FAST-GP: An RTL Functional Verification Framework based on Fault Simulation on GP-GPUs

Nicola Bombieri, Franco Fummi and Valerio Guarnieri
Department of Computer Science
University of Verona
{firstname.lastname}@univr.it

***Abstract*—This paper presents FAST-GP, a framework for functional verification of RTL designs, which is based on fault injection and parallel simulation on GP-GPUs. Given a fault model, the framework translates the RTL code into an injected C code targeting NVIDIA GPUs, thus allowing a very fast parallel automatic test pattern generation and fault simulation. The paper compares different configurations of the framework to better exploit the architectural characteristics of such GP-GPUs (such as thread synchronization, branch divergence, etc.) by considering the architectural characteristics of the RTL design under verification (i.e., complexity, size, number of injected faults, etc.). Experimental results have been conducted by applying the framework to different designs, in order to prove the methodology effectiveness.**

I. Introduction

Fault injection and fault simulation are among the most widely adopted techniques for dynamically verifying RTL designs [1], [2] as they provide a measure of quality of the used testbenches and test patterns [3]. In the past years, several methodologies for automatic test pattern generation (ATPG) have been proposed [4] and different fault models [5] have been proposed for providing a comprehensive measure of the RTL test pattern quality.

Different works have shown that RTL fault coverage is quite close to fault coverage achieved at the gate level when designs are completed and mapped to a technology library [5]. Experimental results have also shown that the designer effort to improve the fault coverage at RTL very likely results in a corresponding improvement of fault coverage at gate level. All these works have underlined, besides the need of accurate fault models, the importance of a fast RTL simulation speed to generate high quality RTL test patterns.

On the other hand, graphics processing units (GPUs) have recently been explored as a new general purpose computing paradigm for accelerating computation intensive EDA applications, such as gate-level fault simulation [6], fault table computation [7], and logic simulation [8].

This paper presents FAST-GP, an RTL functional verification framework for accelerating fault simulation with GP-GPUs. Given a fault model, the RTL design under verification is automatically instrumented (i.e., fault injected) and translated into C code targeting NVIDIA GPUs [9] (see Fig. 1).

The generation process translates the instrumented cycle accurate RTL code into an equivalent instrumented cycle accurate C code. The process guarantees that all and only the faults detectable by the RTL simulation are detectable by

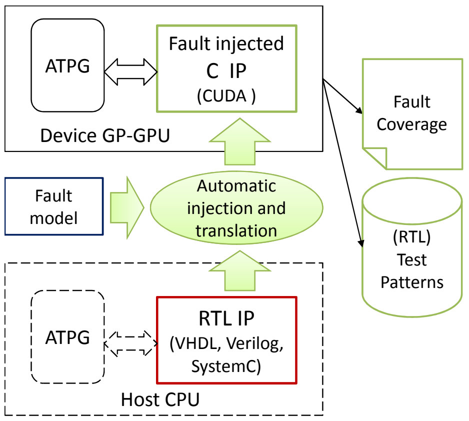
ATPG
Fault injected C IP (CUDA)
Device GP-GPU
Fault Coverage
Fault model
Automatic injection and translation
(RTL) Test Patterns
ATPG
RTL IP (VHDL, Verilog, SystemC)
Host CPU

Fig. 1. RTL fault simulation: from CPUs to many-core GP-GPU devices

the GP-GPU simulation. As a consequence, the fault coverage obtained by the GP-GPU simulation still refers to the given RTL fault model, and the test patterns generated by the GP-GPU simulation (and that detect faults) can be automatically translated into RTL test patterns. The C code is enriched with an high level process scheduler to preserve, in the GP-GPU simulation, the event-driven behavior of the RTL simulation.

The paper also shows how the SIMD characteristics of GP-GPUs can be exploited for RTL fault simulation, by proposing different framework configurations. Finally, the paper reports an analysis of the experimental results to explain how such different configurations better apply depending on the architectural characteristics of the RTL IPs.

The rest of the paper is organized as follows. Section I-A presents the related work. An overview of the FAST-GP architecture is presented in Section II. Section III presents the code instrumentation. Section IV presents the main concepts of the RTL code translation into C CUDA code. Section V present the configurations of parallel code execution. Section VI shows the experimental results, while section VII is devoted to concluding remarks.

