Improving Bio-Inspired Frameworks

Aravind Krishnan Varadarajan

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Michael S. Hsiao, Chair
Haibo Zeng
Cameron D. Patterson

August 20, 2018
Blacksburg, Virginia

Keywords: GPU, Multithreading, RTL, Deep Neural Nets, Testing

Improving Bio-Inspired Frameworks

Aravind Krishnan Varadarajan

(ABSTRACT)

In this thesis, we provide solutions to two different bio-inspired algorithms. The first is enhancing the performance of bio-inspired test generation for circuits described in RTL Verilog, specifically for branch coverage. We seek to improve upon an existing framework, BEACON, in terms of performance. BEACON is an Ant Colony Optimization (ACO) based test generation framework. Similar to other ACO frameworks, BEACON also has a good scope in improving performance using parallel computing. We try to exploit the available parallelism using both multi-core Central Processing Units (CPUs) and Graphics Processing Units(GPUs). Using our new multithreaded approach we can reduce test generation time by a factor of 25× compared to the original implementation for a wide variety of circuits. We also provide a 2-dimensional factoring method for BEACON to improve available parallelism to yield some additional speedup. The second bio-inspired algorithm we address is for Deep Neural Networks. With the increasing prevalence of Neural Nets in artificial intelligence and mission-critical applications such as self-driving cars, questions arise about its reliability and robustness. We have developed a test-generation based technique and metric to evaluate the robustness of a Neural Nets outputs based on its sensitivity to its inputs. This is done by generating inputs which the neural nets find difficult to classify but at the same time is relatively apparent to human perception. We measure the degree of difficulty for generating such inputs to calculate our metric.

Improving Bio-Inspired Frameworks

Aravind Krishnan Varadarajan

(GENERAL AUDIENCE ABSTRACT)

High-level Hardware Design Languages (HDLs) has allowed designers to implement complicated hardware designs with considerably lesser effort. Unfortunately, design verification for the same circuits has failed to scale gracefully in terms of time and effort. Not only has it become more difficult for formal methods due to exponential complexity from increasing path explosion, but concrete test generation frameworks also face new issues such as the increased requirement in the volume of simulations. The advent of parallel computing using General Purpose Graphics Processing Units (GPGPUs) has led to improved performance for various applications. We propose to leverage both the multi-core CPU and the GPGPU for RTL test generation. This is achieved by implementing a test generation framework that can utilize the SIMD type parallelism available in GPGPUs and task level parallelism available on CPUs. The speedup achieved is extracted from both the test generation framework itself and also from refactoring the hardware model for multi-threaded test generation. For this purpose, we translate the RTL Verilog to a C++ and a CUDA compilable program. Experimental results show that considerable speedup can be achieved for test generation without loss of coverage.

In recent years, machine learning and artificial intelligence have taken a substantial leap forward with the discovery of Deep Neural Networks(DNN). Unfortunately, apart from Accuracy and FTest numbers, there exist very few metrics to qualify a DNN. This becomes a reliability issue as DNNs are quite frequently used in safety-critical applications. It is difficult to interpret how the parameters of a trained DNN help store the knowledge from the training inputs. Therefore it is also difficult to infer whether a DNN has learned parameters which might cause an output neuron to misfire wrongly, a *bug*. An exhaustive search of the input space of the DNN is not only infeasible but is also misleading. Thus, in our work, we try to apply test generation techniques to generate new test inputs based on existing training and testing set to qualify the underlying robustness. Attempts to generate these inputs are guided only by the prediction probability values at the final output layer. We observe that depending on the amount of perturbation and time needed to generate these inputs we can differentiate between DNNs of varying quality.

Dedication

Dedicated to my family.

Acknowledgments

Most importantly, I would like to thank Dr. Michael S. Hsiao for taking me under his wing. I had taken 3 courses under him during my first year as a masters student all of which greatly encouraged me to work with him. His course on Verification covers ideas that can be applied across all domains and is one of the courses that I enjoyed the most. I would also like to thank Dr. Haibo Zheng and Dr. Cameron D. Patterson for their kindness in agreeing to serve on my thesis committee. I am also grateful to my ECE Advisor Mrs. Susan Broniak for her support in helping clear any doubt I had related to paperwork throughout my duration of my stay at Virginia Tech. I also thank to the staff of Graduate School for their timely help regarding any administrative issues or questions I faced.

Aravind Krishnan Varadarajan

Blacksburg

August 20, 2018

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Heuristic and metaheuristic algorithms have been increasing in popularity in the last few decades. This is a result of problems increasing in both size and complexity, making many formal and deterministic methods infeasible for large problem sizes due to its exponential time complexity. A common category in heuristics is bio-inspired algorithms. This, as the name indicates, are a naturally occurring phenomenon that has been adapted to solve computer problems. In this thesis, we try to address two issues that many current bio-inspired algorithms face.

- Time: Although these approaches do not scale exponentially with problem size they do tend to take more time as the size increases. The runtime is generally not directly proportional to the problem size. This is evident in heuristic test generation problems, where small but complex circuits might require the algorithm to run for longer in order to produce a satisfactory solution.

- Reliability: Heuristic methods usually do not guarantee an optimal solution or sometimes don't even converge to any solution. There are also questions on reliability concerning reproducibility and quality. This is because of many approximations and the random nature of the approach itself. Given that they are employed in safety-critical applications, some measure of quality or robustness is required to qualify them. An example of such a scenario is Deep Neural Nets were it is possible to generate adversarial

inputs in order to confuse DNNs.

1.1 Contributions of the Thesis

1.1.1 Part 1

Modern day processors and SOCs have grown extremely large in transistor count due to increased design complexity and reduction in transistor size (Moore's Law scaling). The increased circuit complexity and size directly impact design time and effort. A significant chunk of this resources required is taken up by Verification. Estimates suggest that verification takes up to 70% of design time and effort and up to 80% of non-recurring engineering cost [33]. This highlights the need to reduce verification cost in order to reduce design time and to allow innovation in VLSI design accelerate. Verification of hardware models is done during various stages of development. One commonly used metric to evaluate the quality of the verification effort is code coverage or branch coverage. Not only does branch coverage give confidence that all the hardware modules and states have been activated at least once by the underlying input sequence, but it also gives us a test suite that can help us target any particular part of our hardware model.

Unfortunately, test generation for branch coverage is a complicated problem due to complex path constraints in order to reach hard to reach branches and regions in the design. Thus, all traditional methods of test generation, including formal, concrete and concolic (concrete + symbolic), take large amounts of time for large designs. This is especially true for the formal methods which rely on techniques such as Satisfiability (SAT) which is known to have exponential complexity. Both concolic and concrete approaches use simulation to aid test generation. Though hardware simulation scaled better as compared to SAT, size and

complexity of modern designs have increased to a point where simulation of even moderate SOC designs would take multiple hours to complete. Conclic and concrete techniques usually require many iterations of simulation in order to converge at a final test sequence. Therefore these methods also have issues regarding scalability.

Although increased problem size has been the trend over the last three decades across all domains of science, the issues due to the problem size have surfaced only over the last decade. This is as a result of the recent failure of Dennard's Law, which prevents scaling of core frequency/power cannot continue. Modern designs have hit the frequency and power wall which makes it difficult to achieve increased performance from decreased transistor size alone. Therefore, the most commonly used method has been coming up leveraging the increasingly common parallel computing resources. This can be done by either leveraging parallelism already existing in the solution or refactoring the solution to expose parallelism.

Unfortunately, it is hard to fit hardware simulation into the parallel programming model due to the high amount of synchronization and dependencies. Thus hardware simulation is yet to reap the performance benefits from parallel programming advances over the past decade. Given that simulation forms a significant component of test generation, it has struggled to scale with design sizes. Therefore, utilizing parallel computing by extracting parallelism from the verification and test generation framework itself may be more beneficial than merely relying on improving overall performance through improvements in simulation alone. This includes many concrete simulation-based test generation frameworks which usually have multiple independent simulations which can be run in parallel.

The two most commonly used platforms in parallel computing are multi-core Central Processing Units(CPU) and Graphics Processing Units(GPU). For parallelism on the CPU, the application only needs to have independent tasks. This is because each processing unit usually has its dedicated register space, caches, etc. As a result, tasks can be scheduled to

execute on any of these units without any restrictions. Values are shared among tasks using a shared memory space, like a shared cache, or by passing messages to each other using a message passing mechanism.

On the other hand, GPUs impose more requirements on an application in order to deliver performance benefits. A few independent tasks are insufficient. Parallelism must be present in the order of thousands of threads to achieve speedup. The kernel must also exhibit other characteristics such as regular memory access. This is because cores in a GPU execute in an SIMD fashion and groups of cores share register space, caches and also the global memory. Thus it is not possible to assign threads to cores in an arbitrary fashion due to issues such as thread divergence and coalescing memory access.

In this thesis, we first present a framework to automatically convert synthesizable RTL Verilog to native GPU binary for test generation. This along with a test generation framework, BEACON [24], runs entirely on the GPU without any support from the CPU. Frameworks such as BEACON are a good suit for a multi-threaded approach as it exhibits a good amount of parallelism. Also, the amount of work that can be made parallel can be scaled with the increased availability of compute resources. Thus BEACON can utilize architectures such as GPUs with extremely high core count.

Parallelism in BEACON is present in the ACO where there are ants which can explore the design independent of each other. Besides, we have also tried to extract parallelism from the exploration of the individual ants. The methodology used to extract this twofold parallelism is described as a part of this thesis.

In order to allow for BEACON to run on the GPU, the algorithm has to be slightly tweaked. This is because in order to leverage performance from the GPU many criteria have to be met. These include memory coalescing, occupancy, etc. We go over the issues faced when

porting the test generation framework as a whole to the GPU.

Lastly, given that we have implemented a parallel version on both the CPU and GPU we seek to offer a comparison of sorts between the returns from each platform. We hope that this gives an overall idea about how implementations such as these would scale on different platforms.

1. We present a framework to automatically convert synthesizable RTL Verilog to native GPU binary for test generation purpose along with a test generation framework, BEACON [24], which also runs on the GPU itself.

2. We discuss the challenges and benefits that come along with trying to port the RTL concrete test generation framework to Graphic Processor Units.

3. We offer a comparison between a parallel CPU vs. GPU implementations of the same framework.

We include a short discussion on thread divergence, register pressure, instruction to byte ratio and other metrics in order to understand what chokes the performance on GPUs.

1.1.2 Part 2

As a part of this thesis, we also present a new test-generation-based approach metric for evaluating DNNs. Neural networks are a bio-inspired structure to mimic the behavior of neurons in human brains. It is a collection of neurons connected in layers where a neuron in one layer gets as input a weighted response from each of the neurons from its preceding layer. Neural nets are commonly used in artificial intelligence for classification. For a given input, it triggers a neuron in it's output layer to indicate class belongingness.

There exists a training process to make DNNs functional through the process of decreasing loss and thereby increasing accuracy. Unfortunately, these are limited to the response of the DNN to the training dataset alone. Given that most of the DNNs are used in real time and in real-world applications where the input data set might not always be identical to the training dataset, it is critical to ensure that the DNN still produces an acceptable response. Unfortunately, there exists few metrics to qualify a DNN for this purpose automatically. Metrics such as Accuracy and FTest only serve to highlight the effectiveness of the DNN or the training methodology for a given training/testing dataset.

We try to qualify the robustness of the DNN by measuring its response to a wide variety of inputs. A good DNN must be tolerant to small noises and perturbation on its inputs and at the same time avoid classification of samples that are extremely noisy or samples that in no way resembles any classes. It is possible to statically determine the exact input space for a given classification response. However, this assumes access to the values of all the internal parameters of the DNN. Additionally, such a method assumes that that entire input space is a possible real-world scenario. For most applications, this is not true.

In our work, we use a black box approach to tackle this problem. We use a test generation technique to generate new test inputs using the prediction probability values at the output layer and the degree of perturbation as a measure of fitness. The two metrics we define are Max Distortion(Max_D) and Min Distortion(Min_D). Max_D tries to increase the difference between the generated test sample and available training sample while keeping the output of the classifier constant. Min_D tries to force the DNN into misclassifying a test sample while minimizing the delta between the training and the test samples.

1. We present metrics to measure the reliability of DNNs regarding tolerance to varying input patterns.

2. We present an automatic test generator that generates inputs for a given DNN in an attempt to measure the above metrics.

3. We implement the above on a widely used digit recognition database, MNIST, and present our results.

1.2 Thesis Organization

The rest of the thesis is organized as follows.

Chapter II discusses the background for our test generation approach for both RTL Verilog and DNNs. We briefly explain the outline of GPUs and simulation-based verification. We also provide background on how to make parallel a single RTL simulation. For our work on DNN, we provide a brief background on the structure of a neural net and attempts of recent work in trying to test the same.

Chapter III describes the proposed methodology and presents the results for a parallel RTL test generation. We qualify the effectiveness of the same using a parallel implementation of BEACON evaluated on the ITC 99 [9] benchmarks and commonly used open core designs.

Chapter IV describes the methodology used to test DNNs. We provide a discussion on the metrics used to describe the robustness of a DNN. We also present results from the implementation of our work on MNIST.

Chapter V concludes the paper. We present our learnings from our parallel implementation of the test generation framework. We highlight the existing potential and also changes in the simulation framework and the underlying computer architectures that can help to increase performance. We also provide few closing comments on our work in testing DNNs on its potential effectiveness and drawbacks.

Chapter 2

Background

In this chapter we provide the fundamental concepts used in our work. The first part of this section provides relevant background for our work to improve performance of RTL test generation. The second part covers the basics needed to understand DNNs and the idea of reliability and accuracy of a DNN. We also briefly look at all the recent relevant work pertaining to both.

2.1 RTL Test Generation

In this chapter, we provide the fundamental concepts used in our work. The first part of this section provides relevant background for our work to improve the performance of RTL test generation. The second part covers the basics needed to understand DNNs and the idea of reliability and accuracy of a DNN. We also briefly look at all the recent relevant work on both.

2.1.1 Automatic Test Generation

Verification of Hardware Models is performed to ensure that the behavior of the model is consistent with its specification. This means that for any given input, the output of the hardware should be inline with its functional model. In our work, we focus on generating tests

for RTL models for improved branch coverage. Branches in RTL refers to all the *if then else, case* statements that cause the flow of execution to diverge based on the guard statements. Since hardware described inside these branches gets synthesized to gates, coverage of these statements translates to coverage of the gates itself. Further, 100% branch coverage also indicates that every control state in the design is exercised.

Most of the test generation techniques used for hardware testing involves recording the design response with test stimulus. This includes both concolic and concrete methods of testing. Concrete methods are those which involve little to no symbolic analysis of the design and rely mainly on simulation traces to guide test generation. Concolic methods rely on interleaving concrete simulation traces and symbolic analysis. These methods usually simulate the circuit for a limited number of cycles and use the obtained traces along with symbolic constraints of the traces in order to generate inputs for the next round of concrete simulation. Some examples of simulation guided formal methods include [25][32][41]. We can see that both the methods rely on some form of design exploration using a variety of inputs. Naturally, the more the number of input sequences used, the higher the possibility of covering more of the design. However, the number of inputs sequences is limited by the time taken to evaluate the same. With the increase in both design size and complexity, this is turning out to be an issue as:

- Larger the design size, more time taken to evaluate the circuit for a given sequence.
- More complicated the design, more the number of sequences and more iterations of the verification flow needed in order to generate a good test sequence.

In our work, we try to leverage existing parallel computing techniques in order to evaluate more sequences faster to facilitate better design exploration.

2.1.2 Graphics Processing Units

Graphics Processing Units (GPUs) have existed from the 1970s where it was first used to help draw images for video games. From then on till the mid-1990's they were mainly used to render 2D images on the screen. Around this time API based approach along with libraries to use GPUs was starting to become popular. These were widely used from video games even to help operating systems such as Windows and Mac to render its displays. Around this time even 3D rendering was becoming more popular, but most of the workload on the GPU still consisted of video and image processing. From the 2000-2010 is when General Purpose GPUs or GPGPUs started becoming more prevalent. GPUs, from the beginning, was designed for a particular set of operations such as ray tracing or other image and video processing applications. These fall into the category of SIMD or vectorized instructions where a single instruction is applied to a block of data.

Looking at its performance and power efficiency as compared to the CPU many scientists have started exploring ways to utilize GPUs for their application. Over the recent years, many embarrassingly parallel programs have been ported to the GPU leverage these benefits. Even the manufacturers have started to provide C/C++ and directive-based approaches to program the GPUs like a streaming processor to let users leverage the GPUs better. Manufacturers have also gone to the extent of modifying their processor pipelines in order to allow for better general purpose programming.

GPUs differ from traditional Central Processing Units (CPUs) in that they are composed of hundreds of simple processing units as opposed to a small number of high-performance CPUs. This difference is highlighted in Fig 2.1 [29].

