J Electron Test (2010) 26:195–209
DOI 10.1007/s10836-010-5147-x

Fault Table Computation on GPUs

Kanupriya Gulati · Sunil P. Khatri

Received: 30 September 2009 / Accepted: 26 January 2010 / Published online: 12 February 2010

Abstract In this paper, we explore the implementation of fault table generation on a Graphics Processing Unit (GPU). A fault table is essential for fault diagnosis and fault detection in VLSI testing and debug. Generating a fault table requires extensive fault simulation, with no fault dropping, and is extremely expensive from a computational standpoint. Fault simulation is inherently parallelizable, and the large number of threads that a GPU can operate on in parallel can be employed to accelerate fault simulation, and thereby accelerate fault table generation. Our approach, called GFTABLE, employs a pattern parallel approach which utilizes both bit-parallelism and thread-level parallelism. Our implementation is a significantly modified version of FSIM, which is pattern parallel fault simulation approach for single core processors. Like FSIM, GFTABLE utilizes critical path tracing and the dominator concept to reduce runtime. Further modifications to FSIM allow us to maximally harness the GPU's huge memory bandwidth and high computational power. Our approach does not store the circuit (or any part of the circuit) on the GPU. Efficient parallel reduction operations are implemented in our implementation of GFTABLE. We compare our performance to FSIM∗, which is FSIM modified to generate a fault table on a single core processor. Our experiments indicate that GFTABLE, implemented on a single NVIDIA Quadro FX 5800 GPU card, can generate a fault table for 0.5 million test patterns on average 15.68× faster when compared with FSIM∗. With the NVIDIA Tesla server, our approach would be potentially 89.57× faster.

Keywords Fault simulation · Fault table · Hardware acceleration

Responsible Editor: P. Mishra

K. Gulati (✉) · S. P. Khatri
Department of ECE, Texas A&M University,
College Station, TX 77843, USA
e-mail: kgulati@tamu.edu

S. P. Khatri
e-mail: sunilkhatri@tamu.edu

1 Introduction

With increasing complexity and size of digital VLSI designs, the number of faulty variations of these designs are growing exponentially, thus increasing the time and effort required for *VLSI testing* and *debug*. Among the key steps in VLSI testing and debug are fault detection and diagnosis. *Fault detection* aims at differentiating a faulty design from a fault free design, by applying test vectors. *Fault diagnosis* aims at identifying and isolating the fault, in order to analyze the defect causing the faulty behavior, with the help of test vectors which detect the fault. Both detection and diagnosis [3, 30, 33] require the following precomputed information:

- Whether vector v_i can detect fault f_j,
- The list of faults that are detected by applying vector v_i or
- The list of vectors that detect fault f_j.

This information is gathered by pre-computing a *fault table*. In general, a fault table is a matrix $[a_{ij}]$ where

columns represent faults, rows represent test vectors, and $a_{ij} = 1$ if the test vector v_i detects the fault f_j, else $a_{ij} = 0$.

A fault table (also called a *pass/fail fault dictionary* [29]) is generated by extensive *fault simulation*. Given a digital design and a set of input vectors V defined over its primary inputs, fault simulation evaluates (for all i) the set of stuck-at faults F^i_{sim} that are tested by applying the vectors $v_i \in V$. The faults tested by each vector are then recorded in the matrix format of the fault table described earlier. Since the detectability of every fault is evaluated for every vector, the compute time for generating a fault table is extremely large. If a fault is *dropped* from the fault list as soon as a vector successfully detects it, the compute time can be reduced. However, in this case, the fault resolution achieved might be very poor, and insufficient for fault diagnosis. Thus, fault dropping cannot be performed during the generation of the fault table. For fault detection, we would like to find a minimal set of vectors which can maximally detect the faults. In order to compute this minimal set of vectors, the generation of a fault table with limited or no fault dropping is required. For these reasons, fault table generation without fault dropping is usually performed. As a result, the high runtime of fault table generation becomes a key concern, and it is extremely important to explore ways to accelerate fault table generation. The ideal approach should be fast, scalable *and* cost effective.

In order to reduce the compute time for generating the fault table, parallel implementations of fault simulation have been routinely used [8]. Fault simulation can be parallelized by a variety of techniques. These techniques include parallelizing the fault simulation algorithm (*algorithm-parallel techniques* [1, 4, 5]), partitioning the circuit into disjoint components and simulating them in parallel (*model-parallel* techniques [19, 34]), partitioning the fault set data and simulating faults in parallel (*data-parallel* techniques [9, 23, 28]) and a combination of one or more of these [18]. Data parallel techniques can be further classified into *fault-parallel* methods, wherein different faults are simulated in parallel, and *pattern-parallel* approaches, wherein different input patterns (for the same fault) are simulated in parallel. Pattern-parallel approaches, as described in [16, 26], exploit the inherent bit-parallelism of logic operations on computer words. In this paper, we present a fault table generation approach that utilizes a *pattern parallel* approach implemented on Graphics Processing Units (GPUs). Our notion of pattern parallelism includes *bit-parallelism* obtained by performing logical operations on words and *thread level parallelism* obtained by running several GPU threads concurrently.

Graphic Processing Units (GPUs) are designed to operate in a Single Instruction Multiple Data (SIMD) fashion. The key application of a GPU is to serve as a graphics accelerator for speeding up image processing, 3D rendering operations, etc. These graphics acceleration tasks process the same instructions independently on large volumes of data. This is why GPUs employ a SIMD computation semantic. Lately, the application of GPUs for general purpose computations has been actively explored [17, 22]. GPU architectures have been continuously evolving towards higher performance, larger memory sizes, larger memory bandwidths and relatively lower costs. The theoretical performance of GPU has grown from 50 Gflops for the NV40 GPU in 2004 to more than 900 Gflops for Quadro FX 5800 GPU in 2008. This high computing power mainly arises from a fully pipelined and highly parallel architecture, with extremely high memory bandwidths. Maximum GPU memory bandwidths have grown from 42 GB/s for the ATI Radeon X1800XT to 141.7 GB/s for the Quadro FX 5800 GPU. In contrast the theoretical performance of a 3 GHz Pentium4 CPU is 12 Gflops, with a maximum memory bandwidth of 8–10 GB/s to main memory.

Our approach for fault table generation is based on the fault simulation algorithm called FSIM [16]. FSIM was developed to run on a single core CPU. However, since the target hardware in our case is a SIMD GPU machine, and the objective is to accelerate fault table generation, the FSIM algorithm is augmented and its implementation significantly modified to maximally harness the computational power and memory bandwidth available in the GPU. Fault simulation of a logic netlist effectively requires multiple logic simulations of the *true value (or fault free)* simulations, and simulations with faults injected at various gates (typically primary inputs and reconvergent fanout branches as per the checkpoint fault injection model [10]). This is a natural match for the GPU's capabilities, since it exploits the extreme memory bandwidths of the GPU, as well as the presence of several SIMD processing elements on the GPU. Further, the computer words on the latest GPUs today allow 32 or even 64 bit operations. This facilitates the use of bit-parallelism to further speed up fault simulation. For scalability reasons, our approach *does not store the circuit (or any part of the circuit) on the GPU*.

This paper is the first, to the best of the authors' knowledge, to accelerate fault table generation on a GPU platform. The key contributions of this paper are:

– We exploit the match between pattern parallel (bit parallel and also thread parallel) fault simulation

with the capabilities of a GPU (a SIMD-based device) and harness the computational power of GPUs to accelerate fault table generation.
- The implementation satisfies the key requirements which ensure maximal speedup in a GPU. These are:
  - The different threads, which perform gate evaluations and fault injections are implemented such that the data dependencies between threads is minimized.
  - All threads compute identical instructions, but on different data, which conforms to the SIMD architecture of the GPU.
  - Fast parallel reduction on the GPU is employed for computing the logical OR of thousands of words containing fault simulation data.
  - The amount of data transfer between the GPU and the host (CPU) is minimized. To achieve this, the large on-board memory on the recent GPUs is maximally exploited.
- In comparison to FSIM∗ (i.e. FSIM [16] modified to generate the fault dictionary), our implementation is on average 15.68× faster, for 0.5 million patterns, over the ISCAS and ITC99 benchmarks.
- Further, even though our current implementation has been benchmarked on a single NVIDIA Quadro FX 5800 graphics card, the NVIDIA Tesla GPU Computing Processor [21] allows up to eight NVIDIA Tesla GPUs (on a 1U server). We estimate that our implementation, using the NVIDIA Tesla server, can generate a fault dictionary on average 89.57× faster, when compared to FSIM∗.

