
..L448:          # optimization report
                 # LOOP WAS VECTORIZED
                 # PEELED LOOP FOR VECTORIZATION
                 ...
```

Listing 3.8: An example of Intel Compiler Vectorization Report

In Listing 3.9, the `vgatherdps` instructions load 16 single precision values or fewer into a register from the 16 addresses specified in the its first argument. Note that every time a gatter instruction is used, it fetches one cache line,loads all the values that is needed to be gathered, then stores it into a destination vector register. Note that the number of gather instructions depends on how the data is distributed. If the data is in different cache lines, then multiple gather instructions are needed. The `jknzd` instruction checks the value of mask register. If one or more bits are non-zero, then it means that there is still more data to fetch. Zero bits indicate that the data have been gathered, and there is no need jump back to a label and gather more data. The information is helpful when trying to explain the performance of the application in case of dealing with non-contiguous data access.

```
        kmov       %k7, %k2
..L450:
        vgatherdps (%r13,%zmm0,4), %zmm7{%k2}
        jkzd       ..L449, %k2
        vgatherdps (%r13,%zmm0,4), %zmm7{%k2}
        jknzd      ..L450, %k2
..L449:
```

Listing 3.9: Gather instruction interface in IMCI from the generated assembly code for the inner loop in Listing 3.7

The technique explained in this section was also implemented in Vector Pascal (Listing 3.10. In this work, we used this combination in our SIMD circuit simulation algorithm. The Pascal code has been vectorized, compiled for various architectures. Section 3.1, explains the SIMD based simulator architecture in detail. The data structure and the circuit representation is discussed in Chapter 4.

Listing 3.11 is the generated assembly code for the simulator written in pascal. The simulator was compiled targeting MIC architecture. The assembly code only shows for the inner loop. The code has been vectorized.

```
procedure simulation(var state:buf; glev_Num, index:integer);
const stride=256;  {the the amount to do on each core}
type vectarr= array[0..100000,1..stride] of integer;
     pva=^vectarr;
var i,last_index,first_index, count:integer;
    in0,in1,st,com:pva;
begin
```

```
  last_index := glev_Num+index;
  count := glev_num div stride;
  first_index:=last_index mod stride;
  st:=@state[index];in1:=@inp1^[index];in0:=@inp0^[index];com:=@comp^[
    index];
  {$par}
  for i:= 0 to count-1 do
   st^[i]:=lut[com^[i]*4
                        + state[in0^[i]]*2
                        + state[in1^[i]]];

    for i:= index+count*stride to last_index-1 do
     state[i]:=lut[comp^[i]*4
                        + state[inp0^[i]]*2
                        + state[inp1^[i]]];
end;
```

Listing 3.10: The function simulates logic gates at a given level with a look-up table using multi-core and SIMD parallelism

```
label1525f8f1bcf424:;#1
mov            eax         , DWORD ptr [rbp+ -24];#1
cmp            eax         , 256;#1
jg             label1525f8f1bcf426;#1
movsx          r8          , dword ptr [rbp+ -24];#1
imul           r8          , 4;#RLIT;#1
mov            r9          , QWORD ptr [rbp+ -40];#1
mov            r10         , QWORD ptr [rbp+ -48];#1
lea            r12         , [r9+r8];#1
mov            r13         , QWORD ptr [rbp+ -64];#1
lea            r14         , [r8+r9];#1
mov            r15         , QWORD ptr [rbp+ -96];#1
vloadunpackld  zmm13       , [r15+r12]# LDpdw;#1
vloadunpackhd  zmm13       , [r15+r12+64];#1
knot           k1          , k0 ;#1
1:vgatherdps   zmm8{k1}   , [r10+ zmm13*4 ];#1
jknzd          k1          , 1b#VGATHERD;#1
mov            r15         , QWORD ptr [rbp+ -112];#1
vloadunpackld  zmm9        , [r15+r12]# LDpdw;#1
vloadunpackhd  zmm9        , [r15+r12+64];#1
.data;#1
2:.long     4;#1
.text;#1
vbroadcastss   zmm10       , [2b]#DREPR16C;#1
vpmulld        zmm9        , zmm9, zmm10 ;#VPDOP;#1
vpaddd         zmm8        , zmm8, zmm9 ;#VPDOP;#1
mov            r15         , QWORD ptr [rbp+ -80];#1
```

```
vloadunpackld  zmm13       , [r15+r12]# LDpdw;#1
vloadunpackhd  zmm13       , [r15+r12+64];#1
knot           k1          , k0 ;#1
1:vgatherdps   zmm9{k1}   , [r10+ zmm13*4 ];#1
jknzd          k1          , 1b#VGATHERD;#1
.data;#1
2:.long     2;#1
.text;#1
vbroadcastss   zmm10       , [2b]#DREPR16C;#1
vpmulld        zmm9        , zmm9, zmm10 ;#VPDOP;#1
vpaddd         zmm13       , zmm8, zmm9 ;#VPDOP;#1
lea            r15         , [label1525f8f0c92e];#1
knot           k1          , k0 ;#1
1:vgatherdps   zmm8{k1}   , [r15+ zmm13*4 ];#1
jknzd          k1          , 1b#VGATHERD;#1
vpackstoreld   [r13+r14]   , zmm8 #STpi;#1
vpackstorehd   [r13+r14+64], zmm8;#1
add DWORD ptr  [rbp+ -24]  , 16;#1
jmp            label1525f8f1bcf424;#1
label1525f8f1bcf426:;#1
```

Listing 3.11: The Generated assembly code for the inner vectorized loop of simulator written in pascal using the combination of Word packing and Look-up technique

3.1.7 Summary of Methods

Table 3.2, shows the summary of pros and cons of using bit vs. word packing techniques in SIMD or Scalar mode. The memory density is good when the signals are represented as bits instead of words. However, when trying to access the data stored as bits, it takes longer than when stored as words. In the bit-packing technique, the bit extraction takes extra time. Using word-packing in vectorization, multiple signal values can be accessed at the same time.

Table 3.2: Comparison of Bit versus Word packing

	Scalar		SIMD	
	Bit-Packing	Word-Packing	Bit-Packing	Word-Packing
Memory Density	Good	Poor	good	Poor
Access Time	Poor	Good	poor	Very Good
Logic Parallelism	Good	Poor	very good	Medium

3.2 SIMD Based Software Simulator Architecture

In order to benefit from SIMD architectures, we need to have an application with a simple regular pattern. The same operation should be applied to a large number of data elements. This is hard to do if one follows a typical object oriented approach. In such an approach a gate would be represented as an object or record, and wires between gates would be encoded as pointers between objects. This sort of data structure lends itself poorly to SIMD operation, which works better with a set of flat arrays.

Different logic gates perform different Boolean functions but all can be represented as truth table look-up. For any combination of inputs state, we simply use the predefined value stored in the memory. We can thus perform AND, OR, NAND etc in parallel using SIMD instructions which read an aggregate look up tablewhich in turn holds truth tables for binary logic gates. Listing 3.12 is an example of a look-up table array of size 6x4 for two input logic gates. This look-up table array contains 24 entries. The whole array occupies only 24 bytes in memory. The array represents logic gate truth table. Each row represents a logic gate function and its possible 0 or 1 input values. For example, the first four elements (0,0,0,1) represent the output of `AND` logic function. It shows that only if inputs to a `AND` logic gate are 1, the output will be 1, otherwise, it is zero.

```c
char lut[24] = {
    0,0,0,1,
    0,1,1,1,
    1,1,1,0,
    1,0,0,0,
    0,1,1,0,
    1,0,0,1
};
```

Listing 3.12: An example of a look-up table array

To keep the look up table simple, we have used only basic two input logic gates : AND, NAND, XOR, XNOR, OR, NOR. Larger circuits are broken down to the level of two input logic gates. For example, NOT gate is treated as a NAND gate with the same inputs. Listing 3.13 shows the predefined integer types associated with the logic gates that we use as row selectors for the look-up table.

```c
#define AND 0
#define OR 1
#define NAND 2
#define NOR 3
#define XOR 4
#define XNOR 5
```

Listing 3.13: Predefined integer types associated with the logic gates

The gate array netlist contains a circuit specification that includes information related to the path depth of each components, their inputs, outputs, and the component type. To make this simpler, each component in the circuit is given an index ID. The type associated to each component is stored in an array at location ID. In addition to this, each signal in the circuit has its own index corresponding to the index of its driving gate.

To allow efficient parallel access we represent the circuit as 4 contiguous vectors. The first three hold the structure of the circuit: `comp` which holds component types, `inp0, inp1` which identify the two inputs to a component (Figure 3.6). The `inp0` and `inp1` arrays points to a location in `state` array that signal values are stored. The final vector `state` holds current value of internal signals (Figure 3.7). Output signals of logic gates in the same level are stored adjacent to each other. The `Level` notation in Figure 3.7 follows the `shape_vec` array.

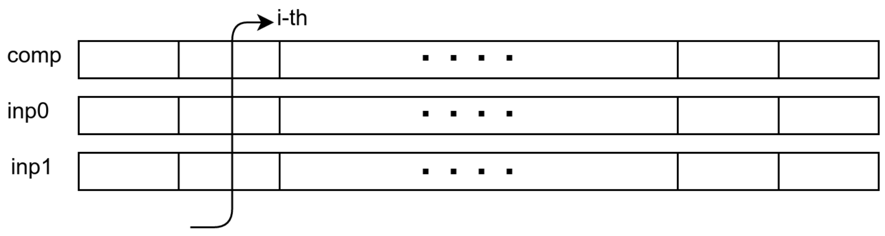


Figure 3.6: Vector arrays to hold the circuit specification (i.e.: logic gate type and input signals)

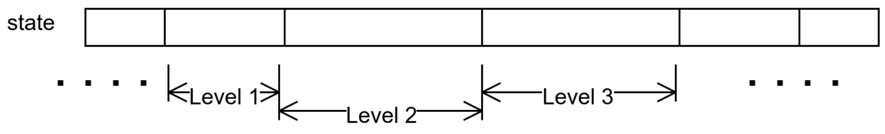


Figure 3.7: Vector array that holds current state signals.

The `state` array contains the current state values of all the signals (Primary Inputs (PIs), Primary Outputs (POs), and intermediate signals). We have also defined separate arrays to update the state of Flip Flips. These arrays points to the location of input and output signal of each Flip Flop in the `state` array. The number of components of the same path depth is kept in the `shape_vec` array. Note that we extract all the relative information from the generated netlist. Our specific netlist already has an array format circuit specification. Listing 3.14 shows the definition of these arrays. The size of these arrays are given in the netlist that is fed into the simulator. The size of `comp`, `inp0`, `inp1` arrays is the same and equal to the number of inputs plus number of gates. `state` array's size is equal to the size of `comp` array plus the number of flip flops in the circuits.

```
csize = gateNum + inputNum; // circuit size
stateSize = csize + numdff; // state vector size

```

```
int * comp = new int[csize];
int * inp0 = new int[csize];
int * inp1 = new int[csize];

int * state = new int[stateSize];

int * ff_in = new int [numdff];
int * ff_out = new int [numdff];
int * shape_vec = new int [maxLevel];
int * outp = new int[outputNum];
```

Listing 3.14: Definition of vector arrays

An example of vector arrays filled with the circuit specification is shown in Figure 3.8. Each location i in arrays `inp0` and `inp1`, contains the ID of the input signals to component in location i in array `comp`, where the type of this component is stored. The current state value of its output signals is stored at index i in `state` array. In this example, the size of `comp`, `inp0`, `inp1` arrays is 6 as the circuit has 2 inputs and 4 logic gates. The size of `state` array is 7 as in addition to 2 PIs and 4 logic gates, the circuit has 2 flip flops as well. The value of input signals are stored in `state[0]` and `state[1]`. These signals are considered to be at level 0 (it is annotated as L0 in the figure). Let's consider the NOR logic gate in the circuit (Figure 3.8) . Its type will be stored as an integer in `comp[2]` and the values of its input signals are held in `state[inp0[2]]` and `state[inp1[2]` that is `state[0]` and `state[7]`. The value of its output signal is stored in `state[2]`.

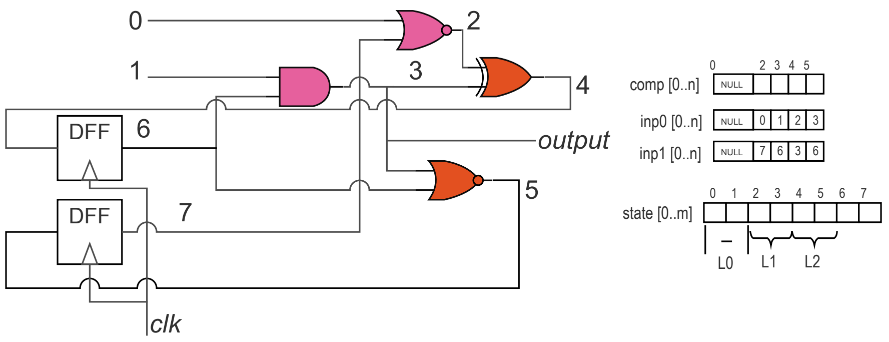


Figure 3.8: An example of a circuit with label. Logic gates of the same level are shown in the same color.

Listing 3.15 shows the main program for the simulator. The simulation function is wrapped in the loop and called at each level (Listing 3.15, line 6). During the simulation, the value of current state signals is calculated as shown in Listing 3.16, line 7. The use of look up table in this calculation, and the way the data is stored in the arrays, allows this calculation to run in SIMD. In order to have an aligned memory access, all the relevant data for a block of 16 components $i..(i+15)$ is stored at the same block of indices in the arrays. This induces adjacent memory access which accelerates the running of the calculation in SIMD on each of multiple threads. To ease the read and write access pattern, we store chunks of information related to each level in the circuit, next to each other in the arrays.

```
int main (int argc, char* argv[]){
 ...
 generate_input (state);
 int index = inputNum;
 for (int level = 1; level < maxLevel; level ++) {
    simulation(state, shape_vec[level],index);
  // points to the start of comps in the next Level
    index += shape_vec[level] ;
 }
 ...
}
```

Listing 3.15: Pseudocode for simulator program. Iteration through levels is sequential Note that the outer for loop start from level 1 as the primary inputs are considered to be at level 0.