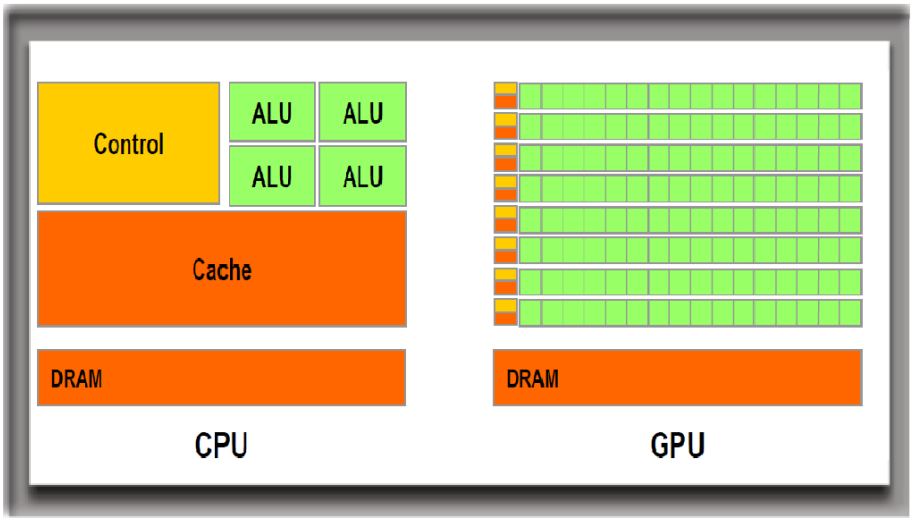

Figure 2.1: Block diagram of CPU and GPU organization.

The primary building blocks of a GPU are the Streaming Processors (SPs). 32 SPs are organized as a warp. These are the smallest units of execution, i.e., groups of 32 SPs execute the same instruction in a Single-Instruction-Multiple-Data (SIMD) fashion. A collection of warps makes a Streaming Multiprocessor (SM). A GPU is composed of multiple SMs. Each SM has its own cache and shared memories. Given that blocks of processors execute the same instruction, the control logic is much simpler as compared to a CPU.

In the CUDA [28] programming model, each GPU kernel is organized as a grid which specifies the number of blocks and the number of threads per block. An SM can run multiple blocks at the same time, but a block cannot span different SMs. Blocks run till completion without being swapped out, and the CUDA runtime scheduler launches a block on an SM only after securing all the resources required. Threads running inside a block can synchronize and pass values to each other through the shared memory. Threads running on different blocks have no way to synchronize with each other and have to write values to the off-chip global memory to pass values. GPUs are designed for high-throughput, and the high GFLOP/sec achieved is as a result of the huge number of Compute units and clever latency hiding and

fast context switching.

The number of threads per block and the number of blocks that can run in parallel can change depending on the kernel. Although various combinations of blocks and threads per block can be launched, the actual number that runs in parallel is governed mainly by registers per thread, shared memory used per thread block and the total number of threads per block. There is a hard limit on all these resources, and the total number of threads that actually run in parallel depends on whichever limit is hit first. It is always possible to estimate the theoretical occupancy using the occupancy calculator provided by the vendor.

Traditional GPU kernels have a total number of threads much greater than the number of SPs in order to leverage this. Other features that improve GPU performance is regular memory access patterns, reduced branching, data reuse between threads and a good mix of floating point and integer instructions. The CUDA profiler for NVIDIA GPUs lets us measure a huge variety of metrics that can help us understand the behavior of a kernel on a GPU. Below are a few:

- Occupancy: Occupancy is the term used to describe the ratio of time spent by warps doing work against the total runtime of the kernel. The reasons for warps stalling can range from unoptimized launch bounds (block and thread count) to stalls due to data unavailability.

- Instruction to Byte ratio: This metric can determine whether an application is compute bound or memory bound. It is the number of arithmetic instructions per byte of data fetched from memory. An ideal instruction to byte ratio is around 3.7:1.

- Instructions per warp: The Instruction to Byte ratio is a theoretical estimate. A more accurate way of measuring whether the kernel is compute or memory bound is by profiling during runtime. The Instructions per warp metric tracks the number of

instructions issued at warp level each cycle. The max Instructions per warp is 2.

- DRAM utilization: The DRAM utilization measures the bandwidth utilization against the max on a scale of 10. Based on the values of DRAM utilization and Instructions per warp we can determine whether the kernel is bounded by memory or compute.

- Branch Efficiency: When threads of a warp try to execute different instructions, one set of threads stall while the other executes.

- Memory Coalescing: When consecutive threads access consecutive addresses of DRAM, GPUs recognizes these patterns and serve these requests in a single memory transaction. Otherwise, it might require multiple transactions per warp which will drastically increase stalls not only on the warp that requests the data but also on other warps due to increased load on DRAM.

2.1.3 BEACON

The test generator we are trying to integrate with our GPU simulation engine is BEACON [24]. BEACON is an ant colony optimization based test generator at producing vectors to improve branch/code coverage. Branch coverage is considered an important verification metric as it indicates how much of the hardware design has been activated by our input stimulus. A 100% coverage, apart from providing confidence that all of the code along with its assertions have been exercised, also helps state justification which is used heavily in ATPG. Even though bio-inspired test generators tend to generate long test sequences, existing fast methods for compacting the resulting test sets exist [17].

In ACO, the problem is represented as a graph with vertices and edges between the vertices. Each vertex is a branching point in our RTL, and all the edge leading out of a vertex are

all the possible divergence or branches possible from a given branch statement in RTL. The edges are directed. In BEACON, each ant is described by a test vector sequence. Once the RTL is converted to its functionally equivalent C++, the start and end vertices are defined using the corresponding function's entry and exit points.

Each cycle of evaluation of the design consists of an ant traversing this graph starting at the *start* node and finishing at the *end* node. We also record the best branch uncovered by an ant at each cycle. The path taken by an ant for a given cycle depends only on its input vector for that cycle and the current state of the circuit. An ant with N cycles traverses this graph N times and records the number of times each branch has been traversed. This is then repeated for all ants. Once all the ants have finished their execution, we go back and update the graph or the *pheromone map* based on the total number of times each branch has been covered by every ant. Since we would like to direct test generation towards uncovered or hard to reach branches, the pheromone value for a branch is inverse proportional to the number of times it has been hit. Once the *pheromone map* has been updated each ant re-evaluates its inputs by going through its recorded design response identifying at which cycle it was able to uncover the hard to reach branches. Then it regenerates its inputs to continue exploration from that point onwards. Fitness is assigned to each ant based on the number of hard to reach branches covered by it. This helps us promote ants with sequences that cover hard to reach branches.

From the above description, we can see that in BEACON there is no dependency between the simulation of different ants. Therefore, this can be done in parallel with no synchronization. The only values the ants collectively modify is the *pheromone map*. However, here too there is no restriction on the order in which the ants update it. Thus we can make this update into an *atomic* operation to avoid race condition and still preserve the final value of the *pheromone map*. After this update, each ant can go on to evaluate its sequence to improve

them for the next round. We could go one step further here by not synchronizing the ants at the end of the update. This would imply that for some ants the *pheromone map* may not adequately reflect the results of all ants for a particular round. However, given that the other ants are going to update it in time for the next round and also because this is a heuristics based approach, it does not cause any issues overall. This allows for far fewer synchronization constraints. Thus we can see that there is a sufficient amount of parallelism to be exploited to warrant a multi-threaded approach.

Algorithm 1 BEACON

```
1: procedure GEN_TEST(num_cycles,num_rounds)
2:     ants ← Initialize_ants()
3:     ph_map ← Initialize_map()
4:     circuit ← Get_circuit_instance()
5:     for each num_rounds do
6:         for each ant in ants do
7:             circuit.Reset()
8:             for each cycle in num_cycles do
9:                 circuit.Eval(ant[cycle])
10:                capture_response(ant, ph_map, circuit)
11:            end for
12:        end for
13:        for each ant in ants do
14:            Update_ph_map(circuit, ph_map)
15:        end for
16:        for each ant in ants do
17:            Evolve(ant, ph_map)
18:        end for
19:    end for
20: end procedure
```

Parallelism on GPUs involves calling a kernel on a GPU from the CPU. Frequent calls to a GPU kernel can add up due to kernel call overhead and data movement between the CPU and the GPU on the PCIE bus. By porting BEACON to the GPU we hope to avoid the overhead of synchronization between GPU and CPU. Algorithm 1 shows the algorithmic flow in BEACON. In this thesis, we extract parallelism from both the test generation algorithm

and also the simulation.

2.1.4 Factoring of a single simulation

It is sometimes possible to find parallelism from just observation of the hardware model from a single simulation if the operation of the hardware model has a pipeline like nature. However, usually, this is not the case. We use Chandy Misra Bryant (CMB), similar to Qian and Deng [34], like approach to break down hardware designs to help us make simulation parallel. CMB is a distributed time discrete event simulation. Here, the overall design is broken down into smaller units usually known as logic processes. Each logic process has its own inputs event queue and internal time stamp. Each logic process reads the value from its input queue with the least time stamp and generates an output with the same time-stamp. This continues until there are no more inputs in the input queue and the process stalls until new inputs arrive. Typical CMB implementation span cores in different nodes even geographical locations.

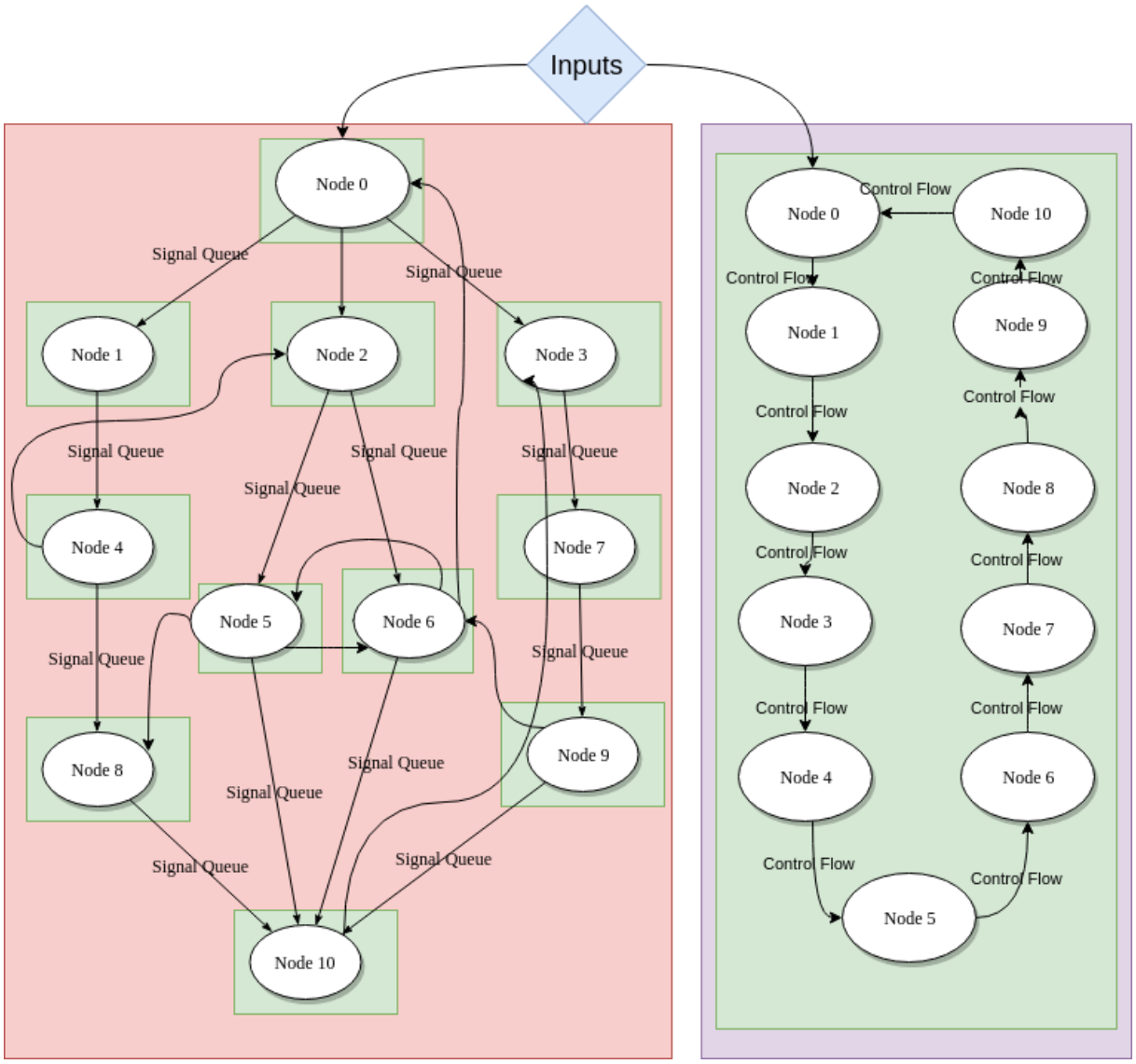

Figure 2.2: Illustration of regular (right) vs distributed (left) model.

Fig 2.2 illustrates the difference between a regular model and a CMB model. The model inside the green box executes all the functions on a single machine in a serial fashion until all its inputs are exhausted. The arrows between nodes here show the control flow. As all the variables exist on the same machine, there is no need for any message passing. The model inside the red box represents the distributed model. Here each green box represents different compute units and they can execute in parallel and pass values to each other as shown using

the arrows. Deadlock is avoided by assuming initial values wherever needed.

Hardware simulation is extremely time-consuming, but the difficulty in exposing parallelism dampens efforts to make it faster using multicore. Most of the modern design follow core-based design, which is a modular plug and play approach to designing hardware for ease of development. So intuitively we can see computations needed to simulate different modules can be made parallel with some synchronization. We hope to use CMB to naturally expose this independence by breaking the design into as small workloads as possible and let each one run at its own pace.

However, this, unfortunately, comes at the cost of increased overhead. To implement such a model on a GPU, we would need to define queues, double buffer signal values, and store values of signals and coverage values across a few timestamps as different logical processes are executing across different time-stamp. There is also the issue of thread divergence. If different threads are to represent different logical processes, then they cannot be scheduled within the same warp. This is because threads in a warp execute in a SIMD fashion. Given that the minimum size of a warp is 32 threads, we end up to wasting a significant amount of compute resources as we have to restrict ourselves to one thread per block. However given that we have SIMD parallelism from our test generation framework, we can design a system on the GPU to use parallelism from both the design and the test generation framework.

The idea behind using CMB for logic simulation is to increase thread count to improve latency hiding on the GPU. The SIMD type of workload we are extracting comes only from our test generation framework as a single logic simulation at the RTL level does not involve SIMD type workload.

2.1.5 Verilator

We use Verilator [38] to help us convert RTL to Cuda. Verilator is a cycle accurate open source code based simulator that can convert synthesizable RTL Verilog to C++. Internally Verilator performs synthesis of the Verilog and thus collapses redundant signals and flattens out the design while generating the equivalent C++ version. Verilator does not just conduct source to source translations; it also generates highly optimized C++ wrapper for the design. Finally, Verilator also automatically instruments the generated C++ code with branch coverage counters.

Listing 2.1: Sample Verilog to verilator translation

```
module greatest(in1, in2, out1, out2, clk, rst);
  input [3:0] in1, in2;
  input clk, rst;
  output reg [3:0] out1, out2;

  always @(posedge clk) begin
    if(rst == 1'b1)begin
      out1 <= 0;
      out2 <= 0;
    end
    else begin
        if(in1 > in2) begin
            out1 <= in1;
            out2 <= out1;
        end
        else begin
          out1 <= in2;
```

```
        out2 <= out1;
      end
    end
  end
endmodule
```

Listing 2.2: Verilated(C++) Verilog

```
    // Variables
    VL_SIG8(__Vdly__out1,3,0);
    //char __VpadToAlign5[3];
    // Body
    __Vdly__out1 = vlTOPp->out1;
    // ALWAYS at greatest.v:7
    if (vlTOPp->rst) {
   __Vdly__out1 = 0U;
   vlTOPp->out2 = 0U;
    } else {
   if (((IData)(vlTOPp->in1) > (IData)(vlTOPp->in2))) {
       __Vdly__out1 = vlTOPp->in1;
       vlTOPp->out2 = vlTOPp->out1;
   } else {
       __Vdly__out1 = vlTOPp->in2;
       vlTOPp->out2 = vlTOPp->out1;
   }
    }
    vlTOPp->out1 = __Vdly__out1;
```

```
}
```

Verilator provides us a C++ source file. This not only helps us to understand more about the code we are trying to optimize but it also helps to simplify the circuit implementation by removing features unused by our application.