Our fault dictionary computation algorithm is implemented in the Compute Unified Device Architecture (CUDA), which is an open-source programming and interfacing tool provided by NVIDIA corporation, for programming NVIDIA's GPU devices. The correctness of our GPU based fault table generator, GFTABLE, has been verified by comparing its results against the results of FSIM∗ (which is run on the CPU).

The remainder of this paper is organized as follows. Previous work in fault simulation and fault table generation has been described in Section 2. Section 3 details the architecture of the GPU device and the programming tool CUDA. Section 4 details our approach for implementing fault simulation and table generation on GPUs. In Section 5 we present results of experiments which were conducted in order to benchmark our approach. We conclude in Section 6.

2 Previous Work

Since efficient fault simulation is a requirement for generating a fault dictionary, we discuss some of the previous work in accelerating fault simulation. Several research efforts have attempted to accelerate fault simulation in a scalable and cost-effective fashion, by exploiting the parallelism inherent in the problem. These efforts can be classified as *algorithm-parallel*, *model-parallel* and *data-parallel*.

Algorithm-parallel efforts distribute the fault simulation workload and/or pipeline the tasks [1, 4, 5, 18], aiming to reduce the frequency of communication and synchronization between processors. In contrast to these approaches, our approach is pattern-parallel. The approach in [18] aims at heuristically assigning fault set partitions to several medium-grain multiprocessors. In [5], the authors present a methodology to predict and characterize workload distributions, which aid in parallelizing fault simulation. The authors of [1] suggest a pipelined design for fault simulation. MARS [4], a hardware accelerator, is based on this design. However, [18] states that MARS is limited in its application for accelerating fault simulation.

In a model-parallel approach [18, 19, 34], the circuit under test is partitioned into several (possibly non-disjoint) components. Each component is assigned to one or more processors. In order to keep the partitioning balanced, dynamic re-partitioning [26, 27] is performed. This increases the complexity of the algorithm thereby impacting simulation time [26, 27].

Several data-parallel approaches for fault simulation have been studied in the literature. These approaches use dedicated hardware accelerators, supercomputers, vector machines or multiprocessors [6, 9, 14, 15, 23, 24, 28, 32]. These hardware accelerated fault simulators require specialized hardware, significant design effort and time, non-trivial algorithm modifications and potentially huge software design efforts. Our approach, in contrast, accelerates fault table generation by using commercial off-the-shelf graphic processing units (GPUs). The ubiquity and ease of programming of GPU devices, along with their extremely low costs compared to hardware accelerators, supercomputers, etc. makes GPUs an attractive alternative for fault table generation.

The underlying algorithm for our GPU based fault table generation engine is based on an approach for accelerating fault simulation called FSIM [16]. FSIM is a data-parallel approach that is implemented on a single core microprocessor. The essential idea of FSIM is to simulate the circuit in a levelized manner from inputs to outputs, and to prune off unnecessary gates

in the early stages. This is done by employing critical path tracing [2, 13] and the dominator concept [7, 12], both of which reduce the amount of explicit fault simulation required. Some details of FSIM are explained in Section 4. We use a modification to FSIM (which we call FSIM∗) to generate the fault table, and compare the performance of our GPU-based fault-table generator, GFTABLE, to those of FSIM∗. Since the target hardware in our case is architecturally different from a single core microprocessor, the original algorithm is redesigned and augmented to maximally exploit the computational power of the GPU.

The application of GPUs for general purpose computations has been actively explored [17, 22]. The approach described in [11] accelerates fault simulation by employing a table look-up based approach on the GPU. Their approach, in contrast to ours, does not target a fault table computation, but only accelerates fault simulation. The data transferred between the GPU and host in [11] only records fault detection (and not the vector that caused detection) and is therefore not sufficient to construct a complete fault table. Also, unlike our approach, [11] does not take advantage of the bit-parallelism available on the GPU. Further, [11] does not employ critical path tracing or the dominator concept, both of which help achieve a substantial reduction in the simulation runtime [16].

An approach which generates compressed fault tables or dictionaries is described in [29]. This approach concentrates on reducing the size taken by the fault table by using compaction [3, 35] or aliasing [31] techniques during fault table generation. Our approach, on the other hand reduces the compute time for fault table generation by exploiting the immense parallelism available in the GPU, and is hence orthogonal to [29].

3 Architecture

In this section we discuss the key architectural aspects of the NVIDIA Quadro FX 5800 GPU device, which is the GPU used in our implementation. We discuss the hardware and programming model for this device.

3.1 Hardware Model

The Quadro FX 5800 architecture has 30 multiprocessors per chip and eight processors (ALUs) per multiprocessor. During any clock cycle, all the processors of a multiprocessor execute the same instruction, but may operate on different data. There is no mechanism to communicate *between* the different multiprocessors. In other words, no native synchronization primitives exist to enable communication between multiprocessors. We next discuss the relevant memories of the device.

A pool of shared memory within each multiprocessor is accessible to all its processors. Each block of shared memory represents 16 banks of single-ported SRAM. Each bank has 1KB of storage and a bandwidth of 32 bits per clock cycle. Furthermore, since there are 30 multiprocessors on a Quadro FX 5800, this results in a total storage of 480 KB. However, if two or more access requests are made to the same bank, a *bank conflict* results. In this case, the conflict is resolved by granting accesses in a serial fashion. Thus, shared memory must be accessed in a fashion such that bank conflicts are minimized.

Global memory is read/write memory that is *not* cached. A single floating point value read from (or written to) global memory can take 400 to 600 clock cycles. Much of this global memory latency can be hidden if there are sufficient arithmetic instructions that can be issued while waiting for the global memory access to complete. Since the global memory is not cached, access patterns can dramatically change the amount of time spent in waiting for global memory accesses. Thus, coalesced accesses of 32-bit, 64-bit, or 128-bit quantities should be performed in order to increase the throughput and to maximize the bus bandwidth utilization. The total on-board memory in Quadro FX 5800 is 4 GB.

3.2 Programming Model

CUDA (Compute Unified Device Architecture), which is used for interfacing with the GPU device, is a new hardware *and* software architecture for issuing and managing computations on the GPU as a data-parallel computing device, without the need of mapping them to a traditional graphics API [20]. CUDA was released by NVIDIA corporation in early 2007.

CUDA's programming model is summarized in Fig. 1. When programmed through CUDA, the GPU is viewed as a compute device capable of executing a large number of *threads* in parallel. Threads are the atomic units of parallel computation, and the code they execute is called a *kernel*. The GPU device operates as a coprocessor to the main CPU, or host. Data-parallel, compute-intensive portions of applications running on the host can be off-loaded onto the GPU device. Such a portion is compiled into the instruction set of the GPU device and the resulting program, called a kernel, is downloaded to the device.

