RTL Test Generation on Multi-Core and Many-Core Architectures

Aravind Krishnan Varadarajan and Michael S. Hsiao
Department of Electrical and Computer Engineering
Virginia Tech, Virginia, USA 24061

Abstract— **Design Verification of complex circuits has become an increasingly time-consuming process. The advent of parallel computing using General Purpose Graphics Processing Units (GPGPUs) has led to enhanced 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. Not all applications can be made parallel due to complex dependencies, making parallelism non-trivial. Besides, large memory bandwidth requirements can become a bottleneck hindering potential speedups. In this paper, we have ported an RTL branch coverage based test generation framework to work within the constraints of a GPGPU environment. Experimental results show that considerable speedup can be achieved for test generation without loss of coverage.**

I. Introduction

Modern day processors and SOCs have grown extremely large in terms of transistor count due to increased design complexity and reduction in transistor size (Moores Law scaling). The increased complexity and circuit size directly impact design and development time and effort. Even moderate SOC designs would take days to run simulations on the whole design. Thus, any activity that uses simulation can end up becoming the bottleneck in the design process. Unfortunately, everything from functional verification to ATPG (Automatic Test Pattern Generation) and diagnostics uses simulation as formal methods scale poorly with design size and can easily become intractable for large circuits.

Estimates suggest that verification takes up to 70% of design time and effort and up to 80% of non-recurring engineering cost [1]. This highlights the need to reduce verification cost and time in order to reduce design time and to allow innovation in VLSI design accelerate.

Given the recent failure of Dennard's Law, 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. In order to achieve more performance, there has been a shift from high-performance single-core designs to multi-core designs. This has inspired and motivated programmers to exploit the parallel programming paradigm in order to continue to improve the performance of their programs. Unfortunately, it is hard to fit hardware simulation into the parallel programming model due to its high amount of synchronization and dependencies. Thus hardware simulation is yet to reap the performance benefits from parallel programming advances over the past decade. This is because of the state machine like behavior which prevents parallelizing across input cycles. There is also complex define use dependencies within every cycle.

Supported in part by NSF Grant 1422054

Given that simulation forms a significant component of hardware verification and test generation, verification and test generation have struggled with scaling 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. This includes many concrete simulation-based test generation frameworks which usually have multiple independent simulations which can be run in parallel.

For parallelism on the CPU alone, the application only needs independent tasks. 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. 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.

The contributions of this paper are as follows.

1) We present a framework to automatically convert synthesizable RTL Verilog to native GPU binary for test generation along with a test generation framework, BEACON [2], which also runs on the GPU itself.
2) We discuss the challenges and benefits that come along with trying to port the concrete RTL test generation framework to Graphic Processor Units.
3) We offer a comparison between a parallel CPU vs. GPU implementations of the same framework.

The rest of the paper is as follows. Section II discusses the background of GPUs and simulation-based verification. Section III describes the proposed methodology. Section IV presents the results, and Section V concludes the paper.

II. Background

Graphics Processing Units (GPUs): 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 better in Fig 1 [13].

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.

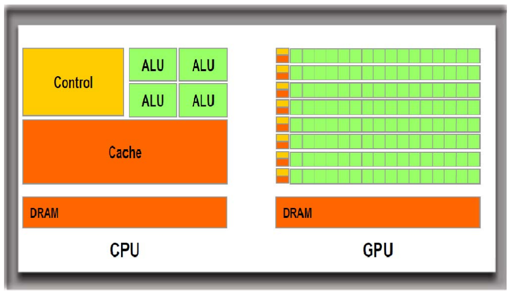

Fig. 1. Block diagram of CPU and GPU organization.

In the CUDA [3] 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 clever latency hiding and fast context switching. Most 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 lets us measure 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.
- 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.

Verilator: We use Verilator [4] 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.

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.

Algorithm 1 BEACON

```
1: procedure GEN_TEST($num\_cycles$,$num\_rounds$)
2:     $ants \leftarrow$ Initialize_ants()
3:     $ph\_map \leftarrow$ Initialize_map()
4:     $circuit \leftarrow$ 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:            Evolve($ant, ph\_map$)
16:        end for
17:    end for
18: end procedure
```

BEACON: The test generator we are trying to integrate with our GPU simulation engine is BEACON. BEACON is an ant colony optimization aimed at producing vector 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. An 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.