Since Verilator flattens the design during the transpiling process, the generated C++ code is designed for sequential execution. Given that this is a code based simulator and not an event-based simulator, the order in which the models are flattened into its final C++ form automatically exposes the dependencies between them. Verilator generated code that can be represented as a Directed Acyclic Graph. Each cycle of evaluation is a call to this graph, and thus in one cycle of evaluation, the dependency never flows backward. We use the above information in helping us to generate logic processes and analyze the dependencies between them.

Given that Verilator provides us with a class wrapper to the circuits with many features what we do not use, we copy their *eval* API alone along with the necessary variable declaration into a much simpler *eval* function which preserves the original semantics. This makes it much easier to handle on a GPU.

2.1.6 Related work

There has been work in adapting EDA problem to multi and many-core architectures, most of them around the 2010 time frame. The most common adaptations have been for simulation. This includes both RTL, gate level and even higher level languages like System C. The main issue for parallel simulation is identifying elements that can run in parallel. Compared to RTL, it is easier to identify these in gate level. One commonly used technique is levelization

of the gate level description and running the gates at the same level in parallel. This has been done by Min and Hsiao [26], Gulati and Khatri [12]. Another commonly used technique in the event-based modeling of the gate level netlist. Here gates are aggregated into a bigger module with a common sensitivity list. Whenever one of the inputs of the module changes, the module is scheduled for execution. Modules can execute at the same time as long as their interfaces are double buffered to avoid a race condition. Thus for a single cycle of simulation, various groups of gates are evaluated in parallel and evaluated efficiently due to its even based nature. This technique is used by Chatterjee et al. [7] and Chatterjee et al. [6]. In [35] the authors use an And Inverted Graph(AIG) representation of the whole circuit and perform simulation the GPU. Using a cycle based simulator and the AIG representation they leverage the available low latency memory on GPUs for performance. There has also been a Distributed Event Discreet Time approach to simulation at both the gate and RTL level as seen in and [34] and [42]. By evaluating across both time of the simulation and space of the circuit, they can expose more parallelism than what is usually available. Other notable results include [27] which maps translated System C description of hardware onto different threads on the GPU.

All the above-described approaches suffer from one of the two issues listed below.

- Difficulty in exposing parallelism: Hardware simulation usually involve large amounts of complex define use dependencies. This is especially true for RTL where the description is at a much higher abstraction level.

- Difficulty in adapting to GPU framework: Regular kernels need to be modified in order to organize the memory operation and instructions into a fashion that fit the GPU architecture. Memory accesses have to, and branch divergence has to be reduced. This again is more important for RTL and more abstracted models. Both [27] and [34] avoid

using more than one thread per threadblock. This leads to around 32× loss in resource utilization.

- Automation of task generation and load balancing across all threads is also a serious issue.

As simulation is mostly used for the sake of verification and test generation, researches started exploring ways to leverage GPU's performance for the application as a whole instead of for just simulation. This provides an advantage as not we no longer are restricted to expose parallelism from the simulation alone. As discussed earlier, since most verification methods tend to use multiple independent simulations, we leverage parallelism from the framework too. Along this line there has been a few attempts such as [12], [19], [23] and [26]. All four of these are parallel implementations of gate level verification frameworks for fault testing. They run threads which test the circuit for both different faults and different sequences. These are variations of Parallel/Serial Faults, Parallel/Serial Sequences approaches. It is interesting to note that they do not attempt to make parallel the simulation itself. Gate-level simulation exhibit similar and repetitive compute formats which are highly preferred in GPUs. They make efficient use of texture memories as seen in [26] and can easily reduce thread divergence as only the values of each gate differ across different threads while the order in which the gates are evaluated remain constant.

While there has been parallelization attempted for gate-level test generators such as [15, 16, 20], there has been very few RTL level test/verification framework that has been made parallel. One of them include [4]. Here a parallel fault, parallel sequence approach is used for generation of test vectors. Unfortunately, the effort to generate GPU code is not automated, and their methodology suffered from kernel size issues. They were unable to scale their efforts for larger circuits. Moreover, the effort to convert the hardware description for CUDA is

not straightforward. In our work, we automate this process using Verilator. Since branch coverage does not have an equivalent "Parallel Faults" dimension, we attempt to extract parallelism from the simulation also. We also provide a comparison with a parallel CPU implementation.

We also note that not all of the above works provide a comparison between their GPU implementation and its corresponding CPU serial implementation. Some of them only report speedup against their commercial implementations. So the amount of speedup obtained by porting to GPU alone remains in question.

2.2 Testing DNNs

2.2.1 Deep Neural Nets

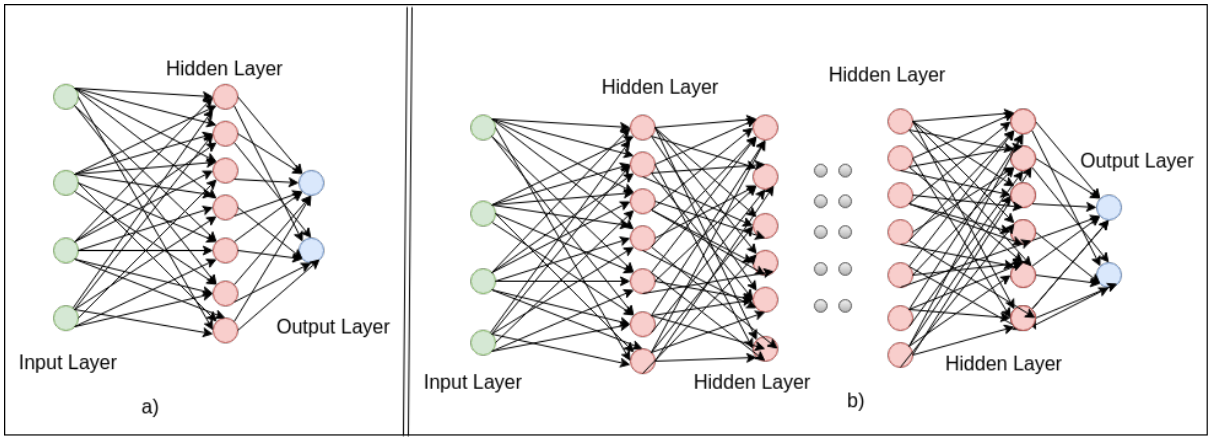

Figure 2.3: Simple Multi Layer Perceptron a) vs Deep Neural Net b).

To understand a Deep Neural Net, we need to start at a simple Neural Net. Neural Nets designed to imitate the function of the network of neurons and nerve endings present in human brains. Neural nets consist of sets of neurons and connections between them. The

neurons are arranged in layers as shown in Fig. 2.3. Each Neuron takes as inputs the weighted values of all the neurons from its immediate preceding layer. It then accumulates this value and depending on its activation function assigns itself a value based on its input.

Assume that there are a total of L layers where the layer l has n_l neurons. The value of the x, the accumulated input, for the the k^{th} neuron on layer l can be computed using the

$$x = \sum_{i=1}^{n_{l-1}} w_{lki} * val(i_{l-1}) \tag{2.1}$$

where w_{lki} denotes the value of the weight between the i_{th} neuron of the $l-1$ layer and the k_{th} neuron in layer l.

The activation function is usually a non-linear function which governs whether a particular neuron is *on* (value > 0) or *off* (value ≤ 0). Commonly used activation functions including *RELU*,

$$val(n_l) = max(x, 0) \tag{2.2}$$

Softmax,

$$val(n_l) = 1/(1 + 1/e^x) \tag{2.3}$$

where x is the accumulated input and $val(n_l)$ is the value of Neuron n in layer l.

The value at the input layer is fed directly by the user. The number of neurons in the input layer is usually the same as the number of inputs. The number of neurons in the output layer is the same as the number of classes for classification, allowing for the output layer to be one hot encoded. This means that one of the output values is 1 and all the other outputs are 0. This encoding is determined by looking at the values of the neurons in the previous layer and identifying the neuron with the max value. They are then normalized so that they represent the probability of class belongingness.

The choice of the activation function, the values of each of the internal weights and also the number of neurons in each layer is a choice that is left to the user. It is up to them to decide a value for these in order to make sure that the NN predicts the correct class belongingness as output.

The process of deciding the values for the internal weights of the NN is known as training. During the starting of this process, the weights are initialized with a random value. Then the NN is provided with an input, and the difference in the output between the expected and the obtained value is used to adjust the values of the weights to reduce this difference. This difference is usually referred to as *loss*, and the objective of the training process is to minimize this. This process is repeated for multiple inputs over multiple iterations until the user is satisfied with the performance or until the performance has saturated. This is usually done using Backpropagation, where a gradient is computed in order to determine how to change the weights to reduce loss. This falls under the supervised category of learning. More about this can be found in [13].

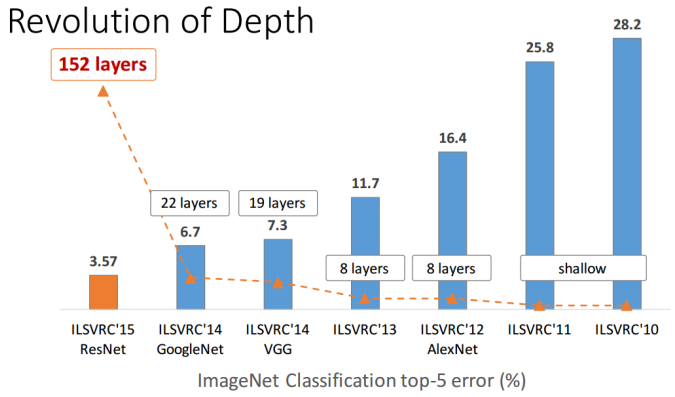

Figure 2.4: Error vs Depth

All the above definitions remain the same for both DNNs and NNs. In fact, for all practical purposes, a DNN is just an NN with more hidden layers. Anything that is possible to express

using a DNN is also possible to represent using an NN as proved by Cybenko [10]. However, in practice it has been observed that with increased depth there has been an increase in accuracy as seen in 2.4 from [14]. Although how exactly increased depth helps in increased performance remains in doubt. This ambiguity can be traced back to the ambiguity in how the learned information is encoded in the trained weights. This is especially true for DNNs where it is difficult to trace how different neurons are affected by input features. Thus there are very few ways to cross-verify whether whatever that has been learned by a DNN is what was expected. Still, due to the effectiveness of DNNs, the ambiguity does not stop researchers from using them extensively. In place of verifying what has been learned by neural nets in order to judge reliability, metrics such as accuracy, loss, confusion matrix and FTest. These metrics only provide data on the performance of the DNN on the available input testing and training data.

Since it does not make sense to train and test a DNN using the same dataset, the available samples are divided into training and testing samples. The training samples are used to train the values of its internal parameters while the testing sample is used to compute the accuracy, FTest, and the Confusion Matrix. Accuracy is computed as

$$accuracy = Number\ of\ samples\ classified\ correct \times 100 / Total\ number\ of\ testing\ samples \tag{2.4}$$

The confusion matrix is the table representation of all the correct classification and misclassifications. Both the rows and the columns of a table represent all available classes. The value of an entry in row r and column c of a table represents the number of samples from class r misclassified as class c.

Testing what has been learned by the parameters during training can be done empirically by observing the change in neuron values and the outputs for a variety of given inputs. It

has been observed that layers in the middle of a DNN can learn complex features than the rest of the layers as observed in [40] and [30]. Thus it makes sense to monitor the excitation of these neurons for a variety of inputs.

2.2.2 Adversarial Generation

All the available metrics only measure averages of the DNN performance and gives no guarantee for a particular input. This is to be expected as it is a heuristics based approach. Nevertheless given its prevalence, especially in image processing applications, we need some form of a qualifier in order to measure how much of a reliability issue is present in a given deep neural net. There have been examples of many adversarial pattern generation that highlight this issue. These methods usually seek to generate an input that a human can classify with little to no ambiguity but a well trained neural net with 95%+ accuracy still has difficulty classifying. There have been both black box methods, which use only the value at the final output layer of the DNN, and white box approaches which can look at all the neuron values to aid input/test generation. These frameworks use a range of techniques from modeling the DNN as an equation and solving them with constraints to using an adversarial DNN to generate inputs for the DNN under test. Unfortunately, none of these methods seek to test or stress the DNN during test generation. In fact, they try to generate tests with least amount of queries to model an attack on the DNN instead of trying to understand the actual limits of the DNN. Thus it ends up highlighting the quality of the Adversarial Input generation framework as opposed to highlighting the quality of the DNN.

To highlight the reliability issue of DNNs Fig 2.5 and 2.6 are examples taken from [2] and [5] respectively. They attack DNNs with high classification accuracy and show that they are easy to fool. By altering a few pixels in these images, it was sufficient to fool the DNNs to

Figure 2.5: Meerkat as Doormat

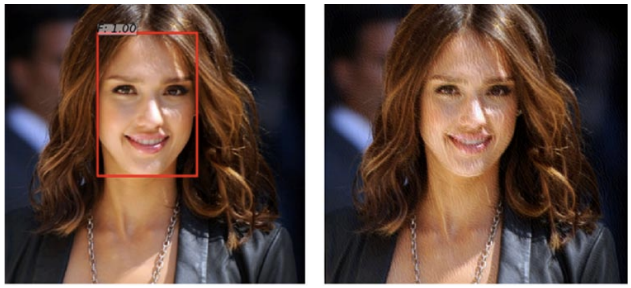

Figure 2.6: Face as not Face

misclassify the images. In our work, we want to ensure that if such manipulations alter the classification results, the proposed metrics will capture it.

2.2.3 Testing Neural Net and related works

There has been very little work on testing of Deep Neural Nets albeit a good number of attempts at adversarial input generation such as [3]. This is because of difficulty in coming up with a *fault* model for DNNs. Given that it is based on heuristics, it is difficult to pinpoint a particular fault as the issue as they are not expected to work at 0% error rate. In addition to that, there have to be domain specific constraints on the test inputs in order for the generated test set to provide value.

Recent efforts in test generation for DNN employ a white box approach. They use coverage as a metric to describe the quality of the generated test. Coverage for a neural net can be based on neuron activation. A neuron is said to be covered if it has been activated at least once during the test sequence. Other test generation goals are built on top of neuron coverage include an adaptation *Modified Condition/Decision Coverage* (MC/DC)[18] for DNNs as shown in [36]. Here the coverage metrics include not only the activation of a given neuron but also try to generate inputs that can conditionally activate neurons in consecutive layers

to prove dependencies between them. The complexity of this, however, is in the order of the number of neurons pairs in the DNN. This also implies white box access to the DNN is available.

In our effort to generate tests for DNNs, we focus only on the input and the output layers of the DNN. This helps us reduce complexity in the test generation process. We hope to provide similar demarcation between DNNs of variable reliability in spite of this.

Chapter 3

Multi and Many Core RTL Test Generation

We use OpenMP 3.0 [11] for the parallel implementation of BEACON on CPU and CUDA Toolkit version 9.2 for implementing on the GPU.

3.1 Methodology

Parallelism in BEACON comes from:

- Simulation of ants, as each ant describes its test inputs completely.
- Updating the state of the Pheromone Map using reduction or atomic operations.
- Evolution of each ant for the next round as it requires no interaction between ants.

Even the initial seeding of ants can be performed in parallel but as we will see later in our results, the generation of random numbers in parallel on CPUs is not always threadsafe, and this degrades performance.

3.1.1 Parallel implementation on CPU

The implementation on multi-core CPUs is rather straightforward. OpenMP is a directive based approach to make existing code parallel. We first query the number of logical cores available for simulation. Then we request for that number of threads for our task and record how many are granted as depending on the resource sharing policy we might not get as many threads as available. We create as many instances of the circuit as the number of concurrent threads. Each ant is assigned one instance, and the values of the circuit are reset before being passed to the next ant. The response of the design to each cycle of each ant is recorded as a part of the ant data structure and not as a part of the circuit.

This is easily achieved by using the *"#pragma omp parallel for"* provided by OpenMP and assigning each ant an ID which is same as the threadID and assigning to it the respective circuit instance. Static scheduling is used as all the simulation are for the same number of cycles.

Similarly, by using the reduction pragma provided by OpenMP, we can accumulate the overall hit count of every branch in the design from all ants in parallel. This hit count can be used to assign a pheromone value for each branch. Each ant can then later compare its branch coverage values against the overall and can decide whether it is unique in the sense that it has uncovered any interesting branches.

Evolving ants for the next stage consists of each ant going through the circuit responses for each of its inputs along with the global hit counters to help it in recomputing its vectors for the next round. Again a simple *"#pragma omp parallel for"* should suffice but with dynamic scheduling as the number of cycles for which new inputs need to be generated varies for each ant.