A. Related work

Many-core architectures have been applied for accelerating computation intensive EDA applications and, in particular, logic simulation [8], [10], [11] and gate-level fault simulation [6], [12], [13]. In [8], the authors present an event-driven gate-level simulator that leverages a design to exploit the benefits of the low switching activity typical of large HW designs. In [10], the authors present a GPU-accelerated logic simulator optimized for large structural netlists. [11] presents a different

978-3-9810801-8-6/DATE12/©2012 EDAA

parallel cycle-based logic simulation algorithm that uses And Inverter Graphs (AIG) as design representations.

Different fault simulation techniques with GP-GPUs have been proposed [6], [12], [13]. In [6], the framework fault simulates all the gates in a particular level of a circuit, including good and faulty circuit simulations, for all patterns, in parallel. Each of the fault simulation threads uses memory lookup. In [12], the authors propose to map a fault simulation algorithm based on the parallel-pattern single-fault propagation (PPSFP) paradigm to many-core architectures. [13] presents a GPU-based fault simulator for stuck-at faults which can report the fault coverage of one to n-detection for any specified integer n using only a single run of fault simulation. All approaches are implemented for NVIDIA CUDA devices (GTX 8800, GT200, and GTX9800, respectively) and achieve speed-up around 30x compared to existing fault simulation engines based on conventional CPUs. Although all these techniques meaningfully apply to fault simulation of gate-level circuits, they do not apply to RTL fault simulation. What is missing, and what is proposed in this paper, is a technique that (i) given a high level (i.e., RTL) fault model which can be represented by code instrumentation, automatically injects faults, and (ii) automatically translates the injected RTL to injected C code targeting GP-GPU architectures (i.e., CUDA) by preserving the fault testability in the GP-GPU simulation.

II. FAST-GP ARCHITECTURE: AN OVERVIEW

FAST-GP addresses the following main issues:

- Given any high level (i.e., RTL) fault model, the faults should be automatically injected, and the injected RTL code should be automatically translated into injected C code targeting GP-GPU architectures.
- The translation technique should guarantee that all and only the faults detectable by the RTL simulation are detectable by the GP-GPU simulation.
- The fault coverage obtained by the GP-GPU simulation should still refers to the given RTL fault model, and the test patterns generated by the GP-GPU simulation (and that detect faults) should be automatically translated into RTL test patterns.

The main flow (illustrated in Figure 1) consists of the following steps:

1) Given any RTL fault model that can be represented by a code instrumentation, the RTL faults are injected into the starting RTL model, as described in Section III.
2) The injected RTL model is translated into a corresponding C description, in order to be compiled and executed within the CUDA framework. This step is detailed in Section IV.
3) The simulation of the injected RTL design is executed in parallel on GPUs through the CUDA framework. This involves selecting a configuration in order to organize parallel execution, as deepened in Section V. The corresponding fault coverage is achieved by parallel simulation of the C description.

III. FAULT INJECTION

The fault injection technique proposed in this paper relies on *injection functions* added to the RTL code. Faults are enumerated, i.e., each fault is associated with a number belonging to the range [1, N]. Fault number 0 is used as special value to simulate the fault-free description, (i.e., no fault is activated). During simulation, an integer-type port (i.e., `fault_port`) is used to activate each fault by driving the number associated with it. According to the adopted fault model, for each type of object that can be injected in the description, an injection function is defined. An example of such a function is as follows:

```
int inject_fault_type(int obj, int start_range
    , int end_range);
```

Parameter `obj` provides the object to be injected. Parameters `start_range` and `end_range` define the range of values to be driven on `fault_port` to activate the associated faults injected into the object. Such functions are implemented so that they check whether the value of `fault_port` belongs to the range [`start_range`, `end_range`]. They then provide the correct or faulty value of the target object accordingly.

The proposed methodology is independent from the specific implementation of these functions. Their complexity can greatly vary according to the fault model being implemented.

IV. TRANSLATION FROM RTL TO C CUDA

FAST-GP generates CUDA C code similarly to [14], [15]. It translates one injected RTL IP into one *injected CUDA kernel* (called `main_IP()`), which can be executed by thousands of threads in different configurations (see Section V).

There is a one-to-one mapping of each RTL concurrent statement (i.e., processes, global action) into C procedures. Each *synchronous* process sp_i is translated into a C procedure $\texttt{psp}_i\texttt{()}$. There is a one-to-one mapping of each RTL sequential statement (i.e., statements inside a process) into C sequential statements. Thus, the C procedure body contains the set of sequential statements of sp_i. In the same way, each *asynchronous* process ap_i is translated into a C procedure $\texttt{pap}_i\texttt{()}$. The procedure body contains the set of sequential statements of ap_i.