```
void simulation (int *state,int glev_Num, int index) {
 int i;
 // last index in state that the last gates in this level resides
 int last_index = glev_Num+index;
 #pragma omp parallel for simd
 for (i = index; i<last_index ; i++ ) {
    state[i]  = lut[comp[i]*4
                + state[inp0[i]]*2
                + state[inp1[i]]];
    }
```

Listing 3.16: The function simulates logic gates at a given level with a lookup table using multi-core and SIMD parallelism

Intel Xeon Phi has 128, 512-bit SIMD registers on each of its cores (32 per thread). Depending on the size of the circuit and its shape, the component of the same path depth will be simulated as 512 bits chunks of data (Figure 3.9). In other words, the load/store, read-/write, as well as above calculations are done in SIMD on 512 bit of data at the same time (Figure 3.9).

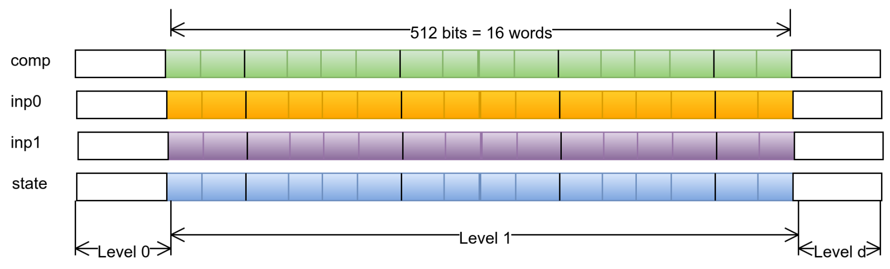


Figure 3.9: Example of performing SIMD operation on 512-bits of data in the integer array

Given an array of size N, on Intel Xeon Phi with 240 threads, each physical thread is allowed to process **N/240** elements of array. On top of this, vectorization allows 16 simultaneous calculations. So, each arithmetic unit only has to do **N/3840** calculations per threads.

Figure 3.10, shows the workload among threads at each level, during the simulation. The curved lines in the figure symbolized the synchronization between threads. At each level, logic gates are divided among threads. The amount of workload for each thread may be different at each level. This depends of the shape of the circuit (distribution of logic gates per level). The sizes of the boxes in the figure shows the amount of workload per level. Each thread performs calculations on piece of data, equal to others. This ensures work balancing among them (Figure 3.11).

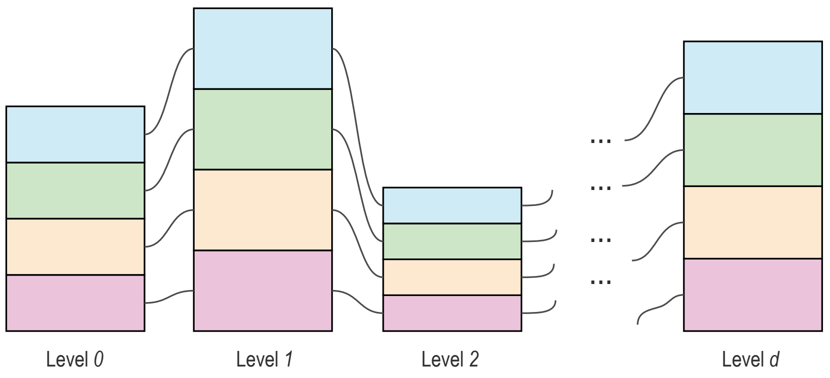


Figure 3.10: An example of workload among threads per level simulation. The curved lines in the figure symbolized the synchronization between threads.

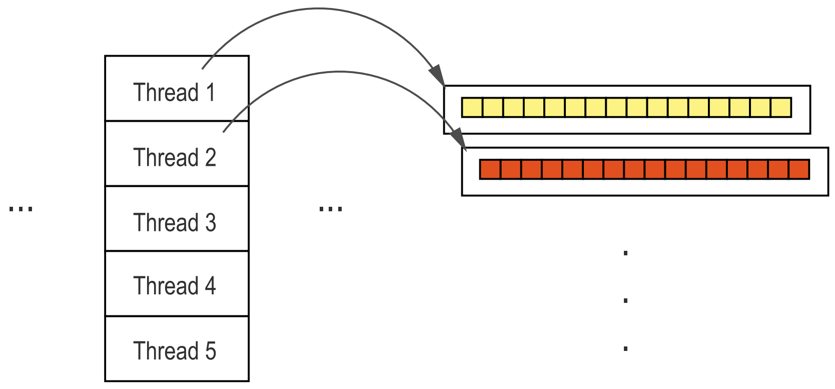


Figure 3.11: An example of workload per thread per level. Each thread is capable of performing a single operation on 16 ints simultaneously.

Chapter 4

Circuit Benchmark Suites

This chapter explores various existing circuit suites, from standard benchmark circuits to synthetic ones. Moreover, I briefly discuss the method that was used to generate netlist for my SIMD simulator (ZSIM). The chapter also provide details of how the circuits are represented.

4.1 Circuit Suites

In order to evaluate the performance of a simulator test circuits are needed. There are several benchmark suites with different level of abstraction and circuit complexity [85]. These suites have examples of 'real' circuits. The number of these test suites is few and within them the examples are of small circuit size. Furthermore, it is clear that a few examples cannot represent all circuit classes [86, 2].

The alternative is to use synthetic circuits. With synthetic benchmark circuits, we can have control over important characteristics of the circuits such as size (total number of logic gates), depth (the maximum number of levels or critical path depth), and shape (relative number of gates in each level). This section explores various existing circuit benchmarks and the concept behind synthetic circuits. Furthermore, we explain the method used to generate our experimental test suites.

4.1.1 Standard Benchmark Circuit

A main reasons for the lack or large benchmark designs is confidentiality concerns in industry. For over a decade, a small collection of circuit benchmark suites have been used widely in publications and to validate algorithms. There are various benchmark suites that describe a digital circuit at different levels of abstraction. Below we explore some existing sources of benchmark circuits.

The Microelectronics Center of North Carolina (MCNC) circuits were sponsored by the ACM/SIGDA. They were collected in 1980ś and are a common source of benchmarks. The circuit suite comprises 205 circuits which range from 24 Logic Elements (LEs) to 7694 LEs. It covers several applications. At least 85 of the 205 circuits are state machines or arithmetic logic units [87]. Out of these 205, 77 multilevel (sequential and combinational) benchmarks were originally available in BLIF(Berkeley Logic Interchange Format)[1], but some of the others have also since been converted into BLIF files. Although these circuits are small, their ease of use and wide acceptance make them a popular choice for comparison.

Another recent benchmark suite is from the Quartus University Interface Program (QUIP). It contains 45 real circuits [89]. The benchmark suite was described by Pistorius et al. in [90] where they proposed the addition of a *black box*[2] directive to the BLIF format which would represent hard blocks. Although these circuits are much larger (up to 134,341 LEs) than the MCNC circuits (7694 LEs), only 7 of them can be synthesised to standard BLIF files of which the largest would have 9,867 LEs.

A recently released suite is eASIC that provides a series of 5 ASIC netlists [91] with the sizes in the range 125k to 1M elements.

The two ISCAS [92] and ITC'99 [93, 94] benchmark sets are the most commonly used ones. The ISCAS sets contain both combinational and sequential circuits. The largest of the circuits has 22179 gates and 1636 flip flops. The full overview of the benchmark sets can be found in [85].

Open Source Circuit Repositories OpenCores [95] is an online repository of open source circuits. A wide range of circuits are available in HDL language that would need conversion to BLIF or any other preferred netlist format. Often the size of the available circuits are small[3] as they are intended to be used as part of a bigger circuit. For example a processor containing around 5k Logic Elements is meant to be stitched to other logic cores in order to shape a complete circuit.

[1]BLIF [88] is used to represent combinational and sequential logic circuits in logic synthesis and verification tools such as Quartus.

[2]In order to enable the representation of the module where specification and logic function is not present, *black box* is used.

[3]They are a collection of Intellectual Property(IP) blocks (a pre-designed module)

4.1.2 Synthetic Benchmark Circuits

Synthetic benchmark have been used as an alternative to standard benchmark circuits [96, 97, 98, 99, 100, 101]. Such circuits are generated using an automated process that allows to specify desired circuit characteristics. Attempts to generate synthetic circuits to be used as benchmark suit for experiments and algorithm validation go back to the 1990s. Circuits constructed using these generators are homogeneous graphs of basic logic gates and latches. Real circuits are often less homogeneous, since they would contain memory elements, PLAs etc. These circuits were, in the main, validated by comparing with MCNC design sets.

However, the main advantage of synthetic circuits is their controllability in terms of characteristic circuit parameters such as size, interconnection structure and functionality. Those parameters have limited influence on each other and can be set separately. This can give control over the overall character of the benchmark circuits. In other words, we can generate a set of circuits of varying sizes with different circuit characteristic. By changing a single characteristic of the circuit to test a particular application, we are able to draw more informative conclusions from the experimental results.

For example, one may want to examine the behaviour of a particular algorithm or architecture on different circuit sizes with the same fan-in count.

Synthetic circuits have been used to test FPGA place and route algorithms [96], logic optimizers [97], and partitioning algorithms [98, 99, 102]. By cloning existing circuit benchmarks, sequential benchmark circuits were also developed to use for testing partitioning, place and route algorithms [100, 101].

To justify the use of synthetic circuits, they have to be able to show realistic properties. Among the different approaches for creating synthetic circuits, we chose Hutton et al.'s [2, 103] method of random circuit generation. They proved and validated their method to be realistic by comparing the examples generated synthetically from properties of the real circuits to real benchmark circuit sets. They termed such circuits 'clones'. The clones were generated based on important characteristics of real circuits such as total wire length after routing and placement, critical path depth, and power consumption.

Although Hutton et al. successfully validated clones that were purely combinational, clones for large sequential circuits were less successful. In this work, our circuit generator is derived from the Hutton et al.'s synthetic circuit generator GEN. Instead of using GEN directly, we use the parametric formulas discussed in Hutton et al. in a synthesizer that we constructed from scratch. For instance, in order to generate sequential circuits, we took the purely combinational circuits and glued them to latches. Thus our circuits are classic state machines with a state vector of latches and a large combinatorial block that feeds these. GEN, in contrast, generates sequential circuits by gluing combinational sub circuits together via flip flops. However, as we use levelisation techniques to flatten the circuit, we would eventually will have the Flip flops in level 0 (Chapter ..) so it is simpler to generate such circuits directly. Our randomly generated circuits are based on the statistical and structural characteristics of circuits described by Hutton et al.

4.2 Synthetic Circuit Generation Algorithm

In this section, we discuss the formulas used as the basis of our synthetic circuit generator. We have used the formulas in [104] to determine reasonable defaults for generating parameters that define the characteristic of a combinational circuit as well as sequential ones. The method and the parameters used were validated against MCNC benchmark circuits and the properties showed realistic behaviour.

4.2.1 Circuit Characteristics

Given N, the size of the circuit, the synthetic generator would use the default formulas to produce a circuit with reasonable characteristics. In this process, number of circuit's primary inputs(*nPI*), outputs(*nPO*) for the given circuit size of(*N*), the circuit delay[4], and the circuit shape[5] is also calculated through the formulas.

To generate sequential synthetic circuits with single global clock, we first calculated the estimated number of flip flops (*nDFF*) in the circuit, and connected the the rest of the combinational circuit to a latched state register of length *nDFF*.

The `GEN` synthetic circuit generator uses the *Rent* parameter that identifies the interconnection complexity of the circuit. This parameter is used in the Rent rule formula. It shows a relationship between the size of the sub circuits and the number of their IOs [105]. Note that the Rent parameter and Rent rule is not directly used in this work. We are only briefly mentioning the formula as it was used in `GEN`.

The Rent rule is based on the following equation:

$$T = tg^p \tag{4.1}$$

[4]circuit delay *d(x)* is the maximum length from the primary input to the point *x*, for a circuit with unit delay model

[5]circuit shape that indicates the distribution of the logic gates at each delay level (excluding primary inputs and outputs) e.g: for a given circuit *c* with combinational delay of 4, the shape could be 12,4,4,2. In circuit *c*, 12 is the number of primary inputs to the circuit, and the remaining numbers of 4, 4, 2 is the number of logic gates at levels 1,2, and 3.

, where T is the number of pins at the boundaries of the sub circuit, and g is the size of the sub circuit, and p (rent parameter) and t are constants. This rule is a very important factor to consider in algorithms for separating the circuit into modules (sub circuits). The rent parameter is low where the partitions are small, the opposite applies when the sub circuits are large. The reason is based on the observation of circuit designers to restrict the number of IO in the circuit that could be fit on a chip. Higher rent parameter in small sub circuits is due to lack of hierarchy in the logic. The larger, the sub circuit, the lower the p.

The number of IOs achieved by using Rent formula works for purely combinational circuits. When dealing with sequential circuits, the size of the circuit (number of logic gates) and IO will not have a statistical correlation, as well as number of flip flops. This is the case when we have several sequential levels (sub circuits). Although, our focus in this work is to consider only one sequential level circuit, we can still use GEN subroutine as the basis of our synthetic circuit generation algorithm. Note that we only allow 2 input logic gates in the circuit, we do not use all of the formulas defined in GEN.

4.2.2 The circ-gen Generation Algorithm

We use circ-gen that uses GEN [100] subroutines to generate sequential synthetic circuits with single global clock. The algorithm is implemented in C language. For the given N, the approximate size of the circuit (nPI + nDFF), we can generate several circuits of different shape and attributes.

The circ-gen program divides into two parts. The first part of the program is loosely based on the Hutton et al. method of synthetic circuit generation and directly uses GEN subroutines to return the number of primary inputs,outputs, flip flops, depth, and the circuit shape. The circuit shape is a vector of integers whose elements specify the sizes of successive layers of logic. In other words, GENs input arguments are n(approximate size of the circuit that includes the total number of logic gate and primary inputs), and f(number of fan-in) and it returns the number of flips flops, inputs, output, maximum delay, and the shape vector.

The second part of the program uses a gluing algorithm that uses the generated parameters from GEN to produce a circuit netlist. The netlist contains all the information related to the circuit in a specific array format that is then read in by the simulator. The gluing algorithm that does the generation of the circuit has the time complexity of $O(n * m) + k)$. This depends on the depth and shape of the circuit (number of logic gates at each depth). k is almost equal to the number of flip flops in the circuit. Below is the summary of the gluing algorithm used in circ-gen:

First we calculate the necessary parameters for a given N. Next, we generate the circuit (connecting logic gates and placing flip flops according to the parameters) based on following rules: each logic gate gets its input from primary inputs or from outputs of logic gates from prior levels. However, at each level, each logic gate is forced to take a signal from the immediately prior level and one from any of the prior levels. By knowing the shape of the circuit, the number of logic gates at each path depth is also clear. However, the type of each logic gate is assigned randomly (it could be XOR, XNOR, AND, OR, NOR, NAND, NOT). Eventually, we connect the circuit to the flips flops. Note that we also generated a second set of synthetic circuit in a way that logic gates would get their inputs from previous level only. The reason for this was to run a few experiments on how the connectivity in the circuit could affect the performance of ZSIM.

4.3 Test circuits

In order to evaluate our SIMD algorithm, we used a combination of test circuits from standard benchmark circuits plus synthetic circuits. We took benchmarks available in BLIF(Berkeley Logic Interchange Format)[6]. For these circuits we used a parser to read the BLIF and then flattened the circuit to generate the internal netlist array used by the simulator. In this section, we first explore each test set, followed by figures showing the characteristic of each set.

4.3.1 IWLS Benchmark circuits

International Workshop on Logic and Synthesis (IWLS) [1] is a benchmark design set that contains the ITC'99 designs [94], the OpenCores designs [95], and a few other designs. We took various designs from IWLS benchmark circuits[7] to compare the performance of our simulator: (a) against some other multi-core simulators reported in the literature; (b) against a simulator from Xilinx. The circuit sizes used are fairly small, yet they are sufficient for us to draw conclusions from.

We took the designs in BLIF format and used a programme we developed in C to flatten and levelise the design. It reads in the BLIF netlist, parses the design, applies the flattening algorithm, levelises it, and generates a netlist in a format to be read in by our simulator.

BLIF format describes a logic gate level circuit in a textual format. Each BLIF file may contain several models, each of which is a flattened hierarchical circuit. Within this logic gate objects act as nodes representing logic function on signals in the model, and driving in turn other signals. A BLIF logic gate approximates to an N input, 1 output PLA description of a logic function. Listing 4.1 is an example of a simple design in BLIF format.

[6]BLIF [88] is used to represent combinational and sequential logic circuits in logic synthesis and verification tools such as Quartus.

[7]The reason for not using MCNC designs [87] was that the majority of circuits used in other published works were from IWLS 2005 benchmark suite.

```
.model simple
.inputs a b c d
.outputs z
.names a b c d z
1--0 1
-1-1 1
0-11 1
.end
```

Listing 4.1: An example of a BLIF netlist

Below is the sum of product equivalent in C notation for the logic gate description shown as a BLIF netlist in Listing 4.1:

$$z = (a\&\ d)|(b\&d)|(\ a\&c\&d) \tag{4.2}$$

The flattening algorithm decomposes each logic gate's description and converts it to a set of two input logic gates. The algorithm is based on binary trees. We create a binary tree of minimum possible depth for each logic gate. The time complexity of the flattening algorithm is $O(nlogn)$. Listing 4.2 is a C notation of the 4 input, 1 output PLA description of a logic function in Listing 4.1 after flattening.