3.1.2 Parallel implementation on GPU

Verilator captures the circuit as a Class which can be instantiated during runtime for use. CUDA allows for classes in its programming model, but a much simpler approach is possible if the BEACON runtime parameters such as a number of ants are known during compile time. We can remove the Class wrapper around the Verilated circuit and use the bare code that describes the circuit to generate CUDA compatible code. We can also statically allocate space on the device for all the circuit's internal variables or allocate them as a part of our kernel stack. We will also have to make sure that all the helper functions that are used in the Verilated C++ have an equivalent "__device__" implementation as "__host__" functions are not accessible from the GPU.

The code size for kernels is usually kept small enough to reduce register pressure. The number of registers used by each thread is important as it can determine the number of threads-blocks that can reside on an SM at any given moment. If one thread-block uses 51% of the registers available on an SM, then only one block can be scheduled on an SM at a time. Thus 49% of the SM remains unused and this drastically affects performance. Keeping the number of registers low can be achieved by using LaunchBounds in our *.cu file or using max_registers flags during compilation. This forces the compiler to uses fewer registers and thus could also adversely affect code performance as seen in [37].

For moderate to large circuits, the number of lines in the Verilator generated model tends to be in the order of thousands if not tens of thousands. This naturally increases register pressure. Thus we have implemented models to either reduce register pressure or allow high register count as a trade-off with low theoretical occupancy. In addition to these, we have implemented a model with BEACON inside GPU along with the simulation to eliminate the cost of synchronization and communication between CPU and GPU.

One or more kernel calls per cycle

All ants are simulated in parallel one cycle at a time. Moving the "for" loop that iterates over all the cycles to the CPU help to reduce the register usage considerably. The CUDA RTL model is split into multiple kernels for circuits which still use a large number of registers. This requires saving the circuit state onto to the GPU global memory every kernel call as the kernel stack is not persistent across calls.

One kernel call per round

All the ants are simulated in parallel for all cycles in one kernel call. Here the circuit signals can be kept thread local as the kernels run to completion of the whole simulation. The trade-off is between possible increased performance due to more registers allocated per thread vs. performance decrease due to theoretical lesser occupancy.

Algorithm 2 BEACON_GPU

```
    global ph_map ← Initialize_map()
 2: procedure GEN_TEST
        id ← blockIdx.x*blockDim.x + threadIdx.x
 4:     ant ← Initialize_ant()
        circuit ← Get_circuit_instance()
 6:     for each num_rounds do
            circuit.Reset()
 8:         for each cycle in num_cycles do
                circuit.Eval(ant[cycle])
10:             capture_response(ant, ph_map, circuit)
            end for
12:         atomic(Update_ph_map(circuit, ph_map))
            Evolve(ant, ph_map)
14:    end for
    end procedure
```

One kernel call for all of the test generation

The complete BEACON framework along with simulation is moved to the GPU, starting from initialization of ants till the last round of simulations and evolution. There is no cost of repeated transfer of the coverage numbers and simulation inputs or kernel launch overheads. This too has high register pressure. Algorithm 2 shows the outline of this implementation.

3.1.3 Execution of BEACON on GPU

All the ants are divided into blocks of threads. Blocks are scheduled on SMs. One SM can support up to 1024 threads or 15 blocks of threads, whichever limit is hit first. It is possible to have insufficient resources to schedule all the blocks to run in parallel. In such cases, the remaining blocks are scheduled once the execution of current blocks has completed. This means that certain ants may have not even started their first round while others may have completed all their rounds. Since the global state is preserved even when the blocks are swapped, there is no loss of information due to running in such a serial fashion. Speedup on a GPU is not directly proportional to thread count. Based on our launch boundsit is possible that at a given point of time in the kernel execution, certain warps are not assigned threads, and certain threads are not assigned warps. In addition to that, warps can go idle due to a number of reasons. In our results section, we have tried to understand what features of our GPU kernel leads to speedup limitations.

3.2 Experiments and Results

All the experiments were conducted on a machine with Intel(R) Core(TM) i7-6700K CPU at 4.00GHz and an NVIDIA GTX 980. The CPU core supports 2 threads per core and has

4 CPU cores with 64GB RAM. The GTX 980 is built with 2048 cores using the Maxwell Architecture and clocked at 1.1GHz. It has 4GB of RAM with 7Gbps bandwidth. All implementation were verified for correctness against its equivalent single thread CPU implementation. Coverage achieved from multi and many-core implementations have also been compared against the original implementation. All the speedup values reported are obtained with no loss in coverage. The circuits under test have been taken from [1] and [9].

3.2.1 CPU speedup considerations

In BEACON there is only a small section of the code that needs to be executed in a serial fashion. The time taken for the parallel portion varies due to circuit size, the number of ants and number of cycles per ant. From Amdhal's law, we know that the maximum theoretical speedup that can be achieved is bound by the serial part of the program.

Random number generation on the CPU is usually computed using dedicated hardware. This is to ensure the quality of the random numbers generated. Unfortunately, on our computing platform, there exists only one random number generator unit. Thus, when attempting to initialize and evolve ants for each round, requests from different threads for random numbers contend for this resource. This worsens the time taken for generating random numbers leading to zero speedup while evolving and initializing ants. There exist few thread safe ways to generate random numbers on CPUs but all such attempts tend to reduce coverage of BEACON as the quality of generation process is reduced as well. This forces us to serialize the input generation part which adds on to the already existing serial section of the code.

Table 3.1: Size of the design

Circuit	Lines of C++	Lines of Verilog	Number of Coverage points
b11	143	151	31
ss_pcm	224	146	19
simple_spi	572	394	66
usb_phy	864	871	139
i2c	1059	934	123
b15	1397	811	141
openmsp430	5445	9430	434
or1200	7474	21563	659
aes128	28260	1406	5139

Table 3.2: Speed Up Comparison

	BEACON param		Execution Time (sec)			Speedup over CPU serial	
Circuit	# Ants	# Cycles	CPU serial	CPU parallel	GPU	GPU	CPU Parallel
b11	8192	1000	9.28	3.07	0.24	38×	3.02×
ss_pcm	4096	1000	8.72	4.11	0.12	68.12×	2.12×
simple_spi	4096	1000	9.83	4.00	0.366	27.30×	2.46×
usb_phy	16384	1000	88.10	24.33	1.98	44.45×	3.62×
i2c	16384	1000	54.02	19.29	1.54	34.85×	2.8×
b15	16384	1000	50.62	20.20	1.39	36.41×	2.50×
openmsp430	4096	1000	72.46	19.14	2.95	24.56×	3.79×
or1200	4096	1000	66.76	17.23	3.71	17.99×	3.87×
aes128	4096	500	112.62	23.98	6.94	16.12×	4.69×

3.2.2 GPU speedup considerations

Usually, only the compute-intensive part of a program is moved to the GPU, and the rest of the operations are handled on the CPU. However, there are various methods for moving simulation to GPU as mentioned in Section III, with pros and cons, as described below:

- Kernel calls every cycle: The GPU kernel call overhead is approximately 10 ms per call. If we were to call kernels every cycle, this overhead becomes a recurring cost. Unless the amount of time taken on the CPU for a kernel is >> one GPU kernel call, we would see no benefits. The time taken for having to recompute all the signals

on the CPU is not significantly higher than on the GPU to overcome such a kernel call overhead. Therefore, even though the theoretical occupancy might reach 100% by calling the kernel every cycle, it does not translate into improved performance, as this does not eventually scale.

- One kernel call per round: Here the recurring cost of kernel calls for each cycle is eliminated, as we call the GPU only once per round. The number of registers per kernel now increases because each round contains many cycles, thus bringing down the theoretical occupancy. However, by allowing more registers per thread, there is a possibility of better performance per thread. Consequently, it improves the overall performance. There is also no longer a requirement of saving the circuit state to the global memory between each call. Nonetheless, the value of coverage counters and simulation inputs have to be transferred to and from the CPU and the GPU between each round. This transfer will reduce potential speedup.

In order to resolve the two issues above, our final approach runs only one kernel in total. The kernel has a larger size relative to the 'One kernel call per round' and this, in turn, increases register pressure further. Fortunately, we have managed to eliminate the memory transfer between the CPU and the GPU as the generation of random numbers can now be done in parallel on the GPU without the hardware issues on the CPU. It also decreases the serial portion of the code significantly. Because there is no way and we do not need to synchronize across different thread-blocks on a GPU, all the threads run in parallel with no need for stalls for synchronization. Fortunately, BEACON is based on heuristics and does not require strict ordering of operations. Thus this adds additional benefit.

Table 3.1 presents the size of each circuit for context .The results for both the CPU and GPU are presented in Table 3.2. For each circuit, the number of ants, the number of cycles

per ant are reported. Next, the execution time for CPU serial, CPU parallel, and the GPU are reported. Finally, the speedups are reported for the two parallel versions.

Consider circuit *or1200*, a RISC processor, using a total of 4096 ants each of length 1000 vectors. The test generation took 66.76 seconds on the single-threaded CPU. With 8 threads, the runtime was reduced to 17.23 seconds. With GPU, the execution time was further reduced to 3.71. We achieved 17.99× speedup with the GPU compared to a serial CPU implementation. On the CPU using 8 threads, we achieved only 3.87× speedup. Interestingly, on another circuit, *b11*, 38× speedup was achieved on the GPU and only 3.02× speedup for the corresponding multi-core CPU implementation. When comparing the speedups across different circuits, *b11* has a better speedup on the GPU as compared to *or1200*, whereas on the CPU it is vice versa. This can be attributed to how different circuit sizes can favor either the CPU or the GPU. Smaller circuits can fit better on the GPU register space thus leading to reduced loads and stores as compared to bigger circuits on a GPU. On the other hand, bigger circuits on the CPU increases the amount of parallelizable code from the circuit simulation and thus provides a greater speedup on the CPU as compared to smaller circuits.

3.2.3 Scaling on CPU vs GPU

Figures 3.1 and 3.2 shows how this approach scales on both the CPU and GPU for *ss_pcm*. Figures 3.3 and 3.4 represent the scaling for *or*1200, a larger circuit. As we can see in both cases with an increase in load GPU scales much better as compared to its corresponding CPU implementation. The difference between the *ss_pcm* and *or*1200 is only in the time taken for a single thread on a GPU vs. CPU. Since this difference is much lesser in *ss_pcm* as compared to *or*1200, the cut off point between a CPU and a GPU implementation is also much lesser for *ss_pcm*. With around 50 ants the GPU implementation breaks even

