nGFSIM : A GPU-Based Fault Simulator for 1-to-*n* Detection and its Applications

Huawei Li[1,2], Dawen Xu[1,3], Yinhe Han[1], Kwang-Ting Cheng[2], Xiaowei Li[1]
[1]Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]Department of Electrical and Computer Engineering, University of California, Santa Barbara
[3]Graduate University of Chinese Academy of Sciences
{lihuawei, xudawen, yinhes}@ict.ac.cn; timcheng@ece.ucsb.edu; lxw@ict.ac.cn

Abstract

We present nGFSIM, 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. nGFSIM, which explores the massive parallelism in the GPU architecture and optimizes the memory access and usage, enables accelerated fault simulation without the need of fault dropping. We show that nGFSIM offers a 25X speedup in comparison with a commercial tool and enables new applications in test selection.

1. Introduction

Results of testing experimental chips have revealed that even though the single stuck-at model is inaccurate to model real defects, a single stuck-at test set with each stuck-at fault detected more than once may be effective for detecting both hard and timing failures [1]. An *n*-detection test set, including *n* different tests for each target stuck-at fault, increases the probability to detect un-modeled defects at the fault site. Generation and evaluation of an n-detection test set have been studied in recent years [2-5].

An efficient *n*-detection fault simulation process is key to the application of *n*-detection tests. First, it can be used to accelerate the *n*-detection test generation process, or minimize the test set for any given *n* [6-7], as fault simulation is always an integrated process in automatic test pattern generation (ATPG) and test compaction. Second, for a given test set, *n*-detection fault simulation can be used for test selection to derive a more compact test set, or for test ordering to obtain a fault coverage curve which is as steep as possible [8-9]. Such ordered test patterns can help reduce the test time for screening defective chips.

Unique detections of failing units for individual *n*-detection test up to *n*=6 were reported in [2]. In [10], a random excitation and deterministic observation (REDO) method is proposed for multiple fault detections, which produces a compact test pattern set and achieves a low defective part level as well. But it is still unclear what is, or how to determine, the best choice for the value of *n* when testing a chip. A test set may have a higher fault coverage for a small *n*, but a lower fault coverage for a large *n*, in comparison with another test set. So it is difficult to judge which one has a better test quality. To achieve more optimized defect-oriented testing, it may be necessary to target a combination of multiple *n's*, e.g., *n*=1, 5, 10, to generate a test set with optimized fault coverage for all the 1-, 5-, and 10-detections. However, this application would require a powerful fault simulation tool which is capable of reporting *n*-detection fault coverage for multiple, specified *n's*.

n-detection fault simulation is usually modified from a single-detection fault simulation process which drops a fault from the fault list only after it is detected at least *n* times. However, specifying a value for *n* in advance will cause inaccuracy in counting the number of detected faults for each test, and result in a non-optimal order of the tests, as illustrated in [11]. In contrast, *n*-detection fault simulation without fault dropping can achieve optimal results, but is computationally expensive. Approximation of optimal results can be obtained by using an *n*-pass, *n*-detection fault simulation process, which has a computational complexity similar to that of *n*-detection fault simulation with fault dropping [11] and does not require to pre-specify the value of *n*.

Graphics Processing Units (GPUs) have recently been explored as a new general-purpose computing platform. Most GPUs follow a single-instruction multiple-data (SIMD) architecture and often can have thousands of threads running concurrently. This feature makes the GPU suitable for acceleration of computation-intensive EDA applications, such as fault simulation [12], fault table computation [13] logic simulation [14], power grid analysis [15], and statistical timing analysis [16].

In this paper, we present *nGFSIM*, 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. With the ability of efficient fault simulation without fault dropping, we further present some applications of *nGFSIM* for *n*-detection test methods. The main contributions and key results of this paper are as follows.

This paper is supported in part by National Natural Science Foundation of China (NSFC) under grant No. 60776031, 60633060, 60921002, and in part by National Basic Research Program of China (973) under grant No. 2005CB321605.

978-1-4244-7207-9/10/$26.00 ©2010 IEEE

- By exploring the massive parallelism in the GPU architecture and optimizing the memory access and usage, *nGFSIM* enables accelerated fault simulation without the need of fault dropping. Given a test set, *nGFSIM* can produce a complete fault detection table that records the per-pattern detectability of faults, and count the number of detections of each fault. Simulation results show that *nGFSIM* is 20X-25X faster for computing *n*-detection fault coverage in comparison with a commercial fault simulation tool.
- With *nGFSIM*'s ability of producing the complete fault detection table without extra cost, we propose a parallel test set ordering algorithm which utilizes the information for ordering the tests to achieve the steepest 1-to-*n* detection fault coverage curves, and also for selecting high-quality 1-to-*n* detection tests from any given large set of test patterns.
- With the applications and experiments enabled by *nGFSIM*, we are able to make the following observations from the results:
  - A deterministically generated test set targeting (*i*+1)-detection always outperforms a deterministically generated test set targeting *i*-detection in terms of their *n*-detection fault coverage for *n* larger than *i*+1. This observation indicates the necessity of considering a larger *n* for *n*-detection to improve the test quality.
  - A random test set, with a large pattern count (say, 32K in the experiment), has a roughly uniform *n*-detection fault coverage for different *n*'s, and achieves a higher *n*-detection fault coverage than a deterministically generated *i*-detection test set when *n* is much larger than *i* (we show the experimental results for *n*=63 and *i*=5).
  - Under the constraint of a limited pattern count for testing, tests selected by our test set ordering algorithm based on *nGFSIM* achieve a superior fault coverage curve for 1-to-*n* detections, in comparison with the one-detection deterministic tests generated by a commercial ATPG tool.

The remainder of this paper is organized as follows. Section 2 introduces the architecture of the GPU used in the implementation of *nGFSIM*. Section 3 discusses parallel fault simulation using GPUs, and presents the detail of the proposed fault simulation method and its mapping to the GPU's architecture. In Section 4, we extend our method to 1-to-*n* detection fault simulation for any specified integer *n*, and propose its applications to test ordering and test selection. Experimental results are given in Section 5, and conclusions are drawn in Section 6.

2. Architecture of the GPU

In this section, we introduce the architecture of NVIDIA GeForce GTX9800 GPU, which is used in our implementation. NVIDIA GeForce GTX9800 GPU is composed of 16 multiprocessors per chip and 8 stream processors per multiprocessor. All the stream processors in a multiprocessor execute the same instruction, but on different data.

Figure 1 shows the hardware hierarchy of GeForce GTX9800 GPU. The total DRAM (device memory) in the GPU is 512 MB, which can be used as global, local, and texure memory. The global and local memory are not cached, while the texture memory is cached by read-only texture caches. The size of a texture cache is 8 KB per multiprocessor, which is shared by all the stream processors in a multiprocessor. There is also a shared memory of 16 KB in each multiprocessor, which is shared by the 8 stream processors.