The RTL interface is translated into a C data structure (`io_struct`). There is a one-to-one mapping of each I/O port composing the RTL interface and the fields of such an I/O data structure. Input test patterns are written to the fields corresponding to input ports. The results of the IP simulations are read from the fields corresponding to output ports.

The generated CUDA C code includes a *dynamic scheduler*.[1] We adopted dynamic scheduling since it allows an immediate proof of functional equivalence between the starting and the generated description. Static scheduling, which will make the C code smaller and faster, is part of our future work. In order to preserve the event-driven behavior of the RTL design in the C code, each RTL signal sig is translated into two C variables (sig and sig_new). This allows to implement RTL events (due to the changing value of the signals) and to mimic the deferred assignment of RTL signals also in the C description. The dynamic scheduler, which is implemented with a C procedure, calls the C procedures

[1]With *dynamic scheduling*, the set and the order of processes to wake up is scheduled at run time at the beginning of each simulated clock cycle. In contrast, *static scheduling* resolves the process scheduling before simulation.

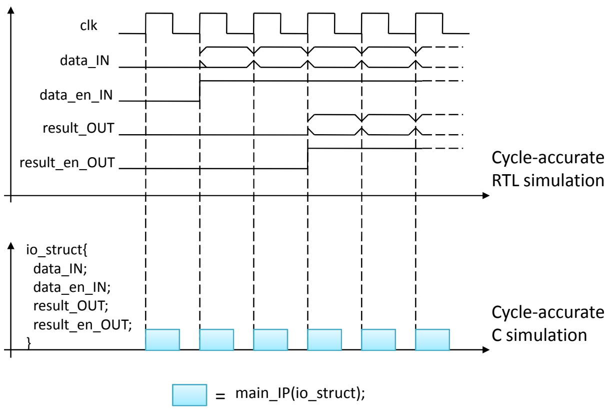

Fig. 2. Cycle-accurate simulation: RTL vs. C

$\texttt{psp}_i\texttt{()}$ and $\texttt{pap}_i\texttt{()}$ in the same sequential order as the corresponding RTL processes were sequentially executed by the HDL scheduler.

The comparison of the cycle accurate execution of the C kernel code and the RTL code is illustrated in Figure 2. For each clock cycle, `main_IP()` is executed, which reads the input test sequences (`data_IN` and `data_en_IN` in the example) provided by the ATPG and returns the resulting outputs (`data_OUT` and `data_en_OUT`) for the fault testability check through the `io_struct` payload. The cycle accurate execution of the C code and the proposed structure of the I/O guarantee an immediate translation of the ”useful” test patterns generated in the C simulation into RTL test patterns.

A simulation run that detects a given fault using a given test sequence has no dependencies with other simulation runs. Therefore, fault simulation can be accelerated by executing simulation runs in parallel along two basic dimensions (i.e., faults and test sequences), as explained in the next Section.

V. Configurations for parallel kernel execution

The kernel *configuration* plays a pivotal role in parallelizing the simulation, since it specifies the dimensions of the grid and thread blocks. Five different kernel configurations have been analyzed, each one providing advantages and drawbacks.

Configuration #1. It arranges faults and test sequences along the two dimensions of the grid. This leads to the creation of thread blocks consisting of only a single thread, which performs the simulation of a given fault on a given input sequence. This configuration is not affected by the problem of branch divergence. However, it results in a significant waste of the computational resources made available by the GPU device. CUDA threads are partitioned for execution into 32-thread warps, and all threads in a warp are executed concurrently. This configuration leads to executing warps consisting of a single thread only, instead of 32. Thus, an exceedingly large number of cores in the GPU device will remain idle during the execution of a warp.

Configuration #2. It creates as many thread blocks as injected faults. Each thread block is responsible for simulating a given fault on an array of test sequences. This configuration yields a much higher utilization range of the computational resources of the GPU device with respect to configuration #1. It also allows to optimize the simulation of a fault by stopping the simulation as soon as a test sequence detects the fault. However, it is affected by branch divergence within the same warp, since different test sequences may lead to different branches taken in the control flow of the C description. It is particularly fitting for designs with few branch conditions.

Configuration #3. It extends simulation capabilities by arranging test sequences along two dimensions, one in the grid and one in the thread blocks. This enables simulation of an extremely large set of parallel test sequences. The major drawback is that it does not allow the aforementioned optimization on fault detection. Once a test sequence detects a fault, all threads within the same block may stop, but this does not apply to other thread blocks simulating the same fault on different test sequences, as CUDA does not allow synchronization between threads in different blocks.