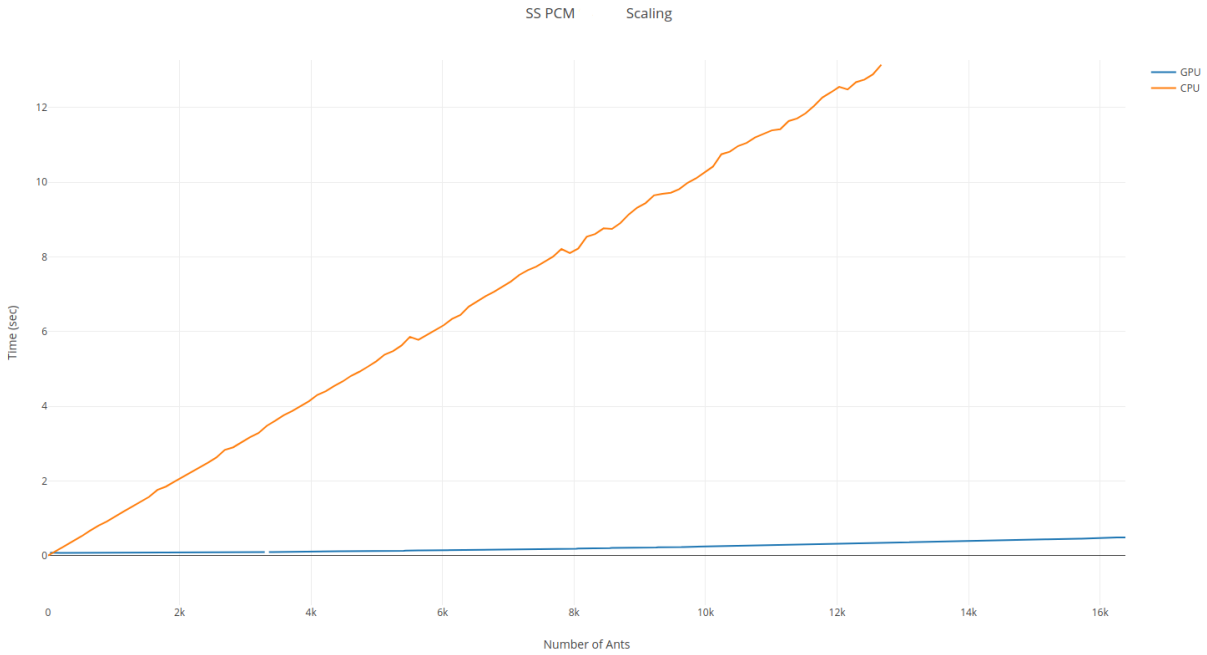

Figure 3.1: Scaling graph of SS_PCM

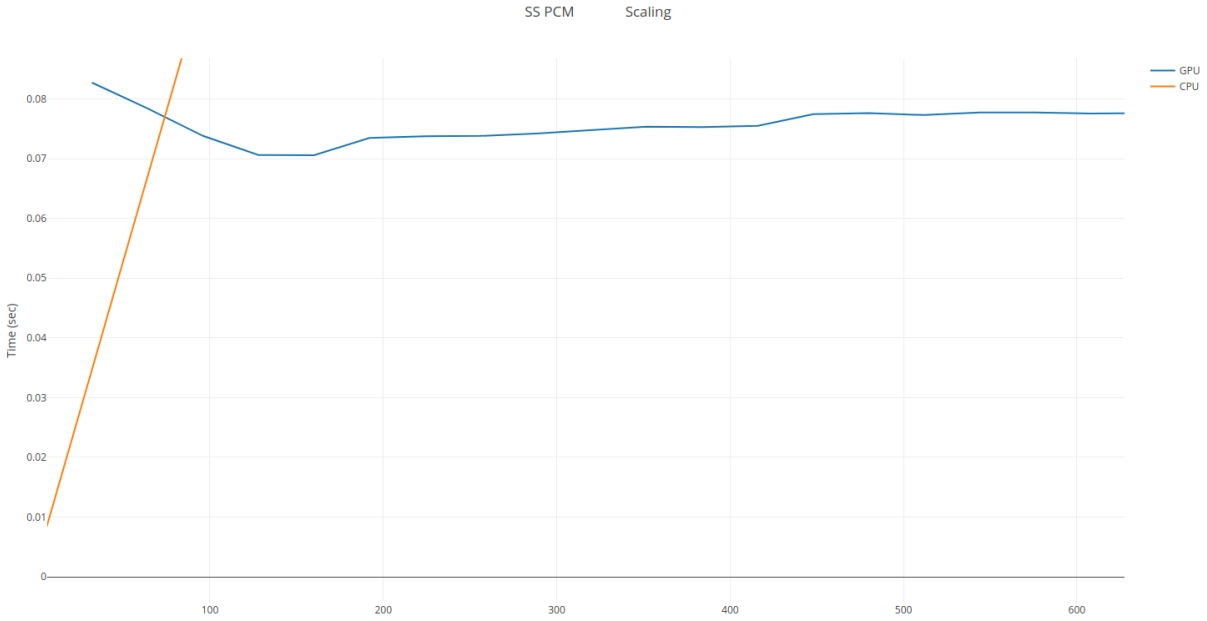

Figure 3.2: Cut off point between CPU and GPU for SS_PCM

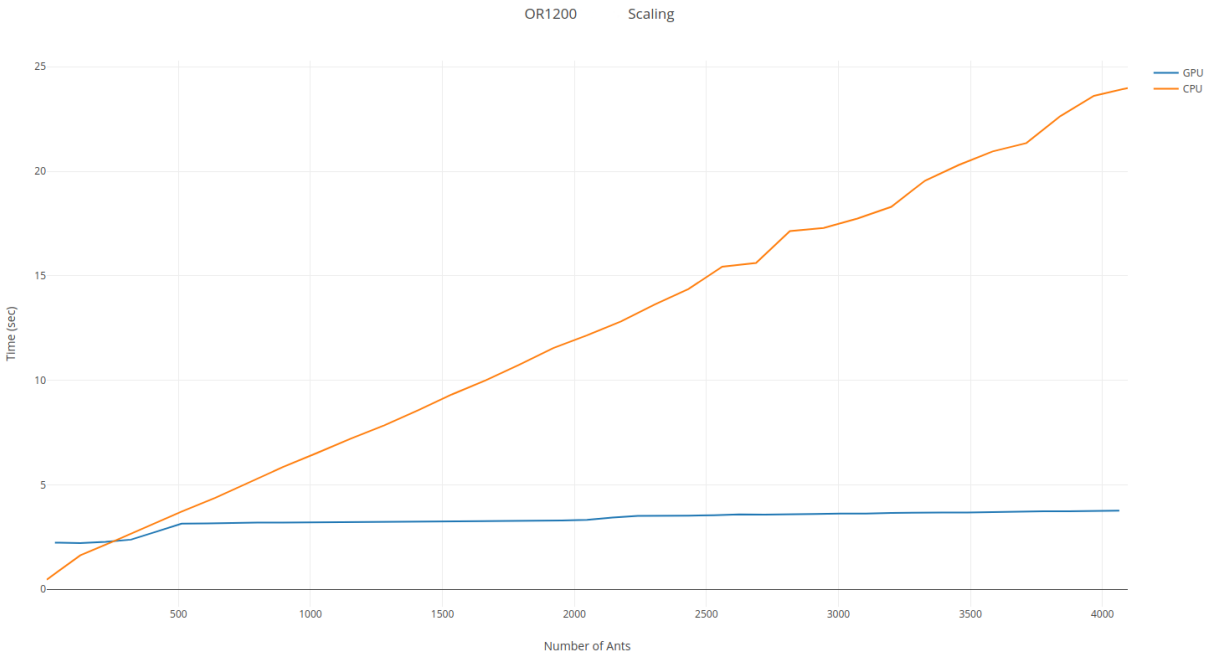

Figure 3.3: Scaling graph of OR1200

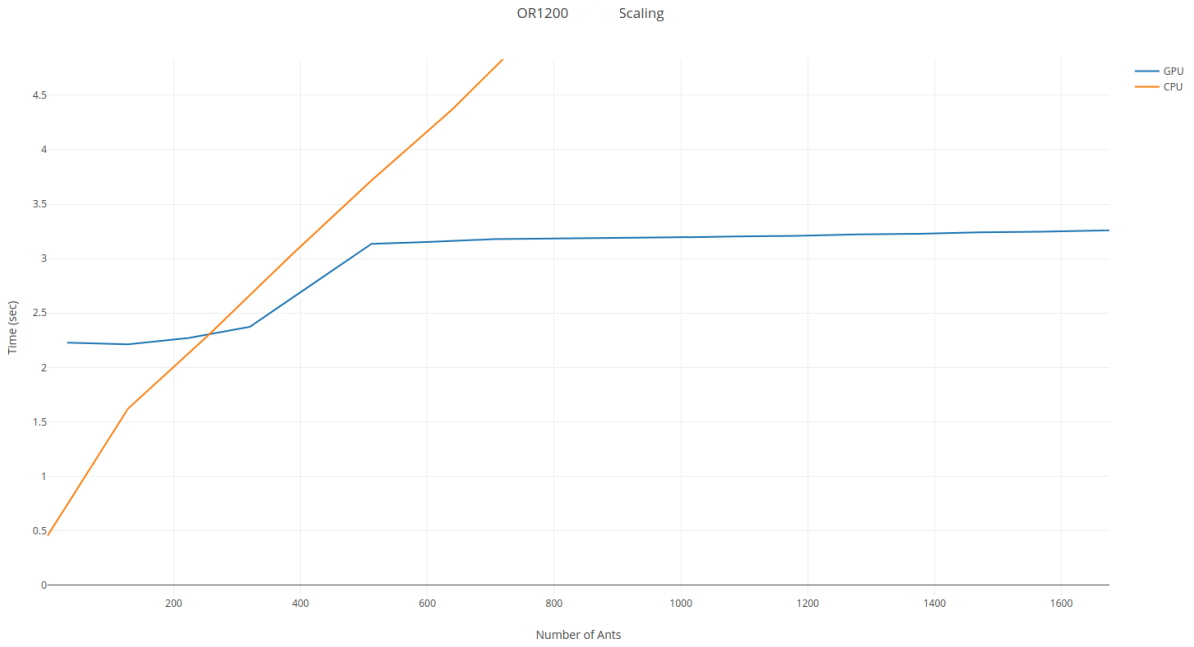

Figure 3.4: Cut off point between CPU and GPU for OR1200

with the CPU implementation for *ss_pcm* whereas in *or*1200 it takes around 250 ants to reach the break-even point. As we will see next, the difference can be attributed to circuit characteristics such as size and behavior.

3.2.4 Metrics from BEACON on GPU

Below are some metrics collected on BEACON for some of the circuits tested. Given that they are of a wide variety, we hope that the metrics provide a good cross-section of issues that may arise.

- Theoretical and Achieved Occupancy: Circuit *b11* exhibits the highest occupancy of 50% as compared to *or1200* which averages only 12%. This delta stems from the difference in the size of the two circuits. *b11* is small enough to allow the complete design reside in the register space of each thread. *or1200*, being a CPU design, has a lot of loads and stores which can lead to memory related stalls.

- Instruction to Byte ratio: For all our circuit's GPU kernels the ratio is around 1:2. This has been computed after analyzing the assembly code of the kernel. This indicates that it is possible for our kernel to be memory bound.

- Instructions per warp and DRAM utilization: The max Instructions per warp is 2. Our kernel averages 0.28 Instructions per warp and a DRAM utilization of 7/10. This again shows that our kernel is memory bound implying most of the warps are idle. This measured metric is true for circuits of larger size including *or*1200, *aes*128 and *openmsp*430. For smaller circuits, the DRAM utilization is much lesser as most of the memory access is on either the register space or the L1 cache. Thus we can see a more considerable speed up for smaller circuits as noted in the Table 3.2.

- Branch Divergence: A widespread and valid concern of modeling RTL Verilog as a CUDA instance in a SIMD fashion as done here is branch divergence. Through translation, the Verilated C++ is littered with *if then else* and *case* statements. This causes branch divergence and significantly affect performance. For the *b11* circuit branch divergence causes around 72%. This contributes to 2.6× less speedup than if there were no divergence to our test generation process.

- DRAM usage: Another bottleneck that we discovered was not having enough memory space on the GPU to store all the variables. This includes signals, coverage instrumentation values (which in most cases is more than the number of signals per simulation) and the test generation outputs. This limits the number of ants and the number of cycles per ant. For GPUs, it is beneficial to run large number ants as it translates to larger thread count which helps in hiding latency and increasing throughput.

- Memory Coalescing: In our work, we have coalesced transaction to the global memory as much as possible. This has helped us improve performance over a factor of at least 1.8×. For larger circuits with more signals, we have observed close to 4× improvement over uncoalesced accesses.

- Data reuse: It is beneficial when different threads access the nearby global addresses as these reads can be cached and later be accessed from the local cache directly. Unfortunately, our framework has nonexistent data reuse between ants.

Use case scenarios for shared memory and texture cache, an optimal mix of integer and floating point operations are absent in BEACON. The latter is because it is difficult to cast bitwise operations, that are common in Verilog, to floating point values. These are features that are preferred by the GPU and can help a kernel achieve even better performance.

3.3 Parallelism from the RTL Design

Extracting each process from the original RTL can be done using verilator. Verilator converts each RTL process or statement into a verilated C++. These statements are grouped together and assigned a process identifier. Each process has its own function and they are called using function pointers. Apart from the function pointer, we also have an input queue which is time sorted and an output buffer. Combinational processes, i.e., those derived from "assign" or "always *" need to execute whenever their inputs change. On the other hand, sequential processes only change for every clock. So the output of a sequential logic process remains valid until the time-stamp of the next clock. The following subsections provide the outline of the steps we followed to convert the RTL simulation to GPGPU Cuda equivalent and used it for verification. We also explain the implementation and the data structures that are created.

3.3.1 Generation of Processes, Interfaces and Dependency Matrix

Each process in the RTL can be either sequential or combinational blocks which have to be simulated. The overall outline is described in Figure 3.5. We first convert the RTL Verilog to C++ using Verilator.

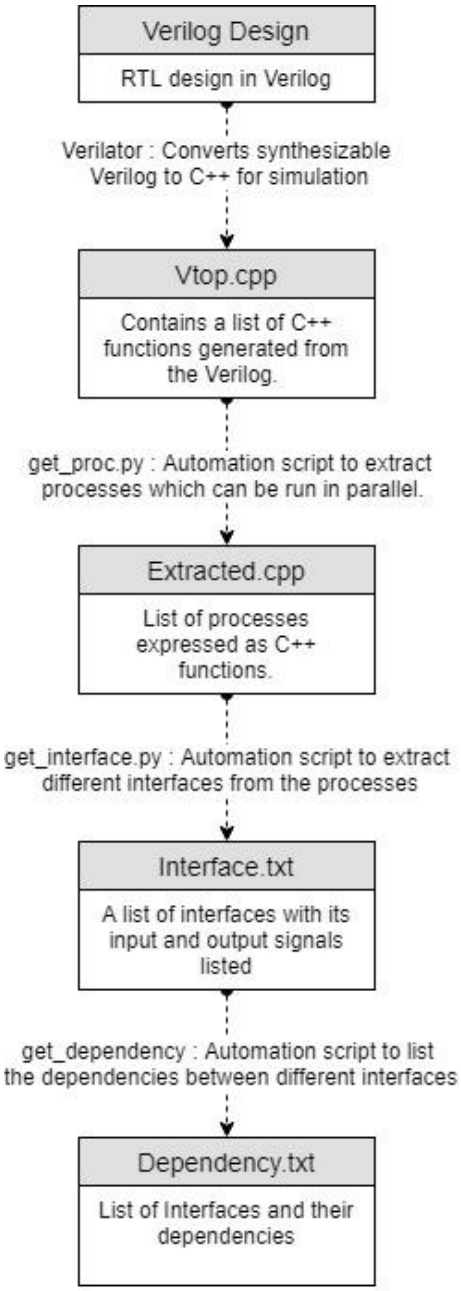

Figure 3.5: GPU accelerated RTL simulation flow.

- Extracting processes as functions: Taking the output of the Verilator, we slice it down to processes by identifying chunks of code that can run together. We have written a python script to process the verilated C++ to generate function pointers from the

RTL process blocks.

- Generating Interfaces: This is an abstraction that we are generating using a python script which parses the process blocks generated above. It is used to represent a process (sequential or combinational) along with its input and outputs defined as a list of arguments. The inputs and outputs are the actual inputs and outputs to the RTL design and they also include the wires which are instantiated between different modules. This helps is identify relationships between different processes and also help with process grouping.

- Generating dependency matrix: The interface generated is parsed to generate a dependency list between different interfaces. This list is used to initialize the connections using a 2-D edge matrix between the interfaces in the CUDA code. Using this we can define the connections between the outputs and inputs of different processes.

3.3.2 Data Structures Used

Fig 3.6 describes the different data structures used.

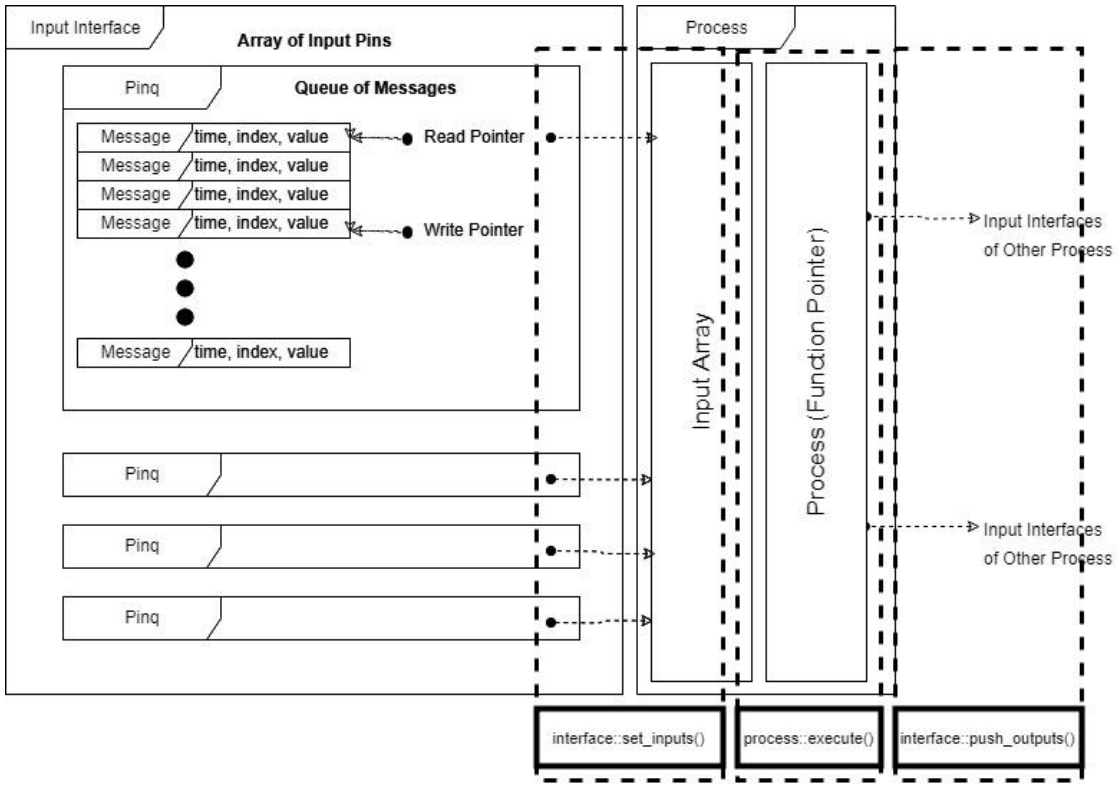

Figure 3.6: High Level Design.

Below is a brief description of the components in the figure,

- Message: This is the fundamental message passing unit, which shares the generated signals across the processes. It contains time of the signal, the pin number index and the value of the pin.
- Input queue: Signals which are defined and used by multiple processes are represented by queues which is a time sorted list of messages. Each queue has a read pointer which has to be updated on read from the input queue if the process executes and a write pointer which is updated on generation of a new timestamped value in the queue.
- Interface: Every process has an interface associated with it. It has a list of queues which maps to the input signals of the process. The interface also contains a variable

which is indicates the least time present in the current input queue. Each process has an input interface and it loads its output into queues from different interfaces.

3.3.3 Code Generation

In order to actually implement the extracted processes, we need to use the dependencies and other information extracted to generate a *.cu implementation that can be execute on a GPU. The code for the Process, Queues and Interfaces cannot be handwritten even for moderately sized circuits as the chance of error is high. Also the number of lines that needs to be handwritten increases very quickly as the number of processes increases. Thus, this process of code generation has been automated. In order to store and retrieve values from Pin Queues and Input/Output buffer of each process we need to allocate space in the global memory of the GPU. This is usually done using the CUDA Malloc API which returns a pointer to the allocated memory. But since for a given circuit, we already know the input/output sizes of each process and the depth of each Queue, we can specify them during compile time itself. The overall flow is as described below.

- We allocate memory for the input and output buffer arrays and all the signal queues required.
- We define read and write pointers for each queues and also time for each process.
- We define the set inputs and clean outputs for each process to set its inputs and shift new outputs to its signal queues respectively.
- Using the above definitions we define the execute which execute the process function for each process extracted from the RTL
- We also define the kernel calls which calls the functions defined above.

3.3.4 Execution of the kernels

Below are the two basic kernels that are executed every clock cycle. We also describes the various levels of parallelism exploited by the kernels.

Listing 3.1: This function gets the inputs from the PinQs to the input array in the process.

```
void Interface::set_inputs(int *inputs, int process_time, int *input_time, int
    &least_input_time){
  int count = 0;
  int set_count = 0;
  least_input_time = process_time + 1;
  for(int i=0; i<num_pins; i++){
    if(process_time >= input_time[i]){
      count += 1;
      set_count += pinq[i].get_value(inputs, input_time, i);
    }
  }
  if(set_count == count){
   return 1;
  }
  return 0;
}
```

Process::execute() – This is the base kernel which exploits block level parallelism. It calls set_inputs to determine if the process is executable. Figure Listing 3.1 displays the code snippet which is called by the input interface. This code tries to set the appropriate value in the input array for the process. If the valid values are available then the process is executed and sets the clean up flag. If the values are not available, it skips the execution. We use only

block level parallelism here as executing multiple processes on a single block would result in thread level execution divergence which would cause some cores to stall and negatively impact performance.

Process::clean_op() - : This operation executes if the clean up signal is set. The output signals for each process are stored in an output array. The output array updates its corresponding signal queues. The update is adding a message, value time pair, to the end of the queue of that particular signal. Since this step involves only moving data, we can try and exploit thread level parallelism.

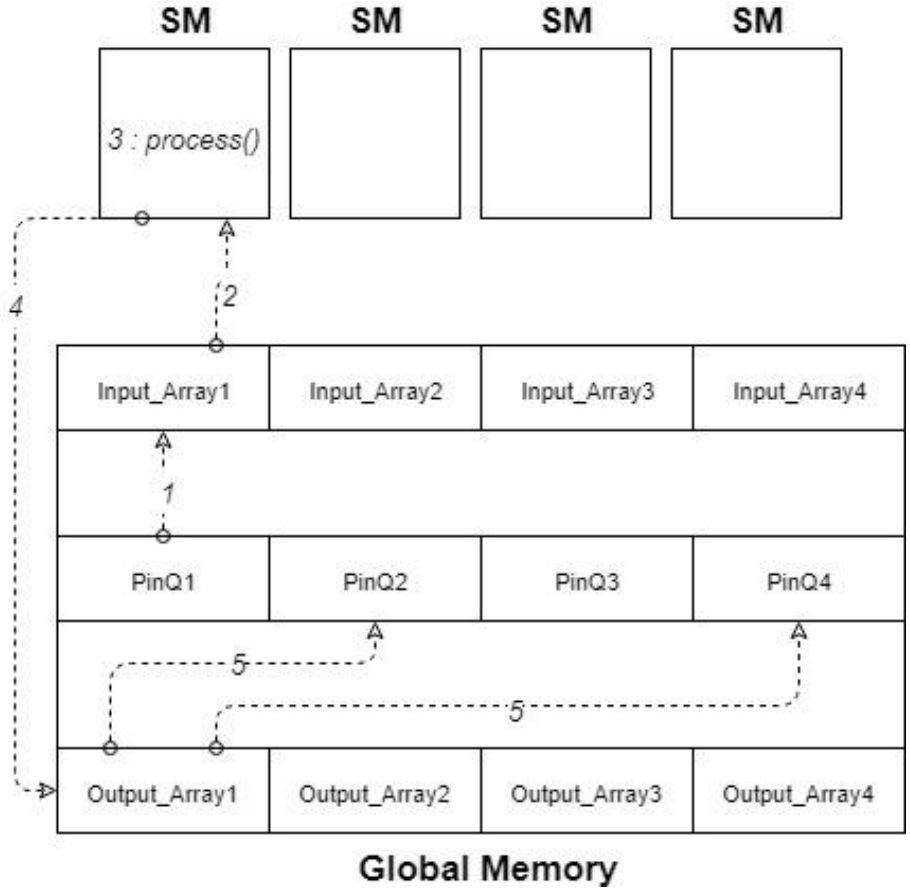

Figure 3.7: Execution flow on the GP-GPU.

The execution flow as shown in the Fig 3.7 for the entire framework is

- Fetching the next set of input values for the process from the signal queues.
- After determining if all values are available, then we execute on the function pointer.
- Execution of the function pointer can be done in parallel on different blocks.
- The output of the process is then stored in an output array.
- The outputs update the signal queues, which is done using multiple threads, thus exploiting thread level parallelism.

Chapter 4

Testing Deep Neural Nets

In this chapter, we provide our metrics, methodology, and results for assessing the robustness and reliability of Neural Nets (NN) and Deep Neural Nets (DNN).

Traditionally, the *testing* of NN/DNNs refers to the measuring of the accuracy of classification of a network for its test inputs. The notion of accuracy depends on the richness of the training set. In our work, however, we try to qualify the reliability and robustness of a DNN for inputs that may lie outside of the available dataset. As we have seen earlier in Chapters 1 and 2, even DNNs with a high level of accuracy can misclassify some inputs when the inputs are just altered a little bit. In fact, in some cases, it may be easier to render a higher-accuracy DNN to misclassify, as we will show in the results. Furthermore, it has also been observed that DNNs may classify seemingly random and uncorrelated inputs as a valid class. Both of these issues are serious concerns. Given that DNNs are key decision-making units in may Artificial Intelligence (AI) frameworks, the behavior of the AI engine can greatly change depending on the DNN's reliability. To do this, we first try to assess how well the trained weights/classifier correspond to the ground truth/human perception. This is followed by attempts to generate corner-case test inputs to check how the DNNs respond to such inputs.

In a nutshell, we start with a sample input, s. Then, we seek to distort s to see how the underlying neural network N will respond. However, there is an infinite number of ways that one can attempt to distort s. Thus, in order to make the technique practical and scalable, we approach this from two different angles. First, we try to distort s as little as possible to

see how much distortion is needed to alter the output of N such that N classifies the slightly distorted input as a different resulting class. Often the slight distortion is not noticeable to the naked eye. If such a small distortion is sufficient to distort the output of N, one can reasonably argue that N is not very reliable and robust. The second angle is the reverse. That is, we try to distort s as much as possible without making N classify it as some other class. Again, such distorted input often results in an input that is drastically different from the original s. Hence, it suffices to say that the distorted s may not resemble the original s. Often this distorted s should not be classified the same way as the original s. These two angles attempt to test how sensitive the neural net is to distorted inputs; hence, how reliable one can take the result.

The exact method used to assess the reliability of a DNN in terms of the above mentioned issues is described in this chapter.

4.1 Metrics

4.1.1 Min Metric

Let us assume that N is a trained neural net and x_i is an input from an available data set where i denotes its class number. x_i is chosen such that the currently trained N can classify x_i to class i. Then, we try to alter/mutate the individual bit values of x_i using any method available with the objective and constraint described below. Let x'_i denote our modified or mutated x_i.

$$S = min(|x'_i, x_i|) \tag{4.1}$$

$$O : i \neq N(x'_i) \tag{4.2}$$

where, S is the constraint to minimize the magnitude of change in the input. $N(x'_i)$ denotes the output class of the classifier. O is our objective of trying to generate an input that is classified as any class other than class i.

The $|x'_i, x_i|$ stand for the Euclidean distance between the two samples. In contrast to the Hamming distance, which measures the mean absolute difference in the value of the features in the input, Euclidean distance measures the actual distance between two samples in the n dimensional space where n is the number of features. This value is the distance which x_i has to be moved in order to make the DNN change its classification.

The choice of x_i makes a considerable difference in the process. Depending on the particular sample, the value of $|x'_i, x_i|$ can change considerably. Also, the samples from different classes have varying levels of tolerance towards input perturbation. Thus, in order to get a proper estimate, we repeat this process $\forall x_i$ in i and $\forall i$ in C where C is the number of output classes. The overall reliability can be an average of the difference between all generated x'_i and x_i and can be denoted by:

$$R_{min} = \sum(|x'_i, x_i|)/T \tag{4.3}$$

where T is the total number of samples used.

Classwise R_{min} can provide confidence to the user on classification to a particular class. Input features are usually expressed in some numerical form as either integers on real numbers. Different input features may have different levels of importance and may represent altogether different entities. We can choose to normalize the inputs or weigh their delta differently depending on domain knowledge.

4.1.2 Max Metric

Using similar notations from earlier, for the Max metric, we try to measure how much perturbation is necessary for a N to start to classify a x_i' as something still belonging to i.

The constrains now become:

$$S = max(|x_i', x_i|) \tag{4.4}$$

$$O : i = N(x_i') \tag{4.5}$$

As for computing R_{max} we try to vary and mutate input x_i as much as we can such that the neural net N still classifies it as class i. Here it is important to note that not every neural net N is trained to produce a $Null$ or classification indicating it does not belong to any existing class. Thus, it is possible to force a N into giving a particular class as its output even though the confidence or the prediction probability is quite low. In our experiments, we show results for such neural nets N that have been trained to classify inputs that do not match any digits as a $Null$ classification.

The equation for computing R_{max} over T inputs is:

$$R_{max} = \sum(|x_i', x_i|)/T \tag{4.6}$$

Finding a x_i' boils down to finding an input that is not x_i but is still classified as i. We already know that there exists many such x_i', as even samples from the training set that belong to i are a valid solution. Thus, in this case, we are guaranteed to find a x_i', albeit an inferior solution.

Our method starts by taking a sample from the original dataset as the initial reference, for both Min and Max cases. Thus, the generated test sample also has a higher chance of being

identifiable by human perception as something that does not look randomly generated. This is important so as to be able to provide the designed of a DNN meaningful feedback.

4.2 Methodology

We use a blackbox approach to test generation in which we try only to probe the values at the output layer of the DNN in order to generate the tests. This is because the user themselves need not train the neural nets used. They might form a part of a cloud service or a locked IP. However, this still allows us to probe not only the reliability of the classifier but also the confidence or the prediction probability of each class. We use this to help guide test generation.

Different methods can be used to generate test inputs that try to honor the constraints mentioned before. They include Genetic Algorithm, simulated annealing, particle swarm optimization, gradient descent, etc. However, since the objective of this work is to evaluate the effectiveness of the metric and not the test generation framework itself, we chose to explore only Genetic Algorithm (GA). However, any other approach may be used.

4.2.1 Genetic Algorithm Setup

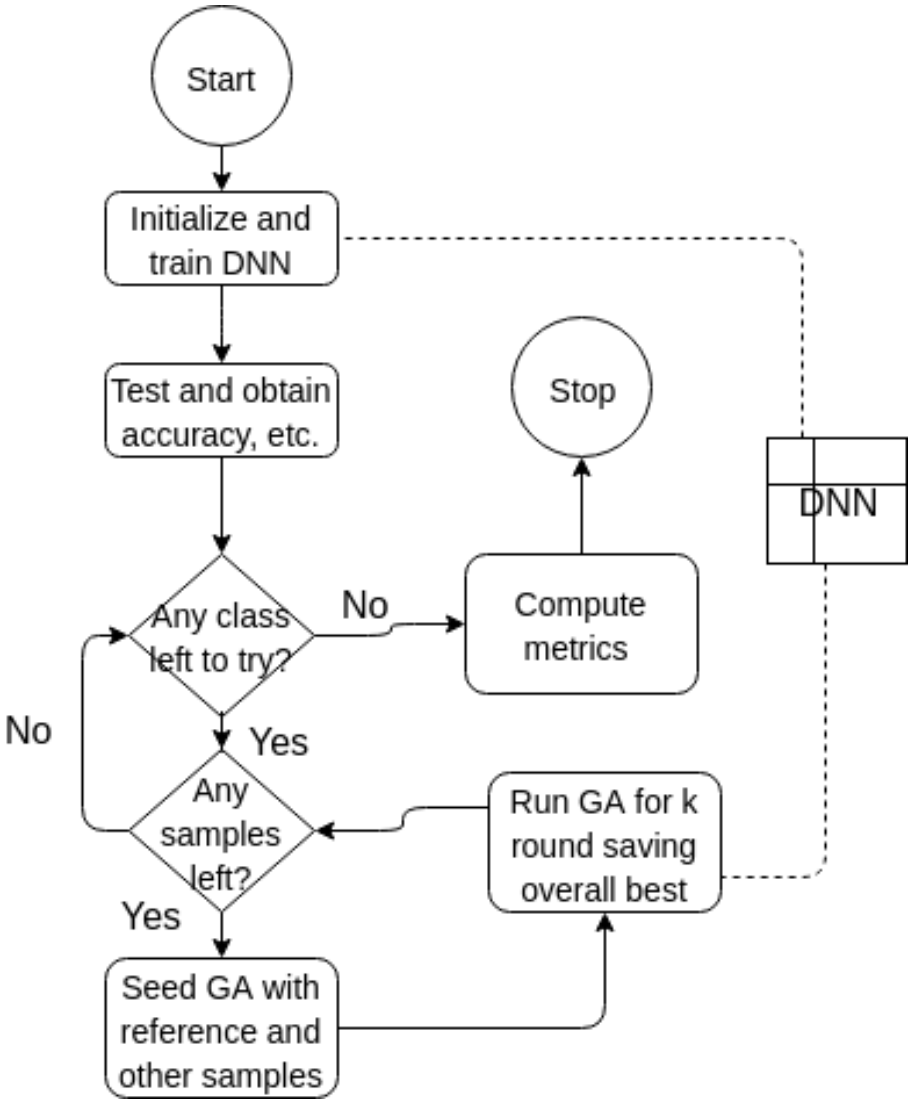

Figure 4.1: Flow chart of Genetic Algorithm.

The overall framework for the test generation is depicted in Fig 4.1. The flow of execution remains the same for targeting both Min and Max objectives. Since most of the modern libraries allow us to save and load DNNs, it is possible to skip the training part of the flow. As mentioned earlier, our training phase consists of training the DNN in question not only to recognize classes but also to try and avoid classification of doubtful inputs. This is achieved

by creating a new class at the output layer that does not correspond to any ground truth. We generate random input samples with this class and include them as part of our training data set in order to help the DNN recognize these later on during test generation.

The seeding of the population, the crossover and the mutation specifics are as follows.

Initial population seeding

The initial seed, especially for this application, seems to play a critical role in test generation effectiveness. In the case of the Min objective, we seed the test generator with the reference sample, x_i. We then fill 1/3 of the population with a variety of samples from the same class as x_i. The remaining 2/3 of the population are filled with samples from all other classes including the random/noise class. Seeding can also leverage direction from the confusion matrix by seeding from classes which have a high number of entries in the matrix.

For the Max objective, the population is also seeded with the reference, x_i, and samples from the original class only fill up to 1/10 of the population. The remaining 9/10 of the population is seeded using samples from other classes, with preference to the noise class. This is because we are trying to distort the image as much as possible.

Crossover

One of the key ways to traverse the input space in GA is using crossover. Thus, without an effective crossover operator, it is nearly impossible to generate good test samples. In our crossover operator, we only change one of the two samples used during crossover. This is because while one of the two samples improve during the crossover process, the other one starts to represent noise which turns out to be detrimental during the following rounds of evolution. Below is the list of crossover functions and how they vary for Min and Max.

- Two-point crossover: For two random numbers within the number of features, all features indices between these numbers are replaced by the values from the second sample. We only copy the features from the second sample to the first sample, where the second sample is more-fit than the first. As a result, we have to conduct crossover for all samples in the population so that we evolve all samples during every iteration. The reason that we do not swap the bits between the two samples is the following: Since in our Min metric we try to minimize the changes in the feature values, the number of features that are replaced should be small.

- Random crossover: The value of each feature in the first sample is replaced by its corresponding value from the second sample based on a probability. The chance of feature change during test generation for Min is $<$ than that during test generation for Max.

- Single point crossover: This is similar to the two-point crossover except that one of the two points is fixed to be either 0 or the index of the last feature.

- Patch crossover: Given that we have applied our work for image classifying DNNs this is designed to fetch patches of the image from the second sample to replace in the first sample. We use patches of two different sizes which is chosen with mixed bias based on whether it is Min or Max generation. For the Min case, we try to use smaller patches, and for the Max case, we bias towards larger patches. The location of these patches is also randomized.

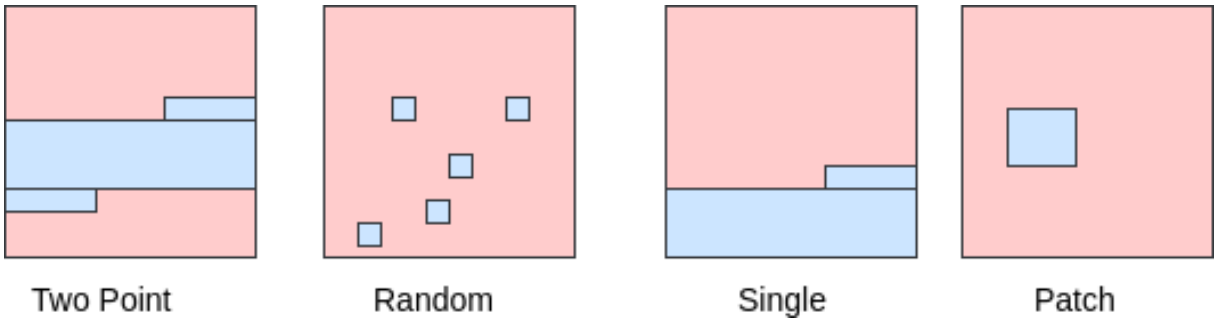

Figure 4.2: Window of different crossover types.

Fig 4.2 shows the window of each of the above crossover functions with the features laid out on a 2D array.

Fitness

The fitness for each test sample is ascertained using the two constraints discussed before. The class belongingness and how different or similar it is from the reference sample. During each test generation, the GA is seeded with the reference sample and the DNN model. It records the prediction probability values and the class belongingness of each test sample during a given round using the response of the DNN.

The equation for Min is:

$$fitness = 100 * 10^{\Delta(predict\_proba)} * belongingness\_bonus/(1 + 10 * \delta) \tag{4.7}$$

The equation for Max is:

$$fitness = 100 * 10^{\Delta(predict\_proba)} * belongingness\_bonus * (10 * \delta) \tag{4.8}$$

Where $\Delta(predict\_proba)$ stands for change in prediction probability of the current sam-

ple compared to the previous round and δ stands for the difference between the generated test sample and the reference sample. The belongingness bonus is either 1 or 4 depending on whether the test sample is classified in the same class as the reference sample or not. Naturally, it is 4 when it is in a different class during Min and the same class during Max.

It is interesting to note that it is possible to conduct test generation using the class belongingness values alone and not use the prediction probability values. However, this might tend to produce a very suboptimal test suite.

Evolution

After every round of fitness calculation, we sort the population based on the same and broadly separate them into three groups. The top 1/3, the bottom 1/6 and the rest.