```
z0 = ~d
z1 = a & z0
z2 = b & d
z3 = ~a
z4 = c & d
z5 = z3 & z4
z6 = z1 + z2
z7 = z6 + z5
```

Listing 4.2: The C notation for the logic gate description of a flattened netlist

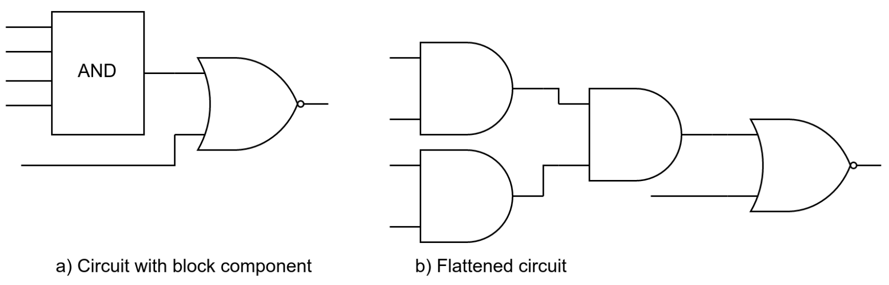


Figure 4.1: An example of circuit flattening

Figure 4.1 shows an example circuit with block component and a flattened version of it. Note that the programme then uses the levelisation algorithm described in section 4.4 to levelise the flattened circuit and generates a netlist.

4.3.2 Randomly Generated circuits

We used our programme `circ-gen` to generate sequential synthetic circuits in a range of sizes from 50 to 500 Million logic gates. Memory limitations of our experimental hardware meant that we were only able to test circuits of less than 200 Million gates.

We generated two sets of different synthetic circuits; `v1` and `v2`. The main difference between the two versions of the generated synthetic circuits is the way in which gates depend on previous logic levels. In `v1` circuits, the inputs to each logic gate can derived any of the previous logic levels and in `v2` circuits, the inputs to each gate come from the immediately prior level. The purpose of using these test circuits is to see how the read/write access pattern of the state vector affects performance. It can be anticipated that `v2` circuits will show more local access than `v1` circuits, and in consequence may have better cache performance. We used compile time a flag in `circ-gen` program to enable the generation of `v2` test circuits.

Figures 4.2, and 4.3, show the distribution of number of flip flops, and IO (input + output), versus the circuit size for BLIF file circuits and both types of synthetic ones. The synthetic circuits were generated in a way that was statistically to the real circuits. It can be observed from the plots that the different test sets overlap in their distributions.

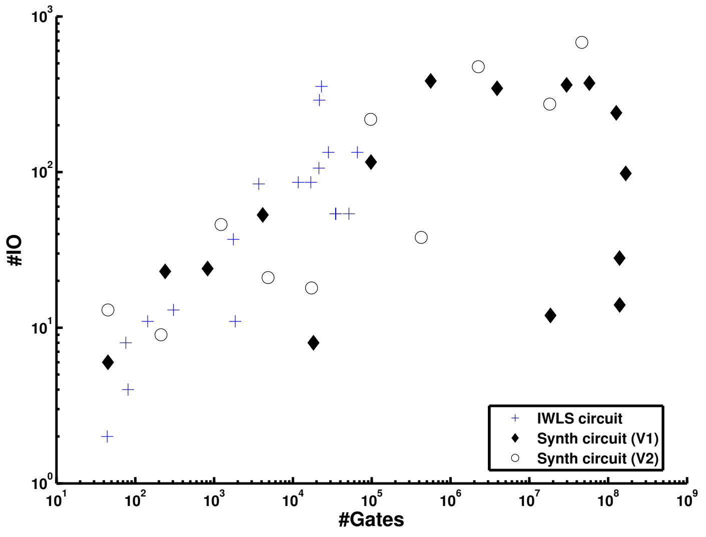


Figure 4.2: Distribution of the number of Input Output signals against the number of logic gates in the test circuit collection.

4.4 Circuit Structure and Representation

In this section, we discuss the levelisation algorithm, followed by the netlist format, and the circuit specification is being represented. We first, discuss the delay model we use and how we handle delay in our simulation.

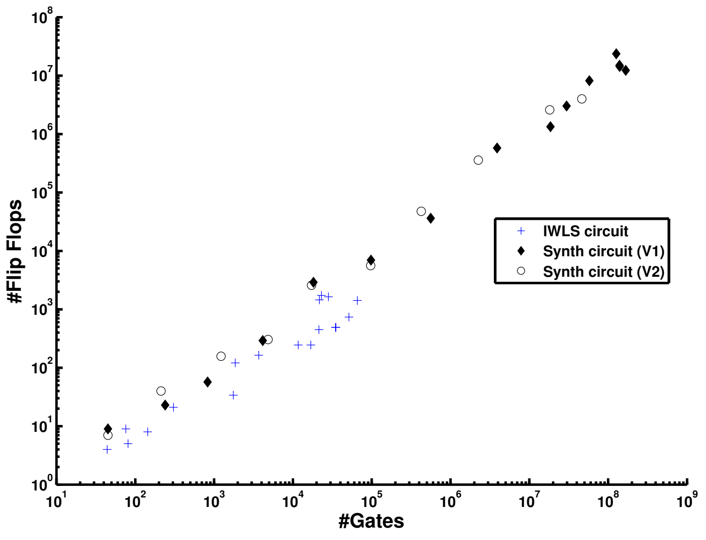


Figure 4.3: Distribution of the number of D type Flip Flops against the number of logic gates in the test collection.

4.4.1 Levels of logic

There are two general ways of simulating digital logic circuits : Cycle based and Event based. In Cycle based simulation, every logic gate in the circuit is simulated during each clock cycle. In reality, not every change on the input signal of a logic gate lead to a change on its output. As a result, it is clear that cycle based simulation leads to extra calculation. In event based simulation, we can reduce the number of computations during the simulation, by excluding those unnecessary calculations. So, a component is simulated, only if there was a change on its input that could possibly lead to a change on its output.

The implementation of any event based algorithm, requires the use of queues and involves some sort of locks. The standard way of implementing the an event based algorithm is to use an event queue that holds the events [5]. During the simulation, events are processed in a specific order. The related calculation/operation is done. Then the new event is inserted back into the queue. The sequential implementation uses priority queue data structure.

Parallelising event based algorithm that uses queue is not straight forward. In parallel, parallel threads will access the queue at the same time. It is possible that multiple simulation threads might try to access the same pending event on the queue. This will result in race condition. To avoid this, the queue insertion and removal operations must be done with mutual exclusion. Using locks (e.g: atomic operations) ensures safe operations [8].

If one wishes to maximise parallelism one should attempt to have a small ratio of lock operations to useful work done. This is a general principle which applies to any parallelisation. Since event queue operations will occur at least once per logic gate simulation, they are likely to be a serious obstacle on a many core machine. And in the context of SIMD operation, it is totally impractical to use locks. There are no SIMD lock instructions available on current Intel CPUs.

Although, there are many existing works on event based simulation, for example [77, 13] , we only focus on Cycle based simulation to take advantage of our SIMD architecture.

In Cycle base simulation there are two rules to consider: first, each logic gate should only be simulated once, and second, inputs to the logic gates should be ready and valid at the time of evaluation. In order to meet this requirements, logic gates in the circuit have to be simulated in a specific order. By giving a rank to each logic gate, we ensure a safe evaluation. In the literature the method is called Rank-ordered and was discussed in Chapter 2. Levelisation is based on the same concept. However, instead of word 'rank', the term 'level' is used. Levelising the circuit in a way that allows parallel simulation of independent components at the same time, was used in [36, 37, 38].

We restrict ourselves to simulating synchronous state machines and make the following further simplifying assumptions:

- All gates are two input, NOT is represented by a NAND with duplicate inputs, 3 input NANDs made up of pair of 2 input ones etc.
- All two input gates have same gate delay, t.

Working backwards from the rising edge of the system clock, the state latches can be affected either by external inputs that feed them directly or by logic gates. No change to an input to a logic gate occurring after a time $-t$ can affect an input to a latch, so it follows that when simulating there can be no dependencies in the last t of the machine cycle in the set of signals that either feed the latches or between the gates the generate signals that feed the latches. Thus all of these gates can in principle be simulated in parallel. Call this set of signals level N. Clearly we can, by induction, apply the same argument to the signals which feed these gates which we will call level $N-1$. Given a netlist we can levelise it as follows:

Step 1. form set of all signals feeding the latches or outputs.

Step 2. push gates whose outputs generate this set onto a stack

Step 3. form set of all signals feeding the set of gates on the top of the stack

Step 4. if this set is empty goto step 5 otherwise goto step 2

Step 5. set n=0

Step 6. pop the stack and label all gates with level n

Step 7. if stack empty terminate, otherwise set n=n+1 and goto step 6

By taking advantage of levelisation in cycle based simulation, we can perform parallel independent calculations of whole levels and only force synchronization between the parallel processors at the end of each level's simulation (Figure 4.4). The following section theoretically explains how the delay can be represented in circuits with non-unit delay gates. Note that this levelisation algorithm only works for circuits with unit-delay gates. By substituting a logic gate with non-unit delay with multiple gates with unit-delay, the simulator and the levelisation algorithm is still usable.

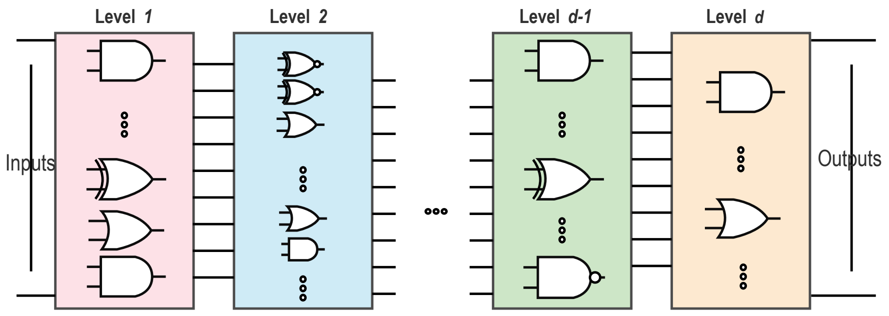


Figure 4.4: Levelisation example in a circuit, each of the coloured blocks can be simulated in parallel

4.4.2 Representing Delay

Depending on the how the synchronization of operations is done among the components in the circuits, digital circuits are divided into two synchronous and asynchronous circuits. In synchronous circuits, a periodic signal known as a clock, controls the operation of all the components in the circuits. In other words, all the calculations must be done at the right time within the duration of each clock cycle. When clock is active, it takes some time for a signal value to settle down and becomes valid. Otherwise, the value is not interpreted correctly during the simulation. Depending on the simulation goal (functional verification, timing verification, and etc), delays must be properly considered to avoid problems.

In asynchronous circuits, there is no single global clock to control the activity of the components. Delays are very important factor in simulating both asynchronous and synchronous circuits. However, we only discuss this matter for synchronous circuits.

In synchronous circuits, the maximum delay of the combinational logic determines the duration of the clock cycle, that is the maximum speed of the whole circuit. The maximum delay (critical path depth) is the longest path from the input to the output. In large circuits, every path in the circuit can have a different delay. When an input of a circuits changes, it takes some time for this change to propagates through the circuit and affects the output. This time is called the propagation delay. As the circuit grows larger, so does the propagation delay.

In larger circuits, an input could go through different paths to reach the output. Every path may have different delays. However, the propagation delay of a single input, is the delay time of a path with longest delay.

For fully characterizing a delay element, we associate a delay model to it. This model is characterized by a set of rules and parameters. These rules and parameters determine the value of the propagation delay in the circuit. They also indicate the accuracy of the model. The choice of the model affects the speed and accuracy of the simulation.

There are different forms of delay model to consider when simulating a circuit at gate level. Different delay models have different levels of computational complexity. The choice of model depends on the goal of the simulation. For example, timing simulation provides a more accurate, detailed, and complete verification of a design. So, a delay model associated with timing simulation has a higher level of complexity that contains a detail timing behaviour of circuit elements. Having a less complex delay model leads to less complicated calculations during the simulation. As a result, the simulation runs faster. For example, when the purpose of simulation is only to verify the functionality of a design, a simpler, less detailed delay model would be useful.

When dealing with circuits at gate level, delay models can be divided into two groups of 1) unit delay and 2) static delay [106]. The simplest form of delay model is where all the logic gates have the same timing behaviour regardless of their type, number of fan-ins,fan-outs, and etc.

The unit delay model considers a unit of time as a delay for each logic gate. Meaning, all the logic gates have the same unit of delay.

In contrast, logic gates have different delays in static delay models. In the simplest form of this delay model, the delay is associated with the type of logic gates in the circuit. However, it is possible to add more detailed timing behaviour to this system. For example, the delay could be defined as a function of load on its output. The difference between this model and the unit delay model is that the delay value can change from a logic gate to another.

Unit delay models have a much less complexity than static delays. Even with unit delay models, the degree of the precision in the simulation is not enough to obtain anything more than relative timings. The simulation is not accurate in terms of an objective time measure like nanoseconds, instead timing is given in terms of an abstract time unit of *gate delays* based on the assumption that all the logic gates in the circuit have the same timing behaviour. By replacing the path or distributed delay to unit delay, the calculation will become simpler. As a result, the simulation time will be reduced. When functionality of the design is more important than the timing correctness, we can simply use any of the above delay modes.

In a static delay model,there is delay associated with each element type in the circuit. In a sophisticated system the delay value can also be a function of the load on the output of the logic gate. Note that complexity makes the high precision timing simulation slow.

Normally, the exact size of delay is unknown in circuits. Delay in circuit may vary with temperature or power supply voltage. The input value on the logic gates can also affect the delay. For example, signal value transitioning from 1 to zero may take longer than zero to 1. Although, in principle delays may be expressed as real numbers, actual circuit delay times of components are typically specified to integer precision in nanoseconds, picoseconds, and etc. We can assume that the relative delays between circuit elements will be approximated as ratios of integers instead of real numbers [107].

In CMOS technology, an AND gate is constructed from a NAND gate and an inverter. So, the delay of an AND gate is sum of the delay of the NAND and the inverter. So, the delay ratio for a 2-input, 1-output AND gate is equal to 3:2:1. So, by adding an extra buffer logic on the output of a logic gate, we can simply add delay (Figure 4.5).

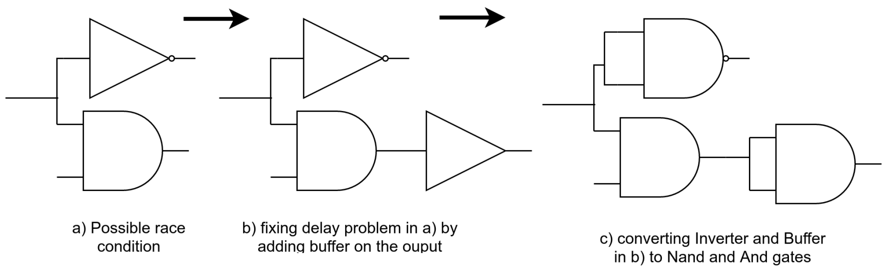

a) Possible race condition
b) fixing delay problem in a) by adding buffer on the ouput
c) converting Inverter and Buffer in b) to Nand and And gates

Figure 4.5: Transformation to two input race free logic

Simplified delay models (e.g: unit delay model) are widely used in research programs (e.g: studying path delay faults). It has also been used in the MCNC [87] benchmark suit library.

In this work, while we ignore the physical properties [8] of the gate, we also ignore the delay of interconnections and consider the delay for logic gates only. We consider unit delay model and assume that all the logic gates have the same delay. Factors such as load capacitance on input and output of the logic gates, and its type do not affect the delay of the element in the circuit.

Although, by simulating unit delay mode, we may encounter problems. However, we can add the extra delay by placing an extra buffer on the output of a logic gate in the circuit. The combination of using levelisation and unit delay in circuit simulation have been used in previous studies. Levelisation technique have been used by Wang et al. [29], Maurer et al. [30], and other authors [31, 5, 32, 33, 34, 35, 40]. Furthermore, Maurer et al. [30], Tang et al. [108], Ahmed et al. [109], and Gonsiorowski et al. [4] used unit-delay model in their simulation, while Chatterjee et al. [37], Kochte et al. [110], considered zero delay model.

[8]Changes in power supply voltage and temperature can affect the delay through the circuit.

4.4.3 Netlist Format

We transformed and processed circuits (IWLS and synthetic circuits) to generate a netlist that can be efficiently used in our parallel simulator. The final netlist format is a description of a circuits that is flattened and levelised and is referred to as the *gate array netlist*. The gate array netlist will then be fed as an input to the SIMD simulator, along with the input data file. Note that the IWLS benchmark suit is in BLIF format and in this thesis, the netlist is referred to as a *gate level netlist*. Figure 4.6 illustrates the simulation process.

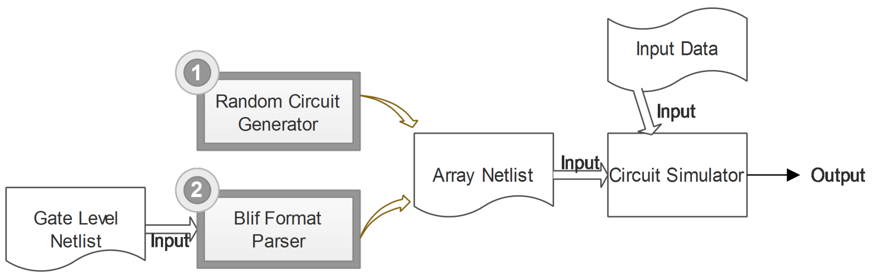


Figure 4.6: Process of transforming Netlists to gate Array Netlist for the Simulator

The gate array netlist is a vector description of the circuit that only contains integers. Each signal in the circuit is identified by its given ID. Listing 4.3 is the netlist format with a fixed order that allows it to be efficiently used by the simulator. The format is used in array netlist fed to the ZSIM simulator. In this listing, `gateNum` is number of logic gates in the circuit, `inputNum` is primary inputs, `outputNum` is primary outputs, `maxLevel` is the depth of the circuit, and `ndff` is the number of flip flops.

```
[gateNum inputNum outputNum maxLevel ndff]
[shape_vec] [comp] [inp0] [inp1] [ff_in] [ff_out] [outp]
```

Listing 4.3: Array netlist format structure

`shape_vec` is a vector that has the shape profile (distribution of logic gates throughout each level, i.e.: the number of components in each path depth). A shape profile for a circuit with D levels of logic can be represented as :

$$shape\_vec = [x_0, x_1, x_2..x_{D-1}] \tag{4.3}$$

, where x_0 is the number of inputs to the circuit, x_1 is the number of logic gates in level 1, and so on. In the shape profile, we do not specify a number of Flip flops for intermediate levels, as we do not consider multiple sub-circuits to be connected by latches. We only have one combinational circuit that is connected to a vector of flip flops some of whose outputs are fed back into the circuit or serve as the final outputs of the simulated module.

The vector `comp` is a list which specifies the types of the logic gates, the vectors `inp0`,`inp1`, and `outp` are the lists of left and right inputs and the output signals. In all cases the index position in the lists identifies the logic gate. The type of components in the circuit is, for simulated circuits, randomly chosen from one of the primary 2 input logic gates (AND, OR, NAND,NOR, XOR,XNOR). NOT is converted to a NAND gate where both inputs are the same.

`ff_in` and `ff_out`, contains indices that point to locations in the current state vector array. Those locations hold the signal ID that goes in/out Flip flops. Listing 4.4 is an example of a gate array netlist for the circuit shown in Figure 4.7:

```
6 3 1 4 2
3 2 2 2
0 0 0 3 0 4 3 2 5
0 0 0 1 2 3 4 0 6
0 0 0 10 9 4 9 5 10
...
```

Listing 4.4: An example of a gate array netlist

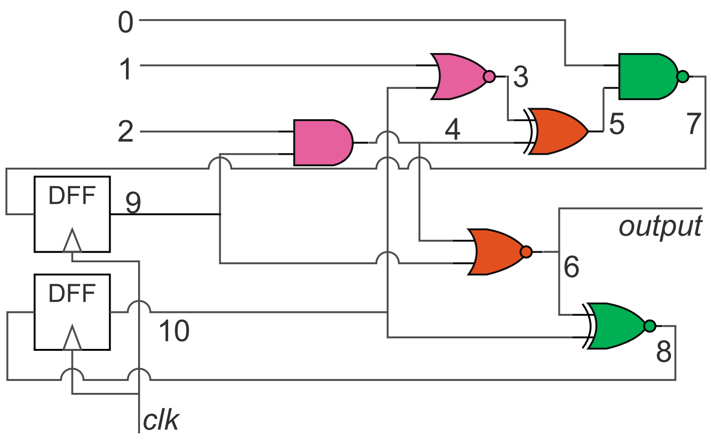


Figure 4.7: An example of a circuit with labels indicating the component ID. Logic gates of the same level are shown in the same color.

4.4.4 Circuit Representation

The gate array netlist is fed into the ZSIM and the information is held in a run-time data structure designed for fast run-time access. The data structure representing the circuit elements and its specification, uses several flat arrays of type integer. The information about the logic gate types is held in `comp` array. The values associated with the gate types are stored as integers. `inp0` and `inp1` arrays point to the source of input signals values to a logic gate in the `state` array. All the current values of signals are stored in the `state` array.

In previous chapter, vector arrays that are used to store circuit information were shown in Figure 3.6 and Figure 3.7 (see Page 53). Each array index is indexed by a logic gate ID. Logic gate `i`'s information is stored in `i_th` element of all 4 arrays. It's gate type is stored in `comp[i]`, and its output signal is held in `state[i]`. Its input values are stored in `state[inp0[i]]` and `state[inp1[i]]`. As previously mentioned, output signal values of logic gates in the same level are stored next to another in the `state` array.

Figure 4.8 is the schematic of data structure used for holding circuit specifications for the example circuit (Figure 4.7). Separate arrays hold data for input signals, logic gate types, and state array. Once the circuit is read in by the simulator, the state array is then updated by the input signals. The length of the arrays are almost the same, except that the state array is extended to keep the output signals derived from latches too. Chapter 3.2 discusses the SIMD simulator architecture in detail.

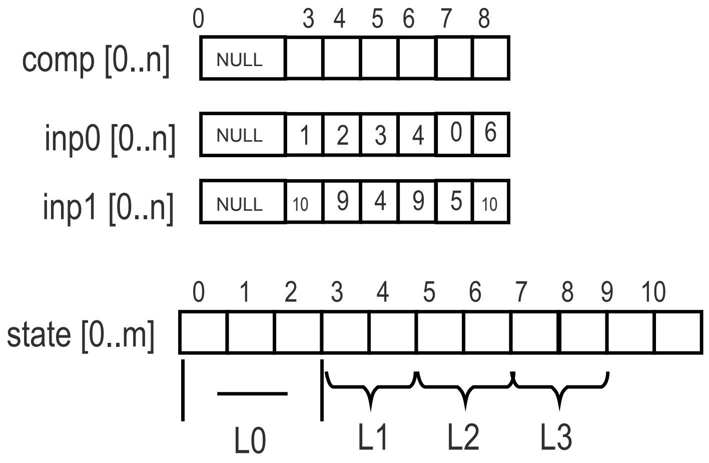


Figure 4.8: An example of the array data structure used in the SIMD simulator.In practical examples the vectors would be much longer.

Chapter 5

Experimental Data

The aim of the experiments was to:

- Verify that the data structures used allow SIMD acceleration, particularly on machines with gather instructions (section 5.3.1).
- Verify that, on sufficiently large circuits, substantial gains could be made from multi-core parallelism (section 5.3.2).
- Show that a simulator using this approach out performed an existing commercial simulator on a standard workstation (section 5.3.3).
- Show that the performance on a cheap Xeon Phi card is competitive with results reported elsewhere on much more expensive super-computers (section 5.3.5).

ZSIM is a cycle based simulator that is compared with several other cycle based as well as event based simulators. ZSIM targets various architectures including Xeon Phi, AMD, and Xeon. Other simulators that are compared with are mainly targeting GPUs. To be able to compare ZSIM with other previous works, the same set of circuit suites were used. For the comparisons to be fair, we used number of event transitions per second as the main metric and not the total execution time. Note that some of these works are quite old. However, these are the most recent works that could be used for comparisons on.

It is always possible to derive from a cycle based simulator the number of events that an event based simulator would have generated when give the same problem. One conditionally compiles in monitoring code that counts the number of actual changes in signal values that occur during the cycle based simulation. When running performance tests this monitoring code is elided but a run using the monitoring code gives the total number of events that would have occurred in an event based simulator on the same circuit with the same inputs. This allows a fair comparison of the two types of simulator using a common metric.

5.1 Test sets

To evaluate the ZSIM[1] , two types of test circuits were used:

1. Circuits from the IWLS benchmark suit [1] which allow direct comparison with other published studies of parallel simulators.

2. Circuits generated by a parametrised circuit synthesizer. The synthesizer used an algorithm that has been shown [2] to generate circuits that are statistically representative of real logic circuits. The synthesizer allowed testing of a range of very large circuits, larger than the ones for which it was possible to obtain open source files.

From the IWLS circuit benchmark suite, circuit designs in BLIF(Berkeley Logic Interchange Format)[2]formats were used. A parser was used to flatten, levelise the circuit designs, and generate a gate level netlist array. The netlist array is then fed to the simulator as a test circuit. Description of both parser and synthetic circuit generator was explained in Chapter 4. Note that for experimental purposes, `circ-gen` also generates a verilog netlist for each synthetic circuits. Figure 5.1 illustrates the experimental setup including the process of generating the test circuits and the simulation itself. The SIMD algorithm was implemented in both Pascal and C++ and ZSIM was compiled with three different compilers (Intel C, Gcc, Vector Pascal).

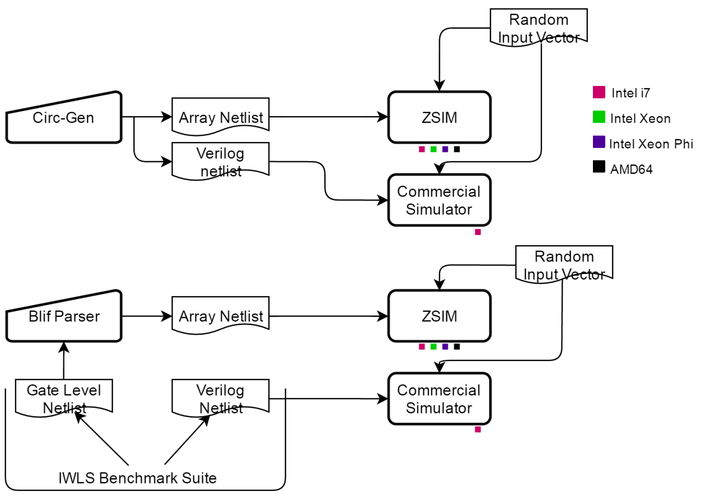


Figure 5.1: Illustration of the experimental setup

[1]my simulator
[2]BLIF [88] is used to represent combinational and sequential logic circuits in logic synthesis and verification tools such as Quartus.

5.2 Experimental Setup

An Intel Xeon, an Xeon Phi coprocessor, and AMD64 were used as a primary platforms, to evaluate performance of the parallel SIMD simulators. To assess the performance, the circuit simulator was ran on different architectures for a varying number of cores over different sizes of circuits.

The Intel Xeon Phi 5110S coprocessor with 60 cores, each operating at 1.053 GHz, an Intel Xeon E5-2620 processor operating at 2 GHz, Intel core i7-2630QM, and AMD Opteron 6366HE (a NUMA machine) were used. The AMD machine has four memory banks, but none of our circuit designs were large enough to require non uniform access memory; they would fit on 1 memory bank. So the other three memory banks on the AMD machine were left unused.

The multicore CPU performance for the Intel Xeon processor was measured on a server with 16 GB of 667 MHz DDR3 memory, based on a two-socket Intel Xeon E5-2620 (Ivy Bridge) CPU for servers. The thermal design power (TDP) of each CPU socket is 95 W. A E5-2620 socket has 6 physical cores clocked at 2 GHz (turbo frequency of 2.5 GHz) with two-way hyper-threading. The vector units of the system support the AVX instruction set with 256-bit vector registers.

The host operating system is SUSE Linux Enterprise Server 11 with kernel version 3.0.76-0.11-default. The simulator program was compiled with the Intel C++ compiler version 15.0.3.

The Xeon Phi coprocessor performance was measured on the same system as the CPU performance. The system contains one Xeon Phi coprocessor of the 5110P series with 60 cores at 1.053 MHz, and 8 GB of GDDR5 RAM at 2.75 GHz. The driver stack is MPSS version 3.1.2. The Xeon Phi itself is running Linux with a memory mapped root file system. Table 5.1 shows the detail specification for the architectures.

In our experiments, input signals are randomly generated for each circuit. We used a high volume of randomly generated input signals, one every clock cycle. We could either generate all the input vectors for all cycles and use them during the simulation. Alternatively we could repeatedly call a function to generate random inputs during each simulation cycle. We chose the second approach and measured the total time it takes to generate input values. The time difference was not significant. Note that the time it takes to generate the random numbers is not a critical aspect in the performance of our simulator. This time can be ignored over many simulation cycles. Using random inputs in simulation was also used in previous studies [36, 11, 4] as an alternative to an officially released stimuli for some standard benchmarks [10].

Throughout the experiments, we focus on the wall-clock time, that measures the total physical elapsed time. Note that in order to measure the elapsed time, the experiments were done several times. The reported timings are in seconds with an uncertainty of 0.01. Results shown throughout this chapter show the timings and averages (rounded up) up to 2 decimal points. During the measurements, ZSIM was configured to simulate without monitoring values of internal nets. We also used other metrics such as event rate. A software counter was used to measure the number of gate transitions. A simple XOR operation between the monitored current `state` array and previous signal values determines if any change has occurred. The counter can be disabled for timing. The metric regardless of how many cores are used, would give us the number of logic events that can be computed per second. Due to the fact that the simulation is deterministic, event counts will be consistent. As each event represent a logic gate transition, the event rate metric can indicate the total simulation speed. The gate per second rate also reflect the magnitude of logic events the simulator is able to compute (on a target platform).

To control for variations in compilers two implementations of the simulator were tested, in C++ and Vector Pascal [111]. Three compilers were also used, `gcc`, `icpc`, and `vpc`.

In order to take advantage of Xeon Phi cores and its wide SIMD registers, we draw advantage from automatic parallelision feature of Vector Pascal. The Vector Pascal compiler generated codes specifically for Xeon Phi coprocessor using `-cpuMIC` flag, and the AMD Opteron machine using `-cpuopteron`.

For `icpc` compiler, the auto-vectorization was enabled using `-O2` flag. Furthermore, to ensure the auto-vectorization was disabled, `-O0` or `-no-vec` was used. To enable auto-vectorization using `gcc` compiler, `-O3` was used.

The parallel simulator written in C++ targeting Xeon Phi was executed natively[3] on MIC. The executables were copied to the Xeon Phi, and the connection to the device via Xeon host is done via `ssh` command. Below is an example of the commands used to copy and connect to a Phi coprocessor (called `mic0`)

The OpenMP programming model that is supported by Intel compiler was used to parallelized the `C++` programs. In order to execute the OpenMP applications on Xeon Phi, the `libiomp5.so` was copied onto the device along with the executables. Note that prior to the execution of the files, the path to the library was set via below command:

```
export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
```

To compile the OpenMP programs, `-openmp` flag was used with `icpc`, and `-fopenmp` was used with `gcc` compiler. Below are the commands used to compile the OpenMP programs:

[3]Offload data transfers usually keep the OS busy. If running the application natively, N cores can be used. On offload mode, N-1 cores can be used as one core is busy for data movement.

```
icpc -mmic -openmp -O2 <filename>
g++ -O2 -fopenmp <filename>
```

Furthermore, the simulator was also compared to a commercial simulator (Xilinx) executing on an Intel i7 machine, also shown on Table 5.1. Note that all the benchmark circuits including synthetic ones are synchronous designs with latches driven by a single clock, hence the commercial simulator and ZSIM both worked on a single clock boundary.

Table 5.1: Specification of Processors Used in the Experiments

Parameter	Intel Xeon Phi	Intel Xeon	AMD Opteron	Intel i7
	Coprocessor 5110P	Processor E5-2620	6366HE	2630QM
Core, Threads	60, 240	6, 12	32, 64	4, 8
Clock Speed	1.053 GHz	2 GHz	1.8 GHz	2 GHz
Memory Capacity	8 GB	16 GB per socket	-	6 GB
Memory Technology	GDDR5	DDR3	DDR3	DDR3
Memory Speed	2.75 GHz (5.5 GT/s)	667 MHz (1333 MT/s)	1333 MHz	
Memory Channels	16	4 per socket	4	2
Memory Data Width	32 bits	64 bits	64 bits (S)	64 bits
Peak Memory Bandwidth	320 GB/s	42.6 GB/s per socket	51.2 GB/s	21.3 GB/s
Vector Length	512 Bits (Intel IMCI)	256 Bits (Intel AVX)	256 Bits (Intel AVX)	256 Bits (Intel AVX)
Data Caches	32 KB L1,	32 KB L1,	16 x 16 KB L1 (S),	32 KB L1 per core
	512 KB L2 per core	256 KB per core,	8 x 2 MB L2 (S), 1024KB	
		15 MB L3 per socket	2 x 6 MB L3 (S)	6144 KB

5.2.1 Platforms

Due to problems such as energy consumption and heat dissipation, gaining extra computational power from architectures based on a single processing unit (CPU), by solely relying on the increase of the CPU clock frequencies, have reached its limit around 2003 [112]. Reducing the size of the transistors and fitting more of them on a single chip to gain more computational power in Microprocessors was challenging. This problem is due to the fundamental limitations in semiconductor device physics that restricts the number of tasks being done in each clock period. The solution adopted by Intel and other firms was to switch to parallel processing models, multicores and many cores, either through systems with multiple processors or having multiple cores on single microprocessor [113]. An example of many core systems are highly parallel graphic processing units (GPU) with as many as hundreds of cores that have throughput oriented architecture. Processors as such that are derived from consumer products are widely accessible at an inexpensive prices. However, they require the software developers to rearrange some of the algorithm's data structure, so that it would benefit from the parallel architecture of the system [114].

5.3 Results

In order to validate the effectiveness of the SIMD ZSIM, the simulator was run over 1000 clock cycles on three main platforms of Intel Xeon, Intel Xeon Phi, and AMD64. The results are compared to some of the published work in the area (Section 5.3.4 and 5.3.5). Further experiments were done to verify the performance of the ZSIM across three compilers; `gcc, Intel C, Vector Pascal`. Unless specifically stated in graphs or tables, results are from the compilation of ZSIM with Intel C.

Note that for the AMD, Xeon and Intel i7, runs were done with numbers of threads increasing by powers of 2 in order to provide data that will fit nicely on a log plot. In consequence for the Xeon and i7 measurements were taken for 1, 2, 4, 8 threads but not 16 as this would have exceeded the number of hyper-threads supported. For the AMD the maximum number of threads tested was 64. For the Xeon Phi the sequence was 1, 5, 25, 125, 240. Full data can be retrieved from the university document repository at doi:10.5525/gla.researchdata.342.

5.3.1 SIMD acceleration

In order to show the effect of vectorization on the performance of the sequential simulator, the ZSIM simulator was run on one core with and without SIMD. In this section, no parallel directive was used. The program is compiled with and without vectorization flag being enabled. Section 5.3.2 will discuss the performance of ZSIM when both SIMD and multicore parallelism were used. For that purpose, OpenMP pragmas were used to create both SIMD and multicore loops. However, SIMD acceleration using single core means taking advantage of SIMD for a sequential code.

Auto vectorization was enabled by compiling the simulation program with -O2. The compiler directive (**#pragma simd**) was used to enforce SIMD vectorization for the `for loop` in the simulation algorithm. SIMD directives give permission to the compiler to vectorize the loop. Note that this **for loop**, loops over the logic gates in the same level.

The results were compared against the non optimized version of the simulator compiled with the -O0 -no-vec flag. The -O0 flag turns off the auto vectorization and any other optimizations. The -no-vec disables any possible SIMD vectorization (in case there was a **#pragma simd** in the program). We measured ZSIM's performance for both synthetic and standard circuits. Some of the experiments were done on two sets of synthetic circuits (V1 [4],V2 [5]). The standard circuits are from IWLS suite.

[4]V1: Synthetic circuits with inputs from any level

[5]V2: synthetic circuits with inputs from previous level only

Table 5.2, shows the ZSIM's vectorization performance for synthetic circuits V1 across different architectures. Using single core with SIMD acceleration enabled, ZSIM's peak performance on Xeon Phi is 10.6, then it reduces to around 1.3. On Intel Xeon, ZSIM achieved speedup of upto 4 for the circuit size of around 10^5 . Then, the performance reduces to 1.7 (Table 5.2). On Intel i7, the maximum achieved speedup is 14.3.

Since the bit-vector length in Xeon Phi is 512, 256 on Xeon and Intel i7, and the data is stored as array of integers, the expected potential speedup enabling SIMD acceleration: on Intel Xeon Phi is 16 (512/32 bit), and 8 (256/32 bit) on Intel Xeon and Intel i7. Meaning 16 logic gates on Xeon Phi and 8 logic gates on Xeon and Intel i7 can be evaluated at the same time. However, the achieved speedup was 10.6 on Xeon Phi, and 4 on Xeon. Note that the speedup ZSIM achieved on Intel i7 is more than what we expected. It could be related to how Intel Compiler optimizes the code on that specific architecture.

The reason for not achieving the desired speedup on both Intel Xeon and Xeon Phi is due to the access patterns. When simulating logic gates of the same level, their inputs may be coming from any other previous levels. Therefore, when accessing the `state` array to retrieve the data, we may have to read the data from various part of the memory. Non-contiguous memory access (read and write) can affect the performance. It is possible that the data that was already loaded in the cache for previous evaluation, is not needed for current calculations. When the required data is in the different cache line, loading the data into cache degrades the performance.

Table 5.2: Single core SIMD performance gain against single core nonSIMD across difference architectures (Synthetic circuits V1)

Circuit Size (gates)	Intel Xeon Phi	Intel Xeon	Intel i7
45	1.4	1.5	1.9
240	2.7	2.8	8.6
828	6.4	3.7	8.3
4140	10.6	3.9	14.3
18274	10.2	4.0	8.1
97814	4.1	3.5	6.3
560251	1.7	2.7	2.1
3892127	1.4	1.9	2.0
18561032	1.4	2.0	2.0
29651508	1.7	2.1	2.0
57765590	1.6	1.9	-
126351272	1.4	1.8	-
139352764	1.5	1.8	-
140083890	1.3	1.7	-
166599511	1.4	1.7	-

In order to further explain this poor performance, the same experiments and tests were done on another set of synthetic circuits. This set of circuits are not ideal, and were only generated to test the impact of read and write access pattern on the performance of ZSIM.

Table 5.3, shows the vectorization performance for synthetic circuits V2 across different architectures. It shows that ZSIM achieved the same speedup of 4.9 on Intel Xeon for the second test circuits. In other words, changing the read and write access pattern did not have a noticeable impact on the peak performance on Intel Xeon. However, the experiments showed that non-contiguous memory access can affect the peak performance hugely. When simulating synthetic circuits with inputs from previous level only, ZSIM achieves the performance with the peak value of 22.3. Using the second set of synthetic circuit, Intel i7 shows almost the same performance as Intel Xeon. As explained before, the reason that ZSIM did not show the expected speedup for synthetic circuits with inputs from any level, was due to cache locality and the need to gather data from various cache lines.

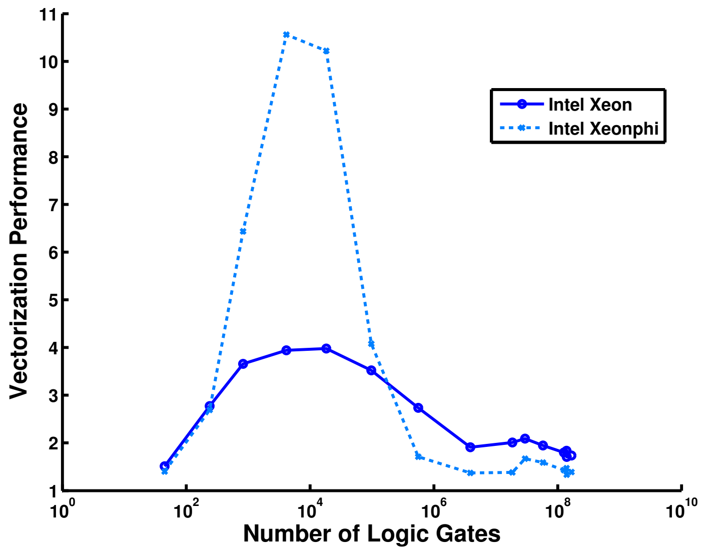
Table 5.3: Single core SIMD performance gain against single core nonSIMD across difference architectures (Synthetic circuits V2)

Circuit Size (gates)	Intel Xeon Phi	Intel Xeon	Intel i7
45	4.1	1.7	1.6
212	9.0	4.2	1.7
1225	13.7	2.4	4.3
4836	21.4	3.8	4.2
17229	22.3	4.9	4.4
96948	16.4	4.2	4.1
425881	19.2	4.1	4.5
2253903	16.4	3.9	3.5
18130360	3.7	3.0	2.5
46256306	6.0	3.2	2.4

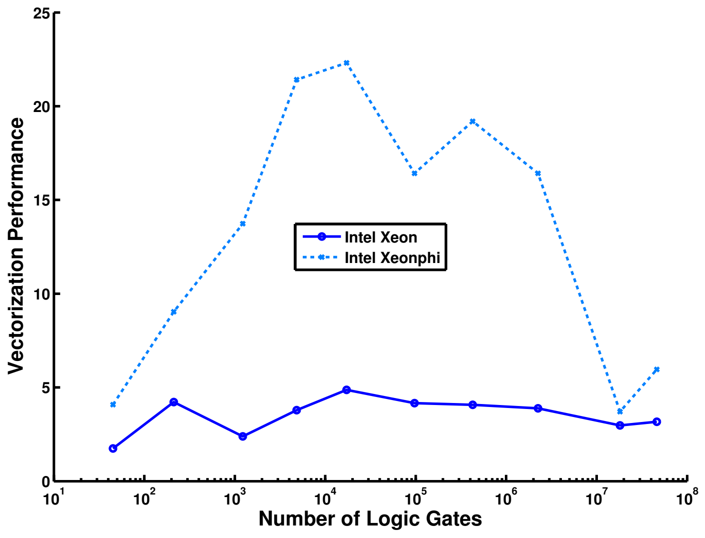
The two plots in Figure 5.2, depicts the vectorization performance of both synthetic circuits (V1 and V2) on Intel Xeon and Xeon Phi.

Figure 5.3, shows the vectorization performance for synthetic circuits V1; with inputs come from any previous levels. The plot compares the speedup of the ZSIM on single core Xeon, Xeon phi (left plot) and on multicore SIMD (right plot)[6]. The plot on the left (Figure 5.3), shows the performance gradually decreases and levels off at 1M logic gates. When the circuit does not fit into the cache, the performance degrades. Note that each logic gate occupies about 4 integers (Figure 4.8). That makes about 16 bytes. With 512 Kbytes L2 cache per core, the maximum size of the circuit that fits the cache on a single core is about 32k gates.

[6]See full data at doi:10.5525/gla.researchdata.342



(a) Synthetic circuits with inputs from any level



(b) Synthetic circuits with inputs from previous level only

Figure 5.2: Comparison of performance gain for single core SIMD against single core non-SIMD on Intel Xeon Phi and Xeon

Although smaller circuits fit into the cache and they benefit from vectorization, they do not benefit from larger number tasks.

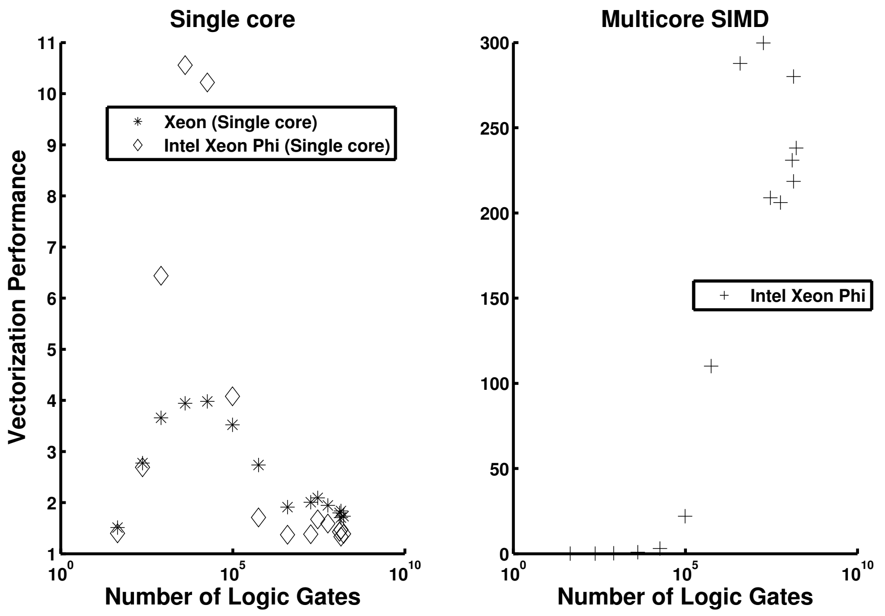


Figure 5.3: Comparison of performance gain for single core SIMD against single core non-SIMD on both Intel Xeon Phi and Xeon, and multicore SIMD against nonSIMD on Intel Xeon Phi (240 threads) for synthetic circuits V1

Table 5.4, shows the ZSIM's vectorization performance for circuits from IWLS suite across different architectures. Using single core with SIMD acceleration enabled, ZSIM's vectorization performance on Xeon Phi is 1.33 to 17.71, with the average of 9.54. On Intel Xeon, ZSIM achieved the average vectorization speedup of 3.47. On Intel i7, the average number is almost the same on Intel Xeon (average of 3.84) as it has the same bit-vector length of 256 as Intel Xeon. ZSIM's performance on Intel i7 fluctuated between 0.89 to 6.58. In some cases, Intel i7 showed a better performance comparing to Intel Xeon. We've seen the same behaviour in Table 5.2 on synthetic circuit V1. As stated before, this could be related to the compiler optimization on that specific architecture. Plot in Figure 5.4 shows the stated vectorization performance results. Note that these circuits have 44 to 65K logic gates.

The log/log plot in Figure 5.5, shows the achieved gate transitions per second on single core for circuits from IWLS suite. The effect of SIMD optimization on both Intel Xeon and Xeon Phi is clearly evident on the plots. However, as the size of these circuits are fairly small, Intel Xeon Phi does not perform as well as Intel Xeon. Although we achieved a higher speedup on Intel Xeon Phi with vectorization, the ZSIM's execution time is still slower than Intel Xeon.

The log/log plot in Figure 5.6, shows the ZSIM's execution time comparison for the two versions of synthetic circuits on Intel Xeon and Intel Xeon Phi, running on single cores of the two machines.

Table 5.4: Single core SIMD performance gain against single core nonSIMD across difference architectures for circuits from IWLS suite

Circuit Size (gates)	Xeon Phi	Xeon	Intel i7
44	1.33	3.09	2.68
76	1.58	3.17	4.65
81	1.59	3.21	0.89
305	2.69	2.94	5.66
1753	5.74	3.14	3.49
1856	7.81	2.77	6.58
3686	10.30	3.76	2.54
11709	12.38	3.94	4.40
16881	13.05	3.78	3.28
21431	15.06	3.51	4.97
21677	13.13	3.87	2.89
23070	17.71	4.03	4.77
28155	13.35	3.95	2.62
34697	12.37	3.07	4.21
35195	12.09	3.84	3.76
51513	10.90	3.79	3.69
65661	11.09	3.05	4.26

Machine	Speedup (Avg)
Xeon Phi	9.54
Xeon	3.47
Intel i7	3.84

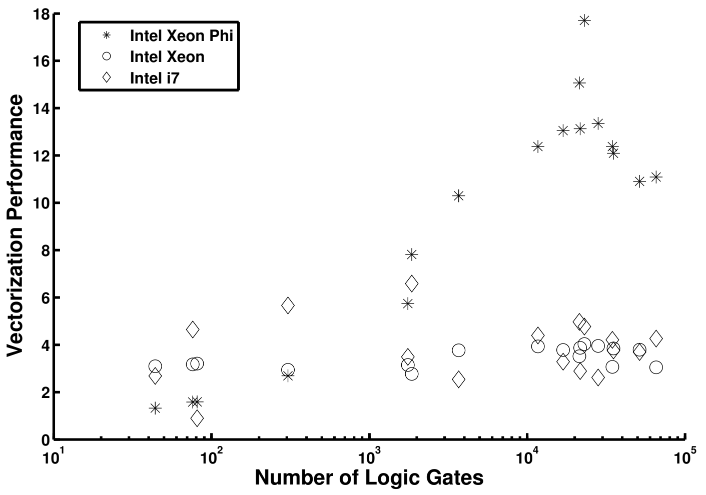


Figure 5.4: Comparison of performance gain for single core SIMD against single core non-SIMD across different architectures for circuits from IWLS suite

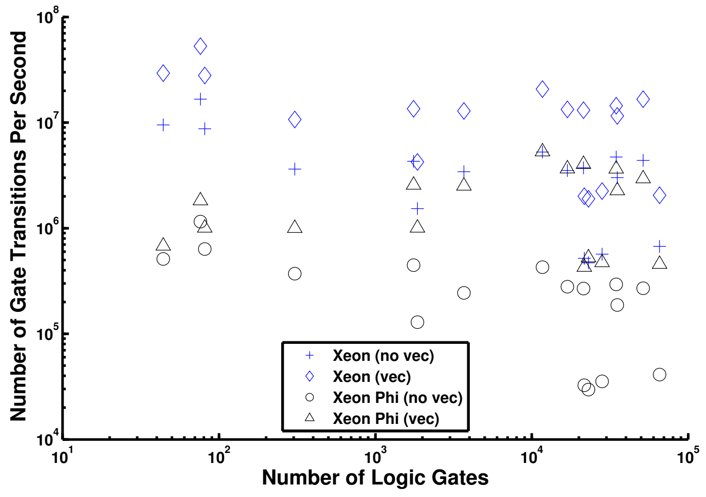


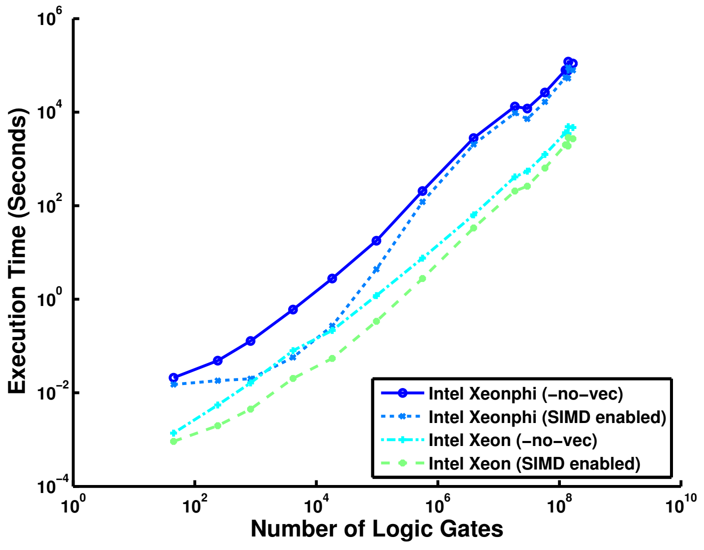
Figure 5.5: Number of gate transitions per second of single core sequential code (SIMD and nonSIMD) on Intel Xeon and Xeon Phi for circuits from IWLS suite

Note that with or without vectorization (on any of the test circuits), the single core ZSIM simulator runs faster on the Xeon than the Intel Xeon Phi (due to the faster clock rate on Intel Xeon). However, the purpose of this experiment was only to show how much improvement the SIMD vectorization would give. Furthermore, in this section, by comparing the test circuits version 1 and 2, it became clear that how the change in the read/write access pattern, can affect the performance.

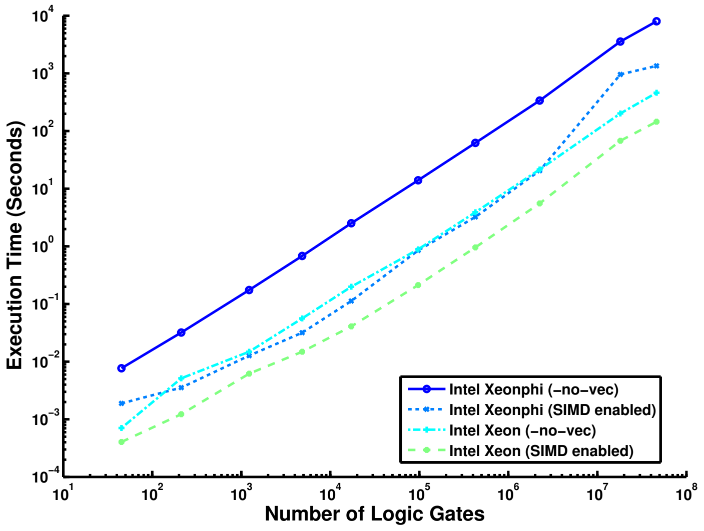
5.3.2 Multi-core acceleration

The log/log plots in Figures (5.7, 5.8), show the effect of multicore parallelism. On Intel Xeon Phi, as we increase the number of threads (from 1 to 240), we clearly see improvements on larger circuits. From the circuit size of 3 millions logic gate, the simulator gains speedup using 240 threads.The larger synthetic circuit (version 1) that was used in these experiment has around 160 millions of logic gates. For this circuit size, ZSIM achieved the speedup of 10 using 240 threads on Intel Xeon phi, in comparison to the baseline (ZSIM on Intel Xeon). This number increases to 299.8, if compared to the sequential version on Intel Xeon Phi.The speedup for synthetic circuits (version 2) was roughly twice as great - presumably because of better cache use.

Table 5.5, reports the vectorization performance across difference architectures using multicores for synthetic circuits V1. Using multicore with SIMD enabled, ZSIM's maximum vectorization performance on Xeon is 11.1, while the performance increases upto 299.8 on Intel Xeon Phi. As stated in the previous section, the expected potential speedup enabling SIMD acceleration: on Intel Xeon Phi is 16, and 8 on Intel Xeon. Using multicore acceleration along with enabling SIMD, it is expected to achieve the maximum performance of on

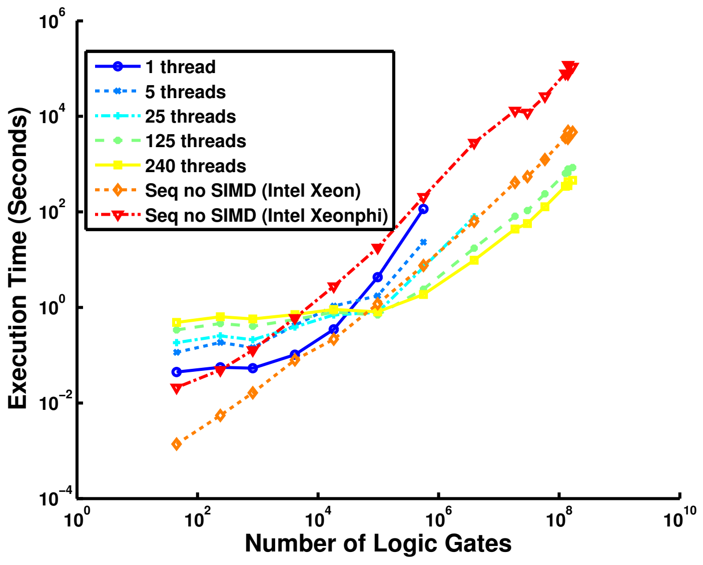


(a) Synthetic circuits with inputs from any level

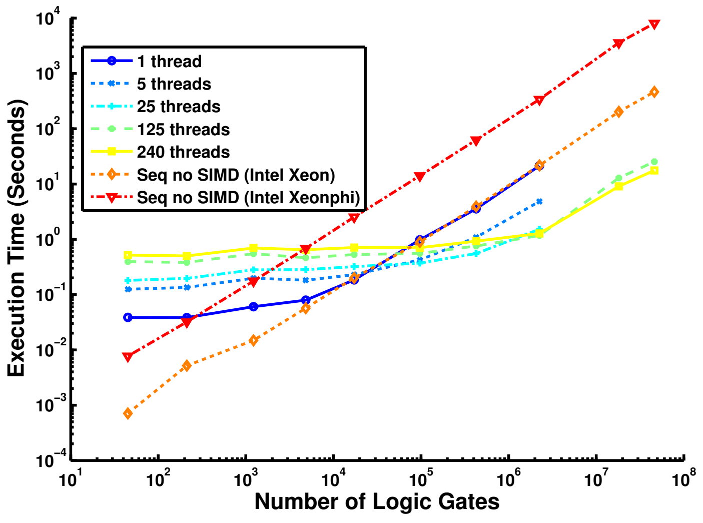


(b) Synthetic circuits with inputs from previous level only

Figure 5.6: Performance comparison of single core SIMD and nonSIMD on Intel Xeon Phi and Xeon

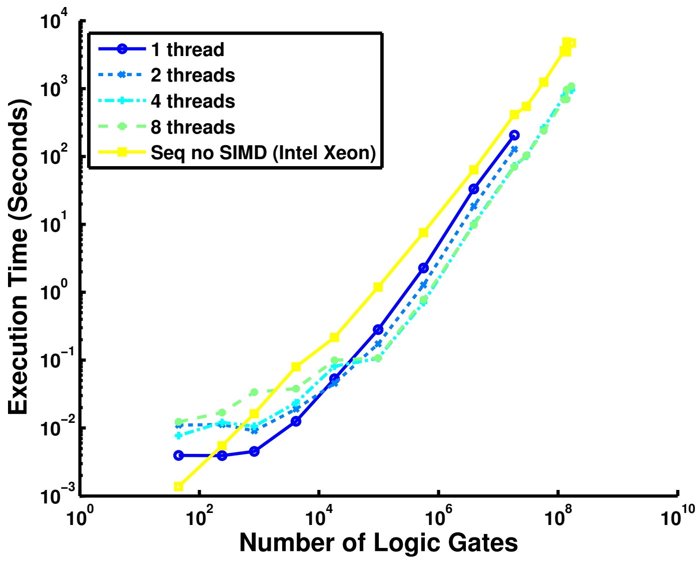


(a) Synthetic circuits with inputs from any level

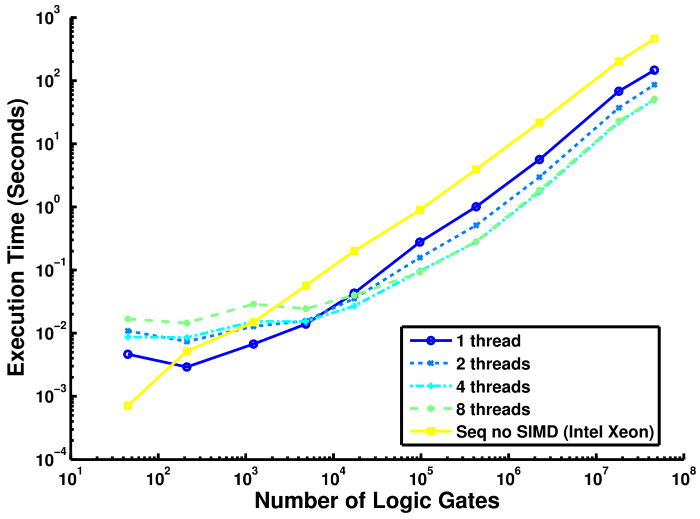


(b) Synthetic circuits with inputs from previous level only

Figure 5.7: Performance comparison of multicore SIMD and single core nonSIMD on Intel Xeon Phi



(a) Synthetic circuits with inputs from any level



(b) Synthetic circuits with inputs from previous level only

Figure 5.8: Performance comparison of multicore SIMD on Intel i7 and single core non-SIMD on Intel Xeon

Intel Xeon Phi $16*60 = 960$ and $8*6 = 48$ on Intel Xeon. Note that 60 and 6 are the number of cores on Xeon Phi and Xeon. What we are calculating is the throughput of the hardware, the number of operations it can perform per cycle, hyper-threading does not increase max number of operations per cycle. It only allows a more balanced workload between load operations and compute operations, to reduce the load latency.

Figure 5.3 (left plot), shows the peak performance of 10 on Intel Xeon Phi and 4 on Intel Xeon. With regards to this numbers, enabling multicore SIMD acceleration, it is expected to gain the maximum performance of 600 on Intel Xeon Phi and 24 on Intel Xeon.

In Figure 5.3 (right plot) that shows the speedup of multicore SIMD on Intel Xeon Phi for synthetic circuits with inputs from any level, the performance peaks at around 300 and not 600 as expected. Table 5.5, shows the maximum performance of 11 on Intel Xeon instead of 24.

When using multiple threads, not all the resources (threads) are always available and free to do the simulation. Furthermore, as the circuit grows larger, there will be memory contention (when cores trying to access part of memory that is not accessible). Table 5.5, shows that although smaller circuit sizes fit into cache, using multiple cores, do not lead to any performance gain. The reason is that there is not enough work to keep the cores busy. As a result of that, the overhead degrades the performance.

Table 5.5: Multicore SIMD performance gain against single core nonSIMD across difference architectures (Synthetic circuits V1)

Circuit Size (gates)	Intel Xeon	Intel Xeon Phi	Intel i7
	(8 threads)	(240 threads)	(8 threads)
45	0.1	0.0	0.06
240	0.3	0.1	0.53
828	0.5	0.2	1.99
4140	2.1	0.8	5.25
18274	2.2	3.1	5.99
97814	11.1	22.0	19.02
560251	9.8	110.1	5.83
3892127	6.5	287.7	5.02
18561032	5.9	299.8	2.78
29651508	5.3	208.9	-
57765590	5.2	206.1	-
126351272	5.2	230.9	-
139352764	5.0	218.5	-
140083890	5.1	280.0	-
166599511	4.3	238.1	-

Using single core, ZSIM achieved the maximum speedup of 22.3 on Xeon Phi, 4.9 on Xeon, and 4.5 on Intel i7 for synthetic circuits V2. When using multiple cores in addition to SIMD acceleration, the expected speedup with respect to these numbers is 1320 on Xeon Phi (22*60), 30 on Xeon (5*6), and 18 on i7 (4.5*4) (explained the reason before). Note that in these calculations, the numbers 60, 6, 4 are the number of cores on each machine. Table 5.6, shows when synthetic circuits with inputs from previous level only are used, the performance gain is around 460.1 on Intel Xeon Phi, 13.9 on Intel Xeon, and 15.71 on Intel i7. On none of the machines, have we achieved the theoretical peak performance. However, the speedup performance when using the synthetic circuits V2 is higher than when testing synthetic circuits V1 (the difference between the two synthetic circuits V1 and V2 was previously explained). Comparing to synthetic circuits V1, the increased speedup performance is due to the change in write/read pattern. Most likely, a lesser number of gather instruction cycles is needed to collect the required data during the simulation.

Table 5.6: Multicore SIMD performance gain against single core nonSIMD across difference architectures (Synthetic circuits V2)

Circuit Size (gates)	Intel Xeon (8 threads)	Intel Xeon Phi (240 threads)	Intel i7 (8 threads)
45	0.0	0.0	0.10
212	0.4	0.1	0.36
1225	0.5	0.3	1.46
4836	2.3	1.1	5.49
17229	5.0	3.6	8.79
96948	9.6	20.0	15.71
425881	13.9	67.7	10.50
2253903	11.8	266.5	4.87
18130360	8.8	394.6	2.96
46256306	9.1	460.1	2.78

Note that Tables 5.6 and 5.5, only show the speedup when the maximum number of threads is used. ZSIM was tested for various circuit sizes on Intel Xeon for various number of threads. For smaller circuits, as the number of threads are increasing, the performance gain reduces. The reason is that there is not enough work keep the threads busy. The overhead of spawning threads reduces the performance. The performance is achieved when using larger circuit sizes. Table 5.7 reports the number of threads and vectorization performance using multicore for synthetic circuits V1 (10k to 560K logic gates) on Intel Xeon Phi. The table shows for larger circuits (from the point where the circuit does not fit into the cache), the performance increases as the number of threads increases. When the circuits are large, there is enough work to keep the cores busy and hide latency.

Note that Table 5.7 shows the SIMD vectorization performance on Intel Xeon Phi for three different circuit size (See full data at doi:10.5525/gla.researchdata.342). The purpose of this table is to show that using multiple threads on Intel Xeon Phi, ZSIM starts gaining speedup performance for circuits of larger than 100,000 gates. Using 240 threads does not improve the speedup on circuit of size 97814. However, from 125 threads to 240, ZSIM gains significant performance on circuit of size 560251

Table 5.7: Multicore SIMD performance gain against single core nonSIMD on Intel Xeon Phi (Synthetic circuits V1)

Circuit Size (gates)	Threads	Intel Xeon Phi
18274	1	7.9
	5	2.6
	25	4.0
	125	3.5
	240	3.1
97814	1	4.2
	5	10.2
	25	22.6
	125	25.8
	240	22.0
560251	1	1.8
	5	8.8
	25	29.8
	125	85.6
	240	110.1

The plot in Figure 5.9 shows the performance gain of multicore SIMD ZSIM for standard circuits (IWLS benchmark Suite). The plot only compares the metric on Intel Xeon and i7 (with 8 threads) with Intel Xeon Phi (125 threads).

Table 5.8 shows the average vectorization speedup using multicores for circuits from the IWLS suite. These circuits are small (less than 65K gates), so we do expect to see any performance gain as we increase the number of threads.

Table 5.8: Multicore SIMD performance gain against single core nonSIMD across difference architectures for circuits from IWLS

Theads	Speedup (Avg)	
	Xeon	Intel i7
1	1.99	2.32
2	1.35	1.85
4	1.31	1.80
8	0.95	1.13

Threads	Xeon Phi (Avg Speedup)
1	2.37
5	1.62
25	1.31
125	0.83

To summarize this section and the previous one, SIMD speedup of up to 11.1 were recorded showing the effectiveness of the data structure and speedup of up to 299.8 were obtained by combination of SIMD and multicore (Figure 5.3).

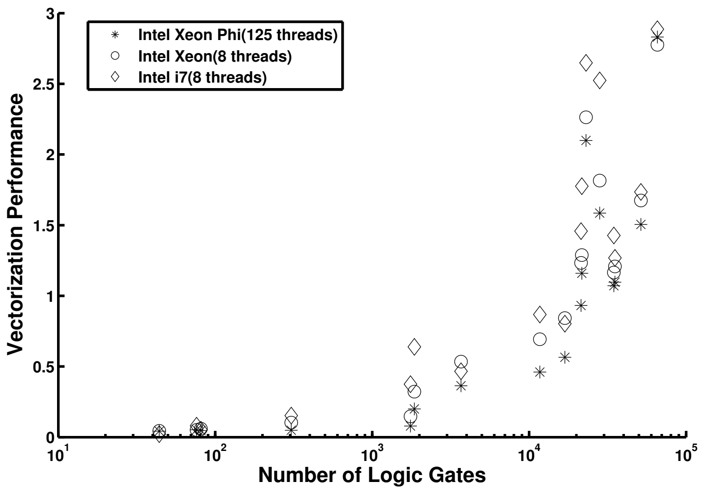


Figure 5.9: Comparison of performance gain for multicore SIMD against single core non-SIMD across different architectures for circuits from IWLS suite

Conclusions relative to Multicore and SIMD accelerations

In both previous sections (5.3.1and 5.3.2), the effect of SIMD acceleration and the use of multiple cores was examined. For the first experiment, the simulator was run on a single core with and without vectorization. Then, the performance of the simulator using multiple cores was evaluated.

We also observed the effect of SIMD on a single core. On Xeon Phi, ZSIM gained the maximum performance of 10.6 on Xeon Phi. However, the total simulation time on Xeon was much less than Xeon Phi. Moreover, the vectorization performance degraded as the circuit size grew larger. With each logic gate occupying 16 bytes (4bytes * 4), only 2K logic gates can fit in L1 and 32K in L2 cache. This explains jumps in the performance when size of the circuit goes beyond 2K, and 32K.

Using multiple cores with SIMD acceleration, the performance increased to almost 300 on Xeon Phi. The results displayed demonstrate the clear performance benefit which could have been achieved through SIMD acceleration and multicore parallelisation. It was observed that gaining full parallelisim on Xeon Phi is not possible for smaller circuits due to the thread scheduling overhead. On larger circuits, increasing the number of cores led to performance gain. However, ZSIM did not gain the full theoretical performance due to memory contention.

Xeon Phi supports aggressive forms of pre-fetching to hide data access latency. Due to random memory access patterns (e.g: evaluating a logic gate with inputs from other levels), pre-fetching data from memory reduces memory latency to some extent. Although not reported,the ZSIM was also tested without the pre-fetching capabilities of the compiler (`no-opt-prefetch`). Software pre-fetching is enabled by `-O2` optimization, and the default degree of aggressive optimization is `opt-prefetch=3`. The default pre-fetching mode was used throughout the experiments and all the reported results on Xeon Phi using multiple cores are with the pre-fetching flag enabled.

Further optimizations such as `-O3`, did not prove to be useful in a sense to improve the performance. In our case, the further optimization led to longer simulation time on multiple cores.

Moreover, to have an efficient parallel simulator, an attempt was made to control the OpenMP Thread Affinity at the runtime. The `KMP_AFFINITY` variable controls how threads are bounds to cores (i.e: spreading threads across cores is done by thread affinity). The three `compact, scatter, or guided` affinity types were used on various occasions. However, the monitored results did not show any significant affect on the performance of ZSIM using 240 cores. It is more likely that having multiple threads gathered on the same core would result in them competing for the shared resources and reducing the performance as it would serialize threads' requests. Distributing the threads among cores using the affinity variables (in the case we tested) did not make a difference in performance.Note that those reported in Chapter 5, are the ones that showed some significance in them.

5.3.3 Comparison with a commercial workstation simulator

The commercial simulator that was used in this work is a software tool that was produced by Xilinx. As part of the Xilinx ISE Design Suite, ISE simulator was used as a HDL simulator. For the experiments the ISE Design Suite version 14.7 was installed on Intel i7 machine. The Xilinx simulator simulates the circuit in verilog format. Note that ZSIM and Xilinx were compared for the same set of test circuits. The set of input vectors to both simulators were random. Moreover, both simulators were run for the same number of cycles. Although the multi-threaded compilation was enabled on Xilinx ISE simulator, the simulation was sequential.

As the Xilinx simulator runs only on standard Intel chips, this comparison did not use the Xeon Phi. The Xilinx simulator could only take circuits of small size. We tested ZSIM simulator's results for IWLS circuits, as well as our synthetic circuits (with inputs from any level), against Xilinx simulator on an Intel i7 machine.

Although it is not fair to compare the performance of ZSIM running on Xeon Phi and Xeon, with the commercial simulator running on Intel i7, Table 5.10, and Table 5.9 show the average speedup of ZSIM against the ISIM (commercial simulator) across different platforms (ZSIM was compiled by Intel compiler on the three machines).

For circuits from IWLS suite (size: 40 to 65K), when running on single core, ZSIM outperforms the commercial simulator by 14.89 to 962.73 , and average performance of 207.68

(with vectorization enabled), and by 4.26 to 1077.73, and average performance of 104.72 without vectorization (Table 5.10). Using multicore with SIMD enabled, ZSIM's performance degrades. Using only 1thread, ZSIM still outperforms the commercial simulator by 7.47 to 263.95 ,and average speedup of 97.68. Using 8 threads, ZSIM's average relative speedup to Xilinx reduces to 45.40 (Table 5.9)[7]

Figure 5.10 shows the number of gate transitions per second for SIMD ZSIM running on Intel i7 (8 threads), Intel Xeon Phi (125 threads), and the commercial simulator on Intel i7 for circuits from IWLS benchmark suite. It shows the poor performance of ZSIM on Intel Xeon Phi for small circuit sizes (note that the size of the test circuits are less than 100K). The event rate metric shows the difference in performance on Intel i7 for both simulators. Figure 5.11, shows the effect of vectorization on the performance when comparing with the number of transitions per second in the non vectorized version of ZSIM. Note that ISIM is the commercial simulator.

The circuits are of a small size for which our simulator works best with small scale 8 threads parallelism. As we increase the number of threads, the overhead due to thread scheduling worsen the performance. The circuits are small, there is not enough work to keep the cores busy and hide latency. However, as stated in previous paragraph, even using one thread on the same machine, our SIMD simulator is much faster than the commercial simulator.

Figure 5.12 shows the ZSIM's execution time in seconds using multiple threads and SIMD enabled on Intel i7. The log/log plot shows how ZSIM runs slower as more number of threads

[7](See full data at doi:10.5525/gla.researchdata.342)

Table 5.9: Speedup of ZSIM using multicore and SIMD against the commercial simulator across different platforms for circuits from IWLS suite

Threads	Xeon Phi
1	6.65
5	4.36
25	3.46
125	2.18

Threads	Xeon	I7
1	78.98	97.68
2	54.53	74.23
4	53.90	72.05
8	38.30	45.40

Table 5.10: Speedup of sequential ZSIM (SIMD and nonSIMD) against the commercial simulator across different platforms for circuits from IWLS suite

Machine		Speedup (Avg)
Xeon Phi	SIMD	26.33
	nonSIMD	4.35
Xeon	SIMD	212.10
	nonSIMD	62.57
i7	SIMD	207.68
	nonSIMD	104.72

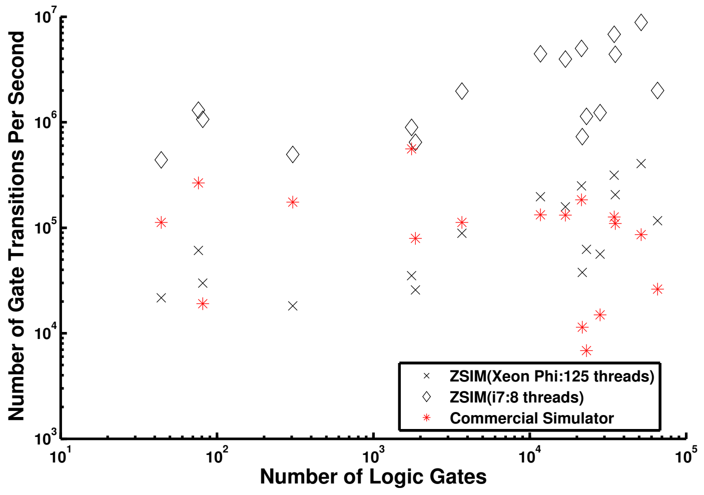


Figure 5.10: Number of gate transitions per second for the commercial simulator ISIM (on Intel i7), and the multicore SIMD ZSIM running on both Intel i7 and Xeon Phi for circuits from IWLS suite

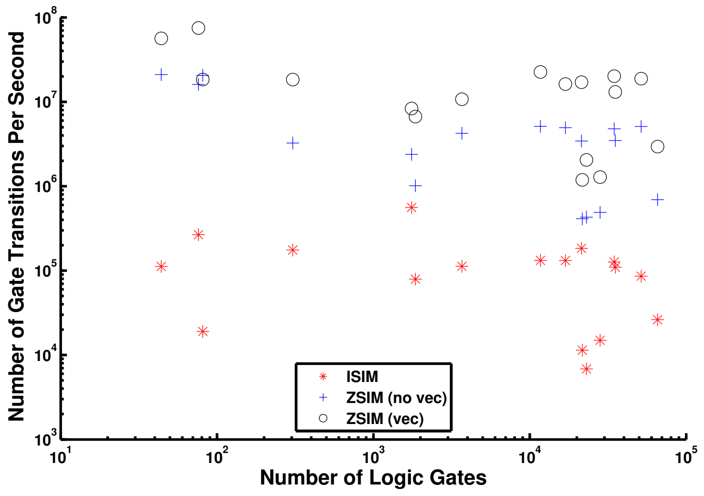


Figure 5.11: Number of gate transitions per second for ZSIM (single core with and without vectorization), and the commercial simulator (ISIM) on Intel i7 for circuits from IWLS suite