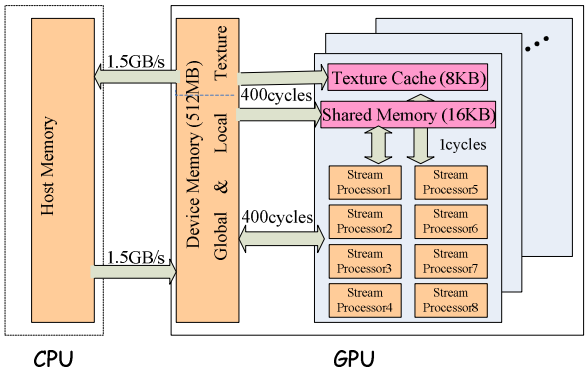


Figure 1 Hardware Hierarchy of GeForce GTX9800

The memory access bandwidth and latencies are also shown in Figure 1. The data transfer bandwidth between the host (CPU) and the GPU is 1.5 GB/s. To maximize the computational power of the GPU, the communication between the host and the GPU should be minimized. Since the global memory is not cached, it takes about 400 cycles to read from or write to it. However, coalesced access in units of 32, 64, or 128 bits can be used which could help reduce the access latency of the global memory by a dozen times. The latency of read from the texture memory, though in worst case is the same as the access latency from the global memory, in the best case is one cycle only (through accessing the texture caches).

NVIDIA provide a programming interface for general purpose computing, which is called Compute Unified Device Architecture (CUDA) [17]. When programming using CUDA, the GPU is viewed as a coprocessor to the CPU. A main program, running on the CPU, off-loads compute-intensive tasks onto the GPU, by invoking a great quantity of threads executing in parallel on the GPU. In CUDA, a thread is the basic execution unit. Each thread has its unique thread ID. Threads are grouped in wraps, which are further grouped into blocks. Each block is assigned to a multiprocessor for execution. A task in a GPU is corresponding to a grid which contains blocks of threads executing the same kernel. For GeForce GTX9800 GPU, the wrap size is 32, and in the best case 16 million threads can be invoked in parallel.

To fully explore the massive parallelism provided by a GPU, a huge number of threads should be invoked in parallel for any efficient GPU-based program. If every thread has to read data from CPU and write results back to CPU, the amount of data transmission between the CPU and the GPU would be very large, which could degrade or even completely dominate the overall performance of the

GPU-based program. So it is desirable to design and implement the program that threads get their main data from and write their results to the global memory of the GPU. Except the initial data and the final results, the data transfer between the CPU and the GPU during and between thread executions should be minimized. It is better to store intermediate results of threads, which need to be accessed by other threads in later computation, in the GPU's global memory, instead of in the host memory.

3. GPU-based Parallel Fault Simulation

3.1 Related Work

The first attempt to use GPUs for fault simulation of combinational circuits was reported in [12], which performs a large number of tiny threads to implement a data-parallel simulation using a conventional forward fault simulation flow for all faulty circuits. This approach, implemented in an NVIDIA GeForce GTX 8800 GPU card, evaluates logic gates in the same logic level in parallel, for both fault-free and faulty circuits. Three types of kernels are implemented: 1) the logic simulation kernel for evaluation of each fault-free gate, 2) the fault simulation kernel for evaluation of each faulty gate and each gate in the transitive fanout of the faulty gate, and 3) the fault detection kernel for comparing the results at each primary output in the transitive fanout of the faulty gate. All the gate evaluations are performed using look-up table based computations. This implementation demonstrated a 35X speedup, in comparison with a CPU-based commercial fault simulation engine running on a 1.5 GHz UltraSPARC-IV+ processor with 1.6 GB of RAM.

Using tiny threads is beneficial for maximizing computation parallelism. However, it also increases the amount of data transmission between the CPU and the GPU. In the implementation detailed in [12], most of the relevant information about the circuit under simulation is not statically stored in the GPU. Therefore, the parameters to be used by each tiny thread, including values of gate type, gate inputs, fault type, and addresses of required data in the global memory, need to be transferred from the CPU to the GPU when invoking a thread. Since a gate is evaluated multiple times for both logic and fault simulation under different groups of patterns, a lot of duplicated data are transferred between the CPU and the GPU. Based on a simple estimation, the number of invoked tiny threads is roughly proportional to the square of the gate count in the circuit. Transferring the required data for this huge amount of threads could account for a significant fraction of the total runtime. In addition, the fault simulation flow used in [12] explicitly simulates each faulty circuit, which is not the most efficient solution.

As an application of fast fault simulation on the GPUs, fault table computation on a GPU was explored in [13] which derives per pattern detectability information for each fault. Fault table generation is very helpful for silicon debug and fault diagnosis. To support this application, the critical path tracing algorithm was adopted in [13] for further acceleration of the GPU-based fault simulation process. However, similar to [12], the information relevant to the circuit under simulation is not statically stored in the GPU, and a great amount of data transfer between the CPU and the GPU is needed in order to provide data for a huge amount of tiny threads.

3.2 The Proposed Fault Simulator on GPUs

This subsection discusses the proposed fault simulation method on the GPU. In contrast to [12-13], we allocate a region in the device memory of GPU as the texture memory to store the levelized netlist of the circuit under simulation. The size of the allocated texture memory depends on the circuit size. The remaining device memory is allocated as the global memory to store the good/faulty values and the detectability arrays of gates. Because the gate information is stored in the GPU, very few parameters need to be transferred from the CPU to the GPU when a thread is invoked. A thread can locate most of the required data within the GPU memory using its own thread ID. With this implementation, the time taken to transfer data between the CPU and the GPU is drastically reduced. It now includes mainly the time to transfer the levelized circuit netlist and the patterns for simulation from the CPU to the GPU at the beginning of the host program, and the time to transfer the good circuit logic values and the detectability arrays reversely at the end of the host program.

The proposed fault simulation method derives the set of detected faults through both forward and backward simulations based on the logic simulation results. We use separate flows for faults on reconvergent fanout stems, and for faults on other lines. By definition, a stuck-at-*x* fault on a fanout stem is equivalent to the multiple-fault that includes all the stuck-at-*x* faults on its fanout branches. Therefore, the detectability of a stuck-at-*x* fault on a fanout stem cannot be derived simply based on the detectability of the stuck-at-*x* faults on its fanout branches. It was observed three decades ago that only faults on reconvergent fanout stems have to be explicitly simulated [18]. For other faults, implicit, backward fault simulation methods such as critical path tracing [19] which can be massively parallelized, could be used to fully take advantage of GPU's massive parallel architecture. Therefore, in our implementation, we use forward compiled simulation to calculate the detectability of the reconvergent fanout stem faults and use backward critical path tracing based simulation, which is significantly faster, for other faults. As the ratio between the number of reconvergent fanout stems and the total number of nodes in a circuit is usually low, this hybrid flow is much more efficient in terms of both computation time and memory usage, in comparison with the conventional all-forward simulation flow for all faults, which is adopted in the GPU-based fault simulation reported in [12].