In order to preserve the best-generated test sets through each round of evolution, we practice elitism in the population. The best set of samples are preserved through each round so that we do not lose the traits of the same. Thus only a few of the top 1/3 is changed during the crossover process. Here even the second sample for crossover comes from the top 1/3.

Since we would like to replicate the features of the elite samples in the rest of the population, the samples lying in-between the top 1/3 and the bottom 1/6 are changed every round. Here one of two samples used for crossover is chosen from the top 1/3 while the other is the original sample itself. The sample taken from the top 1/3 is copied over its current and then crossover is then applied using one of the crossover windows described and its original values as its second argument.

Since we would like to keep on promoting changes in the population, in general, to allow for better crossover attempts, the bottom 1/6 is replaced during every iteration of evolution and is reseeded using the original method.

Mutation is straightforward in our implementation. Each sample has a 10% chance of being mutated, but mutation itself only alters each feature with a 1% chance.

4.3 Experiments and Results

In this section, we explain our experimental setup, discuss the evaluations of sample DNNs, and provide visual examples of the generated test samples. Finally, we describe some observations on the results shown.

4.3.1 Experimental Setup

The dataset we have chosen to experiment on is MNIST [22]. We use Keras [8] Neural Net libraries to model and train our DNN. We also use the NUMPY [31] library for its efficient scientific computations. The framework has been implemented using Python 2.7.

4.3.2 Evaluations and Observations

We implement our framework on multiple DNNs for the same dataset with varying structures and accuracies. Each MNIST sample is a 24×24 greyscale image with each pixel value in the range of 0 - 255. We linearize this 2D array as a 1D array of length 784. The pixel values are normalized to a 0 - 1 range.

Table 4.1 records the accuracy of the DNN for each class. This can help provide context to the generated metrics. The overall accuracy of the DNN under test measuring using the validation data set is 98.5%.

Tables 4.2 and 4.3 report the values of our metrics obtained using our robustness test genera-

Table 4.1: Class-wise accuracy

Digit	Accuracy (%)
0	99.28
1	99.38
2	98.44
3	98.11
4	98.06
5	97.98
6	98.22
7	98.92
8	97.63
9	97.91

Table 4.2: Min. Results

Digit	Mean	Min.	Max.	Std. Dev.
0	5.502	4.121	7.384	0.853
1	4.145	2.803	4.804	0.495
2	5.168	2.906	7.506	1.179
3	4.412	1.312	6.018	0.922
4	4.581	1.767	6.324	1.080
5	4.172	2.135	5.592	0.910
6	4.194	1.100	5.432	0.869
7	4.409	1.900	5.730	0.884
8	4.122	1.597	6.845	0.996
9	3.244	1.988	4.903	0.712

tion framework for Min. and Max., respectively. Entries in each table measure the Euclidean distance between the color of all the pixels that compose the image. We use 60 samples per class in order to compute these metrics. The samples are randomly chosen from the pool of correctly classified training and testing samples.

Min MNIST-Digits

For the digit *zero*, the mean minimal Euclidean distance that a sample has to be altered in order to make its classification unreliable is 5.502. The minimum observed distortion value

Table 4.3: Max. Results

Digit	Mean	Min.	Max.	Std. Dev.
0	13.93	12.99	15.20	0.70
1	12.13	10.58	13.49	0.98
2	14.29	13.24	15.27	0.58
3	13.82	12.50	15.76	0.89
4	12.77	11.36	14.53	0.80
5	13.54	12.47	15.25	0.78
6	12.99	11.81	13.87	0.67
7	12.65	11.07	14.21	0.78
8	12.78	11.15	14.40	0.81
9	12.83	11.75	14.13	0.76

for the digit *zero* is 4.121 while the maximum distortion tolerated by the DNN is 7.384. This means that among the 60 references picked from the initially classified samples, one reference sample required only 4.121 distance worth of distortion to force the DNN to misclassify. For context, the highest possible numeric value for both Min and Max is 28, which would imply that all the pixels that were initially black are now white and vice versa. A value of 1 for Min. could mean all it takes for an image to be misclassified is flipping one completely black pixel white or vice versa. The standard deviation over all the 60 samples for the digit *zero* is 0.853. This suggests that for *zero*, the amount of distortion needed does not vary much and is usually about ± 0.853 distance around the mean

The smallest distortion needed in order to force misclassification of digit 6 over all the samples of all the digits is observed to be a mere 1.100. Digit 1 requires less alteration as its mean distance is just 4.14 and digit 9 is even less at 3.244. As it is difficult to make sense of these values on their own, we have attached samples of generated images. In these images for digit 0: Fig 4.3 and Fig 4.4. Likewise, Fig 4.6 through Fig 4.10 for the other digits. From these figures, we can see how different metric values translate to different levels of reliability in terms of classification. Not all test generation attempts result in an output that represents the desired ground truth. For example, Fig 4.13 looks like an 8 to the human

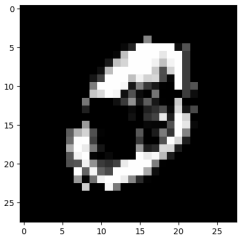

Figure 4.3: Zero as Five

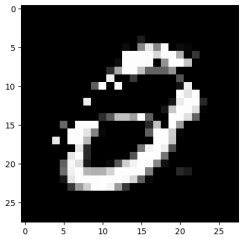

Figure 4.4: Zero as Two

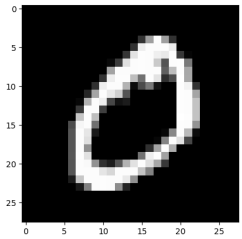

Figure 4.5: Reference

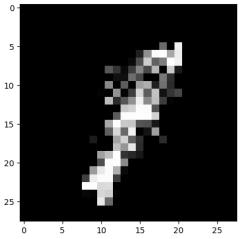

Figure 4.6: One as Nine

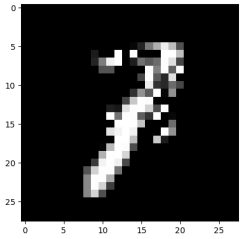

Figure 4.7: One as Seven

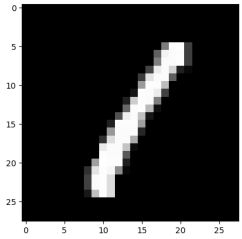

Figure 4.8: Reference

eye even though the test generator tries to alter it as little as possible. Interestingly, it is classified as a 4. Other examples of misclassification are Fig 4.15, Fig 4.18 and Fig 4.21 where the ground truth for the generated test images is not the same as the one represented in the reference image.

From these images, we can see that this DNN (with a 98.5% accuracy based on the standard training set) has issues classifying inputs that are perturbed even just a little bit. Thus, the supposedly accurate DNN may misclassify some images where human perception can easily recognize.

Max MNIST-Digits

The tests generated using the Max criteria reveal a similar story. Fig 4.24 shows an image that was distorted as much as possible from a clean 1 that is still classified as 1 by the DNN. It is easy to note that this image is composed of mostly noise. The same is true for Figures

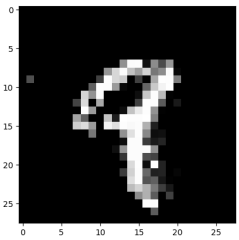

Figure 4.9: Nine as eight

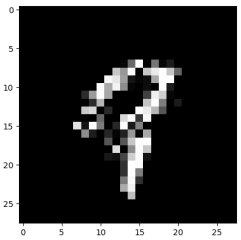

Figure 4.10: Nine as Four

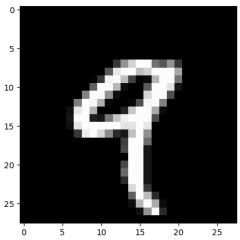

Figure 4.11: Reference

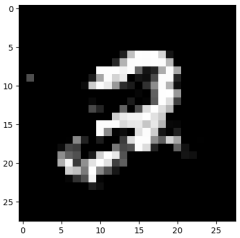

Figure 4.12: Two as Three

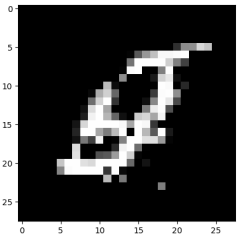

Figure 4.13: Two as Four

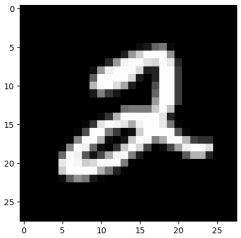

Figure 4.14: Reference

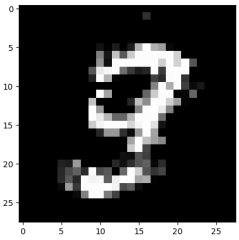

Figure 4.15: Failed Min : Three as Nine

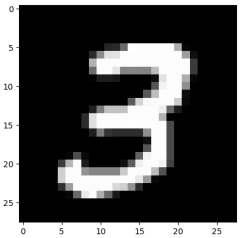

Figure 4.16: Reference 3

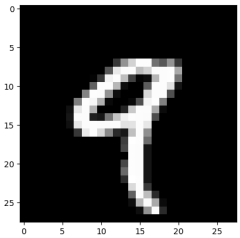

Figure 4.17: Reference 9

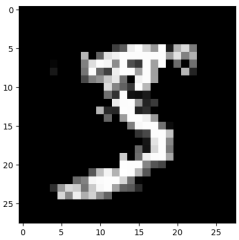

Figure 4.18: Failed Min : Five as Three

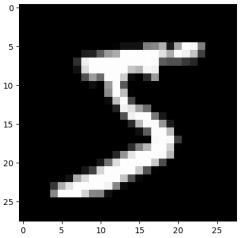

Figure 4.19: Reference 5

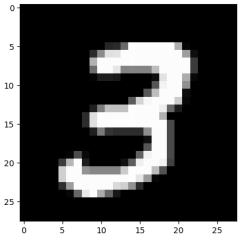

Figure 4.20: Reference 3

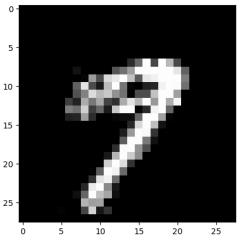

Figure 4.21: Failed Min : Seven as Nine

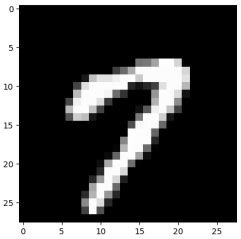

Figure 4.22: Reference 7

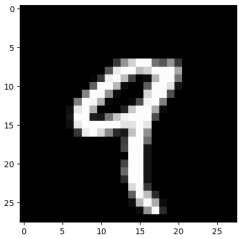

Figure 4.23: Reference 9

4.26 and 4.28 which are still classified as 6 and 8, respectively, even though they have been distorted significantly from the original clean 6 and 8. As mentioned earlier, the DNN is trained to classify the 10 digits as well as the non-digit, and it received with 98.5% accuracy from the training set. Despite including noise samples to train the DNN, the DNN still chooses to classify these noisy inputs as valid digits. In fact, the prediction probabilities for these images are > 0.8 in favor of digits.

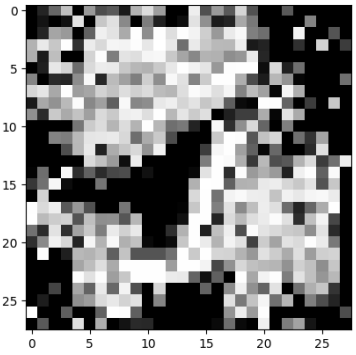

Figure 4.24: Nothing as One

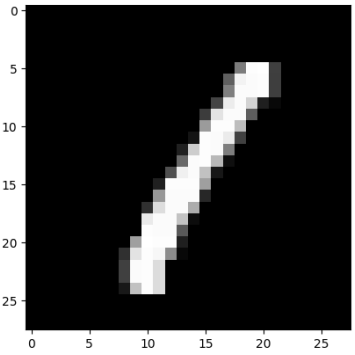

Figure 4.25: Reference

One interesting observation while generating the Max. test cases is seen in Figures 4.30 and 4.32 which represent digits 7 and 4. Here the contours of the digits 7 and 4 are preserved in the distorted images, but the pixels are inverted to some degree. There have been theories that DNNs internally learn to observe image boundaries in order to aid classification, and

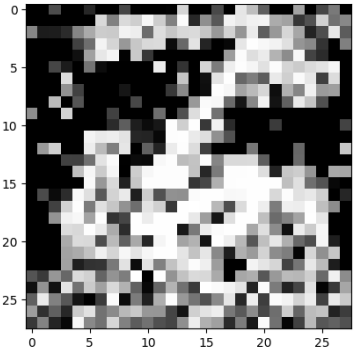

Figure 4.26: Nothing as Six

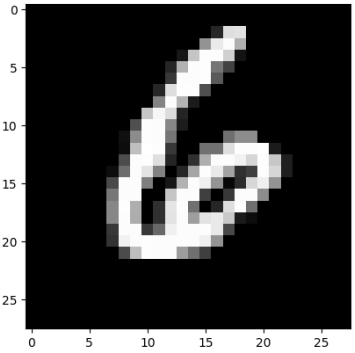

Figure 4.27: Reference

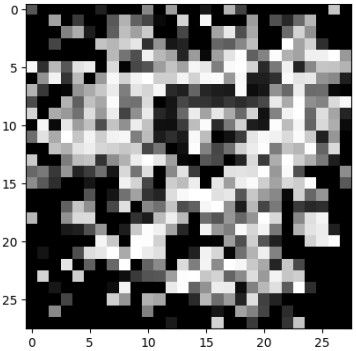

Figure 4.28: Nothing as eight

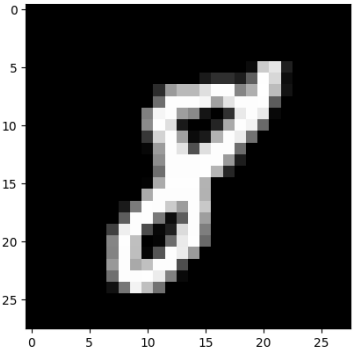

Figure 4.29: Reference

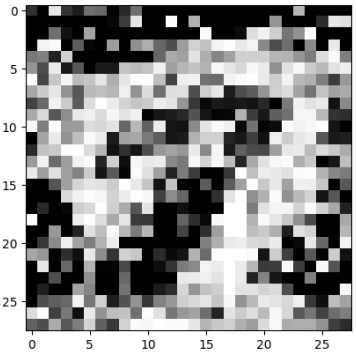

Figure 4.30: Nothing as Seven

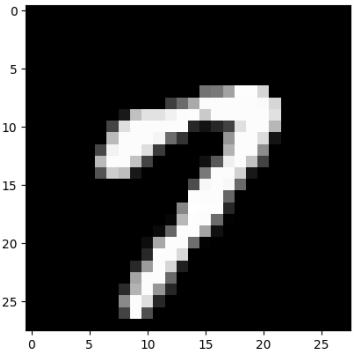

Figure 4.31: Reference

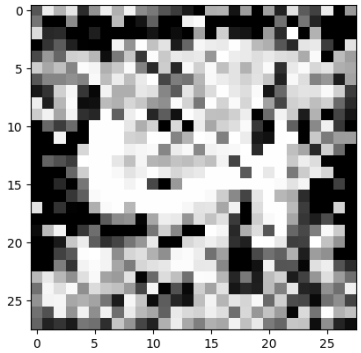

Figure 4.32: Nothing as Four

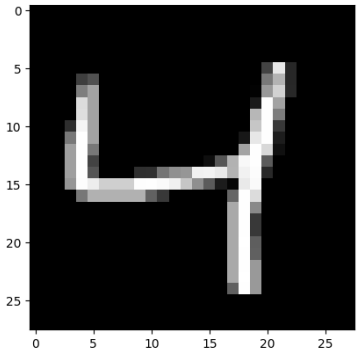

Figure 4.33: Reference

this observation reinforces that theory.

Min MNIST-Fashion

MNIST is considered one of the easier datasets to classify as compared to CIFAR [21], which is composed of hundreds of classes and millions of images. Unfortunately even a highly performant DNN that classifies a simple dataset is unreliable as observed here.

In addition to assessing the robustness of a particular DNN, we also wish to contrast the robustness of two different DNNs for this case. To this effect, we have evaluated the proposed approach on DNNs with different levels of accuracy. We note that the MNIST-Digits is an easy dataset that even an MLP, with 1 hidden layer can be trained to classify with high accuracy. Thus, instead of the MNIST-Digits, we have chosen the MNIST-Fashion database [39] which is is a collection of clothes and shoes. It is difficult to achieve high accuracy for this dataset as items such as shirts, t-shirts, shoes, and boots can look similar and present a challenge to the classifiers.

The two DNNs used have 95% and 72% accuracy, respectively. The lower accuracy is achieved by changing the number of layers, the training set size and the number of epochs of training.