Hardware simulation is extremely time-consuming, and the difficulty in exposing parallelism dampens efforts to make it faster using a multithreaded approach. In this paper, we extract parallelism from the test generation algorithm itself. BEACON is a concolic test generation framework which involves running multiple simulations independent of each other. With little modification to the algorithm, parallelism is also extracted from tasks such as generation of new inputs and even updating of global states.

In BEACON, each ant is described by a test vector sequence, and fitness is assigned to it based on the number of hard to reach branches covered by it. All the branches in the design are assigned pheromone value based on the number of times all ants have covered it during a particular round of simulation. Hard to reach branches are characterized by the amount of pheromone or lack thereof. This is used to guide test generation in order to promote vectors to try and reach uncovered branches. There is no dependency between the simulation of different ants, and this can be done in parallel with no synchronization. Only updating the pheromone map requires any form of synchronization. Even recomputing the vectors for the next iteration of simulation can be done in parallel. Thus there is sufficient amount of parallelism to be exploited to warrant a multi-threaded approach. By porting BEACON to GPU, we hope to avoid the overhead of synchronization between GPU and CPU. Algorithm 1 shows the algorithmic flow in BEACON.

Related work

There has been GPU based gate level [5] simulation that focuses on error injection and use the GPU to run multiple independent simulations and get speedup. Each thread executes a simulation of one injected error. Since only one error is injected, the divergence may not be significant. This enables them to leverage thread level parallelism. A similar evaluation of design across different test and test generation for the design has been implemented on multicore systems using SystemC as opposed to Verilog RTL in [6]. There has been work done to extract speed up from a single RTL simulation on a GPU as seen in [7].

III. Methodology

We use OpenMP 3.0 [8] for the parallel implementation of BEACON on CPU and CUDA Toolkit version 9.2 for implementing on the GPU.

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 thread safe, and it degrades performance.

Parallel implementation on CPU

The implementation on a multi-core CPUs is 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 (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 vector along with the global hit counters to help recompute the vectors for the next round. Again a simple *"#pragma omp parallel for"* suffice but with dynamic scheduling as the number of cycles for which new inputs need to be generated varies for each ant.

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 the 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 [11].

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 try and 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.

A. 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 reduces 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.

B. 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
```

C. 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, until the last round of simulations. There is no cost of repetitive 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.

The ideal number of ants and cycles per ant depends on the circuit. More ants provide more parallelism but with diminishing returns. More cycles per ant might improve coverage. Nonetheless, we should try to completely utilize the GPU DRAM.

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 have 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 bounds and theoretical occupancy calculations, it 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 our results, we have tried to understand what features of our GPU kernel leads to speedup limitations.

IV. 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 [9] and [10]. We avoid comparison with other algorithms as they may be misleading. They use different platforms and different coverage goals and sometimes gate level models which behave differently when porting to a GPU.

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 exists 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.

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 $\approx$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 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 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 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 reduces 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.

The results for both the CPU and GPU are presented in Table I. For each circuit, the number of lines of code, 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$\times$ speedup with the GPU compared to a serial CPU implementation. On the CPU using 8 threads, we achieved only 3.87$\times$ speedup. Interestingly, on another circuit, *b11*, 38$\times$ speedup was achieved on the GPU and only 3.02$\times$ 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.

The framework scales much better with the number of ants on the GPU vs. CPU, as time taken increases linearly on the CPU, while remaining sub-linear on the GPU.

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 sections 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 it is possible that our kernel is memory bound.
- Instructions per warp and DRAM utilization: Our kernel averages 0.28/2 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 holds true for circuits of larger size including or1200, aes128 and openmsp430. 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 larger speed up for smaller circuits as noted in the Table I.
- Branch Divergence: A widespread and valid concern of modeling RTL Verilog as a CUDA instance 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$\times$ less speedup than if there were no divergence to our test generation

TABLE I
SPEED UP COMPARISON

Circuit	# of lines of C++/CUDA	BEACON param		Execution Time (sec)			Speedup over CPU serial	
		# Ants	# Cycles	CPU serial	CPU parallel	GPU	GPU	CPU Parallel
b11	143	8192	1000	9.28	3.07	0.24	38×	3.02×
ss_pcm	224	4096	1000	8.72	4.11	0.12	68.12×	2.12×
simple_spi	572	4096	1000	9.83	4.00	0.366	27.30×	2.46×
usb_phy	864	16384	1000	88.10	24.33	1.98	44.45×	3.62×
i2c	1059	16384	1000	54.02	19.29	1.54	34.85×	2.8×
b15	1397	16384	1000	50.62	20.20	1.39	36.41×	2.50×
openmsp430	5445	4096	1000	72.46	19.14	2.95	24.56×	3.79×
or1200	7474	4096	1000	66.76	17.23	3.71	17.99×	3.87×
aes128	28260	4096	500	112.62	23.98	6.94	16.12×	4.69×

process. Interestingly for the ss_pcm circuit, there is little branch divergence leading to better speedup.

- DRAM usage: Another bottleneck is 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 , 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.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 preferred by the GPU.

V. CONCLUSION

In our work, we have extracted parallelism from the test generation algorithm and not from factoring a single simulation. Running high volumes of native Verilog simulation, though prevalent, has 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. Porting the framework itself to the GPU allows us to overcome various issues seen when trying to parallelize RTL test generation, giving us around 20× speedup even for larger circuits.

The issue of choosing GPU vs. CPU for parallelism depends on factors such as cost and size of the circuit. The GPU has difficulty scaling for larger circuits, it has been noted that similar GPU test generation has had issues in fitting the kernel in device memory [12]. The exact design size limit depends on the circuit, the number of ants and number of cycles resulting in multiple permutations. But, it also provides significant performance enhancements for small to medium designs. CPUs, on the other hand, scale better for large implementations, without limiting DRAM size issues, albeit the problem of random number generation. 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. Thus, the GPU implementation should scale gracefully for tomorrow's GPUs.

REFERENCES

[1] Rashinkar, P., Paterson, P., Singh, L. (2007). System-on-a-chip verification: methodology and techniques. Springer Science & Business Media.

[2] M. Li, K. Gent, and M. S. Hsiao, "Design validation of RTL circuits using evolutionary swarm intelligence," in IEEE International Test Conference (ITC), 2012, pp. 1-8.

[3] John Nickolls, Ian Buck, Michael Garland, Kevin Skadron, "Scalable Parallel Programming with CUDA", ACM Queue, vol. 6 no. 2, March/April 2008

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

[5] M. Li and M. S. Hsiao, "High-Performance Diagnostic Fault Simulation on GPUs," 2011 Sixteenth IEEE European Test Symposium, Trondheim, 2011, pp. 210-210.

[6] K. Gulati and S. P. Khatri. Towards acceleration of fault simulation using graphics processing units. In 2008 45th ACM/IEEE Design Automation Conference, pages 822827, June 2008. doi: 0.1145/1391469.1391679.

[7] Hao Qian, Yangdong Deng. Accelerating RTL Simulation with GPUs. ICCAD &#39;11 Proceedings of the International Conference on Computer- Aided Design Pages 687-693.

[8] Dagum, Leonardo and Menon, Rames, "OpenMP: An Industry-Standard API for Shared-Memory Programming", IEEE Comput. Sci. Eng. vol.5 no.1 pp. 46-55, January 1998

[9] S. Davidson, ITC99 benchmark circuits - preliminary results, in Proceedings International Symposium, Circuits & Systems, p. 1125, 1999.

[10] OpenRISC web page. http://www.opencores.org.

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

[12] N. Bombieri, F. Fummi and V. Guarnieri, "FAST-GP: An RTL functional verification framework based on fault simulation on GP-GPUs," 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2012, pp. 562-565. doi: 10.1109/DATE.2012.6176532

[13] NVIDIA, CUDA C Programming Guide 6.1, 2014.