The pseudocode of the host program, called *GFSIM* which runs on the CPU and use the GPU as a co-processor is shown below as Algorithm 1, while *P* is the number of patterns to be simulated. A pre-processing step is done on the CPU to levelize the circuit and identify reconvergent

stems (in line 2). The patterns to be simulated can be either randomly generated or read from a pattern file (in line 3). After the levelized netlist and the patterns are transferred to the GPU, logic simulation (in line 8), forward simulation for detectability calculation on fanout reconvergent stems (in line 10-13), and backward simulation for detectability calculation on other faults (in line 14) are processed serially.

Algorithm 1: ***Pseudocode of the Host Program***

```
1. GFSIM(P) {
2. (NumLevel, LevelList, NumStem, StemList) = CircuitPrepocessing().
3. PatternGeneration.
4. Transfer the patterns and levelized netlist from CPU to GPU.
5. v=0
6. while v<P do
7.   v=v + 512*LNumBlock
8.   Invoke LogicSim (l, g_l) on GPU. //Logic simulation on GPU.
9.        ns=0
10.       while ns<NumStem do
11.       ns= ns + FNumBlock
12.       Invoke ForwardDetect(s, c, l_s) on GPU //Detectability of
stems.
13.       end while
14.  Invoke BackwardDetect(l, g_l) on GPU //Detectability of other gates.
15.  Transfer detectability arrays from GPU to CPU.
16. end while
}
```

Three kernels, `LogicSim`, `ForwardDetect`, `BackwardDetect`, are designed for the three tasks respectively. The pseudocodes of these kernels are not listed in this paper due to the space limitation. Instead of using tiny kernels, each of which performs a single gate evaluation, as implemented in [12], these kernels evaluate all gates in one level. Such strategy reduces the data transmission between the CPU and the GPU, but at the same time may achieve less computation parallelism than [12]. The parameters to be transferred from the CPU to the GPU when invoking the threads are level ID (l) and gate count in the level (g_l) for kernels `LogicSim` and `BackwardDetect`, and stem ID (s), fanout-cone ID (c), , and level ID of the stem (l_s) for kernel `ForwardDetect`. The numbers of threads that can be invoked in parallel for these kernels are up to 256* (*LNumBlock*, *FNumBlock*, *BNumBlock*) respectively, subject to the memory requirements of the kernels. Here, *LNumBlock*, *FNumBlock*, *BNumBlock* are the number of blocks in the grids of the three kernels, and a block includes 8 wraps, or 256 threads in the GPU. Since the memory requirement for the backward simulation is empirically only 3 times of that required for logic simulation, *BNumBlock* usually equals *LNumBlock* unless the circuit size or the pattern number is too large to perform the backward simulation in a single pass. Finally, the fault table, which is composed of the detectability arrays including per pattern detectability information for all faults simulated in the current pass, are transferred from the GPU to the CPU (in line 15).

In *GFSIM*, data-parallel simulation on all gates in a single logic level for multiple patterns can be achieved. Each thread provides 32-pattern parallelism itself by using bit-parallelism on 32-bit *int*. In addition, a thread also provides gate-parallel operations for multiple (or a group of) gates in the same level by reading/writing their data from/to the global memory in one memory access. Each half-wrap (i.e., 16 threads) is used to evaluate the same group of gates for 32*16=512 patterns in parallel. *GFSIM* assigns one block to evaluate all gates in one logic level concurrently. Since a block includes 256 threads, or 16 half-wraps, the gates in one level are partitioned into 16 groups, each of which are operated by the threads in a half-wrap. This arrangement allows coalesced accesses in 512-bit quantities to reduce the access latency of the global memory. In GeForce 9800, thread synchronization could only be achieved for threads in the same block. As a block is used to operate all the gates in one level in parallel, synchronization points are inserted in the kernels when the computation in one level is finished, where threads in a block are suspended until they all reach the synchronization point.

The forward fault simulation flow needs multiple simulation passes, because a single pass can only handle a limited number of faults. As pointed out earlier, the forward flow considers only faults on reconvergent fanout stems. Since exactly one of the two stuck-at faults of a gate line is activated by any pattern, we simply complement the good value of a fanout reconvergent stem, which would inject the activated fault between the two stuck-at faults on the stem, and start the forward detectability computation in kernel `ForwardDetect`.

After the detectability of the reconvergent fanout stem faults are derived by `ForwardDetect`, we use a level-by-level backward fault simulation flow to calculate the detectability of all other faults, and the detectability of faults in the same level can be calculated concurrently by kernel `BackwardDetect`. The backward detectability calculation is based on the fault equivalence principle. The general rules of the backward detectability calculation are given below.

1) The detectability of a primary output fault is `TRUE` if the fault is activated by the fault-free value. Otherwise, it is `FALSE`.

2) The detectability of a gate input fault, if the fault is activated by the fault-free value, and all the other inputs of the gate have a non-controlling value, is equal to that of the corresponding gate output fault. Otherwise, it is `FALSE`.

3) The detectability of a non-reconvergent fanout stem is `TRUE` if any one of the detectability of its fanout branches is `TRUE`. Otherwise, it is `FALSE`.

The above rules are equivalent to those of sensitive gate inputs in a critical path tracing method [19].

While calculating a fault's detectability, the GPU fetches the relevant data in the global memory, such as the fault-free value of the gate and the detectability of the faults on its output. The fault detectability and the fault-free value for the same gate are stored in a continuous memory space so that threads in a block can sequentially fetch the required data with the access coalescing.

The netlist information of the circuit is frequently needed for each thread, so the texture memory, which is cached, is used to store the netlist. While the size of a texture cache is 8 KB, cache miss may occur when the circuit under simulation is large. In the implementation, locality of memory access is well achieved by threads in a block. If the circuit is too large to be kept in the texture memory (which could be in the order of hundreds of MB to several GB), partitions can be performed and multiple passes of data transfer from the CPU to the GPU's texture memory will be needed. Even for a very large circuit, as long as the data of a fanout-cone can fit into the texture memory, the proposed algorithm can work, possibly at the cost of an increased time for data transfer from the CPU to the GPU. On the other hand, if the data of a fanout-cone cannot fit into the texture memory, forward fault simulation of this fanout-cone will have to be processed in the CPU. With the trend of rapid increase in GPU's global memory size, this special case would less likely to occur for modern designs with a typical circuit topology.

In summary, each thread reads the required data from the texture cache, or the global memory, and writes the results into the global memory, all based on its own thread ID and simple parameters like level ID and gate count in a level. No other data transfer between the CPU and the GPU is required when invoking threads as long as the texture memory (not the texture cache) can accommodate the entire circuit netlist. All threads in a grid of blocks compute identical instructions on different data, which conforms to the SIMD architecture of the GPU.

4. *n*-Detection Fault Simulation and its Application to Test Ordering

4.1 *n*-Detection Fault Simulation for Any Specified *n*

GFSIM does not experience much difference in runtime if fault dropping is implemented or not for the benchmark circuits we tried. With the computational power provided by a GPU, critical path tracing used in the backward fault simulation can derive the detectability of all non-reconvergent-fanout-stem faults for a large number of patterns in a single pass. Our experiments show that without fault dropping, less than 1% of the total runtime is spent on the backward fault simulation which handles the majority of the faults. Furthermore, the large pattern parallelism (512**xNumBlock*) in *GFSIM* further reduces any potential gain of fault dropping. For circuits in which the ratio of faults on reconvergent fanout stems is relatively large, employing the fault-dropping strategy for faults on reconvergent fanout stems could be helpful to reduce the runtime of the forward fault simulation flow. However, our experiments show that *GFSIM* is highly efficient even without fault dropping, primarily due to the fact that the ratio of faults which could benefit from fault dropping is relatively small.

The nature of no fault dropping in the proposed method enables a single run of *n*-detection fault simulation for multiple *n*'s of interest. Using an additional 32-bit *int* per fault for counting detection times, *GFSIM* is extended to *nGFSIM*, producing outputs as complete as those by the conventional *n*-detection fault simulation flow without fault dropping. Furthermore, since the accurate number of detections of each fault is recorded in *nGFSIM*, the *i*-detection fault coverage for any *i* between 1 and *n* can be easily calculated in post-processing after the fault simulation is completed. Note that *n* can be up to 2^{32}-1 as a 32-bit *int* is used for each fault to store the detection count.

4.2 Test Set Ordering for Max 1-to-n Detections

The per-pattern details of the fault detectability information for all faults produced by *nGFSIM*, can be easily converted into per-fault detectability information for all patterns, which can be used for test pattern ordering. Test set ordering using such information was discussed in [11], which iteratively selects one test at a time that detects the maximum number of faults that are not yet detected in order to obtain a fault coverage curve which is as steep as possible. A test set following such an order can detect defective chips earlier.

Algorithm 2: ***Test Set Ordering for Max 1-to-n Detections***
1. T = the test set, F_{targ} = the target fault set, M= size of F_{targ}.
2. let $F_{targ}(i)$ be the set of faults at least i-detection covered by T, let Tr be the set of selected tests, let $Fr(i)$ be the set of faults i-detection covered by Tr, let $F_{det}(t_k)$ be the set of faults detected by a test t_k.
3. Apply *nGFSIM* to obtain: $F_{det}(t_k)$ for every $t_k \in T$, and $F_{targ}(i)$, i=1~n.
4. Set $Tr = \Phi$, $Fr(i) = \Phi$, u=1; set $w(n)$=1, $w(i)$=M*$w(i+1)$, i=(n-1)~1.
5. For $\forall t_k \in T$, do // Pattern-parallel can be achieved on a GPU.
6. $X_k(i)$= $Fr(i) \cap F_{det}(t_k)$, i=1~n; .$X_k(0)$= $F_{det}(t_k)$-$\sum_{i=1\sim n} X_k(i)$.
7. Priority(t_k) = $\sum_{i=u\sim n} (w(i)*| X_k(i\text{-}1) |)$.
8. end For
9. Find $t_s \in T$, such that Priority(t_s) is maximum.
10. Remove t_s from T, add t_s at the end of Tr.
11. $Fr(n)$=$Fr(n) \cup X_s(n\text{-}1)$; $Fr(i)$=$Fr(i) \cup X_s(i\text{-}1)$-$X_s(i)$, i=2~n-1;
12. $Fr(1)$=$Fr(1) \cup X_s(0)$.
13. while ($|F_{targ}(u)|$ - $\sum_{i=u\sim n} |Fr(i)|$) = 0 do // If u-detection is maximized.
14 u=u+1.
15. end while
16. If $T \neq \Phi$ & $u \neq (n+1)$, go to step 5.

The objective of test set ordering can be generalized to order test patterns for achieving the *steepest 1-to-n detection fault coverage curves.* Using a greedy strategy to meet this objective, tests are ordered to achieve the steepest fault coverage curves for *i*-detection where *i* is incrementally increased from 1 to *n* and *i*-detection has a higher priority than (*i*+1)-detection. We call such a procedure as *test set ordering for Max 1-to-n detections.* Algorithm 2 is proposed for this procedure.

The at-least-*i*-detection fault sets of the test set, $F_{targ}(i)$, includes all the faults that are detected at least *i* times by the test set. Before test set ordering starts, $F_{targ}(i)$, i=1 to n, are already known by running *nGFSIM* (in line 3). For a test set Tr including the current selected test patterns, $Fr(i)$ is used to record the faults that are detected exactly *i* times by Tr. For a test t_k, $F_{det}(t_k)$ is the set of faults detected by t_k. Let $X_k(0)$ be the set of new faults detected by t_k, $X_k(0)$= $F_{det}(t_k)$ - $\sum_{i=1}^{n} Fr(i)$. Let $X_k(i)$ be the intersection of $Fr(i)$ and $F_{det}(t_k)$ while i>0. If t_k is selected, $X_k(i\text{-}1)$ will contribute to

the expansion of *i*-detection fault set *Fr*(*i*); however, the (*i*-1)-detection fault set *Fr*(*i*-1) is shrunk (in line 11-12). Note the shrink of *i*-detection fault set *Fr*(*i*) does not mean the decrease of *i*-detection fault coverage, since all the faults in set *Fr*(*i*+1) are detected more than *i* times and still counted in the *i*-detection fault coverage. In Algorithm 2, we simply use $\sum_{i=1\sim n} X_k(i)$ to represent $\sum_{i=1}^{n} X_k(i)$, and use $\sum_{i=u\sim n}(w(i)*|X_k(i-1)|)$ to represent $\sum_{i=u}^{n}(w(i)*|X_k(i-1)|)$.

When all the at-least-*u*-detection fault sets (sum of *Fr*(*i*), *i*=*u* to *n*) reach their maximum ($F_{targ}(u)$), the procedure exits and the rest of the patterns are dropped and will not be included in the ordered set (in line13-16).

In the procedure of test set ordering for Max 1-to-*n* detections, there are *n* optimization objectives, i.e., maximizing *i*-detection fault coverage for each *i* from *1* to *n*. Since detecting the faults already included in *Fr*(*i*), *Fr*(*i*+1), …, *Fr*(*n*) more times will not further contribute to the increase of *i*-detection fault coverage, the optimization sub-objective of maximizing *i*-detection fault coverage can be represented by: *Finding t_s to maximize*:

$$\sum_{k=0}^{i-1}|X_s(k)|$$

To meet the *n* sub-objectives simultaneously and use decreasing priorities with the increase of *i*, the i^{th} sub-objective can be reduced to: maximize $X_s(i\text{-}1)$, and assigned a weight $w(i) = M*w(i+1)$, while *M* is the total number of target faults, and $w(n)$ is 1 (in line 4). With a weight at least *M* times those of the sub-objectives with lower priorities, it guarantees the absolute dominance of *i*-detection fault coverage improvement over (*i*+1)-detection fault coverage improvement. So the integrated optimization objective of test set ordering for Max 1-to-*n* detections will be: *Finding t_s to maximize*:

$$\sum_{i=1}^{n}(w(i)*|X_s(i-1)|).$$

Therefore, Algorithm 2 will first order test patterns targeting the steepest 1-detection fault coverage curve. After the 1-detection fault coverage is maximized (i.e., the 1st sub-objective is satisfied), it continues to order the rest patterns to target the steepest 2-detection fault coverage curve. The procedure continues until the n^{th} sub-objective is satisfied. When several tests contribute the same to the improvement of 1-to-*i* detection fault coverage, the one with greatest contribution to the improvement of (*i*+1)-detection fault coverage will be selected.