are used. This is due to the fact that the number of circuits used in this experiment are very small. As stated before, the poor performance is due to the overhead of thread scheduling. Note that on Intel i7, ZSIM was able to achieve the peak performance of 19.02 on Intel i7 for larger circuit sizes (shown in Table 5.5).

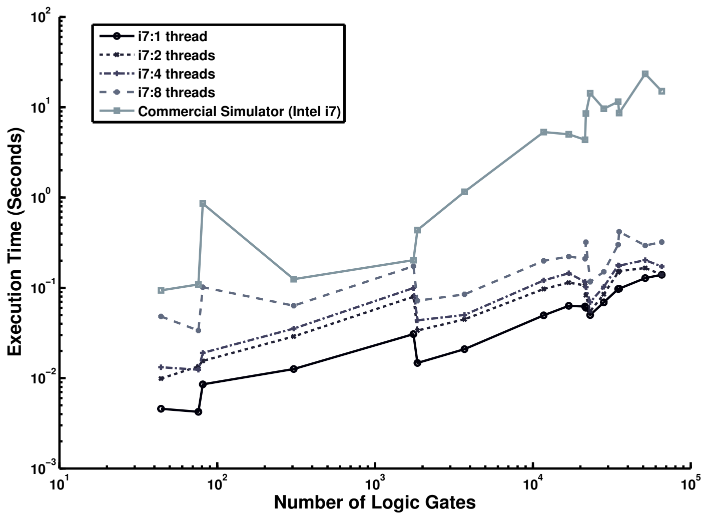


Figure 5.12: Performance comparison of ZSIM (multicore SIMD) and the commercial simulator ISIM on Intel i7 for circuits from IWLS suite

We performed the same experiments with synthetic circuits. Figure 5.13, shows the event rate per second metric for larger circuits (synthetic circuits). Using 8 threads on Intel i7, SIMD ZSIM achieves the maximum event rate of 474M for circuits of size 50 to 30M logic gates. For synthetic circuits (with inputs from any level), sequential ZSIM outperforms the commercial simulator by 17.40. to 148.40 , and average performance of 66.81 (without vectorization), and by 92.93 to 1204.88, and average performance of 511.39 with vectorization enabled (Table 5.11). Using multicore with SIMD enabled, ZSIM's performance degrades. Using only 1thread, ZSIM still outperforms the commercial simulator by 16.47 to 1189.43 ,and average speedup of 447.4 (Table 5.12). Note that the numbers 66.81, 511.39, and 447.4 are the average speedup of circuits of size 45 to 97814 given in Table 5.11 and Table 5.12. We investigated the results from performing the above experiments with synthetic circuits. The results show the commercial simulator drastically slows down as we increase the circuit size. Note that this commercial simulator is sequential event based and the activity rate of the circuit hugely affects it performance.

5.3.4 Comparison with simulations on GPUs

There are several existing papers on logic gate level circuit simulation acceleration on GPUs, though the results reported in the literature are for comparatively small circuits. Table 5.13

Table 5.11: Speedup of sequential ZSIM (SIMD and nonSIMD) against the commercial simulator on Intel i7 (Synthetic circuits V1)

Circuit Size (gates)	SIMD	nonSIMD
45	92.93	49.61
240	275.23	32.11
828	145.21	17.41
4140	686.15	47.83
18274	1204.88	148.40
97814	663.96	105.49

Table 5.12: Speedup of ZSIM using multicore and SIMD against the commercial simulator on Intel i7 (Synthetic circuits V1)

Circuit Size (gates)	1 thread	8 threads
45	16.47	3.45
240	74.22	17.19
828	126.87	34.75
4140	509.10	251.53
18274	1189.43	889.96
97814	768.34	2007.29

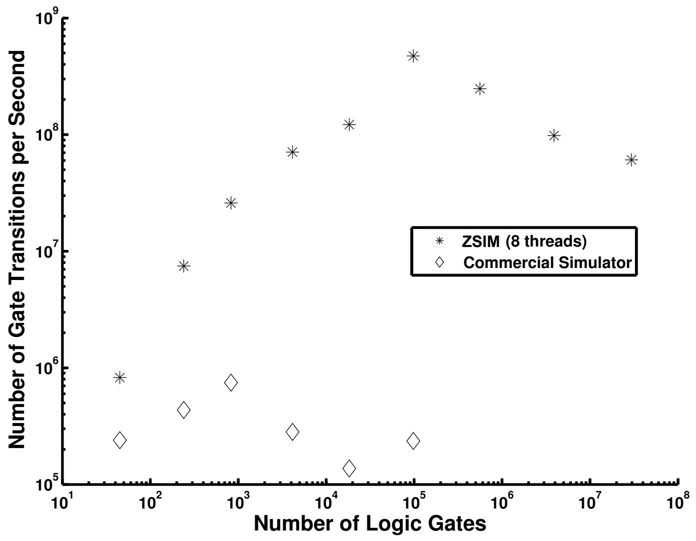


Figure 5.13: Number of gate transitions per second for the commercial simulator ISIM and multicore SIMD ZSIM on Intel i7 for synthetic circuits V1 (with inputs from any level)

shows the characteristic of the GPUs that were used in the literature and are compared to ZSIM in this section.

Table 5.13: Characteristic comparisons of GPUs used in the reviewed simulators (see Section 2.9

	Sen et al.	Zhu et al.	Yuxuan et al.	Chatterjee et al.
GPU Model	Quadro FX 3800	GTX 280	GTX 465	8800 GT
Cores	192	240	352	112
Clock Frequency	1204 MHz	1296 MHz	1215 MHz	1500 MHz
Launch date	March 2009	June 2008	May 2010	October 2007

In [12], the authors use partitioning and replication in conjunction with levelisation in order to handle the problem that the GPUs provide a small amount of shared memory. They therefore put groups of gates into logical blocks which then simulated on these Nvidia blocks since all threads in a block share the same shared memory. For the experiments they used Quadro FX 3800 GPU (Table 5.13).

It is possible to directly compare the performance of my data structure with the result they report for two of their circuits. Table 5.14 shows that when my data structure is run even on one core of a standard Intel i7, the performance substantially exceeds the results reported from [12], when we use a common metric of nano seconds per gate simulation. Note that the comparisons are valid as we are comparing the same circuits and both simulators are cycle based.

Table 5.14: Comparison of time per gate simulation for Intel i7 and Nvidia Quadro FX3800 GPU

Design	Time per gate simulation (nano seconds)	
	Alpsen et. al PAR2 (Nvidia GPU)	Chimeh Intel i7
aes-core	3.56	1.88
system-cdes	50.67	1.90

It is also worth noting that the mentioned paper reports results only on comparatively small circuits well under a million gates. So, the applicability of their technique to large circuits is unclear.

Zhu et al. report simulation on a 2.66 GHz Intel Core2 Duo server with an NVidia GTX 280 graphics card [10] (Table 5.13). It is difficult to compare their results with those reported in this thesis for two reasons.

- There is a little overlap between the circuit models this thesis reports results for and the ones they report.

- For the ones they do report, the random test patterns that they used only change the data once in every 5 cycles, whereas in this thesis data is changed at every cycle.

Given these limitations the only feasible basis to use is to compare the speedup they report between their baseline sequential simulator and their parallel simulator with the speedup obtained between the Xilinx simulator and the parallel ZSIM on circuits of the same size. Their best results for inputs changing randomly every 5 cycles was a speedup factor of 270. The best result for circuits of comparable size (the largest that was attempted to be run on the commercial simulator) was a speedup of 260.68 on Xeon Phi (using 125 threads) and a speedup of 1683.49 running SIMD on Intel Xeon using 8 threads. This was for a circuit of 100K gates in each case. This test is in the small circuit range for ZSIM, the range in which standard Intel processors outperform the more highly parallelised Xeon Phi. Even in this range of size, before the data structure is used to its full advantage, ZSIM gets comparable results to [10].

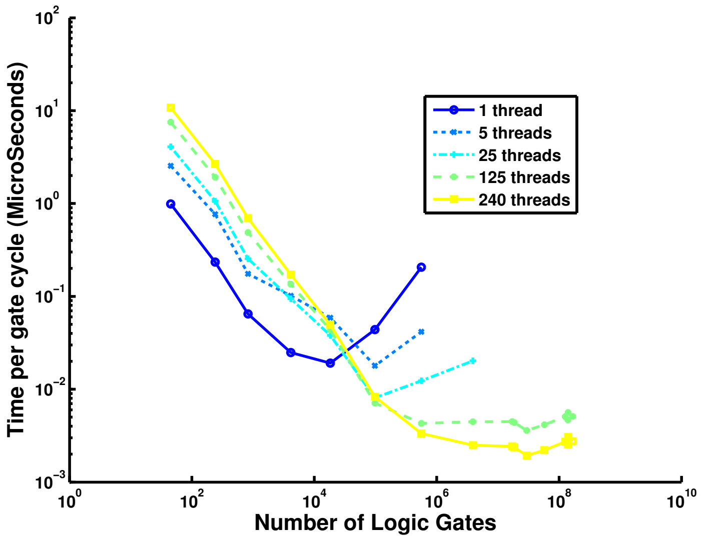
A paper by Yuxuan et al. [9], introduced a strategy to extract and partition the circuit in order to compile it to GPUs. Their technique is based on the levelisation and clustering into blocks as was in [12]. They presented comparison on the Intel Core Duo T2400 processors with 1.8 GHz frequency and the NVIDIA GTX 465 (Table 5.13).

Table 5.15: Time per gate simulation and Gate Transition per second for Parallel Oblivious simulation (Yuxuan et. al) on NVIDIA GTX 465

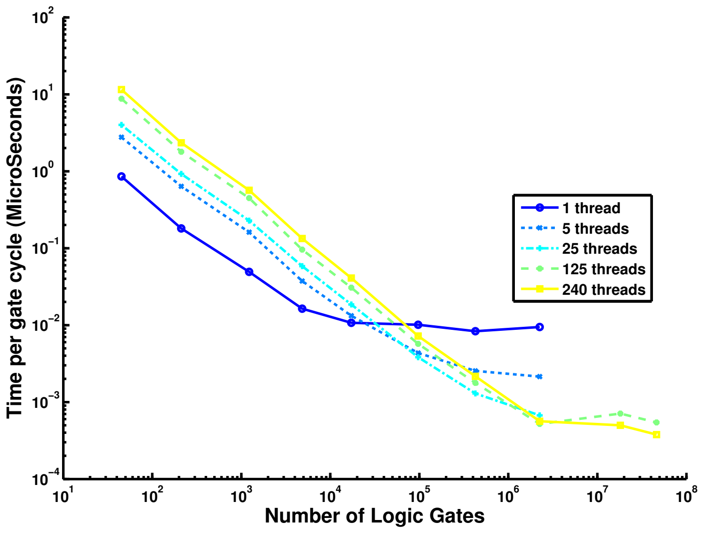
Design	Time per gate cycles (nano sec)	Gate transition per sec
LDPC	6.67	1.50E+008
DES3	2.15	4.65E+008
Or1200	13.30	7.51E+007
OpenSparc	2.17	4.59E+008

They achieved gate cycle times (Table 5.15) comparable to the peak performance of MIC in shown in Table 5.16. Figure 5.14 shows that the fast time per gate cycle for circuit with inputs from any level is around 1.93ns (Table 5.16). This reflects the lower task dispatch cost in CUDA relative to Xeon Phi. The Xeon Phi achieves it best performance on large circuits where the task dispatch cost can be spread over more gates. It is unclear whether the approach of Yuxuan et al. would be usable of the larger circuits studied here.

Chatterjee et al. report simulation on NVIDIA 8800GT GPU with 14 multiprocessors (Table 5.13) for full detail). Due to no overlap of test circuits, in order to compare the performance of ZSIM with the GCS simulator [11], we compare our speedup relative to the Xilinx simulator mentioned in section 5.2 to the speedup that Chatterjee et al. report relative to a commercial simulator. Their GCS simulator outperforms their commercial simulator by between 4 to 44 times with an average speedup of 13. ZSIM running SIMD parallelism on one core Intel i7, outperforms the Xilinx simulator by an average factor of 356 (Table 5.17).



(a) Synthetic circuits with inputs from any level



(b) Synthetic circuits with inputs from previous level only

Figure 5.14: Comparison of time per gate cycle of multicore SIMD ZSIM on Intel Xeon Phi

Table 5.16: Time Per gate Cycle of multicore SIMD ZSIM on Intel Xeon Phi using 240 threads (Synthetic circuits V1)

Circuit Size (gates)	Time Per gate Cycle (ns)
45	10755.28
240	2663.24
828	695.85
4140	171.46
18274	49.43
97814	8.30
560251	3.33
3892127	2.49
18561032	2.36
29651508	1.93
57765590	2.20
126351272	2.72
139352764	2.54
140083890	3.05
166599511	2.75

Note that in Table 5.17, my average speedup is for sequential SIMD ZSIM on Intel i7 against the commercial simulator. However, the average speedup for Chatterjee et al. is for the GCS simulator reported in their paper against the commercial simulator. The speedup of the larger circuit with 100K gates was previously reported in prior sections.

Table 5.17: Comparison of average speedup relative to commercial simulator

	Chatterjee et al.	Chimeh
Circuit size range	17K-1M	44 - 100K
Average speedup	13	356

Conclusions relative to GPUs

GPUs can achieve comparable gate cycle per second rates to the Xeon Phi. But this is only been demonstrated on the GPUs for the relatively small circuits. Although it is not explained in the literature why small circuits have been used in GPU experiments, it is hypothesized that the relatively small local memory on GPUs motivates experiments to select problems that are easier to map to the local memory. Another possibility is simply the lack of large scale circuits designs like the ones I have used. Whatever the case, it is clear from the reported results in this thesis that Xeon Phi can be extended to the circuits of around 100 millions of gates. Furthermore, with the ZSIM algorithm no partitioning is needed due to the automatic cache handling feature on Xeon Phi. We therefore move on to examine the relative performance of Xeon Phi against supercomputers which are capable of handling 100 millions of gates.

5.3.5 Comparison with simulation on the IBM Blue Gene

In this section, comparison is made on some of ZSIM's results with reported work on parallel simulation on a supercomputer (Gonsiorowski et al. [4]). They used a discrete event simulation framework that allows simulations to be run in parallel, called ROSS (Rensselaer Optimistic Simulation System), a modular time wrap system. The paper reports the performance of this framework executing parallel event based simulation (based on the time wrap protocol) using a message passing interface on Blue Gene/L.

Blue Gene/L Architecture

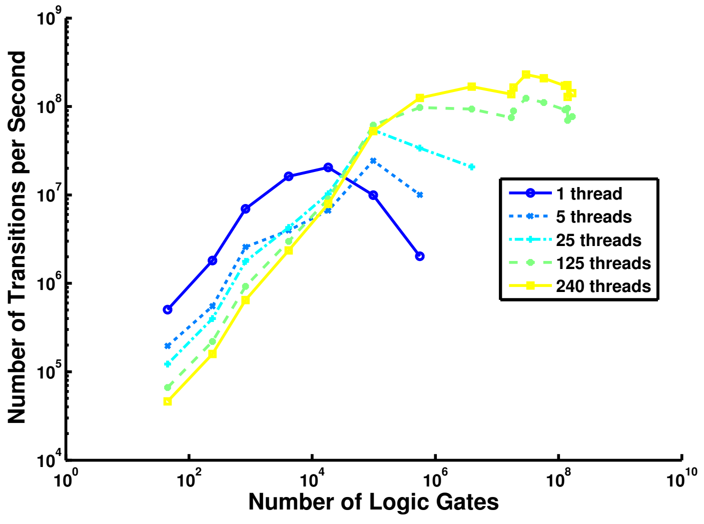
The experiments were done on two machines (IBM Blue Gene/L, and Intel X5650). The Blue Gene/L has upto 1024 cores, each performing at 700 MHZ clock rate. However, we only aim to compare our results with the ones ran on the supercomputer Blue Gene. Table 5.18, compares some of the characteristics of both Intel Xeon Phi and Blue Gene/L, in terms of the price per rack and the size, in addition to the number of available cores.

To evaluate the simulation performance, the number of gate transitions per second between ZSIM and [4] are compared. More specifically, we are comparing the event metric for our largest circuit (with over 160 millions of gates) with their 216 million gates circuit. Table 5.19 compares the event rate data taken from the mentioned paper with the event rate measured in ZSIM. Here, we are only showing the results for the maximum circuit size. On Blue Gene/L with 1024 cores, they achieved an event rate of 116 million events per second, whileZSIM achieved an event rate of 142 million events per second (Table 5.19).Figure 5.15 shows the event rate per second for ZSIM (multicore and SIMD) for the two versions of synthetic circuits on Intel Xeon Phi. Note that these are actual transitions in contrast to Figure 5.14 that shows time per gate cycle.

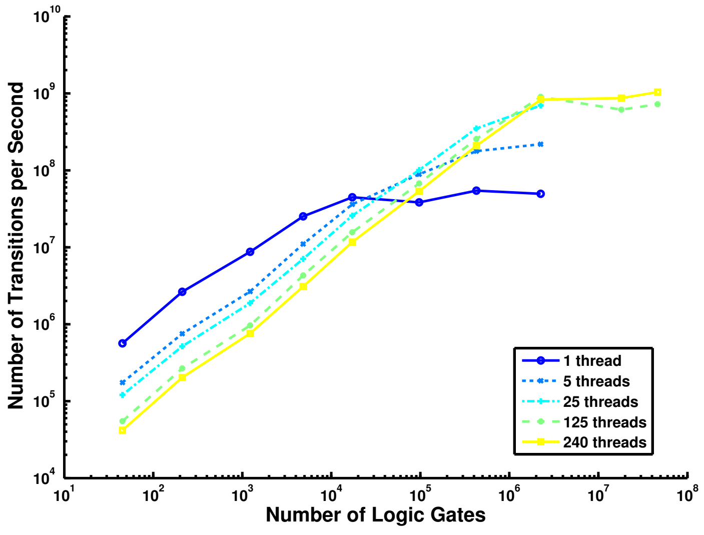
ZSIM achieves better performance on many fewer cores at much lower cost. The Xeon Phi clock speed is slightly higher than that of the Blue Gene, but the main gain comes from the ability of our data-structure to handle both SIMD and multicore parallelism with low synchronization overhead.

5.3.6 Comparative Analysis Across Compilers

Figure. 5.16(left plot) shows the number of gate transitions per second for 64 threads on the multicore AMD Opteron for both C and Pascal versions compiled with `gcc` and `vpc`.



(a) Synthetic circuits with inputs from any level



(b) Synthetic circuits with inputs from previous level only

Figure 5.15: Comparison of number of transitions per second for multicore SIMD ZSIM on Intel Xeon Phi

Table 5.18: Characteristic comparison of Intel Xeon phi and IBM Blue Gene/L

Parameter	IBM Blue Gene/L	Intel Xeon phi
Cores	1024	60
Clock Speed	700 MHz/core	1.053 GHz/core
Price	\$0.8m - \$1.3m	\$1600.00 - \$2649.00
Size	2m height x 1m width	24.61cm x 11.12cm x 3.86cm

Table 5.19: Comparison of number of events per second (IBM Blue Gene/L vs. Intel Xeon Phi)

Machine	Number of gates	Cores/Threads	Event rate (millions/sec)
Blue Gene/L	$\simeq$ 216 million	512	60
		1024	116
Xeon Phi	$\simeq$ 160 million	125	76.8
		240	142

Whereas the right plot in Figure. 5.16, shows the event rate per seconds for the Pascal and C versions on the Xeon Phi using `vpc` and `icpc`. The algorithm shows the same acceleration scaling properties across languages and machines. The performance curves have the same shape in both figures as we increase the circuit size. On both machines C is faster than Pascal.The Intel C compiler on Xeon Phi shows a bigger advantage than `gcc` on the AMD machine.The peak performance leveled off at around 1M logic gates in all cases. Figure 5.17 shows the number of gate transitions per second for 8 threads on the multicore Intel i7 for C version of the simulator compiled with `gcc` and `icpc`. The plot shows that Intel C compiler performs better on Intel i7 comparing to `gcc`. However, after a 1M gate, the performance difference on both compilers keeps reducing.

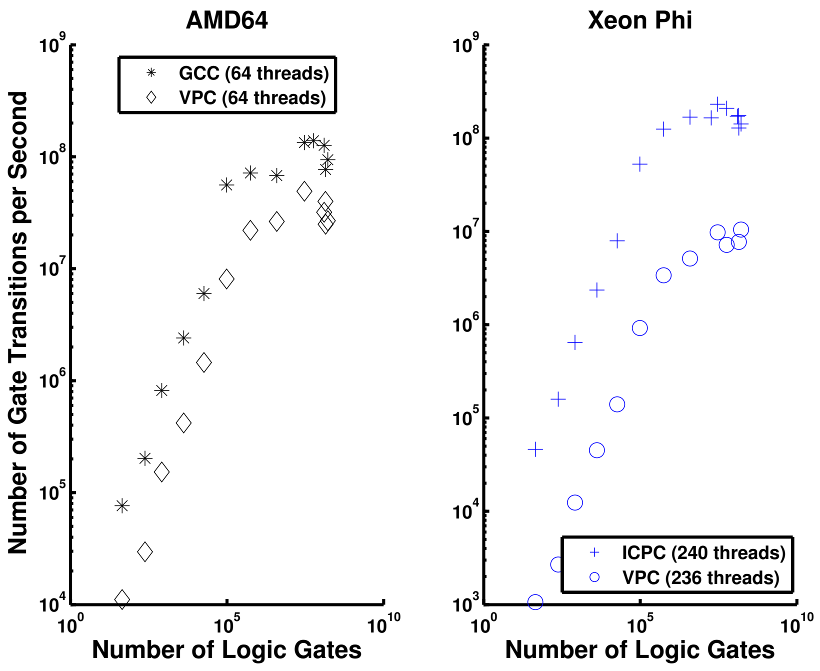


Figure 5.16: Comparison of number of transitions per second of multicore SIMD ZSIM across different compilers on both AMD Opteron and Xeon Phi machine

Figure 5.18, compares event rate per second of the parallel simulator on both AMD and Xeon Phi for the circuit size of 170 million gates. The plot only shows the event rate upto the maximum number of cores on each machine. The plot shows the almost linear relation of the number of threads with the performance on Xeon Phi using Intel compiler. However, the

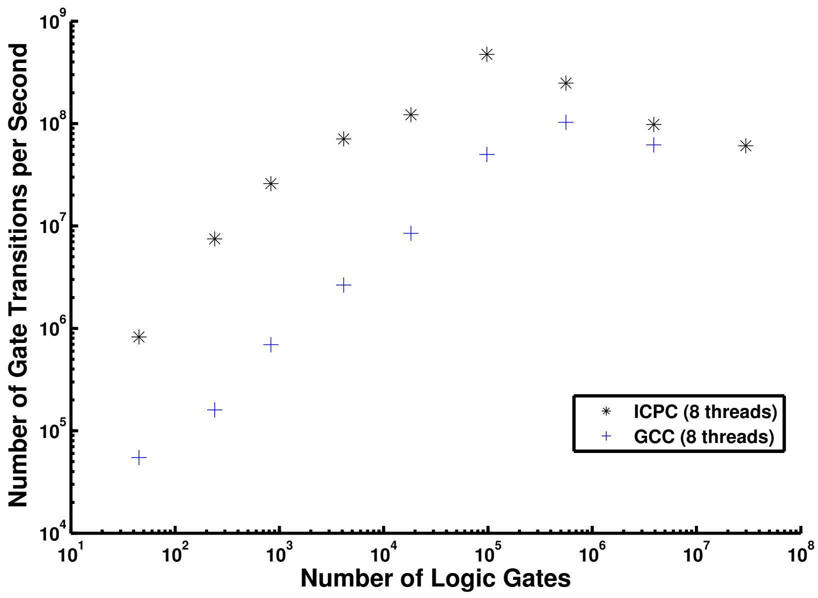


Figure 5.17: Comparison of number of transitions per second of multicore SIMD ZSIM across different compilers on Intel i7 machine

other compilers do not have the linear relation. In other words, using GCC and VP compilers, the simulator stopped scaling after 16 threads on AMD, and 60 threads on Xeon Phi. Vector Pascal compiler shows a better performance on AMD64 than Xeon Phi. In a previous study of ours [14], Vector Pascal was also compared to other commercial compilers in addition to Intel C. The comparison to Intel C compiler had the same result.

For 16 and 64 threads, the Vector Pascal and gcc compiler did not show any improvement on AMD64. On Intel Xeon Phi, using Vector Pascal compiler, event rate leveled off at 2 orders of magnitude threads.

Furthermore, I have also measured the performance of the simulator written in pascal by changing the default task scheduling for the Vector Pascal (thread semaphores). I found that using spin-locks (busy waiting [8]) on Xeon Phi was efficient up to a certain circuit size (1M logic gates). As I increased the size of the circuit, there was no performance gain with the busy wait flag enabled (Figure 5.19). Using 64 threads on Xeon Phi, with busy waiting, the ZSIM achieved the maximum performance of 10.6, and 9.02 using 236 threads (Table 5.20). The table only shows the speedup for circuits under 30M logic gates. For circuits more than 1M, the speedup is not significant.

[8]Simulator was compiled with the `busy wait` flag on

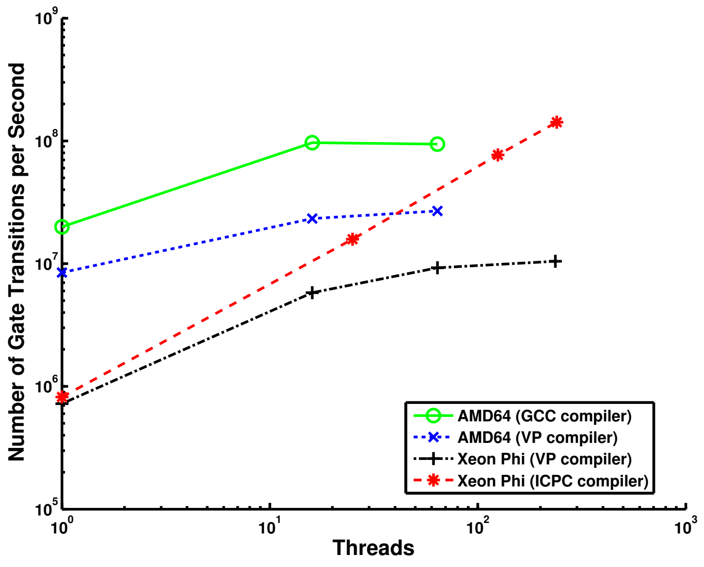


Figure 5.18: Comparison of number of transitions per second of multicore SIMD ZSIM on Intel Xeon Phi and AMD Opteron, compiled by both Vector Pascal and Intel compiler for circuit size of 170M gates

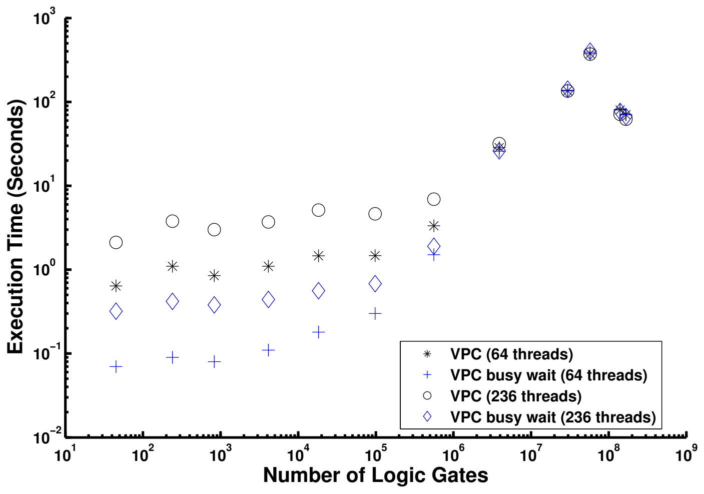


Figure 5.19: Performance comparison of multicore SIMD ZSIM compiled by Vector Pascal compiler using semaphores versus busy waiting on Intel Xeon Phi

Table 5.20: Comparison of performance gain using semaphores versus busy waiting on Intel Xeon Phi for circuits under 30M gates

Circuit size	Threads	Speedup
45	64	9.14
	236	6.59
240	64	12.22
	236	9.02
828	64	10.63
	236	7.89
4140	64	10.00
	236	8.41
18274	64	8.11
	236	9.14
97814	64	4.90
	236	6.81
560251	64	2.21
	236	3.64
3892127	64	1.10
	236	1.22
29651508	64	0.98
	236	0.95

Chapter 6

Conclusion

In this thesis, an architecture without explicit locks for accelerating logic gate simulation was proposed. The proposed architecture targets multicore machines with gather instruction support such as Xeon Phi. This data structure reduces the synchronization overhead, while increases the possibility of SIMD and parallel operations. This was applied on the state the of art, Intel Xeon Phi technology [47]. The combination of this data structure and the Xeon Phi chip is a cost effective solution for simulation acceleration. The architecture was tested on various architectures on large circuit sizes as well as smaller ones.

The thesis statement was :

This thesis is that it is possible, by the use of simple regular data structures to obtain, with SIMD shared memory multiprocessors, simulation speed, that are as good as or better than other workstation technologies and more cost effective than small super computer cluster for the same task.

It was proven that the thesis statement is correct. The results in Section 5.3 show that performance of ZSIM is comparable to performance reported for simulators on Blue Gene/L, comparable and faster than reported for software simulators targeting GPUs and much faster than a workstation commercial simulator Xilinx.

6.1 Research Contributions

This work was motivated by the availability of multicore SIMD processors with gather instructions. As stated above, this allows more efficient high performance logic simulations than alternative software architectures. An architecture for logic circuit simulation targeting SIMD machines was proposed. Moreover, previous works in the area of parallel logic level circuit simulation were reviewed. A larger range of synthetic circuits, broader than what is reported in the logic simulation acceleration literature were generated. The performance of the proposed SIMD simulator was evaluated on various architectures such as Intel Xeon Phi, Intel Xeon, and an AMD64 machine.

The ZSIM simulator was implemented in two different programming languages and its performance was compared across various compilers including Intel C, Vector Pascal, and gcc. When the ZSIM performance was compared across these compilers, Intel C proved to be more efficient. Above a certain level of parallelism, between 16 and 64 threads, the Vector Pascal and gcc compiled implementations did not show further speed improvement when running on AMD64. In contrast, when using the Intel compiler optimised for it, the Xeon Phi implementation showed continuing improvements up to the maximum number of cores used.

The thesis compared the vectorization performance in ZSIM[1] with SIMD acceleration on Single core and multicore. Enabling SIMD acceleration and multicore, ZSIM gained peak performance of 299.8 on Intel Xeon Phi and 11.1 on Intel Xeon. Using only vectorization, ZSIM achieved the maximum performance of 10.6 on Intel Xeon Phi and 4 on Intel Xeon.

It was shown that this combination (the proposed architecture on SIMD machine) is much faster than, and can handle much bigger circuits than a widely used commercial simulator (Xilinx) running on a workstation.

We compared the achieved performance with similar pre-existing work on logic simulation using GPUs and supercomputers. The results presented in this dissertation show that the ZSIM simulator running on a Xeon Phi gives comparable simulation performance to the IBM Blue Gene supercomputer at very much lower cost. The experimental results have shown that the Xeon Phi is competitive with simulation on GPUs and allows the handling of much larger circuits than have been reported for GPU simulation.

When targeting Xeon Phi architecture, the automatic cache management of the Xeon Phi, handles and manages the on-chip local store without any explicit mention of the local store being made in the architecture of the simulator itself. However, targeting GPUs, explicit cache management in program increases the complexity of the software architecture. Furthermore, one of the strongest points of the ZSIM simulator is its portability. Note that the same code was tested on both AMD and Xeon Phi machines. The same architecture that efficiently performs on Xeon Phi, was ported into a 64 core NUMA AMD Opteron.

The performance gains were also compared with the performance of the proposed algorithm in this thesis with stereo vision algorithms on the same compilers and machines from a previous study [14]. The algorithms in that study showed a much greater performance on the GPU than what was achieved on either the Intel or AMD machines. This contrasts with

[1]my simulator

the results in this thesis which show a clear advantage for the Xeon Phi. The reported results were for identical algorithms. Note that ZSIM was compared to the previous published work on the GPU. These studies used different algorithms from ZSIM, so either the Xeon Phi is better for logic simulation than GPUs or the ZSIM architecture is an advance on previous work.

To conclude, the two main achievements are restated as following: The primary achievement of this work was proving that the ZSIM architecture was faster than previously published logic simulators on low cost platforms. The secondary achievement was the development of a synthetic testing suite that went beyond the scale range that was previously publicly available, based on prior work that showed the synthesis technique is valid [2].

6.2 Future Work and Limitations

This research is a self contained piece of work that was done within the timing and resource constraint available for the PhD work. It is not a production simulator and does not handle a wide range of input formats (handles BLIF formats only). Although it does not recognize all the input formats that a more sophisticated commercial product would recognize, the ZSIM simulator handles considerably larger circuits than the Xilinx reference simulator. The experimental results that were demonstrated are within the limited availability of the benchmarks that could have been tested with limited time and resources, in particular the limited availability of real circuit designs in the 100 million gate range.

A further direction of research could be to investigate the performance of the ZSIM simulator on a machine with cluster of Xeon Phis, as we only had access to a single Xeon Phi. Furthermore, it is worth evaluating the impact of using the combination of direct logic with word packing technique in the simulator instead of using look-up table and word packing.

Another direction would be to see how to handle in a more realistic way the different timing in logic gates. Although it was shown in principle that different timing delays in logic gate can be handled by adding extra buffer layers, it would be useful to verify what impact that has in overall performance. However, given the order of magnitude speedup the simulator achieved, the impact of adding extra buffers to the design can be expected to be relatively small.

References

[1] C. Albrecht. IWLS 2005 Benchmarks. [Online]. Available: http://iwls.org/iwls2005/benchmarks.html

[2] M. Hutton, J. Rose, and D. Corneil, "Automatic generation of synthetic sequential benchmark circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, no. 8, pp. 928–940, Aug 2002.

[3] R. L. Geiger, P. E. Allen, and N. R. Strader, *VLSI design techniques for analog and digital circuits.* New Delhi TMH c, 1990.

[4] E. Gonsiorowski, C. Carothers, and C. Tropper, "Modeling large scale circuits using massively parallel discrete-event simulation," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, Aug 2012, pp. 127–133.

[5] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with gp-gpus," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, July 2009, pp. 557–562.

[6] J. Reinders. (2012) An overview of programming for intel xeon processors and intel xeon phi coprocessors. [Online]. Available: http://download.intel.com/newsroom/kits/xeon/phi/pdfs/overview-programming-intel-xeon-intel-xeon-phi-coprocessors.pdf

[7] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," in *ACM SIGGRAPH 2008 Papers*, ser. SIGGRAPH '08. New York, NY, USA: ACM, 2008, pp. 18:1–18:15. [Online]. Available: http://doi.acm.org/10.1145/1399504.1360617

[8] M. Chimeh, C. Hall, and J. O'Donnell, "Optimisation and parallelism in synchronous digital circuit simulators," in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, Dec 2012, pp. 94–101.

[9] Z. Yuxuan, W. Tingcun, K. Yaowen, F. Xiaoya, Z. Meng, and Z. Lili, "Logic simulation acceleration based on gpu," in *Mixed Design of Integrated Circuits and Systems*

(MIXDES), 2011 Proceedings of the 18th International Conference, June 2011, pp. 608–613.

[10] Y. Zhu, B. Wang, and Y. Deng, “Massively parallel logic simulation with gpus,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 3, pp. 29:1–29:20, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1970353.1970362

[11] D. Chatterjee, A. DeOrio, and V. Bertacco, “Gate-level simulation with gpu computing,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 3, pp. 30:1–30:26, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1970353.1970363

[12] A. Sen, B. Aksanli, and M. Bozkurt, “Speeding up cycle based logic simulation using graphics processing units,” *j-INT-J-PARALLEL-PROG*, vol. 39, no. 5, pp. 639–661, Oct. 2011. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article&issn=0885-7458&volume=39&issue=5&spage=639

[13] J. Wang, D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, “Parallel discrete event simulation for multi-core systems: Analysis and optimization,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1574–1584, June 2014.

[14] M. Chimeh, P. Cockshott, S. B. Oehler, A. Tousimojarad, and T. Xu, “Compiling Vector Pascal to the XeonPhi,” *Concurrency and Computation: Practice and Experience*, 2015.

[15] R. L. Page, “Brief history of flight simulation,” *SimTecT 2000 Proceedings*, pp. 11–17, 2000.

[16] (2002) The wake of the sleeping giant-verification. [Online]. Available: http://www.mentor.com

[17] L. W. Nagel, “Spice2: A computer program to simulate semiconductor circuits,” Ph.D. dissertation, EECS Department, University of California, Berkeley, 1975. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1975/9602.html

[18] S. Fuller and L. Millett, “Computing performance: Game over or next level?” *Computer*, vol. 44, no. 1, pp. 31–38, Jan 2011.

[19] N. Corporation. (2012) Nvidia geforce gtx 680. [Online]. Available: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

[20] H. Angepat, D. Chiou, E. S. Chung, and J. C. Hoe, *FPGA-Accelerated Simulation of Computer Systems*. Morgan and Claypool, 2014. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6894333

[21] E. Pugh, L. R. Johnson, and J. H. Palmer, *IBM’s 360 and Early 370 Systems*. Cambridge, MA, USA: MIT Press, 1991.

[22] L. Li and M. Thornton, *Digital System Verification:A Combined Formal Methods and Simulation Framework.* Morgan and Claypool, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6813106

[23] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, “Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 15:1–15:32, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534925

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[25] C. Chang, J. Wawrzynek, and R. W. Brodersen, “Bee2: a high-end reconfigurable computing system,” *IEEE Design Test of Computers*, vol. 22, no. 2, pp. 114–125, March 2005.

[26] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[27] J. Spars and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[28] H. Foster, D. Lacey, and A. Krolnik, *Assertion-Based Design*, 2nd ed. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[29] Z. Wang and P. M. Maurer, “Lecsim: a levelized event-driven compiled logic simulator,” in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, Jun 1990, pp. 491–496.

[30] P. M. Maurer, “Two new techniques for unit-delay compiled simulation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 9, pp. 1120–1130, Sep 1992.

[31] L. Suresh, N. Rameshan, M. S. Gaur, M. Zwolinski, and V. Laxmi, “Acceleration of functional validation using gpgpu,” in *Electronic Design, Test and Application (DELTA), 2011 Sixth IEEE International Symposium on*, Jan 2011, pp. 211–216.

[32] D. Liu, “Function verification of combinational arithmetic circuits,” Master’s thesis, University of Massachusetts Amherst, May 2015.

[33] K. Gulati and S. P. Khatri, “Towards acceleration of fault simulation using graphics processing units,” in *Proceedings of the 45th Annual Design Automation Conference*,

ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 822–827. [Online]. Available: http://doi.acm.org/10.1145/1391469.1391679

[34] L.-T. Wang, N. Hoover, E. Porter, and J. Zasio, "Ssim: A software levelized compiled-code simulator," in *Design Automation, 1987. 24th Conference on*, June 1987, pp. 2–8.