Configuration #4. It creates as many thread blocks as test sequences, so that each block is responsible for simulating a given set of test sequences on all injected faults. This configuration is affected by branch divergence within the same warp, since different faults are activated, thus leading to different branches in the control flow. Such divergence pertains only to the execution of the injection function implementing a fault. Since all threads in the block are associated to the same test sequence, there is no branch divergence caused by different inputs. Configuration #4 suits designs with many conditional instructions. This configuration cannot be applied when the number of injected faults exceeds the constraint on the size of a thread block.

Configuration #5. It allows for an extensive set of test sequences and, together with #3, is the configuration that yields the best utilization range of the HW resources provided by the GPU device. Nevertheless, this configuration is the most prone to branch divergence. In fact, different faults being activated in the same warp introduce branch divergence, as well as different test sequences being simulated in the same warp do. Just like the previous one, this configuration cannot be applied when the number of injected faults exceeds the constraint on the size of a thread block.

VI. Experimental Results

We present the results obtained by applying the proposed framework to different RTL models. The model features are reported in Table II. The last three columns of the Table indicate the lines of code of the fault-free and the injected CUDA C description, and the size of the compiled CUDA C kernel object file, respectively.

The adopted fault model relies on the injection of mutants that alter the design functionality. These mutants operate on operators, assignments and constants in the description. They mimic the introduction of design errors, thus modeling a permanent fault.

Experiments have been performed on a 64-bit Linux server with six 2.8 GHz CPU cores and equipped with a NVIDIA GeForce GTX 460 device.

The left part of Table I reports the results in terms of simulation time speedup. Column *Faults* lists the number of injected faults. Column *Sequences* indicates the number of test sequences. Column *Cov* provides the fault coverage achieved. Columns *RTL time* and *C serial time* indicate the simulation

Model	Faults (#)	Sequences (#)	Cov (%)	RTL time (s)	C serial time (s)	CUDA C time (s)	Speed-up RTL vs CUDA C (x)	Speed-up C serial vs CUDA C (x)	Cfg #1 (s)	Cfg #2 (s)	Cfg #3 (s)	Cfg #4 (s)	Cfg #5 (s)
8b10b	475	131K	94.8	567.9	63.7	1.1	516x	58x	97.15	2.6	1.1	9.3	24.3
adpcm	176	8M	98.1	12,609	617.7	8.8	1,436x	70x	290.9	8.8	8.8	11.3	10.6
dist	142	512K	91.0	8,598	2,087	9.9	867x	210x	290.0	9.9	21.9	35.8	39.7
qnt	477	1M	92.6	13,341	466.0	10.2	1,302x	45x	320.1	10.2	12.5	34.6	41.6
rle	321	4M	94.9	7,297	328.5	3.7	1,956x	88x	133.2	3.7	4.1	9.3	13.3
root	54	845K	92.2	3,364	1,503	10.8	312x	139x	614.0	12.8	10.8	48.4	43.5

TABLE I
EXPERIMENTAL RESULTS.

Model	LoC VHDL	PIs	POs	Gates	LoC CUDA ff	LoC CUDA inj	Kernel size (KB)
8b10b	277	9	10	503	1,006	1,986	7,653
adpcm	284	66	35	24,412	297	652	773
dist	325	130	64	40,663	726	1,052	2,891
qnt	358	24	16	17,645	311	728	1,253
rle	519	16	19	2,493	472	923	1,432
root	192	66	64	7,802	210	394	717

TABLE II
FEATURES OF RTL MODELS.

time of the SystemC RTL description and the C description executed serially (i.e., by a single CPU), respectively. Column *CUDA C time* shows the simulation time with CUDA by employing the most efficient kernel configuration. The following two columns provide the speed-up between the RTL and the CUDA C and between the C serial and the CUDA C simulation, respectively.

The achieved speed-up with respect to the SystemC RTL simulation ranges from two to three orders of magnitude. This is due to the great CUDA architecture performance as well as to the low efficiency of SystemC RTL simulation [16]. On the other hand, the speed-up remains within two orders of magnitude with respect to the C serial simulation. The great variety between the simulation speed-up depends on the architectural characteristics of the models. For example, the high amount of computation performed by *dist* makes it an ideal candidate for a greater speed-up. The lower amount of computation and granularity of operations performed (single bit level) in *8b10b* give a lower speed-up.