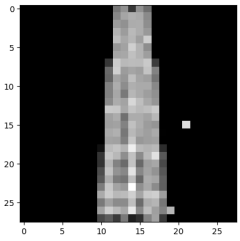

Figure 4.34: Bad DNN: Dress as Trouser

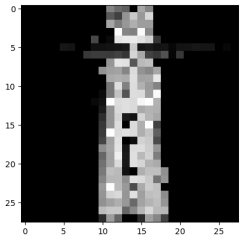

Figure 4.35: Good DNN: Dress as Trouser

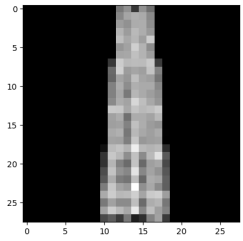

Figure 4.36: Reference

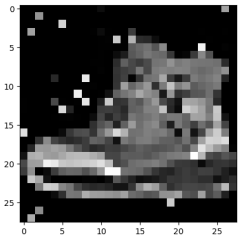

Figure 4.37: Bad DNN: shoe as sandals

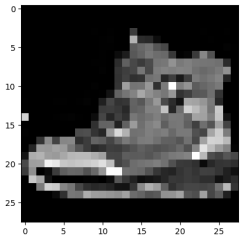

Figure 4.38: Good DNN: shoe as sandals

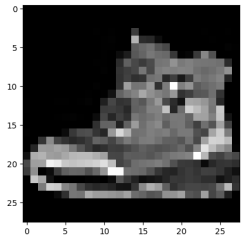

Figure 4.39: Reference

One would expect the DNN with higher accuracy to be more robust and reliable. Below are some images presented from tests generated using the Min Metric.

Figures 4.34 - 4.36 show clearly how just flipping a few pixels is enough to force the DNN with poorer accuracy to misclassify as compared to the DNN with higher accuracy. However, it is interesting to note that this is not always the case.

Figures 4.37 - 4.39 present a counterexample where the higher distortion is required to misclassify an input for the DNN with lower accuracy as compared to the DNN with higher accuracy. This shows that the accuracy metric alone does not tell the whole picture of a DNN.

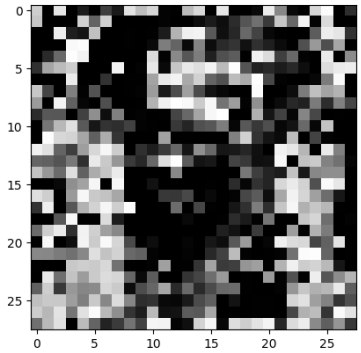

Figure 4.40: Bad DNN: Nothing as T-Shirt

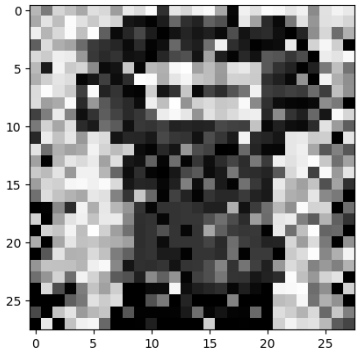

Figure 4.41: Good DNN: Nothing as T-Shirt

Max MNIST-Fashion

For MNIST-Fashion, Figures 4.40 - 4.43 are some Max test inputs generated for both the DNNs. Figures 4.40 and 4.42 are samples generated for the DNN with lower accuracy while Figures 4.41 and 4.43 are samples generated for the DNN with higher accuracy. Upon closer inspection of these images, it is possible to observe more distortion in the images generated for the DNN with lower accuracy as compared to the images generated for the DNN with higher accuracy. In this scenario, the DNN with lower accuracy performs worse in our Max test as compared to the DNN with higher accuracy. This behavior is probably more in-line with expectations.

From the test generation outputs for two DNNs of different levels of accuracy, we can see how our methods discriminate between them regarding reliability. We believe that such metrics offer a new perspective to assessing DNNs, in addition to the traditional metrics of accuracy.

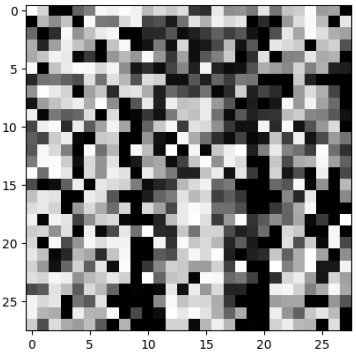

Figure 4.42: Bad DNN: Nothing as Trousers

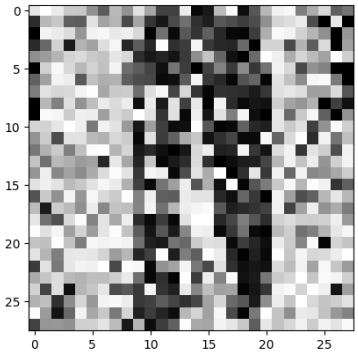

Figure 4.43: Good DNN: Nothing as Trousers

Chapter 5

Conclusion and Future Work

In this chapter, we draw conclusions from our work. We then go over the bottlenecks and drawbacks. Finally, we note all possible extensions and future work that can be undertaken.

5.1 Summary of the thesis

5.1.1 Summary of RTL Test generation on many and multi-core architectures

Although the algorithm for simulation-based test generators based on ant-colony optimization seem embarrassingly parallel, there is still a variety of issues when trying to parallelize it. In our work, first, we have extracted parallelism from the test generation framework itself and not from a single simulation. Running high volumes of native Verilog simulation, though prevalent, has always been an issue due to the cost of EDA licenses. Using an open source code based simulator allows us to run as many simulations as needed. By porting to a GPU framework, we see around 20× speedup. Thus the issue of choosing GPU vs. CPU for parallelism depends on several factors such as cost and size of the circuit. The GPU has difficulty scaling for larger circuits. It also provides significant performance enhancements for small to medium designs. CPUs, on the other hand, scale better for large implementations. In fact, it has been noted that similar GPU test generation has had issues

in fitting the kernel in device memory [4]. We might have to use MPI in place of OpenMP for a parallel implementation as the cores may no longer have a shared memory space. The multi-core CPU also does not suffer from issues such as branch divergence and are much easier to implement successfully as compared to the GPU. In terms of pricing, a GPU is generally much cheaper as compared to a high-end multi-core CPU. The choice then boils down to resource availability, problem size, and finances. Given that GPUs can work well for even medium to moderately sized circuits, test generation on GPU looks to be the better and less expensive choice, unless the problem size is prohibitively large. We should also keep in mind that the GTX 980 is a desktop GPU and not a server or a workstation GPU. Thus, the GPU implementation should scale gracefully for tomorrow's GPUs.

5.1.2 Summary of Testing DNNs

In this thesis, we have presented two new metrics, Min and Max, that are different from the existing Accuracy and Loss values commonly used to rate DNNs. We have also demonstrated the effectiveness of this metric to help assess the robustness and reliability of DNNs. From these generated tests we observe the reliability issues commonly seen with DNNs. With this work, we help to measure the same to provide the user with concrete feedback on the behavior of their DNNs. The new metrics can be used along with existing accuracy values to allow measuring the performance of a DNN. Thus it can help rate DNNs along a new dimension and perhaps help improve overall system safety in many important applications that use DNNs.

5.2 Limitations and Future Work

5.2.1 RTL Test generation on many and multi core architectures

Below are the biggest limitations of multi and many-core test generation.

- Working with the rest of the verification framework: Verification and Validation in the industry currently exists as a part of a bigger framework which consists of a lot of code and collateral designed to interact with the design under test. Thus any parallel test generation approach needs to identify ways to work with the rest of the framework if it hopes to be integrated into the industry.
- GPU Ram Size: One major limiting factor for porting test generation frameworks to the GPU is the GPU memory size. Given that GPU ram is in the order of a few GBs, based on the size of the design and input sequences, it is not hard to go over this limit. In fact, in our work, bigger circuits have issues scaling for a large number of simulations. This forces us to explore ways to factor the test generation algorithm in ways to not increase memory requirements.
- Suitable frameworks: The effectiveness of this approach is dependent upon how suitable the test generation framework is to a parallel approach. If there is no room for parallelism, or if the frameworks have requirements that go against the architectural restrictions of either the CPU or the GPU, then this approach is no longer applicable.

Below are some of the future works that can be conducted based on our current results:

- Create a more suitable algorithm: Although the Beacon presents embarrassing parallelism, it was not originally written for a parallel approach. This is true for many test

generation frameworks. It is possible to rewrite them in such a way to not only expose more parallelism but also write them in a way that can leverage the multi-threaded approach to improve test generation to improve coverage. One possible method, for BEACON, would be to organize ants as smaller colonies, where each colony target a particular set of branches and multiple colonies would have fewer reasons to communicate with each other. Such an implementation might fit the GPU as colonies can be organized as blocks which cannot synchronize, and ants inside colonies can be organized as warps in a block, which can coordinate using shared memory inside an SM.

- Beacon is a completely stochastic approach to test generation. It would be interesting to attempt a similar implementation for concolic test generation frameworks and record performance trade-off.

- Given that one major issue in performance is the memory bottleneck, if we were able to rewrite the translated RTL with fewer reads per arithmetic operation, then we might be able to overcome this. One other method to potentially increase performance would be to conduct an in-situ synthesis of the RTL model to a gate level description. This might lead to better performance as gate-level test generation lends parallelism through the levelization process that we can leverage for speedup.

5.2.2 Testing DNNs

Below are the limitation for our approach to testing DNNs:

- Our approach does not consider the values of the neurons in the hidden layer. Thus in out attempt to generate inputs based on our metrics, it is possible that we never exercise a set of neurons in the hidden layers. Thus it is entirely possible that if these

are the neurons that have *bad* weights, then we will not know for sure if we have attempted to excite them ever.

- It is possible for the values of these metrics to vary based on test generation approach. Furthermore, the test generation framework itself might not always be able to provide results that can help the user. Thus we would need to verify that this approach does not vary much based on the test generation methodology used and thus can provide reliable results.

Below are some directions of the possible future works:

- As mentioned earlier one possible future work would be experimenting with different test generation methods to improve the metrics reliability and accuracy.
- Another possible extension would be to implement this metric on non-image applications, with the aid of a domain expert to help us judge to outputs of our test generator.
- Lastly, it would be of immense benefit if we can discriminate between two DNNs of the same accuracy using Min and Max values. This information will prove to be highly valuable as it can help the user choose a DNN that actually performs better even though the *accuracy* values may not reflect this.

Bibliography

[1] assorted. *Opencores web page.* URL http://www.opencores.org.

[2] RICHEEK AWASTHI. *Breaking Deep Learning with Adversarial examples using Tensorflow*, 2018. URL http://cv-tricks.com/how-to/breaking-deep-learning-with-adversarial-examples-using-tensorflow/.

[3] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Exploring the space of black-box attacks on deep neural networks. *CoRR*, abs/1712.09491, 2017. URL http://arxiv.org/abs/1712.09491.

[4] N. Bombieri, F. Fummi, and V. Guarnieri. Fast-gp: An rtl functional verification framework based on fault simulation on gp-gpus. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 562–565, March 2012. doi: 10.1109/DATE.2012.6176532.

[5] Avishek Joey Bose and Parham Aarabi. Adversarial attacks on face detectors using neural net based constrained optimization. *CoRR*, abs/1805.12302, 2018. URL http://arxiv.org/abs/1805.12302.

[6] D. Chatterjee, A. DeOrio, and V. Bertacco. Gcs: High-performance gate-level simulation with gpgpus. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 1332–1337, April 2009. doi: 10.1109/DATE.2009.5090871.

[7] D. Chatterjee, A. DeOrio, and V. Bertacco. Event-driven gate-level simulation with gp-gpus. In *2009 46th ACM/IEEE Design Automation Conference*, pages 557–562, July 2009. doi: 10.1145/1629911.1630056.

[8] François Chollet et al. Keras. https://keras.io, 2015.

[9] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *IEEE Design Test of Computers*, 17(3):44–53, Jul 2000. ISSN 0740-7475. doi: 10.1109/54.867894.

[10] George Cybenko. *Universal approximation theorem*, 1989. URL https://en.wikipedia.org/wiki/Universal_approximation_theorem.

[11] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998. ISSN 1070-9924. doi: 10.1109/99.660313.

[12] K. Gulati and S. P. Khatri. Towards acceleration of fault simulation using graphics processing units. In *2008 45th ACM/IEEE Design Automation Conference*, pages 822–827, June 2008. doi: 10.1145/1391469.1391679.

[13] Simon Haykin. *Neural Networks: A Comprehensive Foundation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2007. ISBN 0131471392.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

[15] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Sequential circuit test generation using dynamic state traversal. In *Proceedings European Design and Test Conference. ED TC 97*, pages 22–28, March 1997. doi: 10.1109/EDTC.1997.582325.

[16] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Application of genetically engineered finite-state-machine sequences to sequential circuit atpg. *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems, 17(3):239–254, March 1998. ISSN 0278-0070. doi: 10.1109/43.700722.

[17] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Fast static compaction algorithms for sequential circuit test vectors. *IEEE Transactions on Computers*, 48(3):311–322, March 1999. ISSN 0018-9340. doi: 10.1109/12.754997.

[18] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.

[19] M. A. Kochte, M. Schaal, H. Wunderlich, and C. G. Zoellin. Efficient fault simulation on many-core processors. In *Design Automation Conference*, pages 380–385, June 2010. doi: 10.1145/1837274.1837369.

[20] D. Krishnaswamy, M. S. Hsiao, V. Saxena, E. M. Rudnick, J. H. Patel, and P. Banerjee. Parallel genetic algorithms for simulation-based sequential circuit test generation. In *Proceedings Tenth International Conference on VLSI Design*, pages 475–481, Jan 1997. doi: 10.1109/ICVD.1997.568180.

[21] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[22] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[23] H. Li, D. Xu, Y. Han, K. Cheng, and X. Li. ngfsim : A gpu-based fault simulator for 1-to-n detection and its applications. In *2010 IEEE International Test Conference*, pages 1–10, Nov 2010. doi: 10.1109/TEST.2010.5699235.

[24] M. Li, K. Gent, and M. S. Hsiao. Design validation of rtl circuits using evolutionary

swarm intelligence. In *2012 IEEE International Test Conference*, pages 1–8, Nov 2012. doi: 10.1109/TEST.2012.6401556.

[25] L. Liu and S. Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011. doi: 10.1109/DATE.2011.5763253.

[26] Li Min and Michael Hsiao. *High-Performance Diagnostic Fault Simulation on GPUs*.

[27] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. Scgpsim: A fast systemc simulator on gpus. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 149–154, Jan 2010. doi: 10.1109/ASPDAC.2010.5419903.

[28] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500.

[29] NVIDIA. *CUDA C Programming Guide 6.1*, 2014. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[30] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 2018. doi: 10.23915/distill.00010. https://distill.pub/2018/building-blocks.

[31] Travis E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition, 2015. ISBN 151730007X, 9781517300074.

[32] S. Pinto and M. S. Hsiao. Rtl functional test generation using factored concolic execution. In *2017 IEEE International Test Conference (ITC)*, pages 1–10, Oct 2017. doi: 10.1109/TEST.2017.8242038.

[33] Rashinkar Prakash, Paterson Peter, and Leena Singh. *System-on-a-Chip Verification*. Springer US, 2002. ISBN 978-0-7923-7279-0.

[34] H. Qian and Y. Deng. Accelerating rtl simulation with gpus. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 687–693, Nov 2011. doi: 10.1109/ICCAD.2011.6105404.

[35] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert. Parallel cycle based logic simulation using graphics processing units. In *2010 Ninth International Symposium on Parallel and Distributed Computing*, pages 71–78, July 2010. doi: 10.1109/ISPDC.2010.26.

[36] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. Testing deep neural networks. *CoRR*, abs/1803.04792, 2018. URL http://arxiv.org/abs/1803.04792.

[37] V Volkov. *Better performance at lower occupancy*, 2010. URL www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf.

[38] Snyder Wilson. *Verilator*. URL https://www.veripool.org/wiki/verilator.

[39] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017. URL http://arxiv.org/abs/1708.07747.

[40] Jason Yosinski, Jeff Clune, Anh Mai Nguyen, Thomas J. Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *CoRR*, abs/1506.06579, 2015. URL http://arxiv.org/abs/1506.06579.

[41] Y. Zhou, T. Wang, T. Lv, H. Li, and X. Li. Path constraint solving based test generation for hard-to-reach states. In *2013 22nd Asian Test Symposium*, pages 239–244, Nov 2013. doi: 10.1109/ATS.2013.52.

[42] Yuhao Zhu, Bo Wang, and Yangdong Deng. Massively parallel logic simulation with gpus. *ACM Trans. Des. Autom. Electron. Syst.*, 16(3):29:1–29:20, June 2011. ISSN 1084-4309. doi: 10.1145/1970353.1970362. URL http://doi.acm.org/10.1145/1970353.1970362.