[35] H. Somakumar, "Concurrency-aware compiler optimizations for hardware description languages," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. Volume 18, Issue 1, pp. 10:1–10:16, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2390201

[36] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert, "Parallel cycle based logic simulation using graphics processing units," in *Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on*, July 2010, pp. 71–78.

[37] D. Chatterjee, A. DeOrio, and V. Bertacco, "Gcs: High-performance gate-level simulation with gpgpus," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 1332–1337.

[38] A. Perinkulam, "Logic simulation using graphics processors," Master's thesis, University of Massachusetts Amherst, January 2007.

[39] V. Bertacco and D. Chatterjee, "High performance gate-level simulation with gp-gpu computing," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, April 2011, pp. 1–3.

[40] T. Hashiguchi, Y. Mori, M. Toyonaga, and M. Muraoka, "Yapsim: Yet another parallel logic simulator using gp-gpu." SASIMI 2016, 2015.

[41] Top500. [Online]. Available: http://www.top500.org/

[42] F. Tso, D. White, S. Jouet, J. Singer, and D. Pezaros, "The glasgow raspberry pi cloud: a scale model for cloud computing infrastructures," pp. 108–112, 2013. [Online]. Available: http://eprints.gla.ac.uk/83064/

[43] T. Shanley, *Infiniband.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[44] (2013) Visit to the national university for defense technology changsha. [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf

[45] Intel. [Online]. Available: http://ark.intel.com/

[46] A. Khajeh-Saeed and S. P. J. B. Perot, “A comparison of multi-core processors on scientific computing tasks,” *Innovative Parallel Computing: Foundations and Applications of GPU, Manycore, and Heterogeneous Systems*, 2012.

[47] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[48] J. Reinders, J. Jeffers, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition*. Boston, MA, USA: Morgan Kaufmann Publishers Inc., 2016.

[49] G. Meister, “A survey on parallel logic simulation,” University of Saarland, Department of Computer Science, Misra J, Tech. Rep., 1993.

[50] L. Soulé and T. Blank, “Parallel logic simulation on general purpose machines,” in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, ser. DAC ’88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 166–171. [Online]. Available: http://dl.acm.org/citation.cfm?id=285730.285757

[51] W. Baker, A. Mahmood, and B. Carlson, “Parallel event-driven logic simulation algorithms: tutorial and comparative evaluation,” *Circuits, Devices and Systems, IEE Proceedings -*, vol. 143, no. 4, pp. 177–185, Aug 1996.

[52] Y. Matsumoto and K. Taki, “Parallel logic simulation on a distributed memory machine,” in *Design Automation, 1992. Proceedings., [3rd] European Conference on*, Mar 1992, pp. 76–80.

[53] N. Manjikian and W. M. Loucks, “High performance parallel logic simulations on a network of workstations,” *SIGSIM Simul. Dig.*, vol. 23, no. 1, pp. 76–84, Jul. 1993. [Online]. Available: http://doi.acm.org/10.1145/174134.158469

[54] H. K. Kim and S. M. Chung, “Parallel logic simulation using time warp on shared-memory multiprocessors,” in *Proceedings of the 8th International Symposium on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1994, pp. 942–948. [Online]. Available: http://dl.acm.org/citation.cfm?id=645604.662585

[55] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal, “Logic emulation with virtual wires,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 16, no. 6, pp. 609–626, Jun 1997.

[56] M. M. Denneau, “The yorktown simulation engine,” in *Proceedings of the 19th Design Automation Conference*, ser. DAC ’82. Piscataway, NJ, USA: IEEE Press, 1982, pp. 55–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=800263.809186

[57] Y.-I. Kim, W. Yang, Y.-S. Kwon, and C.-M. Kyung, "Communication-efficient hardware acceleration for fast functional simulation," in *Design Automation Conference, 2004. Proceedings. 41st*, July 2004, pp. 293–298.

[58] G. Pfister, "The yorktown simulation engine: Introduction," in *Design Automation, 1982. 19th Conference on*, 1982, pp. 51–54.

[59] R. M. Fujimoto, *Parallel and Distribution Simulation Systems*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[60] M. L. Bailey, J. V. Briner, Jr., and R. D. Chamberlain, "Parallel logic simulation of vlsi systems," *ACM Comput. Surv.*, vol. 26, pp. 255–294, September 1994. [Online]. Available: http://doi.acm.org/10.1145/185403.185424

[61] A. Bataineh, F. Ozguner, and I. Szauter, "Parallel logic and fault simulation algorithms for shared memory vector machines," in *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on*, Nov 1992, pp. 369–372.

[62] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, Sept 1979.

[63] R. E. Bryant, "Simulation of packet communication architecture computer systems," Cambridge, MA, USA, Tech. Rep., 1977.

[64] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3916.3988

[65] D. Jefferson, *Fast concurrent simulation using the time warp mechanism.* Santa Monica, Calif. : Rand Corp. [Online]. Available: http://hdl.handle.net/2027/inu.39000004008277

[66] Brookgpu. [Online]. Available: https://graphics.stanford.edu/projects/brookgpu/start.html

[67] U. W. Smith, S. and M. R. Mercer, "An analysis of several approaches to circuit partitioning for parallel logic simulation," in *In Proceedings of the International Conference on Computer Design*, 1987, pp. 664–667.

[68] S. Karthik and J. A. Abraham, "Distributed vlsi simulation on a network of workstations," in *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer &Amp; Processors*, ser. ICCD '92. Washington, DC, USA: IEEE Computer Society, 1992, pp. 508–511. [Online]. Available: http://dl.acm.org/citation.cfm?id=645461.654888

[69] E. H. Frank, “Exploiting parallelism in a switch-level simulation machine,” in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, ser. DAC ’86. Piscataway, NJ, USA: IEEE Press, 1986, pp. 20–26. [Online]. Available: http://dl.acm.org/citation.cfm?id=318013.318018

[70] K. Hering, R. Reilein, and S. Trautmann, “Cone clustering principles for parallel logic simulation,” in *Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on*, 2002, pp. 93–100.

[71] R. Brayton and A. Mishchenko, “Abc: An academic industrial-strength verification tool,” in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, ser. CAV’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 24–40. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_5

[72] A. Mishchenko, R. Brayton, and S. Jang, “Global delay optimization using structural choices,” in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’10. New York, NY, USA: ACM, 2010, pp. 181–184. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723144

[73] A. Mishchenko, S. Chatterjee, and R. Brayton, “Dag-aware aig rewriting a fresh look at combinational logic synthesis,” in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC ’06. New York, NY, USA: ACM, 2006, pp. 532–535. [Online]. Available: http://doi.acm.org/10.1145/1146909.1147048

[74] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, “Sat sweeping with local observability don’t-cares,” in *Design Automation Conference, 2006 43rd ACM/IEEE*, 2006, pp. 229–234.

[75] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, “Reducing structural bias in technology mapping,” in *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, Nov 2005, pp. 519–526.

[76] Aiger format. [Online]. Available: http://fmv.jku.at/aiger/

[77] B. Wang, Y. Zhu, and Y. Deng, “Distributed time, conservative parallel logic simulation on gpus,” in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, June 2010, pp. 761–766.

[78] Y. Deng, Y. Zhu, and W. Bo, *GPU Solutions to Multi-scale Problems in Science and Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. Asynchronous Parallel Logic Simulation on Modern Graphics Processors, pp. 517–541. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16405-7_32

[79] S. Holst, M. E. Imhof, and H.-J. Wunderlich, "High-throughput logic timing simulation on gpgpus," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 3, pp. 37:1–37:22, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2714564

[80] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: A high-performance, low memory, modular time warp system," in *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, ser. PADS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 53–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=336146.336157

[81] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," *ACM Trans. Model. Comput. Simul.*, vol. 9, no. 3, pp. 224–253, Jul. 1999. [Online]. Available: http://doi.acm.org/10.1145/347823.347828

[82] E. Gonsiorowski, "Modelling large scale circuits using massively parallel discrete-event simulation," University Rensselaer Polytech Institute, Tech. Rep., 2013.

[83] E. J. Gonsiorowski, J. M. LaPre, and C. D. Carothers, "Improving accuracy and performance through automatic model generation for gate-level circuit pdes with reverse computation," in *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM PADS '15. New York, NY, USA: ACM, 2015, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/2769458.2769463

[84] D. Kim, M. Ciesielski, and S. Yang, "Multes: Multilevel temporal-parallel event-driven simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 845–857, June 2013.

[85] J. Harlow, "Overview of popular benchmark sets," *Design Test of Computers, IEEE*, vol. 17, no. 3, pp. 15–17, Jul 2000.

[86] P. D. Kundarewich and J. Rose, "Synthetic circuit generation using clustering and iteration," in *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '03. New York, NY, USA: ACM, 2003, pp. 245–245. [Online]. Available: http://doi.acm.org/10.1145/611817.611875

[87] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," 1991.

[88] B. L. Synthesis and B. Verification Group, University of California. Berkeley logic interchange format (blif). [Online]. Available: https://www.ece.cmu.edu/~ee760/760docs/blif.pdf

[89] (2005) Benchmark designs for the quartus university interface program (quip). [Online]. Available: http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/links/quip_benchmarks.pdf

[90] J. Pistorius and M. Hutton, "Benchmarking method and designs targeting logic synthesis for fpgas," 2007.

[91] easics. [Online]. Available: https://code.google.com/p/eprize1/wiki/BenchmarkData

[92] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Circuits and Systems, 1989., IEEE International Symposium on*, May 1989, pp. 1929–1934 vol.3.

[93] F. Corno, M. S. Reorda, and G. Squillero, "Rt-level itc'99 benchmarks and first atpg results," *IEEE Design Test of Computers*, vol. 17, no. 3, pp. 44–53, Jul 2000.

[94] (1999) Itc99 benchmarks web site. [Online]. Available: http://www.cad.polito.it/tools/itc99.html

[95] Opencores. [Online]. Available: http://www.opencores.org

[96] J. Darnauer and W. W.-M. Dai, "A method for generating random circuits and its application to routability measurement," in *Proceedings of the 1996 ACM Fourth International Symposium on Field-programmable Gate Arrays*, ser. FPGA '96. New York, NY, USA: ACM, 1996, pp. 66–72. [Online]. Available: http://doi.acm.org/10.1145/228370.228380

[97] K. Iwama, S. Sawada, K. Hino, and H. Kurokawa, "Random benchmark circuits with controlled attributes," in *Proceedings of the 1997 European Conference on Design and Test*, ser. EDTC '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 90–. [Online]. Available: http://dl.acm.org/citation.cfm?id=787260.787642

[98] J. Pistorius, E. Legai, and M. Minoux, "Generation of very large circuits to benchmark the partitioning of fpga," in *Proceedings of the 1999 International Symposium on Physical Design*, ser. ISPD '99. New York, NY, USA: ACM, 1999, pp. 67–73. [Online]. Available: http://doi.acm.org/10.1145/299996.300026

[99] P. Verplaetse, "Synthetic Benchmark Circuits for Timing-driven Physical Design Applications."

[100] M. D. Hutton, J. S. Rose, and D. G. Corneil, "Automatic generation of synthetic sequential benchmark circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 8, pp. 928–940, 2002.

[101] P. D. Kundarewich and J. Rose, "Synthetic circuit generation using clustering and iteration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 6, pp. 869–887, 2004.

[102] D. Stroobandt, P. Verplaetse, S. Member, and J. V. Campenhout, "Generating Synthetic Benchmark Circuits for Evaluating CAD Tools," vol. XX, pp. 1–14, 2000.

[103] M. Hutton, J. P. Grossman, J. Rose, and D. Corneil, "Characterization and parameterized random generation of digital circuits," in *Proceedings of the 33rd Annual Design Automation Conference*, ser. DAC '96. New York, NY, USA: ACM, 1996, pp. 94–99. [Online]. Available: http://doi.acm.org/10.1145/240518.240537

[104] M. Hutton, J. Rose, J. Grossman, and D. Corneil, "Characterization and parameterized generation of synthetic combinational benchmark circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, no. 10, pp. 985–996, Oct 1998.

[105] B. Landman and R. L. Russo, "On a pin versus block relationship for partitions of logic graphs," vol. C-20, no. 12, pp. 1469–1479, Dec 1971.

[106] M. Bellido, *Logic-timing simulation and the degradation delay model.* London: Imperial College Press, 2006.

[107] C. Mead and L. Conway, *Introduction to VLSI Systems.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1979.

[108] T. Tang and X. Zhou, "Accurate timing simulation of mixed-signal circuits with a dynamic delay model," in *Proceedings of the Int. Workshop on Computer-Aided Design, Test, and Evaluation for Dependability (Beijing, P.R.C*, 1996, pp. 309–311.

[109] O. S. Ahmed, M. F. Abu-Elyazeed, M. B. Abdelhalim, H. H. Amer, and A. H. Madian, "Logic picture-based dynamic power estimation for unit gate-delay model cmos circuits," *Circuits and Systems*, vol. 4, 2013.

[110] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin, "Efficient fault simulation on many-core processors," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 380–385. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837369

[111] P. Cockshott, "Vector pascal reference manual," *SIGPLAN Not.*, vol. 37, no. 6, pp. 59–81, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/571727.571737

[112] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[113] W.-m. Hwu, K. Keutzer, and T. G. Mattson, "The concurrency challenge," *IEEE Des. Test*, vol. 25, no. 4, pp. 312–320, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MDT.2008.110

[114] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095421