A thread block (equivalently referred to as a block) is a batch of threads that can co-operate together by efficiently sharing data through some fast shared

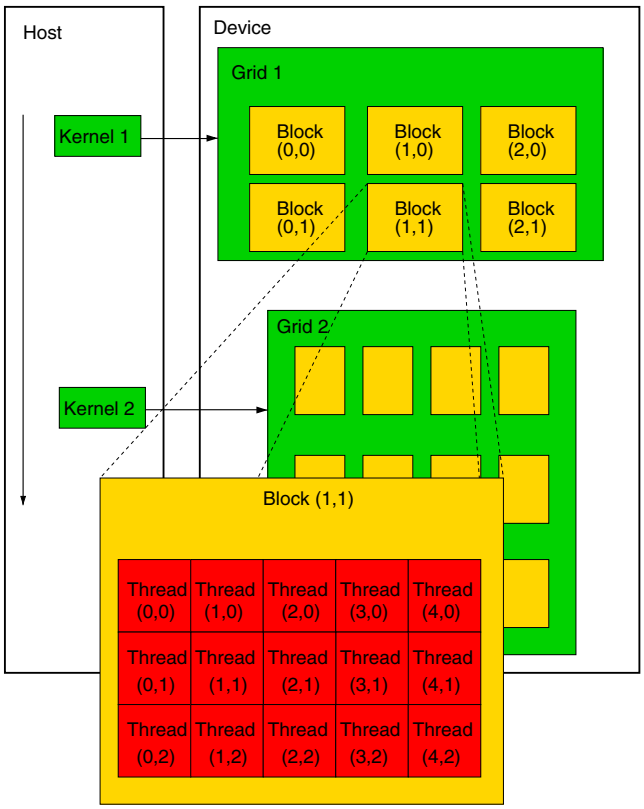

Fig. 1 Programming model of CUDA

memory and synchronize their execution to coordinate memory accesses. Users can specify *synchronization points* in the kernel, where threads in a block are suspended until they all reach the synchronization point. Threads are grouped in warps, which are further grouped in blocks. Threads have identity numbers *threadIDs* which can be viewed as a one, two or three dimensional value. All the warps composing a block are guaranteed to run on the same multiprocessor, and can thus take advantage of shared memory and local synchronization. Each warp contains the same number of threads, called the warp size, and is executed in a SIMD fashion; a *thread scheduler* periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources. In case of the Quadro FX 5800, the warp size is 32. Thread blocks have restrictions on the *maximum* number of threads in them. In a Quadro FX 5800 the maximum number of threads grouped in a thread block is 512.

A thread block can be executed by a single multiprocessor. However, blocks of same dimensionality (i.e. orientation of the threads in them) and size (i.e number of threads in them) that execute the same kernel can be batched together into a *grid* of blocks. The number of blocks in a grid is referred to as *dimgrid*. A grid of thread blocks is executed on the device by executing one or more blocks on each multiprocessor using time slicing. However, at a given time, at the most 1,024 threads can be active in a single multiprocessor in the GPU Quadro 5800 device.

4 Approach

In order to maximally harness the high computational power of the GPU, our fault table generation approach is designed in a manner that is aware of the GPU's architectural and functional features and constraints. For instance, the programming model of a GPU is the Single Instruction Multiple Data (SIMD) model, under which all threads must compute identical instructions, but on different data. GPUs allow extreme speedups if the different threads being evaluated have minimal data dependencies or global synchronization requirements. Our implementation honors these constraints and maximally avoids data or control dependencies between different threads. Further, even though the GPU's maximum bandwidth to/from the on-board memory has dramatically increased in recent GPUs (∼141.7 GB/s in the NVIDIA Quadro FX 5800), the GPU to host communication in our implementation is done using the PCIe 1.0a standard, with a data rate of ∼500 MB/s for 16 lanes. Therefore, our approach is implemented such that the communication between the host and the GPU is minimized.

In this section, we provide the details of our GFTABLE approach. As mentioned earlier, we modified FSIM [16] (which only performs fault simulation) to generate a complete fault table, and refer to this version as FSIM∗. The underlying algorithm for GFTABLE is a significantly re-engineered variant of FSIM∗. We next discuss FSIM∗, along with our key modifications to capitalize on the parallelism available in a GPU.

4.1 Definitions

We first define some of the key terms with the help of the example circuit shown in Fig. 2. A *stem* (or *fanout stem*) is defined as a *line* (or net) which fans out to more than one gate. All primary outputs of the circuit are defined as stems. For example in Fig. 2, the stems are k and p. If the fanout branches of each stem are cut off, this induces a partition of the circuit into *fanout free regions* (FFRs). For example, in Fig. 2, we get two FFRs as shown by the dotted triangles. The output of any FFR is a stem (say s), and the FFR is referred to as FFR(s). If all paths from a stem s pass through a

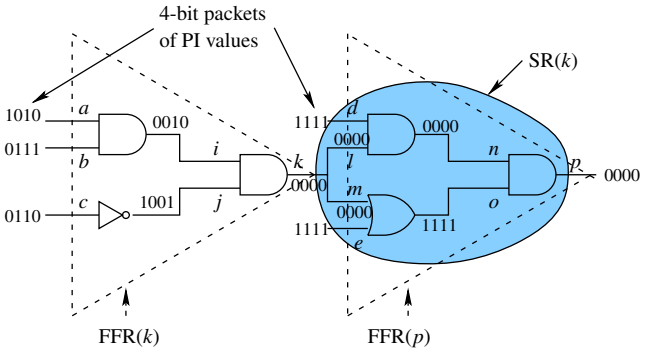

Fig. 2 Example circuit

line l before reaching a primary output, then the line l is called a *dominator* of the stem s. If there are no other dominators between the stem s and dominator l, then line l is called the *immediate dominator* of s. In the example, p is an immediate dominator of stem k. The region between a stem s and its immediate dominator is called the *stem region* (SR) of s, and is referred to as SR(s). Also, we define a *vector* as a two-dimensional array with a length equal to the number of primary inputs, and a width equal to P, the *packet size*. In Fig. 2, the vectors are on the primary inputs a, b, c, d and e. The packet size is $P = 4$. In other words, each vector consists of P fault patterns. In practice, the packet size for bit-parallel fault simulators is typically equal to the word size of the computer on which the simulator is implemented. In our experiments, the packet size (P) is 32.

If the change of the logic value at line s is observable at line t, then *detectability* D(s, t) = 1, else D(s, t) = 0. If a fault f injected at some line is detectable at line t, then *fault detectability* FD(f, t) = 1, else FD(f, t) = 0. If t is a primary output, the (fault) detectability is called a *global (fault) detectability*. The *cumulative detectability* of a line s, CD(s), is the logical OR of the fault detectabilities of the lines which merge at s. The i^{th} element of CD(s) is defined as 1 iff there exists a fault f (to be simulated) such that FD(f, s) =1 under the application of the i^{th} test pattern of the vector. Otherwise, it is defined as 0. The following five properties hold for cumulative detectabilities:

- If a fault f (either s-a-1 or s-a-0) is injected at a line s and no other fault propagates to s, then CD(s) = FD(f, s).
- If both s-a-0 and s-a-1 faults are injected at a line s, CD(s) = (11..1).
- If no fault is injected at a line s and no other faults propagate to s, then CD(s) = (00..0).
- Suppose there is a path from s to t. Then CD(t) = CD(s) · D(s, t), where · is the bitwise AND operation.
- Suppose two paths $r \rightarrow t$ and $s \rightarrow t$ merge. Then CD(t) = (CD(r)D(r, t) + CD(s)D(s, t)), where + is the bitwise OR operation.

Further details on detectability and cumulative detectability can be found in [16].

The *sensitive inputs* of a unate gate with two or more inputs are determined as follows:

- If only one input k has a Dominant Logic Value (DLV), then k is sensitive. AND and NAND gates have a DLV of 0. OR and NOR gates have a DLV of 1.
- If all the inputs of a gate have a value $\overline{DLV}$, then all inputs are sensitive,
- Otherwise no input is sensitive.

Critical path tracing (CPT), which was introduced in [1], is an alternative to conventional forward fault simulation. The approach consists of determining paths of *critical lines*, called critical paths, by a backtracing process starting at the POs for a vector v_i. Note that a *critical line* is a line driving the sensitive input of a gate. Note that the POs are critical in any test. By finding the critical lines for v_i, one can immediately infer the faults detected by v_i. CPT is performed after fault free simulation of the circuit for a vector v_i has been conducted. To aid the backtracing, sensitive gate inputs during fault free simulation are marked.

For FFRs, CPT is always exact. In both approaches described in the next section, FSIM∗ and GPU-TABLE, CPT is used only for the FFRs. An example illustrating CPT is provided in the sequel.

4.2 Algorithms: FSIM∗ and GFTABLE

The algorithm for FSIM∗ is displayed in Algorithm 1. The key modifications for GFTABLE are explained in text in the sequel. Both FSIM∗ and GFTABLE maintain three major lists, a fault list (FL), a stem list (STEM_LIST) and an active stem list (ACTIVE_STEM), all on the CPU. The stem list stores all the stems {s} whose corresponding FFRs ({$FFR(s)$}) are candidates for fault simulation. The active stem list stores stems {s^*} for which at least one fault propagates to the immediate dominator of the stem s^* . The stems stored in the two lists are in the ascending order of their topological levels.

It is important to note that the GPU can never launch a kernel. Kernel launches are exclusively performed by the CPU (host). As a result, if (as in the case

Algorithm 1 Pseudocode of FSIM∗

```
1:  FSIM∗(N){
2:  Set up Fault list FL.
3:  Find FFRs and SRs.
4:  STEM_LIST ← all stems
5:  Fault table [a_ik] initialized to all zero matrix.
6:  v=0
7:  while v < N do
8:     v=v + packet width
9:     Generate one test vector using LFSR
10:    Perform fault free simulation
11:    ACTIVE_STEM ← NULL.
12:    for each stem s in STEM_LIST do
13:       Simulate FFR using CPT
14:       Compute CD(s)
15:       if (CD(s) ≠ (00...0)) then
16:          Simulate SRs and compute D(s, t), where t is the
             immediate dominator of s.
17:          if (D(s, t) ≠ (00...0)) then
18:             ACTIVE_STEM ← ACTIVE_STEM + s.
19:          end if
20:       end if
21:    end for
22:    while (ACTIVE_STEM ≠ NULL) do
23:       Remove the highest level stem s from
          ACTIVE_STEM.
24:       Compute D(s, t), where t is an auxiliary output which
          connects all primary outputs.
25:       if (D(s, t) ≠ (00...0)) then
26:          for (each each fault f_i in FFR(s)) do
27:             FD( f_i, t) = FD( f_i, s) · D(s, t).
28:             Store FD( f_i, t) in the i^th row of [a_ik]
29:          end for
30:       end if
31:    end while
32: end while
33: }
```

of GFTABLE), a conditional evaluation needs to be performed (lines 15, 17, and 25 for example), the condition *must* be checked by the CPU, which can then launch the appropriate GPU kernel if the condition is met. Therefore, the value being tested in the condition must be transferred by the GPU back to the CPU. The GPU operates on T threads at once (each computing a 32-bit result). Hence, in order to reduce the volume of data transferred and to reduce it to the size of a computer word on the CPU, the results from the GPU threads are reduced down to one 32-bit value before being transferred back to the CPU.

The argument to both the algorithms is the number of test patterns (N) over which the fault table is to be computed for the circuit. As a preprocessing step, both FSIM∗ and GFTABLE compute the fault list FL, award every gate a *gate_id*, compute the level of each gate and identify the stems. The algorithms then identify the FFR and SR of each stem (this is computed on the CPU). As discussed earlier, the stems and the corresponding FFRs and SRs of these stems in our example circuit are marked in Fig. 2. Let us consider the following five faults in our example circuit: a s-a-0, c s-a-1, c s-a-0, l s-a-0 and l s-a-1, which are added to the fault list FL. Also assume that the fault table generation is carried out for a single vector of length 5 (since there are five primary inputs) consisting of 4-bit wide packets. In other words, each vector consists of four patterns of primary input values. The fault table $[a_{ij}]$ is initialized to the all zero matrix. In our example, the size of this matrix is $N \times 5$. The above steps are shown in lines 1 through 5 of Algorithm 1. The rest of FSIM∗ and GFTABLE are within a while loop (line 7) with condition $v < N$, where N is the total number of patterns to be simulated and v is the current count of patterns which are already simulated. For both algorithms, v is initialized to zero (line 6).

4.2.1 Generating Vectors (Line 9)

The test vectors in FSIM∗ are generated using an LFSR-based pseudo-random number generator on the CPU. For every test vector, as will be seen later, fault free and faulty simulations are carried out. Each test vector in FSIM∗ is a vector (array) of 32-bit integers with a length equal to the number of primary inputs (NPI). In this case, v is incremented by 32 (packet-width) in every iteration of the while loop (line 8).

Each test vector in GFTABLE is a vector of length NPI and width $32 \times T$, where T is the number of threads launched in parallel in a grid of thread blocks. Therefore, in this case, for every while loop iteration, v is incremented by $T \times 32$. The test vectors are generated on the CPU (as in FSIM∗) and transferred to the GPU memory. In all the results reported in this paper, both FSIM∗ and GFTABLE utilize identical test vectors (generated by the LFSR-based pseudo-random number generator on the CPU). In all examples, the results of GFTABLE matched those of FSIM*. The GFTABLE runtimes reported always include the time required to transfer the input patterns to the GPU and the time required to transfer results back to the CPU.

4.2.2 Fault Free Simulation (Line 10)

Now, for each test vector, FSIM∗ performs fault free or true value simulation. Fault free simulation is essentially the logic simulation of every gate, carried out in a

forward levelized order. The fault free output at every gate, computed as a result of the gate's evaluation, is recorded in the CPU's memory.

Fault free simulation in GFTABLE is carried out in a forward levelized manner as well. Depending on the gate type and the number of inputs, a separate kernel on the GPU is launched for T threads. As an example, the pseudocode of the kernel which evaluates the fault free simulation value of a two input AND gate is provided in Algorithm 2. The arguments to the kernel are the pointer to global memory, *MEM*, where fault free values are stored, and the gate_id of the gate being evaluated (*id*) and its two inputs (a and b). Let the thread's (unique) *threadID* be t_x. The data in *MEM*, indexed at a location ($t_x + a \times T$) is ANDed with the data at location ($t_x + b \times T$) and the result is stored in *MEM* indexed at location ($t_x + id \times T$). Our implementation has a similar kernel for every gate in our library.

Since the amount of global memory on the GPU is limited, we store the fault free simulation data in the global memory of the GPU for at most L gates of a circuit. Note that we require two copies of the fault free simulation data, one for use as a reference and the other for temporary modification to compute faulty circuit data. For the remaining gates, the fault free data is transferred to and from the CPU, as and when it is computed or required on the GPU. This allows our GFTABLE approach to scale *regardless of the size of the given circuit*.

Figure 2 shows the fault free output at every gate, when a single test vector of packet width 4 is applied at its five inputs.

Algorithm 2 Pseudocode of the kernel for logic simulation of two-input AND gate

```
logic_simulation_kernel_AND_2(int*MEM, int id,
int a, int b){
t_x = my_thread_id
MEM[t_x+id*T]=MEM[t_x+a*T]·MEM[t_x+b*T]
}
```

4.2.3 Computing Detectabilities and Cumulative Detectabilities (Lines 13, 14)

Next, in the FSIM* and GFTABLE algorithms, for every stem s, CD(s) is computed. This is done by computing the detectability of every fault and line in FFR(s) by using Critical Path Tracing and the properties of cumulative detectabilities discussed in Section 4.1.

This step is further explained by the help of Fig. 3. The FFR(k) from the example circuit is copied four

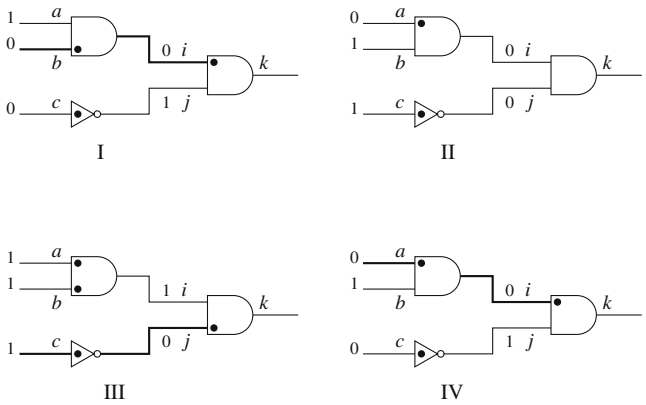

Fig. 3 CPT on FFR(k)

times,[1] one for each pattern in the vector applied. In each of the copies, the sensitive input is marked using a bold dot. The critical lines are darkened. Using these markings, the detectabilities of all lines at stem k can be computed as follows: D(a, k) = 0001, since out of the four copies, only in the fourth copy a lies on the sensitive path (i.e., a path consisting of critical lines) backtraced from k. Similarly we compute the following:

D(b, k) = 1000; D(c, k) = 0010; D(i, k) = 1001; D(j, k) = 0010; D(k, k) = 1111; D(a, i) = 0111; D(b, i) = 1010 and D(c, j) = 1111.

Now for the faults in FFR(k) (i.e., a s-a-0, c s-a-0 and c s-a-1), we compute the FDs as follows:

FD(a s-a-0, k) = FD(a s-a-0, a) · D(a, k).

For every test pattern, the fault a s-a-0 can be observed at a only when the fault free value at a is different from the stuck at value of the fault. Among the four copies in Fig. 3, only the first and third copy have a fault free value of '1' at line a, and thus fault a s-a-0 can be observed only in the first and third copies. Therefore FD(a s-a-0, a) = 1010. Therefore, FD(a s-a-0, k) = 1010 · 0001 = 0000. Similarly, FD(c s-a-0, k) = 0010 and FD(c s-a-1, k) = 0000.

Now, by definition

CD(k) = (CD(i) · D(i, k) + CD(j) · D(j, k)) and CD(i) = (CD(a) · D(a, i) + CD(b) · D(b, i)).

From the first property discussed for CD, CD(a) = FD(a s-a-0, a) = 1010, and by definition CD(b) = 0000. By substitution and similarly computing CD(i) and CD(j), we compute CD(k) = 0010.

[1] This is because the packet width is 4.

The implementation of the computation of detectabilities and cumulative detectabilities in FSIM∗ and GFTABLE is different, since in GFTABLE, all computations for computing detectabilities and cumulative detectabilities are done on the GPU, with every kernel executed on the GPU launched with T threads. Thus a single kernel in GFTABLE computes T times more data, compared to the corresponding computation in FSIM∗. In FSIM∗, the backtracing is performed in a topological manner from the output of the FFR to its inputs, and is not scheduled for gates driving zero critical lines in the packet. We found that this pruning reduces the number of gate evaluations by 42% in FSIM∗ (based on tests run on four benchmark circuits). In GFTABLE, however, T times more patterns are evaluated at once, and as a result, no reduction in the number of scheduled gate evaluations were observed for the same four benchmarks. Hence, in GFTABLE, we perform a brute-force backtracing on *all* gates in an FFR.

As an example, the pseudocode of the kernel which evaluates the cumulative detectability at output k of a two-input gate with inputs i and j is provided in Algorithm 3. The arguments to the kernel are the pointer to global memory, CD, where cumulative detectabilities are stored, pointer to global memory, D, where detectabilites on the immediate dominator, and the gate_id of the gate being evaluated (k) and its two inputs (i and j). Let the thread's (unique) *threadID* be t_x. The data in CD and D, indexed at a location $(t_x + i \times T)$ and $(t_x + j \times T)$, and the result computed as per

$$\mathrm{CD}(k) = (\mathrm{CD}(i) \cdot \mathrm{D}(i, k) + \mathrm{CD}(j) \cdot \mathrm{D}(j, k))$$

is stored in CD indexed at location $(t_x + k \times T)$. Our implementation has a similar kernel for two,three and four input gates in our library.

Algorithm 3 Pseudocode of the kernel to compute CD of the output k of two-input gate with inputs i and j

```
CPT_kernel_2(int * CD, int * D, inti, intj, intk){
t_x = my_thread_id
CD[t_x + k * T]=CD[t_x + i * T] · D[t_x + i * T] + CD[t_x +
j * T] · D[t_x + j * T]
}
```

4.2.4 Fault Simulation of SR(s) (Lines 15, 16)

In the next step, the FSIM∗ algorithm checks that $\mathrm{CD}(s) \neq (00...0)$ (line 15), before it schedules the simulation of SR(s) until its immediate dominator t, and the computation of D(s, t). In other words, if $\mathrm{CD}(s) = (00...0)$, it implies that for the current vector, the frontier of all faults upstream from s has died before reaching the stem s, and thus no fault can be detected at s. In that case, the fault simulation of SR(s) would be pointless.

In the case of GFTABLE, the effective packet size is $32 \times T$. T is usually set to more than 1000 (in our experiments it is ≥10K), in order to take advantage of the immense parallelism available on the GPU and to lower the overhead of launching a kernel and accessing global memory. The probability of finding $\mathrm{CD}(s) = (00...0)$ in GFTABLE is therefore very low (∼0.001). Further, this check would require the logical OR of T 32-bit integers on the GPU, which is an expensive computation. As a result, we bypass the test of line 15 in GFTABLE, and always schedule the computation of SR(s) (line 16).

In simulating SR(s), explicit fault simulation is performed in the forward levelized order from stem s to its immediate dominator t. The input at stem s during simulation of SR(s) is CD(s) XORed with fault free value at s. This is equivalent to injecting the faults which are upstream from s and observable at s. After the fault simulation of SR(s), the detectability D(s, t) is computed by XORing the simulation output at t with the true value simulation at t. During the forward levelized simulation, the immediate fanout of a gate g is scheduled only if the result of the logic evaluation at g is different from its fault free value. This check is conducted for every gate in all paths from stem s to its immediate dominator t. On the GPU, this step would involve XORing the current gate's T 32-bit outputs with the previously stored fault free T 32-bit outputs. It would then require the computation of a logical reduction OR of the T 32-bit results of the XOR into one 32-bit result. This is because line 17 is computed on the CPU, which requires a 32-bit operand. In GFTABLE, the reduction OR operation is a modified version of the highly optimized tree-based parallel reduction algorithm on the GPU, described in [25]. The approach in [25] effectively avoids bank conflicts and divergent warps, minimizes global memory access latencies and employs loop unrolling to gain further speedup. Our modified reduction algorithm has a key difference compared to [25]. The approach in [25] computes a SUM instead of a logical OR. The approach described in [25] is a breadth first approach. In our case employing a breadth first approach is expensive, since we need to detect if *any* of the $T \times 32$ bits is not equal to 0. Therefore, as soon as we find a single non-zero entry we can finish our computation. Note that performing this test sequentially would be extremely slow in the worst case. We therefore equally divide

the array of T 32-bit words into smaller groups of size Q words, and compute the logical OR of all numbers within a group using our modified parallel reduction approach. As a result, our approach is a hybrid of a breadth-first and a depth-first approach. If the reduction result for any group is not (00...0), we return from the parallel reduction kernel and schedule the fanout of the current gate. If the reduction result for any group, on the other hand, is equal to (00...0), we compute the logical reduction OR of the next group and so on. Each logical reduction OR is computed using our reduction kernel, which takes advantage of all the optimizations suggested in [25] (and improves [25] further by virtue of our modifications). The optimal size of the reduction groups was experimentally determined to be $Q = 256$. We found that when reducing 256 words at once, there was a high probability of having at least one non zero bit, and thus there was a high likelihood of returning early from the parallel reduction kernel. At the same time, using 256 words allowed for a fast reduction within a single thread block of size equal to 128 threads. Scheduling a thread block of 128 threads uses four warps (of warp size equal to 32 threads each). The thread block can schedule the four warps in a time-sliced fashion, where each integer OR operation takes four clock cycles, and therefore making optimal use of the hardware resources.

Despite using the above optimization in parallel reduction, the check can still be expensive, since our parallel reduction kernel is launched after every gate evaluation. To further reduce the runtime, we launch our parallel reduction kernel after every G gates. During in-between runs, the fanout gates are always scheduled to be evaluated. Due to this, we would potentially do a few extra simulations, but this approach proved to be significantly faster when compared to either performing a parallel reduction after every gate's simulation or scheduling every gate in SR(s) for simulation in a brute-force manner. We experimentally determined the optimal value for G to be 20.

In the next step (lines 17 and 18), the detectability D(s, t) is tested. If it is not equal to (00...0), stem s is added to the ACTIVE_STEM list. Again this step of the algorithm is identical for FSIM∗ and GFTABLE, however the difference is in the implementation. On the GPU, a parallel reduction technique, as explained in the last paragraph, is used for testing if D(s, t) ≠ (00...0). The resulting 32-bit value is transferred back to the CPU. The *if* condition (line 17) is checked on the CPU and if it is true, the ACTIVE_STEM list is augmented on the CPU.

For our example circuit, SR(k) is displayed in Fig. 4. The input at stem k is 0010 (CD(k) XORed with fault

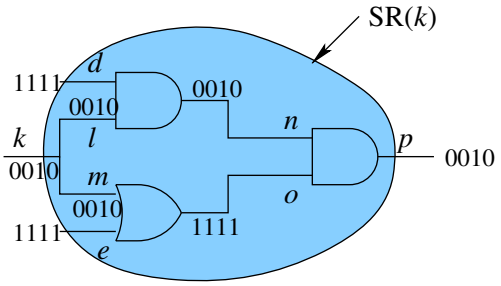

Fig. 4 Fault simulation on SR(k)

free value at k). The two primary inputs d and e have the original test vectors. From the output evaluated after explicit simulation until p, D(k,p) = 0010 ≠ 0000. Thus, k is added to the active stem list.

Due to space constraints, the details of performing CPT on FFR(p) and computing the detectabilities and cumulative detectabilities are skipped. The resulting values are listed below:

D(l, p) = 1111; D(n, p) = 1111; D(d, p) = 0000; D(m, p) = 0000; D(e, p) = 0000; D(o,p) = 0000; D(d, n) = 0000; D(l, n) = 1111; D(m, o) = 0000; D(e, o) = 1111; FD(l s-a-0, p) = 0000; FD(l s-a-1, p) = 1111; CD(d) = 0000; CD(l) = 1111; CD(m) = 0000; CD(e) = 0000; CD(n) = 1111; CD(o) = 0000 and CD(p) = 1111.

Since CD(p) ≠ (0000) and D(p, p) ≠ (0000), the stem p is added to ACTIVE_STEM list.

4.2.5 Generating the Fault Table (Line 22–31)

Next, FSIM∗ computes the global detectability of faults (and stems) in the backward order, i.e., it removes the highest level stem s from the ACTIVE_STEM list (line 23) and computes its global detectability (line 24). If it is not equal to (00..0) (line 25), the global detectability of every fault in FFR(s) is computed and stored in the [a_{ij}] matrix (lines 26–28).

The corresponding implementation in GFTABLE maintains the ACTIVE_STEM on the CPU, and like FSIM∗, first computes the global detectability of the highest level stem s from ACTIVE_STEM list, but on the GPU. Also, another parallel reduction kernel is invoked for D(s, t), since the resulting data needs to be transferred to the CPU for testing if the global detectability of s is not equal to (00..0) (line 25). If true, the global detectability of every fault in FFR(s) is computed on the GPU and transferred back to the CPU to store the final fault table matrix on the CPU.

The complete algorithm of our GFTABLE approach is displayed in Algorithm 4.

Algorithm 4 Pseudocode of GFTABLE

```
GFTABLE(N){
Set up Fault list FL.
Find FFRs and SRs.
STEM_LIST ← all stems
Fault table [a_ik] initialized to all zero matrix.
v=0
while v < N do
  v=v + T × 32
  Generate using LFSR on CPU and transfer test vector to GPU
  Perform fault free simulation on GPU
  ACTIVE_STEM ← NULL.
  for each stem s in STEM_LIST do
    Simulate FFR using CPT on GPU // bruteforce backtracking on all gates
    Simulate SRs on GPU
    // check at every Gth gate during
    // forward levelized simulation if fault frontier still alive,
    // else continue with for loop with s ← next stem in STEM_LIST
    Compute D(s, t) on GPU, where t is the immediate dominator of s. // computed using hybrid parallel reduction on GPU
    if (D(s, t) ≠ (00...0)) then
      update on CPU ACTIVE_STEM ← ACTIVE_STEM + s
    end if
  end for
  while (ACTIVE_STEM ≠ NULL) do
    Remove the highest level stem s from ACTIVE_STEM.
    Compute D(s, t) on GPU, where t is an auxiliary output which connects all primary outputs. // computed using hybrid parallel reduction on GPU
    if (D(s, t) ≠ (00...0)) then
      for (each each fault f_i in FFR(s)) do
        FD(f_i, t) = FD(f_i, s) · D(s, t). // computed on GPU
        Store FD(f_i, t) in the i^th row of [a_ik] // stored on CPU
      end for
    end if
  end while
end while
}
```

5 Experimental Results

As discussed previously, pattern parallelism in GFTABLE includes both *bit parallelism*, obtained by performing logical operations on words (i.e. packet size is 32) and *thread-level parallelism*, obtained by launching T GPU threads concurrently. With respect to bit parallelism, the bit width used in GFTABLE implemented on the NVIDIA Quadro FX 5800 was 32. This was chosen to make a fair comparison with FSIM∗, which was run on a 32-bit, 3.6 GHz Intel CPU running Linux (Fedora Core 3), with 3 GB RAM. It should be noted that Quadro FX 5800 also allows operations on 64-bit words.

With respect to thread-level parallelism, launching a kernel with a higher number of threads in the grid allows us to better take advantage of the immense parallelism available on the GPU, and reduces the overhead of launching a kernel and hides the latency of accessing global memory. However, due to a finite size of the global memory there is an upper limit on the number of threads that can be launched simultaneously. Hence we split the fault list of a circuit into smaller fault lists. This is done by first topologically listing the circuit from the primary inputs to the primary outputs. We then collect the faults associated with every Z (=100) gates from this list, to generate the smaller fault lists. Our approach is then implemented in an iterative fashion, where a new fault list is targeted in a new iteration. We statically allocate global memory for storing the fault detectabilities of the current faults (faults currently under consideration) for all threads launched in parallel on the GPU. Let the number of faults in the current list being considered be F, and the number of threads launched simultaneously be T, then $F \times T \times 4$B of global memory is used for storing the current faults detectabilities. As mentioned previously, we statically allocate space for two copies of fault free simulation output for at most L gates. The gates of the circuit are topologically sorted from the primary outputs to the primary inputs. The fault free data (and its copy) of the first L gates in the sorted list are statically stored on the GPU. This further uses $L \times T \times 2 \times 4$B of global memory. For the remaining gates, the fault free data is transferred to and from the CPU, as and when it is computed or required on the GPU.

Further, the detectabilities and cumulative detectabilities of all gates in the FFRs of the current faults, and for all the dominators in the circuit, are stored on the GPU. The total on-board memory on a single NVIDIA Quadro FX 5800 is 4 GB. With our current implementation, we can launch $T = 16$K threads in parallel, while using $L = 32$K gates. Note that the complete fault dictionary is never stored on the GPU, and hence the number of test patterns used for generating the fault table can be arbitrarily large. Also, since GFTABLE does not store the information of the entire circuit on the GPU, it can handle arbitrary sized circuits.

Table 1 Fault table generation results with $L = 32$K

Circuit	# Gates	# Faults	FSIM∗	GFTABLE	Speedup	GFTABLE-8	Speedup
c432	196	524	0.77	12.60	16.43×	0.13	93.87×
c499	243	758	0.75	8.40	11.20×	0.13	64.00×
c880	443	942	1.15	17.40	15.13×	0.20	86.46×
c1355	587	1,574	2.53	23.95	9.46×	0.44	54.03×
c1908	913	1,879	4.68	51.38	10.97×	0.82	62.70×
c2670	1,426	2,747	1.92	56.27	29.35×	0.34	167.72×
c3540	1,719	3,428	7.55	168.07	22.26×	1.32	127.20×
c5315	2,485	5,350	4.50	109.05	24.23×	0.79	138.48×
c6288	2,448	7,744	28.28	669.02	23.65×	4.95	135.17×
c7552	3,719	7,550	10.70	204.33	19.10×	1.87	109.12×
b14_1	7,283	12,608	70.27	831.27	11.83×	12.30	67.60×
b14	9,382	16,207	100.87	1,502.47	14.90×	17.65	85.12×
b15	12,587	21,453	136.78	1,659.10	12.13×	23.94	69.31×
b20_1	17,157	31,034	193.72	3,307.08	17.07×	33.90	97.55×
b20	20,630	35,937	319.82	4,992.73	15.61×	55.97	89.21×
b21_1	16,623	29,119	176.75	3,138.08	17.75×	30.93	101.45×
b21	20,842	35,968	262.75	4,857.90	18.49×	45.98	105.65×
b17	40,122	69,111	903.22	4,921.60	5.45×	158.06	31.14×
b18	40,122	69,111	899.32	4,914.93	5.47×	157.38	31.23×
b22_1	25,011	44,778	369.34	4,756.53	12.88×	64.63	73.59×
b22	29,116	51,220	399.34	6,319.47	15.82×	69.88	90.43×
Average					15.68×		89.57×

The results of our current implementation, for ten ISCAS benchmarks and 11 ITC99 benchmarks, for 0.5M patterns, are reported in Table 1. All runtimes reported are in seconds. The fault tables obtained from GFTABLE, for all benchmarks, were verified against those obtained from FSIM∗, and were found to verify with 100% fidelity. Column 1 lists the circuit under consideration, Columns 2 and 3 lists the number of gates and (collapsed) faults in the circuit. The total runtimes for FSIM∗ and GFTABLE are listed in Columns 4 and 5 respectively. The runtime of GFTABLE includes the *total* time taken on both the GPU and the CPU, and the time taken for *all* the data transfers between the GPU and the CPU. In particular, the

Table 2 Fault table generation results with $L = 8$K

Circuit	# Gates	# Faults	FSIM∗	GFTABLE	Speedup	GFTABLE-8	Speedup
c432	196	524	0.73	12.60	17.19×	0.13	98.23×
c499	243	758	0.75	8.40	11.20×	0.13	64.00×
c880	443	942	1.13	17.40	15.36×	0.20	87.76×
c1355	587	1,574	2.52	23.95	9.52×	0.44	54.37×
c1908	913	1,879	4.73	51.38	10.86×	0.83	62.04×
c2670	1,426	2,747	1.93	56.27	29.11×	0.34	166.34×
c3540	1,719	3,428	7.57	168.07	22.21×	1.32	126.92×
c5315	2,485	5,350	4.53	109.05	24.06×	0.79	137.47×
c6288	2,448	7,744	28.17	669.02	23.75×	4.93	135.72×
c7552	3,719	7,550	10.60	204.33	19.28×	1.85	110.15×
b14_1	7,283	12,608	70.05	831.27	11.87×	12.26	67.81×
b14	9,382	16,207	120.53	1,502.47	12.47×	21.09	71.23×
b15	12,587	21,453	216.12	1,659.10	7.68×	37.82	43.87×
b20_1	17,157	31,034	410.68	3,307.08	8.05×	71.87	46.02×
b20	20,630	35,937	948.06	4,992.73	5.27×	165.91	30.09×
b21_1	16,623	29,119	774.45	3,138.08	4.05×	135.53	23.15×
b21	20,842	35,968	974.03	4,857.90	5.05×	170.46	28.50×
b17	40,122	69,111	1,764.01	4,921.60	2.79×	308.70	15.94×
b18	40,122	69,111	2,100.40	4,914.93	2.34×	367.57	13.37×
b22_1	25,011	44,778	647.15	4,756.53	7.35×	113.25	42.00×
b22	29,116	51,220	915.87	6,319.47	6.90×	160.28	39.43×
Average					12.88×		69.73×

Table 3 Fault table generation results with $L = 16K$

Circuit	# Gates	# Faults	FSIM∗	GFTABLE	Speedup	GFTABLE-8	Speedup
c432	196	524	0.73	12.60	17.33×	0.13	99.04×
c499	243	758	0.75	8.40	11.20×	0.13	64.00×
c880	443	942	1.03	17.40	16.89×	0.18	96.53×
c1355	587	1,574	2.53	23.95	9.46×	0.44	54.03×
c1908	913	1,879	4.68	51.38	10.97×	0.82	62.70×
c2670	1,426	2,747	1.97	56.27	28.61×	0.34	163.46×
c3540	1,719	3,428	7.92	168.07	21.22×	1.39	121.26×
c5315	2,485	5,350	4.50	109.05	24.23×	0.79	138.48×
c6288	2,448	7,744	28.28	669.02	23.65×	4.95	135.17×
c7552	3,719	7,550	10.70	204.33	19.10×	1.87	109.12×
b14_1	7,283	12,608	70.27	831.27	11.83×	12.30	67.60×
b14	9,382	16,207	100.87	1,502.47	14.90×	17.65	85.12×
b15	12,587	21,453	136.78	1,659.10	12.13×	23.94	69.31×
b20_1	17,157	31,034	193.72	3,307.08	17.07×	33.90	97.55×
b20	20,630	35,937	459.82	4,992.73	10.86×	80.47	62.05×
b21_1	16,623	29,119	156.75	3,138.08	20.02×	27.43	114.40×
b21	20,842	35,968	462.75	4,857.90	10.50×	80.98	59.99×
b17	40,122	69,111	1,203.22	4,921.60	4.09×	210.56	23.37×
b18	40,122	69,111	1,399.32	4,914.93	3.51×	244.88	20.07×
b22_1	25,011	44,778	561.34	4,756.53	8.47×	98.23	48.42×
b22	29116	51,220	767.34	6,319.47	8.24×	134.28	47.06×
Average					14.49×		82.80×

transfer time includes the time taken to transfer the following.

- The test patterns which are generated on the CPU (CPU → GPU).
- The results from the multiple invocations of the parallel reduction kernel (GPU → CPU).
- The global fault detectabilities over all test patterns for all faults (GPU → CPU) and
- The fault free data of any gate which is not in the set of L gates (during true value and faulty simulations) (CPU ↔ GPU).

Column 6 reports the speedup of GFTABLE over FSIM∗. The average speedup over the 21 benchmarks is reported in the last row. On average, GFTABLE is 15.68× faster than FSIM∗.

By using the NVIDIA Tesla server housing up to eight GPUs [21], the available global memory increases by 8×. Hence we can potentially launch 8× more threads simultaneously, and set L to be large enough to hold the fault free data (and its copy) for all the gates in our benchmark circuits. This allows for a ∼8× speedup in the processing time. The first three items of the transfer times in the list above will not scale, and the last item will not contribute to the total runtime. In Table 1, Column 7 lists the projected runtimes when using a 8 GPU system for GFTABLE (referred to as GFTABLE-8). The projected speedup of GFTABLE-8 compared to FSIM∗ is listed in Column 8. The average potential speedup is 89.57×.

Tables 2 and 3 report the results with $L = 8K$ and 16K, respectively. All columns in Tables 2 and 3 report similar entries as described for Table 1. The speedup of GFTABLE and GFTABLE-8 over FSIM∗ with $L = 8K$ is 12.88× and 69.73×, respectively. Similarly, the speedup of GFTABLE and GFTABLE-8 over FSIM∗ with $L = 16K$ is 14.49× and 82.80× respectively.

6 Conclusion

In this paper, we have presented our implementation of fault table generation on a GPU, called GFTABLE. Fault table generation requires fault simulation without fault dropping, which can be extremely computationally expensive. Fault simulation is inherently parallelizable, and the large number of threads that can be computed in parallel on a GPU can therefore be employed to accelerate fault simulation and fault table generation. In particular, we implemented a pattern parallel approach which utilizes both bit-parallelism and thread-level parallelism. Our implementation is a significantly re-engineered version of FSIM, which is a pattern parallel fault simulation approach for single core processors. At no time in the execution is the entire circuit (or a part of the circuit) required to be stored (or transferred) on (to) the GPU. Like FSIM, GFTABLE utilizes critical path tracing and the dominator concept to reduce explicit simulation time. Further modifications to FSIM allow us to maximally

harness the GPU's impressive computational resources and huge memory bandwidth. We compared our performance to FSIM∗, which is FSIM modified to generate a fault table. Our experiments indicate that GFTABLE, implemented on a single NVIDIA Quadro FX 5800 GPU card, can generate a fault table for 0.5 million test patterns, on average 15.68× faster when compared FSIM∗. With the NVIDIA Tesla server [21], our approach would be potentially 89.57× faster.

References

1. Abramovici A, Levendel Y, Menon P (1983) A logic simulation engine. In: IEEE transactions on computer-aided design, vol 2, pp 82–94
2. Abramovici M, Menon PR, Miller DT (1983) Critical path tracing—an alternative to fault simulation. In: DAC '83: proceedings of the 20th conference on Design automation. IEEE, Piscataway, pp 214–220
3. Abramovici M, Breuer MA, Friedman AD (1990) Digital systems testing and testable design. Computer Science, New York
4. Agrawal P, Dally WJ, Fischer WC, Jagadish HV, Krishnakumar AS, Tutundjian R (1987) MARS: a multiprocessor-based programmable accelerator. IEEE Des Test 4(5):28–36
5. Amin MB, Vinnakota B (1997) Workload distribution in fault simulation. J Electron Test 10(3):277–282
6. Amin MB, Vinnakota B (1999) Data parallel fault simulation. IEEE Trans Very Large Scale Integr (VLSI) Syst 7(2): 183–190
7. Antreich K, Schulz M (1987) Accelerated fault simulation and fault grading in combinational circuits. IEEE Trans Comput-Aided Des Integr Circuits Syst 6(5):704–712
8. Banerjee P (1994) Parallel algorithms for VLSI computer-aided design. Prentice Hall, Englewood Cliffs. ISBN-13: 978-0130158352 (ISBN-10: 0130158356)
9. Beece DK, Deibert G, Papp G, Villante F (1988) The IBM engineering verification engine. In: DAC '88: proceedings of the 25th ACM/IEEE conference on design automation. IEEE Computer Society, Los Alamitos, pp 218–224
10. Bossen DC, Hong SJ (1971) Cause-effect analysis for multiple fault detection in combinational networks. IEEE Trans Comput 20(11):1252–1257
11. Gulati K, Khatri SP (2008) Towards acceleration of fault simulation using graphics processing units. In: Proceedings of the 45th annual conference on design automation, pp 822–827
12. Harel D, Sheng R, Udell J (1987) Efficient single fault propagation in combinational circuits. In: Proceedings of the international conference on computer-aided design ICCAD, pp 2–5
13. Hong SJ (1979) Fault simulation strategy for combinational logic networks. In: Proceedings of eighth international symposium on fault-tolerant computing, pp 96–99
14. Ishiura N, Ito M, Yajima S (1987) High-speed fault simulation using a vector processor. In: Proceedings of the international conference on computer-aided design ICCAD
15. Kitamura Y (1989) Exact critical path tracing fault simulation on massively parallel processor AAP2. In: Proceedings of the 1989 IEEE/ACM international conference on computer-aided design, pp 474–477
16. Lee HK, Ha DS (1991) An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagation. In: Proceedings of the IEEE international test conference on test. IEEE Computer Society, Washington, pp 946–955
17. Luebke D, Harris M, Govindaraju N, Lefohn A, Houston M, Owens J, Segal M, Papakipos M, Buck I (2006) GPGPU: general-purpose computation on graphics hardware. In: SC '06: proceedings of the 2006 ACM/IEEE conference on supercomputing. ACM, New York, p 208
18. Mueller-Thuns R, Saab D, Damiano R, Abraham J (1993) VLSI logic and fault simulation on general-purpose parallel computers. In: IEEE transactions on computer-aided design of integrated circuits and systems, vol 12, pp 446–460
19. Narayanan V, Pitchumani V (1992) Fault simulation on massively parallel SIMD machines: algorithms, implementations and results. J Electron Test 3(1):79–92
20. NVIDIA CUDA Homepage. http://developer.nvidia.com/object/cuda.html
21. NVIDIA Tesla GPU Computing Processor. http://www.nvidia.com/object/IO_43499.html
22. Owens J (2007) GPU architecture overview. In: SIGGRAPH '07: ACM SIGGRAPH 2007 courses. ACM, New York, p 2
23. Ozguner F, Daoud R (1988) Vectorized fault simulation on the cray X-MP supercomputer. In: IEEE International conference on computer-aided design, 1988. ICCAD-88. Digest of Technical Papers, pp 198–201
24. Ozguner F, Aykanat C, Khalid O, (1988) Logic fault simulation on a vector hypercube multiprocessor. In: Proceedings of the third conference on hypercube concurrent computers and applications. ACM, New York pp 1108–1116
25. Parallel Reduction. http://developer.download.nvidia.com/~reduction.pdf
26. Parkes S, Banerjee P, Patel J (1995) A parallel algorithm for fault simulation based on PROOFS. In ICCD '95: proceedings of the 1995 international conference on computer design. IEEE Computer Society, Washington, p 616
27. Patil S, Banerjee P (1991) Performance trade-offs in a parallel test generation/fault simulation environment. IEEE Trans Comput-Aided Des 10:1542–1558
28. Pfister GF (1982) The Yorktown simulation engine: introduction. In: DAC '82: proceedings of the 19th conference on design automation. IEEE, Piscataway, pp 51–54
29. Pomeranz I, Reddy SM (1992) On the generation of small dictionaries for fault location. In: ICCAD '92: 1992 IEEE/ACM international conference proceedings on computer-aided design. IEEE Computer Society, Los Alamitos, pp 272–279
30. Pomeranz I, Reddy SM (2008) A same/different fault dictionary: an extended pass/fail fault dictionary with improved diagnostic resolution. In: DATE '08: proceedings of the conference on design, automation and test in Europe. ACM, New York, pp 1474–1479
31. Pomeranz I, Reddy S, Tangirala R (1992) On achieving zero aliasing for modeled faults. In: Proc. [3rd] European conference on design automation, pp 291–299
32. Raghavan R, Hayes J, Martin W (1988) Logic simulation on vector processors. In: Computer-aided design, digest of technical papers. IEEE international conference on, pp 268–271
33. Richman J, Bowden KR (1985) The modern fault dictionary. In: Proc. international test conference, pp 696–702
34. Tai S, Bhattacharya D (1993) Pipelined fault simulation on parallel machines using the circuitflow graph. In: Computer design: VLSI in computers and processors, pp 564–567
35. Tulloss RE (1980) Fault dictionary compression: recognizing when a fault may be unambiguously represented by a single failure detection. In: Proc test conference, pp 368–370

Kanupriya Gulati is a Ph.D. Candidate in the Department of Electrical and Computer Engineering at Texas A&M University, College Station. She received her BE degree in Computer Engineering from Delhi College of Engineering, India in 2003 and her MS degree in Computer Engineering from Texas A&M University, College Station in 2006. She has been employed as an intern with several companies including Strategic CAD Laboratories, Intel, Cadence Research Laboratories, Mentor Graphics and Atrenta. Currently, for her doctoral degree, her research efforts include various aspects of VLSI design like optimizing logic synthesis and accelerating CAD algorithms in hardware. Kanupriya is a recipient of the NVIDIA fellowship for 2008–09 and one best student paper award. She is an IEEE student member.

Sunil P. Khatri received his B.Tech (EE) degree from IIT Kanpur, his M.S.(ECE) degree from the University of Texas, Austin, and the Ph.D. in EECS from the University of California, Berkeley. He worked at Motorola, Inc for 4 years., where he was a member of the design teams of the MC88110 and PowerPC 603 RISC microprocessors. Sunil is currently an Assistant Professor in ECE at Texas A&M University. His research interests include logic synthesis, novel VLSI design approaches to address issues such as power, cross-talk, hardware acceleration of CAD algorithms and cross-disciplinary applications of these topics. He has coauthored about 100 technical publications, five United States Patent awards, one book and a book chapter. His work has received three best paper awards and two best paper nominations. Sunil's research is supported by Intel Corporation, Lawrence Livermore National Laboratories, the National Science Foundation, Accelicon, Inc. and Nascentric, Inc.