In addition, a variable *u* is used to record the current sub-objective that has the highest priority in all the unsatisfied sub-objectives. Each time a new test is selected, the number of faults in the union set of *Fr*(*u*), *Fr*(*u*+1), …, *Fr*(*n*) is compared with $|F_{targ}(u)|$, to judge if *u* needs to be increased (in line 13-15). Therefore, the integrated optimization objective is dynamically adjusted to: *Finding t_s to maximize*:

$$\sum_{i=u}^{n}(w(i)*|X_s(i-1)|).$$

with the increase of *u*.

It should be noted that the time complexity of Algorithm 2 is $O(M*N_T^2)$, where *M* is the number of faults and N_T is the number of test patterns in set *T*. Fortunately, a pattern-parallel algorithm is developed to accelerate the steps in line 5-8 using the GPU, which in the best case reduces the time complexity to $O(M*N_T)$.

Finally, the steepness of the obtained fault coverage curve depends on the completeness of fault detectability information for all the patterns, i.e., $F_{det}(t_i)$ for every test t_i obtained from the fault simulation procedure. The *n*-pass *n*-detection fault simulation procedure presented in [11] improves the accuracy in comparison with *n*-detection fault simulation with fault dropping, but still produces only approximated results to those without fault dropping. As mentioned before, the proposed *nGFSIM* produces complete *n*-detection results in a single run for a large enough *n* (up to the maximum value represented by the number of bits used for counting detection times per fault, for instance, 32 bits for $n=2^{32}-1$ in our experiments). So using the results produced by *nGFSIM* enables finding the optimal solution for test set ordering with a highly efficient procedure.

4.3 *n*-Detection Test Selection

Using *nGFSIM*, a huge set of patterns can be simulated in seconds. It provides the capability to select test patterns for *n*-detection testing based on fault simulation and test set ordering. If we add an additional constraint on the number of selected test patterns (for limiting the test application time/cost), Algorithm 2 can be slightly modified and then used to produce a high quality test set from any possible pattern sources – either generated randomly or deterministically – whose pattern count exceeds the constraint. In the implementation of *n*-detection test generation, we use the following pseudocode to replace the last line in Algorithm 2:

If $|Tr|<$ MaxNumTest & $u \neq n+1$, go to step 5.

If the original pattern set is large enough and has good test quality in terms of final 1-to-*n* detection fault coverage, the modified Algorithm 2 could get a more compacted test set. Such a compact set could get 1-to-*n* detection fault coverage even higher than those test sets generated by an existing deterministic *n*-detection ATPG tool.

5. Experimental Results

The 1-to-*n* detection fault simulation kernels of *nGFSIM* were implemented on an NVIDIA GeForce GTX9800 graphics card. Similar to [12-13], our current implementation is limited to combinational circuits and employs a 2-state logic for simulation. The implementation can be easily extended to accommodate a 3-state logic, by using two bits instead of one bit to record the state of each signal. Several experiments have been done on IWLS benchmark circuits (originally from ISCAS'89 and ITC'99). In the following subsections, fault

simulation results from the following four sources are compared: 1) 1-to-*n* detection fault simulation results from running *nGFSIM*; 2) single-detection fault simulation results on a GPU reported in [12], denoted as *1-GPU*; 3) *n*-detection fault simulation results using a single-thread commercial ATPG tool, denoted as *n-Comm*; 4) *n*-detection fault simulation results from running the *n*-pass *n*-detection fault simulation process [11], denoted as *n-pass*. Due to the large difference of configurations between the GPU used in [13] and ours (device memory: 4 GB vs. 512 MB, memory bandwidth (BW): ~141.7 GB/s vs. 70 GB/s, number of multiprocessors: 30 vs. 16), we only compare the runtimes with those in [12] in detail.

Table 1 Running Platform Configuration.

	nGFSIM	*1-GPU*[12]	*n-pass*[11]*	*n-Comm*
CPU	Core 2 1.8GHz	3.6GHz	Core 2 1.8GHz	4-processor (Intel) 3.00GHz
Memory	2GB	3GB	2GB	6.5G
GPU	GTX9800, core clk at 675MHz	GTX8800, core clk at 575MHz	n/a	n/a
M-processors	16	16	n/a	n/a
Stream processors	16*8=128 1.688 GHz	16*8=128 1.35 GHz	n/a	n/a
Memory	512MB	768MB	n/a	n/a
Memory BW	70 GB/sec	86 GB/sec	n/a	n/a

**n-pass* was run on the CPU of GTX 9800 without using the GPU.

Table 1 summarizes the platform configuration for these four programs/tools. Due to the unavailability of the GTX8800 GPU (which was an earlier generation of the GTX family), we use a GTX9800 GPU based platform for *nGFSIM* and try our best effort to make a fair comparison with the results of [12] which were based on a GTX8800 GPU. In the experiments, the main differences of *nGFSIM*'s platform configurations from those of *1-GPU* [12] are with a smaller device-memory capacity (512 MB vs. 768 MB), a lower memory bandwidth (70 GB/s vs. 86 GB/s), faster clock frequencies (both no more than 1.25 times) for the cores and stream processor. As the main bottleneck of parallel fault simulation on GPUs is the capacity of the device-memory, which limits the number of threads that can be invoked in parallel, it should be more than fair to make the runtime comparison directly because *nGFSIM* was running with a device-memory of only 2/3 of the size used by *1-GPU* in [12].

5.1 Performance Evaluation of Fault Simulation

In the first experiment, we compare the fault simulation runtimes on 32K random patterns, for 10 larger IWLS benchmark circuits.

The results of *nGFSIM* and *1-GPU* are summarized in Table 2. Because the benchmark circuits were synthesized into gate-level netlists separately (as we did not have access to the netlists reported in [12]), there is a minor difference in the gate counts listed in Columns 2 and 3 respectively. Columns 4 and 5 show the total number of collapsed faults (denoted as "Total") (based on the netlists we synthesized) and the fraction of them on reconvergent fanout stems (denoted as "RS.%") respectively. The runtimes in seconds obtained from *nGFSIM* and from [12] are given in Columns 6 and 7 respectively. These times include the time spent by the CPU to schedule the threads as well as the time taken to transfer data between the CPU and the GPU. Column 8 shows the speedup achieved by *nGFSIM* in comparison with [12]. On average, *nGFSIM* achieves a 5.6X speedup in comparison with [12], or a 4.5X speedup even if factoring in the clock frequency difference.

Table 2 Simulation Results of 32K Random Patterns.

Circuit	No. of Gates		No. of Faults		Runtime(sec.)		Speedup
	1-GPU	*nGFSIM*	Total	RS.%	*1-GPU*	*nGFSIM*	
b22	34060	18580	44694	26.6%	60.485	8.94	6.76x
b21	22470	12507	29990	25.8%	46.583	6.23	7.47x
b17_1	51340	27490	61408	15.5%	19.028	6.87	2.76x
b17	51045	28524	64858	17.1%	17.866	11.00	1.62x
s38417	10821	12939	26176	18.0%	8.234	1.04	7.92x
s38584	8995	21798	30664	12.6%	7.883	1.58	4.99x
s35932	10537	13215	28708	23.1%	5.434	0.51	10.65x
s15850	984	6553	9298	14.5%	0.420	0.62	0.67x
s9234_1	1261	3253	4572	14.8%	2.043	0.39	5.23x
s5378	1682	2382	3534	15.5%	1.961	0.24	8.17x
Average			30390	18.3%			5.62x

The significant reduction in data transmission time and the efficiency achieved by the critical path tracing based fault detectability computation process are the two main reasons for the additional 4.5X speedup of *nGFSIM* in comparison with *1-GPU*. It should be noted that without using tiny threads to explore the maximum computation parallelism, and also because of the smaller size of the global memory, *nGFSIM* has significantly fewer active threads running in parallel than those of *1-GPU*.

Table 3 Scalability Evaluation of *GFSIM*: Runtime (sec.).

No. Gates	Total	Sum of Runtimes of Individual Subcircuits	Growth Rate
25k (1 pass)	1.57	1.55	1.3%
50k (2 passes)	3.49	3.37	3.6%
100k (4 passes)	24.26	22.25	9.0%
150k (8 passes)	41.87	37.42	11.8%

It's desirable to experimentally evaluate the scalability of *nGFSIM*. Due to the lack of publicly available large benchmark circuits, we combined some netlists together into a single netlist to create larger examples for experiments. The GPU runtimes for four examples created this way are shown in Table 3. One virtual circuit we created for this purpose is a single larger circuit consisting of the 10 benchmark circuits in Table 2 which are connected in parallel. The results of this virtual circuit are shown in the last row of Table 3. For comparison, Column

3 shows the sum of the GPU runtimes of the individual sub-circuits that constitute the virtual circuit. With a larger gate count, *nGFSIM* can no longer simulate all 32K random patterns in one pass on our current GPU platform. Instead, 8 passes are needed for the largest one, each of which simulates 4K patterns. The total simulation time is 41.87 seconds, which is about only 1.12 times the sum of those for the 10 sub-circuits. The GPU runtime increases almost linearly with the increase of the circuit size. This result is encouraging, indicating the applicability of *nGFSIM* to modern industrial designs. In all the above experiments, *nGFSIM* achieved a 100% coalesced memory access.

To further evaluate the performance of *n*-detection fault simulation against existing tools, the runtimes consumed by *nGFSIM*, *n-Comm*, and *n-pass* respectively, are compared in Table 4. While *nGFSIM* can calculate the *i*-detection fault coverage for all *i*'s from 1 to any specified integer *n* (<=2^{32}-1), *n-Comm* and *n-pass* calculate the specific *n*-detection fault coverage for a given *n*. The runtime of *n-pass* increases significantly with the increase of *n*, and it takes hours for fault simulation when *n* exceeds 64 for large circuits. Although *n-pass* reduces fault simulation time of each fault under the tests, the overall runtime of *n-pass* is much longer because it adopts a process not friendly to implementation of pattern-parallelism or fault-parallelism, and relies on reordering of tests for each pass. The runtime of *n-Comm* increases slowly with the increase of *n*. Some values of *n* are randomly selected to illustrate such increase of *n-Comm*. The last columns of Table 4 show the speedup achieved by *nGFSIM* in comparison with *n-Comm*. It indicates that *nGFSIM* is on average 20X-25X faster.

It was reported in [12] that *1-GPU* achieves a 35X speedup in comparison with a CPU-based commercial fault simulation engine. It should be noted that their commercial engine for comparison was run on a CPU with a much lower frequency and a much smaller RAM (see Section 3.1) in [12] than those of *n-Comm*.

5.2 Evaluation of *n*-Detection Fault Coverage

In the second experiment, we evaluate the *n*-detection fault coverage of different test sets. The commercial ATPG tool, *n-Comm*, was used to generate five test sets for circuit b17, each of which targets the highest *i*-detection fault coverage, while 1<=*i*<=5. To increase the probability of detecting un-modeled defects, no compaction is used during test generation. In addition, a random test set is also generated for comparison. Then, these 6 test sets are simulated by *nGFSIM* to get their 1-to-511 detection fault coverage. Experimental results are listed in Table 5, in which Column 2 gives the pattern count of each test set, and the other columns show *n*-detection fault coverage for different *n*'s. The six fault coverage curves are compared in Figure 2.

It can be seen that the test set generated to target (*i*+1)-detection always outperforms those test sets generated to target *i*- or less-than-*i*-detection in terms of the fault coverage of any larger-than-*i*-detection. This indicates the opportunity for improving the test quality by considering a large *n* using the *n*-detection test methods. It's very interesting to note that the random test set, with a much larger pattern count (32K), has a much more flat curve. For this particular experiment, in comparison with the deterministic test sets, the random test set, though has a low 1-detection fault coverage, achieves a higher *n*-detection coverage than all five deterministic test sets when *n* is larger than 63. This may imply that pseudo-random patterns used in logic build-in-self-test (BIST) could be better than deterministic tests generated to target a small *n*, for detecting subtle un-modeled defects. It will also be interesting to study the correlation between *n*-detection and coverage of un-modeled defects for some specific *n*'s, which would be part of our future work.

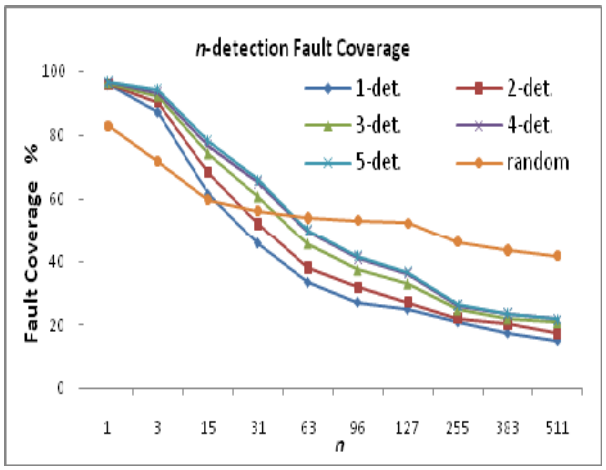


Figure 2 ***n*-Detection Fault Coverage Curves for Circuit b17.**

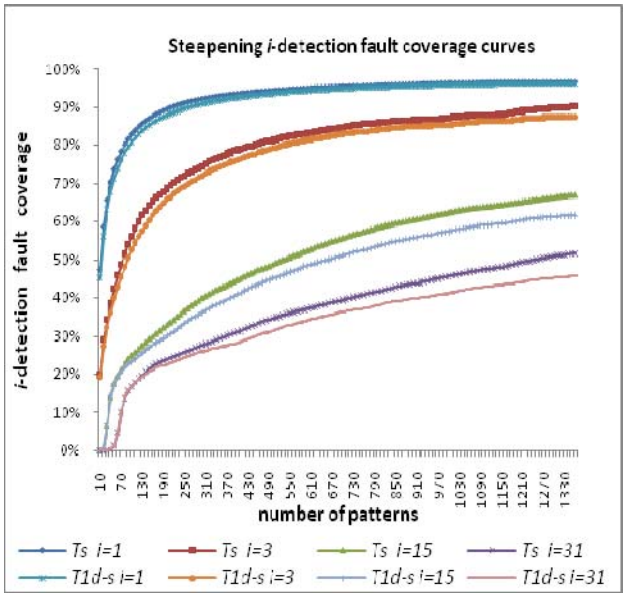
5.3 Test Pattern Ordering and Selection

As mentioned in Section 4.3, *nGFSIM* provides the capability to select test patterns for *n*-detection testing based on fault simulation and test set ordering. We tried 511-detection test set generation using *nGFSIM* and Algorithm 2. For comparison, we also used *n-Comm* to generate two deterministic test sets: a 1-detection test set, T_{1d}, and a 511-detection test set, T_{511d}. Again, no compaction is used during deterministic test generation in order to increase the probability of detecting un-modeled defects. We denote the number of patterns in T_{1d} as N_{1d}. Then, we use *nGFSIM* and Algorithm 2 to select N_{1d} patterns from T_{511d}, to obtain a new test set, T_s, which attempts to optimize the 1-to-511 detection coverage with a pattern count limit of N_{1d}.

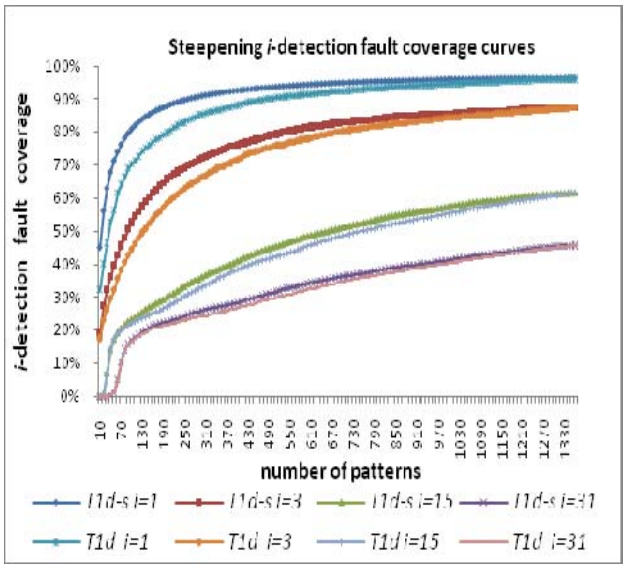
The 1-to-511 detection coverage for both T_{1d} and T_{511d} are computed by *nGFSIM*, and compared with those of T_s in Table 6. Column 3 shows the pattern count. Column 4 shows the runtime of test generation for T_{1d} and T_{511d}, or test ordering (Algorithm 2 on the GPU) for T_s. The other columns show *n*-detection fault coverage for a number of increasing *n*'s. Since *n*-detection fault coverage of T_s outperforms that of T_{1d} for all *n* up to around 383, it demonstrates that by test set ordering, we can generate a

test set with much better test quality under the constraint of a pattern count limit.

Finally, T_{1d} is ordered using Algorithm 2 for maximum 1-to-31 detections. The resulting sorted set is denoted as $T_{1d\text{-}s}$. The fault coverage curves of 1-detection, 3-detection, 15-detection and 31-detection for T_s and T_{1d} of circuit b17, are plotted and compared in Figure 3, where the *x*-axis is the pattern index. The *i*-detection fault coverage curve of T_s is always above that of $T_{1d\text{-}s}$, as shown in Figure 3(a), while the *i*-detection fault coverage curve of $T_{1d\text{-}s}$ is always steeper than that of T_{1d}, as shown in Figure 3(b).



(a) T_s vs. $T_{1d\text{-}s}$



(b) $T_{1d\text{-}s}$ vs. T_{1d}

Figure 3 Steepness of b17's Fault Coverage Curves.

6. Conclusions

In this paper we present *nGFSIM*, a GPU-based fault simulator for 1- to *n*-detection of stuck-at faults. Experimental results show that *nGFSIM* achieves an additional 4.5X speedup in comparison with a recently developed GPU-based fault simulator [12], which computes only fault detection (without per-pattern details) and 1-detection fault coverage. *nGFSIM* also demonstrates an average of 20X-25X speedup for computing *n*-detection fault coverage, in comparison with a commercial fault simulation tool.

We further demonstrate applications enabled by *nGFSIM* to *n*-detection test methods. Our results indicate the benefits of considering multiple *n*'s when using *n*-detection fault coverage for evaluating the quality of tests. Tests selected by our test set ordering algorithm based on *nGFSIM* demonstrate higher fault coverage for 1-to-*n* detections, in comparison with the 1-detection test set deterministically generated by a commercial ATPG tool.

7. References

[1] E.J. McCluskey, C.-W. Tseng, "Stuck-fault tests vs. actual defects", *Proc. Int'l Test Conf.*, 2000, pp.336 -342.

[2] S. Venkataraman, S. Sivaraj, E. Amyeen, S. Lee, A. Ojha, R. Guo, "An Experimental Study of N-Detect Scan ATPG Patterns on a Processor", *Proc. VLSI Test Symposium*, 2004, pp.23-29.

[3] H. Tang, G. Chen, S. M. Reddy, C. Wang, J. Rajski and I. Pomeranz, "Defect Aware Test Patterns", *Proc. DATE*, 2005, pp.450-455.

[4] B. Benware, C. Schuermyer, N. Tamarapalli, K.-H. Tsai, S. Ranganathan, R. Madge, J. Rajski and P. Krishnamurthy, "Impact of multiple-detect test patterns on product quality", *Proc. Int'l Test Conf.*, 2003, pp.1031-1040.

[5] I. Pomeranz, "N-Detection Under Transparent-Scan", *Proc. Design Automation Conference*, 2005, pp.129-134.

[6] I. Pomeranz, S. M. Reddy, "Forming N-Detection Test Sets Without Test Generation", *ACM Trans. Des. Autom. Electron. Syst.*, Vol.12, No.2, Article 18, April 2007.

[7] K. R. Kantipudi, V. D. Agraval, "On the Size and Generation of Minimal N-Detection Tests", *Proc. VLSID*, 2006, pp.425-430.

[8] X. Lin, J. Rajski, I. Pomeranz, and S. M. Reddy, "On static test compaction and test pattern ordering for scan designs", *Proc. Int'l Test Conf.*, 2001, pp.1088-1097.

[9] I. Pomeranz, and S. M. Reddy, "A Measure of Quality for n-Detection Test Sets", *IEEE Trans. on Computers*, Vol. 53, No. 11, Nov. 2004, pp.1497-1503.

[10] M.R. Grimaila, S. Lee, J. Dworak, et al, "REDO - Random Excitation and Deterministic Observation-First Commercial Experiment", *Proc. VLSI Test Symposium*, 1999, pp. 268-274.

[11] I. Pomeranz, S. M. Reddy, "n-Pass n-Detection Fault Simulation and Its Applications", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21, No. 8, pp.980-986, 2002.

[12] K. Gulati , S. P.Khatri, "Towards Acceleration of Fault Simulation using Graphics Processing Units", *Proc. Design Automation Conference*, 2008, pp.822-827.

[13] K. Gulati , S. P.Khatri, "Fault Table Computation on GPUs", *J. Electronic Testing*, Vol.26, No.2, pp.195-209, 2010.

[14] D. Chatterjee, A. DeOrio and Valeria Bertacco, "GCS: High-Performance Gate-Level Simulation with GP-GPUs", *Proc. DATE*, 2009.

[15] Z. Feng, P. Li, "Multigrid on GPU: Tacking Power Grid Analysis on Parallel SIMT Platform", *Proc. ICCAD*, 2008, pp.647-654.

[16] K. Gulati and S. Khatri, "Accelerating Statistical Static Timing Analysis using Graphics Processing Units", *Proc. ASPDAC*, 2009, pp.260 – 265.

[17] NVIDIA. CUDA Complete Unified Device Architecture, 2007.

[18] S. J. Hong, "Fault simulation strategy for combinational logic networks", *Proc. Int'l Symp. on Fault Tolerant Comp.*, 1978, pp.96-99.

[19] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing-An alternative to fault simulation", *Proc. Design Automation Conf.*, 1983, pp.214-220.

[20] H. K. Lee, D. S. Ha, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol.15, No.9, pp.1048-1058, 1996.

Table 4 Runtime Comparison for 1-to-*n* Detection Fault Simulation on 32K Random Patterns.

Circuit	*nGFSIM*	*n-Comm* (sec.)					*n-pass* (sec.)		*nGFSIM* Speedup on *n-Comm*				
	Arbitrary $n<=2^{32}-1$	$n=2^4-1$	$n=2^6-1$	$n=2^8-1$	$n=2^{10}-1$	$n=2^{12}-1$	$n=2^4-1$	$n=2^6-1$	$n=2^4-1$	$n=2^6-1$	$n=2^8-1$	$n=2^{10}-1$	$n=2^{12}-1$
b22	8.94	91.76	94.24	95.36	96.92	99.84	430	3545	10.26x	10.54x	10.67x	10.84x	11.17x
b21	6.23	49.26	49.96	51.54	60.38	63.10	386	1970	7.91x	8.02x	8.27x	9.69x	10.13x
b17_1	6.87	93.81	94.16	94.22	98.02	110.50	723	5222	13.66x	13.71x	13.71x	14.27x	16.08x
b17	11.00	96.10	96.00	98.22	99.74	118.30	841	5377	8.74x	8.73x	8.93x	9.07x	10.75x
s38417	1.04	21.16	22.39	24.76	25.52	32.56	345	2137	20.35x	21.53x	23.81x	24.54x	31.31x
s38584	1.58	25.71	25.84	27.06	27.86	34.65	396	2720	16.27x	16.35x	17.13x	17.63x	21.93x
s35932	0.51	43.56	45.32	42.14	44.89	46.85	356	2914	85.41x	88.86x	82.63x	88.02x	91.86x
s15850	0.62	8.55	8.29	8.71	9.92	12.68	79	441	13.79x	13.37x	14.05x	16.00x	20.45x
s9234_1	0.39	5.50	5.54	5.83	6.27	7.85	49	242	14.10x	14.21x	14.95x	16.08x	20.13x
s5378	0.24	2.69	2.88	2.99	3.22	4.04	23	113	11.21x	12.00x	12.46x	13.42x	16.83x
Average									20.17x	20.73x	20.66x	21.96x	25.06x

Table 5 1-to-511 Detection Fault Coverage of 1-to-5 Detection Test Sets for Circuit b17.

Test set	Pattern Count	*n*-Detection Fault Coverage (%)									
		n=1	*n=3*	*n=15*	*n=31*	*n=63*	*n=96*	*n=127*	*n=255*	*n=383*	*n=511*
1-det.	1354	96.16	87.44	61.62	45.78	33.31	27.03	24.78	20.91	17.32	15.01
2-det.	2048	96.23	90.37	68.30	52.09	38.10	31.78	27.18	22.18	20.44	17.35
3-det.	2976	96.57	92.44	74.41	60.87	45.73	37.37	33.09	24.80	22.15	20.86
4-det.	3328	96.76	93.49	76.95	65.12	49.72	41.00	36.14	26.09	23.43	21.73
5-det.	3360	96.78	94.53	78.37	65.93	50.08	41.65	36.61	26.33	23.55	21.82
random	32 K	83.16	71.93	59.72	56.26	54.16	53.33	52.52	46.21	43.56	41.65

Table 6 1-to-31 Detection Fault Coverage Comparison between Deterministic ATPG and Test Selection.

Circuit	Test Set	Runtime (sec.)	Pattern Count	*n*-Detection Fault Coverage (%)										
				n=1	*n=3*	*n=7*	*n=15*	*n=31*	*n=63*	*n=96*	*n=127*	*n=255*	*n=383*	*n=511*
b22	T_{1d}	2.51	1018	97.78	90.00	77.99	61.47	45.20	33.69	29.68	26.15	16.25	10.90	3.48
	T_{511d}	104.04	32768	97.82	96.79	95.75	94.26	92.68	91.23	89.72	87.37	79.22	73.40	66.87
	T_s	90.50	1018	97.82	93.69	84.50	71.64	53.90	37.32	31.47	27.21	16.42	10.60	3.18
b21	T_{1d}	1.91	1050	98.72	90.90	78.13	65.16	49.02	36.69	30.79	26.57	15.71	10.75	5.18
	T_{511d}	63.25	32768	98.98	98.37	97.29	95.60	93.46	91.28	89.69	88.36	80.43	74.10	68.67
	T_s	63.00	1050	98.98	94.88	84.92	71.97	55.44	39.20	31.56	27.29	16.35	11.69	4.69
b17_1	T_{1d}	2.88	1566	97.39	87.50	76.67	63.57	47.39	34.72	29.20	26.30	22.65	19.74	16.85
	T_{511d}	108.16	32768	97.46	96.58	94.77	92.16	88.60	82.34	78.05	74.95	64.70	58.46	52.20
	T_s	153.70	1566	97.46	90.82	78.94	67.03	51.93	36.13	30.47	27.00	23.18	19.90	16.85
b17	T_{1d}	4.8	1354	96.16	87.44	76.31	61.62	45.78	33.31	27.03	24.78	20.91	17.32	15.01
	T_{511d}	165.08	32768	96.71	95.77	94.36	92.49	89.03	82.73	78.61	75.75	66.66	60.03	54.47
	T_s	148.00	1354	96.71	90.20	78.80	67.03	51.72	36.89	29.50	26.17	21.33	17.27	14.85