The right part of Table I reports the simulation time for each kernel configuration on each model description. Configuration #1 is consistently the slowest, since it creates too few threads, thus exploiting only a minimal part of the available computational resources of the GPU device. Configurations #2 and #3 prove to be the most efficient. #2 achieves best results when its optimization prevents the simulation of a large number of test sequences as soon as a fault is detected. Conversely, #3 performs better when a large number of test sequences are simulated before detecting a fault, thus rendering the previous optimization ineffective. Configurations #4 and #5 follow right behind, hampered by a greater influence of branch divergence.

The main limitation to the proposed methodology lies in the limit to the maximum number of instructions per kernel function, which amounts around to 2 million [17]. This is a limit imposed by the CUDA framework which affects the complexity of the instrumented C code representing the behavior of the injected design. In case of complex descriptions, where a very high number of faults are injected, the solution we propose is to perform partial instrumentations at a time, in order to keep size of the kernel function within the limit. This issue will be addressed in our future work.

VII. CONCLUDING REMARKS

In this paper we presented FAST-GP, an RTL functional verification framework for accelerating fault simulation with GP-GPUs. Given an RTL fault model, the framework automatically injects faults in the RTL design under verification and translates the RTL code into C code targeting NVIDIA GPUs. The translation mechanism guarantees that the RTL fault testability is preserved in the GP-GPU simulation, and that the useful test patterns generated in such a parallel simulation can be directly translated into RTL test patterns. Finally, the paper shows and discusses different framework configurations and the corresponding results.

REFERENCES

[1] L. Dongwoo and N. Jongwhoa, "A novel simulation fault injection method for dependability analysis," *IEEE Design and Test of Computer*, vol. 26, no. 6, pp. 50–61, 2009.

[2] N. Bombieri, F. Fummi, and V. Guarnieri, "Accelerating RTL fault simulation through RTL-to-TLM abstraction," in *Proc. of IEEE ETS*, 2011, pp. 117–122.

[3] S. Park, L. Chen, P. K. Parvathala, S. Patil, and I. Pomeranz, "A functional coverage metric for estimating the gate-level fault coverage of functional tests," in *Proc. of IEEE ITC*, 2006, pp. 1–10.

[4] Z. Liang, I. Ghosh, and M. Hsiao, "A framework for automatic design validation of RTL circuits using ATPG and observability-enhanced tag coverage," *IEEE Trans. on CAD*, vol. 25, no. 11, pp. 2526–2538, 2006.

[5] P. Thaker, V. Agrawal, and M. Zaghloul, "A test evaluation technique for VLSI circuits using register-transfer level fault modeling," *IEEE Trans. on CAD*, vol. 22, no. 8, pp. 1104–1113, 2003.

[6] K. Gulati and S. P. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Proc. of ACM/IEEE DAC*, 2008, pp. 822–827.

[7] ——, "Fault table computation on GPUs," *J. Electron. Test.*, vol. 26, pp. 195–209, April 2010.

[8] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with GP-GPUs," in *Proc. of ACM/IEEE DAC*, 2009, pp. 557–562.

[9] NVIDIA, "Cuda home page," http://www.nvidia.com/object/cuda_home_new.html.

[10] D. Chatterjee, A. DeOrio, and V. Bertacco, "GCS: high-performance gate-level simulation with GP-GPUs," in *Proc. of ACM/IEEE DATE*, 2009, pp. 1332–1337.

[11] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert, "Parallel cycle based logic simulation using graphics processing units," in *Proc. of IEEE ISPDC*, 2010, pp. 71–78.

[12] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin, "Efficient fault simulation on many-core processors," in *Proc. of ACM/IEEE DAC*, 2010, pp. 380–385.

[13] 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 *Proc. of IEEE ITC*, 2010, pp. 1–10.

[14] N. Bombieri, F. Fummi, and G. Pravadelli, "Abstraction of RTL IPs into embedded software," in *Proc. of ACM/IEEE DAC*, 2010, pp. 24–29.

[15] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: a fast SystemC simulator on GPUs," in *Proc. of ACM/IEEE DAC*, 2010, pp. 149–154.

[16] W. Ecker, V. Esen, L. Schonberg, T. Steininger, M. Velten, and M. Hull, "Impact of description languages, abstraction layer, and value representation on simulation performance," in *Proc. of ACM/IEEE DATE*, 2007, pp. 767–772.

[17] *CUDA C Programming Guide*, NVIDIA, http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf.