Towards Acceleration of Fault Simulation using Graphics Processing Units

Kanupriya Gulati and Sunil P. Khatri
Department of ECE, Texas A&M University, College Station, TX 77843
kgulati@tamu.edu and sunilkhatri@tamu.edu

ABSTRACT

In this paper, we explore the implementation of fault simulation on a Graphics Processing Unit (GPU). In particular, we implement a fault simulator that exploits thread level parallelism. Fault simulation is inherently parallelizable, and the large number of threads that can be computed in parallel on a GPU results in a natural fit for the problem of fault simulation. Our implementation fault-simulates all the gates in a particular level of a circuit, including good and faulty circuit simulations, for all patterns, in parallel. Since GPUs have an extremely large memory bandwidth, we implement each of our fault simulation threads (which execute in parallel with no data dependencies) using memory lookup. Fault injection is also done along with gate evaluation, with each thread using a different fault injection mask. All threads compute identical instructions, but on different data, as required by the Single Instruction Multiple Data (SIMD) programming semantics of the GPU. Our results, implemented on a NVIDIA GeForce GTX 8800 GPU card, indicate that our approach is on average 35× faster when compared to a commercial fault simulation engine. With the recently announced Tesla GPU servers housing up to eight GPUs, our approach would be potentially 238× faster. The correctness of the GPU based fault simulator has been verified by comparing its result with a CPU based fault simulator.

Categories and Subject Descriptors: B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance
General Terms: Performance
Keywords: Fault Simulation, Graphics Processing Units.

1. INTRODUCTION

Fault simulation is an important step of the VLSI design flow. Given a digital design and a set of input vectors V defined over its primary inputs, fault simulation evaluates the number of stuck-at faults F_{sim} that are tested by applying the vectors V. The ratio of F_{sim} to the total number of faults in the design F_{total} is a measure of the fault coverage. The task of finding this ratio is often referred to as *fault grading* in the industry. For today's complex digital designs with N logic gates (N is often in the several million), the number of faulty variations of the design can be dramatically higher. Therefore, it is extremely important to explore ways to accelerate fault simulation. The ideal fault simulation approach should be fast, scalable, *and* cost effective.

Parallel processing of fault simulation computations is an approach that has routinely been invoked to reduce the compute time of fault simulation [1]. Fault simulation can be parallelized by a variety of techniques. The techniques include parallelizing the fault simulation algorithm (*algorithm-parallel techniques* [2, 3, 4]), partitioning the circuit into disjoint components and simulating them in parallel (*model-parallel* techniques [5, 6]), partitioning the fault set data and simulating faults in parallel (*data-parallel* techniques [7, 8, 9, 10, 11, 12, 13]) and a combination of one or more of these techniques [14]. Data parallel techniques can be further classified into *fault-parallel* methods, wherein different faults are simulated in parallel, and *pattern-parallel* approaches, wherein different patterns of the same fault are simulated in parallel. In this paper, we present an accelerated fault simulation approach that invokes data parallelism. In particular, *both* fault and pattern parallelism are exploited by our method. The method is implemented on a Graphics Processing Unit (GPU) platform.

Graphic Processing Units (GPUs) are designed to operate in a Single Instruction Multiple Data (SIMD) fashion. By design, the key application of a GPU is to serve as a graphics accelerator for speeding image scanning, image processing, 3D rendering operations, etc., as required of a graphics card in a CPU. In general, these graphics acceleration tasks process the same operations (i.e. instructions) independently on large volumes of data. This is why GPUs employ a SIMD computation semantic. In recent times, the application of GPUs for general purpose computations has been actively explored [15, 16, 17]. The rapid increase in the number and diversity of scientific communities exploring the computational power of GPUs for their data intensive algorithms has had a key contribution in encouraging GPU manufacturers to design more powerful, easily programmable and flexible GPUs. In addition, the development of open-source programming tools and languages for interfacing with the GPU platforms has further fueled the growth of general purpose GPU (GPGPU) applications. Further, 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 500 Gflops for G80 GPU (which is used in the GeForce 8800 GTX graphic card) in 2007 [18]. This high computing power mainly arises from a fully pipelined and highly parallel architecture, with extremely high memory bandwidths. GPU memory bandwidths have grown from 42 GB/s for the ATI Radeon X1800XT to 86.4 GB/s for the NVIDIA GeForce 8800 GTX GPU. In contrast the theoretical performance of a 3 GHz Pentium4 CPU is 12 Gflops, with a memory bandwidth of 6 GB/s to main memory.

Fault simulation of a logic netlist effectively requires multiple logic simulations of the netlist, with faults injected at various gates (typically primary inputs and reconvergent fanout branches). An approach for logic simulation which can be used for fault simulation, uses look-up table (LUT) based computations. In this approach the truth table for all the gates in a library are stored in the memory, such that multiple processors can perform multiple gate level (logic) simulations in parallel. This is a natural match for the GPU capabilities, since it exploits the extreme memory bandwidths of the GPU, as well as the presence of several computation elements on the GPU. Several faults, and several patterns for these faults, can be simulated using the same LUT. In this way, both fault parallelism and pattern parallelism is employed. The key point to note is that the *same* operation (of looking up gate output values in the memory) is performed on independent data (different faults and different patterns for every fault). In this way, the SIMD computing paradigm of the GPU is exploited maximally by fault simulation computations that are LUT-based.

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

- We *exploit the novel match between fault and pattern parallel fault simulation with the capabilities of a GPU, a SIMD-based device*, and harness the computational power of GPUs to accelerate fault simulation.
- The implementation satisfies all the key requirements which ensure maximal speedup in a GPU
  - The different threads, which perform gate evaluations and fault injection, are implemented so that there are no data dependencies between threads.
  - All gate evaluation threads compute identical instructions, but on different data, which exploits the SIMD architecture of the GPU.
  - The gate evaluation is done using a LUT, which exploits the extremely large memory bandwidth of GPUs.
- Our fault simulation algorithm is implemented in a manner which is aware of the specific constraints of the GPU platform, such as the use of texture memory for table lookup, memory coalescing, use of shared memory etc., thus maximizing the speedup obtained.
- In comparison to a state-of-the-art commercial fault simulation tool, our implementation is on average $\sim 35\times$ faster for fault simulating 32K patterns over 30 IWLS benchmarks [19].
- Further, even though our current implementation has been benchmarked on a single NVIDIA GeForce GTX 8800 graphics card, the recently announced NVIDIA Tesla GPU Computing Processor [20] allows up to eight NVIDIA Tesla GPUs (a 1U server). We show that our implementation, using the NVIDIA Tesla server, can perform fault simulation on average $\sim 240\times$ faster, when compared to the same commercial tool.

Our fault simulation 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 GPU device used for our implementation and benchmarking is NVIDIA GeForce 8800 GTX. The correctness of our GPU based fault simulator has been verified by comparing its results against a CPU based serial fault simulator.

The remainder of this paper is organized as follows: Some previous work in fault simulation 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 LUT based fault simulation in GPUs. In Section 5 we present results from experiments which were conducted in order to benchmark our approach. We conclude in Section 6.

2. PREVIOUS WORK

Over the last three decades, several research efforts have attempted to accelerate the problem of fault simulation in a scalable and cost-effective fashion, by exploiting the parallelism inherent in the problem These efforts can be divided into *algorithm-parallel*, *model-parallel* and *data-parallel*.

Algorithm-parallel efforts aim at parallelizing the fault simulation algorithm, distributing workload and/or pipelining the tasks, such that the frequency of communication and synchronization between processors is reduced [14, 2, 3, 4]. In contrast to these approaches, our approach is data-parallel. In [14], the authors aim at heuristically assigning fault set partitions (and corresponding circuit partitions) to several medium-grain multiprocessors. This assignment is based on a performance model developed by comparing the communication (message passing or shared memory access) to computation ratio of the multiprocessor units. The results detailed in [14] are based on an implementation on a multiprocessor prototype with up to 8 units. Our results, on the other hand, are based on off-the-shelf GPU cards (the NVIDIA GeForce GTX 8800 GPU). The authors of [2] present a methodology to predict and characterize workload distribution, which can aid in parallelizing fault simulation. The approach discussed in [3] suggests a pipelined design, where each functional unit performs a specific task. MARS [4], a hardware accelerator, is based on this design. However, the application of the accelerator to fault simulation has been limited [14].

In a model-parallel approach [5, 6, 14], the circuit to be simulated is partitioned into several (possibly non-disjoint) components. Each component is assigned to one or more processors. Further, in order to keep the partitioning balanced, dynamic re-partitioning [21, 22] is performed. This will increase algorithm complexity and may impact simulation time [21, 22].

Numerous data-parallel approaches for fault simulation have been researched in the past. These approaches use dedicated hardware accelerators, supercomputers, vector machines or multiprocessors [7, 8, 9, 10, 11, 12, 13]. There are several hardware accelerated fault simulators in the literature, but they require specialized hardware, significant design effort and time, and non-trivial algorithm and software design efforts as well. In contrast to these approaches, our approach accelerates fault simulation by using off-the-shelf commercial 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 simulation.

The application of GPUs for general purpose computations has been actively explored [15, 16, 17]. However, to the best of our knowledge, the use of GPUs for fault simulation is non-existent.

3. ARCHITECTURE

In this section we discuss the architectural aspects of the NVIDIA GeForce 8800 GTX GPU device, which is the GPU used in our implementation. We discuss the hardware model, memory model and the programming model for this device. This discussion is provided to help the reader understand the details of our implementation of a fault simulator on the GPU.

3.1 Hardware Model

The GeForce 8800 GTX architecture has 16 multiprocessors per chip and 8 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 describe the memory organization of the device.

3.2 Memory Model

Each multiprocessor has on-chip memory of the following four types [23, 24]:

- One set of local 32-bit *registers* per processor. The total number of registers per multiprocessor is 8192.
- A parallel data cache or *shared memory* that is shared by all the processors of a multiprocessor. The size of this shared memory per multiprocessor is 16 KB and it is organized into 16 banks.
- A read-only *constant cache* that is shared by all the processors in a multiprocessor, which speeds up reads from the constant memory space.
- A read-only *texture cache* that is shared by all the processors in a multiprocessor, which speeds up reads from the texture memory space.

The local and global memory spaces are implemented as read-write regions of the device memory and are not cached. These memories are optimized for different uses. The local memory of a processor is used for storing data structures declared in the instructions executed on that processor. The 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.

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 by 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 texture cache is optimized for spatial locality. In other words, if instructions that are executed in parallel read texture addresses that are close together, then the texture cache can be optimally utilized. A texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. Device memory reads through *texture fetching* (provided in CUDA for accessing texture memory) present several benefits over reads from global or constant memory.

We next discuss the GPU programming and interfacing tool.

3.3 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 [23, 24]. CUDA was released by NVIDIA corporation in early 2007.

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 cooperate together by efficiently sharing data through some fast shared 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 GeForce 8800 GTX, the warp size is 32. Thread blocks have restrictions on the *maximum* number of threads in them. In a GeForce 8800 GTX, the maximum number of threads grouped in a thread block is 512. However, the number of threads in a thread block, *dimblock*, is decided by the programmer, who must ensure that i) the maximum number of threads allowed in the block is 512 ii) *dimblock* a multiple of the warp size.

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 768 threads can be active in a single multiprocessor. When deciding the *dimblock* and *dimgrid* values, the restriction on the number of registers being used in a single multiprocessor (i.e. 8192) has to be carefully monitored. If this limit is exceeded, the kernel will fail to launch.

The GeForce 8800's synchronization paradigm is *local* to a thread block, and is very efficient. However, threads belonging to different thread blocks of even the same grid cannot synchronize.

We next discuss our implementation of fault simulation engine exploiting thread level parallelism on a GPU.

4. OUR APPROACH

GPUs allow extreme speedups if the different threads being evaluated have no data dependencies. 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. Also, GPUs have an extremely large memory bandwidth, allowing multiple memory lookups to be performed in parallel.

Since fault simulation requires multiple (faulty) copies of the same circuit to be simulated, it forms a natural match to the capabilities of the GPU. Also, each gate evaluation *within a specific level in the circuit* can be performed independently of other gate evaluations. As a result, if we implement each gate evaluation within a logic level on a GPU thread, these threads will naturally satisfy the condition required for speedup in the GPU (which requires that threads have no data dependencies). Also, we implement fault simulation on the GPU, which allows each of the gate evaluations in a fault simulator to utilize the same thread code, with no conditional computations between or within threads. In particular, we implement pattern-parallel and fault-parallel fault simulation. Fault injection is also done along with gate evaluation, with each thread using a different fault injection mask. This maximally exploits the SIMD computing semantics of the GPU platform. Finally, in order to exploit the extreme memory bandwidths offered by GPUs, our implementation of the gate evaluation thread uses a memory lookup based logic simulation paradigm.

Fault simulation of a logic netlist consists of multiple logic simulations of the netlist with faults injected at specific nets. In the next three subsections we discuss i) GPU based implementation of logic simulation at a gate, ii) fault injection at a gate and iii) fault detection at a gate. In the fourth subsection we discuss the implementation of fault simulation for a circuit. This uses the implementations described in the first three subsections.

4.1 Logic Simulation at a Gate

Logic simulation on the GPU is implemented using a look-up table (LUT) based approach. In this approach, the truth tables of all gates in the library are stored in a LUT. The output of the simulation of a gate of type G is computed by looking up the LUT at the address corresponding to the sum of the gate offset of G (G_{off}) and the value of the gate inputs.

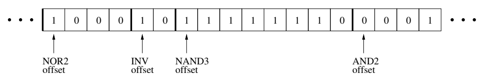

Figure 1: Truth Tables stored in a Look-up Table

Figure 1 shows the truth tables for a single NOR2, INV, NAND3 and AND2 gate stored in a one-dimensional look-up table. Consider a gate g of type NAND3 with inputs A, B and C and output O. For instance if ABC = '110', O should be '1'. This is performed by reading the value stored in the LUT at the address NAND3$_{off}$ + 6. Thus, the value returned from the LUT will be the value of the output of the gate being simulated, for the particular input value. LUT based simulation is a fast technique, even when used on a serial processor, since any gate (including complex gates) can be evaluated by a single lookup. Since the LUT is typically small, these lookups are usually cached. Further, this technique is highly amenable to parallelization as will be shown in the sequel. Note that in our implementation each LUT enables the simulation of 2 identical gates (with possibly different inputs) simultaneously.

In our implementation of the LUT based logic simulation technique on a GPU, the truth tables for all the gates are stored in the texture memory of the GPU device. This has the following advantages:

- Texture memory of a GPU device is cached as opposed to shared or global memory. Since the truth tables for all library gates will typically fit into the available cache size, the cost of a lookup will be one cycle (which is 8192 bytes per multiprocessor).
- Texture memory accesses do not have coalescing constraints as required in case of global memory accesses, making the gate lookup efficient.
- In case of multiple look-ups performed in parallel, shared memory accesses might lead to bank conflicts and thus impede the potential improvement due to parallel computations.
- Constant memory accesses in the GPU are optimal when all lookups occur at the same memory location. This is typically not the case in parallel logic simulation.
- The latency of addressing calculations is better hidden, possibly improving performance for applications like fault simulation that perform random accesses to the data.

- The CUDA programming environment has built-in texture fetching routines which are extremely efficient.

Note that the allocation and loading of the texture memory requires non-zero time, but is done only once for a gate library. This runtime cost is easily amortized since several million lookups are typically performed on a given design (with the same library).

The GPU allows several threads to be active in parallel. Each thread in our implementation performs logic simulation of 2 gates, of the same type (with possibly different input values) by performing a single lookup from the texture memory.

The data required by each thread is the offset of the gate type in the texture memory and the input values of the 2 gates. For example, if the first gate has a 1 value for some input, while the second gate has a 0 value for the same input, then the input to the thread evaluating these 2 gates is '10'. In general, any input will have values from the set {00, 01, 10, 11}, or equivalently an integer in the range [0,3]. A 2 input gate therefore has 16 entries in the LUT, while a 3 input gate has 64 entries. Each entry of the LUT provides outputs for both the gates. Each LUT entry is a word. Our gate library consists of an inverter as well as 2, 3 and 4 input NAND, NOR, AND and OR gates. As a result, the total LUT size is 4+4×(16+64+256) = 1348 words. Hence the LUT fits in the texture cache (which is 8192 bytes per multiprocessor). Simulating more than 2 gates simultaneously per thread does not allow the LUT to fit in the texture cache, hence we only simulate 2 gates simultaneously per thread.

The data required by each thread is organized as a 'C' structure *type struct threadData*, is stored in the global memory of the device for all threads. The global memory, as discussed in Section 3, is accessible by all processors of all multiprocessors. Each processor executes multiple threads simultaneously. This organization would thus require multiple accesses to the global memory. Therefore, it is important that the memory coalescing constraint for a global memory access is satisfied. In other words, memory accesses should be performed in sizes equal to 32-bit, 64-bit, or 128-bit values. In our implementation the *threadData* is *aligned* at 128-bit (= 16 byte) boundaries to satisfy this constraint. The data structure required by a thread for simultaneous logic simulation of a pair of identical gates with up to 4 inputs is:

typedef struct __align__(16){
int *offset*; // Gate type's offset
int *a*; int *b*; int *c*; int *d*;// input values
int m_0; int m_1; // fault injection bits
} threadData;

The first line of the declaration defines the structure type and byte alignment (required for coalescing accesses). The elements of this structure are : the offset in texture memory (type integer) of the gate which this thread will simulate, the input signal values (type integer) and variables m_0 and m_1 (type integer). Variables m_0 and m_1 are required for fault injection and will be explained in the following subsection. Note that the total memory required for each of these structures, 1 × 4 bytes for the offset of type int + 4 × 4 bytes for the 4 inputs of type integer and 2 × 4 bytes for the fault injection bits of type integer. The total storage is thus 28 bytes, which is aligned to a 16 byte boundary, thus requiring 32 byte coalesced reads.

The pseudocode of the kernel (the code executed by each thread) for logic simulation is given in Algorithm 1. The arguments to the routine *logic_simulation_kernel* are the pointers to the global memory for accessing the *threadData* (*MEM*) and the pointer to the global memory for storing the output value of the simulation (*RES*). The global memory is indexed at a location equal to the thread's unique *threadID* = t_x, and the *threadData* data is accessed. The index I to be fetched in the LUT (in texture memory) is then computed by summing the gate's offset and the decimal sum of the input values for each of the gates being simultaneously simulated. Recall that each input value ∈ {0, 1, 2, 3}, representing the inputs of both the gates. The CUDA inbuilt single-dimension texture fetching function $tex1D(LUT,I)$ is next invoked to fetch the output values of both gates. This is written at the t_x location of the output memory *RES*.

Algorithm 1 Pseudocode of the kernel for logic simulation

```
$logic\_simulation\_kernel(threadData *MEM, int *RES)\{$
$t_x = my\_thread\_id$
$threadData\ Data = MEM[t_x]$
$I = Data.offset + 4^0 \times Data.a + 4^1 \times Data.b + 4^2 \times Data.c + 4^3 \times Data.d$
$int\ output = tex1D(LUT, I)$
$RES[t_x] = output$
}
```

m_0	m_1	Meaning
–	11	Stuck-at-1 Mask
11	00	No Fault Injection
00	00	Stuck-at-0 Mask

Table 1: Encoding of the Mask Bits

4.2 Fault Injection at a Gate

In order to simulate faulty copies of a netlist, faults have to be *injected* at appropriate positions in the copies of the original netlist. This is performed by masking the appropriate bits by using a fault injection mask.

Our implementation parallelizes fault injection by performing a masking operation on the output value generated by the lookup (Algorithm 1). This masked value is now returned in the output memory *RES*. Each thread has it own masking bits m_0 and m_1, as shown in the *threadData* structure. The encoding of these bits are tabulated in Table 1.

The pseudocode of the kernel to perform logic simulation followed by fault injection is identical to pseudocode for logic simulation (Algorithm 1) except for the last line which is modified to read

$RES[t_x] = (output\ \&\ Data.m_0) \parallel Data.m_1$

$RES[t_x]$ has thus been appropriately masked for stuck-at-0, stuck-at-1 or no fault injected. Note that the two gates being simulated in the thread correspond to the same gate of the circuit, simulated for different patterns. The kernel which executes logic simulation followed by fault injection is called *fault_simulation_kernel*.

4.3 Fault Detection at a Gate

For an applied vector at the PIs, in order for a fault f to be detected at a primary output gate g, the good-circuit simulation value of g should be different from the value obtained by the faulty-circuit simulation at g, for the fault f.

In our implementation, the comparison between the output of a thread that is simulating a gate driving a circuit primary output, and the good circuit value of this primary output has to be performed efficiently. The modified *threadData_Detect* structure and the pseudocode of the kernel for fault detection are shown below.

typedef struct __align__(16) {
int *offset*; // Gate type's offset
int *a*; int *b*; int *c*; int *d*;// input values
int *Good_Circuit_threadID*; // The thread ID which computes
//the Good circuit simulation
} threadData_Detect;

Algorithm 2 Pseudocode of the kernel for fault detection

```
$fault\_detection\_kernel(threadData\_Detect * MEM, int * GoodSim, int * Detect, int * faultindex)\{$
$t_x = my\_thread\_id$
$threadData\_Detect\ Data = MEM[t_x]$
$I = Data.offset + 4^0 \times Data.a + 4^1 \times Data.b + 4^2 \times Data.c + 4^3 \times Data.d$
$int\ output = tex1D(LUT, I)$
if ($t_x == Data.Good\_Circuit\_threadID$) then
    $GoodSim[t_x] = output$
end if
$\_\_synch\_threads()$
$Detect[faultindex] = (output \oplus GoodSim[Data.Good\_Circuit\_threadID])?1:0$
}
```

The pseudocode of the kernel for fault detection is shown in Algorithm 2. This kernel is only run for the primary outputs of the design. The arguments to the routine *fault_detection_kernel* are the pointers to the global memory for accessing the *threadData_Detect* structure (*MEM*), a pointer to the global memory for storing the output value of the good circuit simulation (*GoodSim*) and a pointer in memory (*faultindex*) to store a 1 if the simulation performed in the thread results in fault detection (*Detect*). The first four lines of Algorithm 2 are identical to those of Algorithm 1. Next, a thread computing the good-circuit simulation value will write its output to global memory. Such a thread will have its *threadID* identical to the Data.Good_Circuit_threadID. At this point a thread synchronizing routine, provided by CUDA, is invoked. If more than one good circuit simulation (for more than one pattern) is performed simultaneously, the completion of all the writes to the global memory has to be ensured before proceeding. The thread synchronizing routine guarantees this. Once all threads in a block have reached the point where this routine is invoked, kernel execution resumes normally. Now all threads, including the thread which performed the good circuit simulation, will read the location in the global memory which corresponds to its good circuit simulation value. Thus, by ensuring the completeness of the writes prior to the reads, the thread synchronizing routine avoids write-after-read (WAR) hazards. Next, all threads compare the output of the logic simulation performed by them to the value of the good-circuit simulation. If these values are different, then the thread will write a 1 to a location indexed by its *faultindex*, in *Detect*, else it will write a 0 to this location. At this point the host can copy the *Detect* portion of the device global memory. All faults listed in the *Detect* vector are detected.

4.4 Fault Simulation of a Circuit

Our GPU-based fault simulation methodology is parallelized using the two data parallel techniques, namely fault parallelism and pattern parallelism. Given the large number of threads that can be executed in parallel on a GPU, we use both these forms of parallelism simultaneously. This section describes the implementation of this two-way parallelism.

Given a logic netlist, we first levelize the circuit. By levelization we mean that each gate of the netlist is assigned a level which is one more than the maximum level of its input gates. The primary inputs are assigned a level '0'. Thus, $Level(G) = \max(\forall_{i \in fanin(G)} Level(i))$ + 1. The maximum number of levels in a circuit is referred to as L. The number of gates at a level i is referred to as W_i. The maximum number of gates at any level is referred to as W_{max}, i.e. (W_{max} = $\max(\forall_i(W_i))$). Figure 2 shows a logic netlist with primary inputs on the extreme left and primary outputs on the extreme right. The netlist has been levelized and the number of gates at any level i are labeled W_i. We perform data-parallel fault simulation on all logic gates in a single level simultaneously.

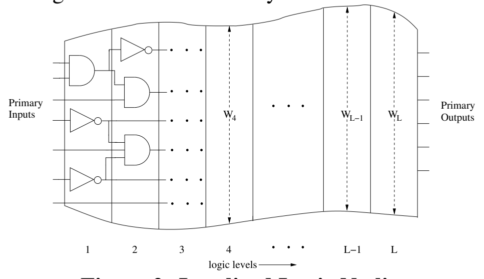

Figure 2: Levelized Logic Netlist

Suppose there are N vectors (patterns) to be fault simulated for the circuit. Our fault simulation engine first computes the good circuit values for all gates, for all N patterns. This information is then transferred back to the CPU, which therefore has the good circuit values at each gate for each pattern. In the second phase, the CPU schedules the gate evaluations for the fault simulation of each fault. This is done by calling i) *fault_simulation_kernel* (with fault injection) for each faulty gate G, ii) the same *fault_simulation_kernel* (but without fault injection) on gates in the transitive fanout (TFO) of G, and iii) *fault_detection_kernel* for the primary outputs in the TFO of G.

We reduce the number of fault simulations by making use of the good circuit values of each gate for each pattern. Recall that this information was returned to the CPU after the first phase. For any gate G, if its good circuit value is v for pattern p, then fault simulation for the stuck-at-v value on G is not scheduled in the second phase. In our experiments, the results include the time spent for the data transfers from CPU $\leftrightarrow$ GPU in all phases of the operation of out fault simulation engine. GPU runtimes also include all the time spent by the CPU to schedule good/faulty gate evaluations.

A few key observations are made at this juncture.

- Data-parallel fault simulation is performed on all gates of a level i simultaneously
- Pattern-parallel fault simulation is performed on N patterns for any gate simultaneously.
- For all levels other than the last level, we invoke the kernel *fault_simulation_kernel*. For the last level we invoke the kernel *fault_detection_kernel*.
- Note that no limit is imposed by the GPU on the size of the circuit, since the entire circuit is never statically stored in GPU memory.

5. EXPERIMENTAL RESULTS

In order to perform T_S logic simulations plus fault injections in parallel, we need to invoke T_S *fault_simulation_kernel*s in parallel. The total DRAM (off-chip) in the NVIDIA GeForce GTX 8800 is 768MB. This off-chip memory can be used as global, local and texture memory. Also the same memory is used to store CUDA programs, context data used by the GPU device drivers, drivers for the desktop display and NVIDIA control panels. With the remaining memory, we can invoke T_S = 16M *fault_simulation_kernel*s in parallel. The time taken for 16M *fault_simulation_kernel*s is 45.509 ms. The time taken for 16M *fault_detection_kernel*s is 80.901 ms. The fault simulation results obtained from the GPU implementation were verified against a CPU based serial fault simulator, and were found to verify with 100% fidelity.

We ran 60 large IWLS benchmark [19] designs, to compute the speed of our GPU based fault simulation tool. We fault simulated 32K patterns for all circuits. We compared our runtimes to those obtained using a commercial state-of-the-art fault simulation tool. The licensing agreement for this tool requires that we do not mention the name of the tool in this paper. This tool was run on a 1.5 GHz UltraSPARC-IV+ processor with 1.6 GB of RAM, running Solaris 9.

The results for our GPU based fault simulation tool for 30 representative IWLS benchmarks are shown in Table 2. Column 1 lists the name of the circuit. Column 2 lists the number of gates in the mapped circuit. Columns 3 and 4 list the number of primary inputs and outputs for these circuits. The number of collapsed faults F_{total} in the circuit are listed in Column 5. Columns 6 list the runtimes, in seconds, for simulating 32K patterns, using the commercial tool. The runtimes obtained from our implementation using a single GPU is listed in Column 7. These times include the time spent by the CPU to schedule good/faulty gate evaluations. The time taken to transfer data between the CPU and GPU is also accounted for in the GPU runtimes listed. In particular, the data transfered from the CPU to the GPU is the 32K patterns at the primary inputs, and LUT data consisting of 1348 words as explained in Section 4.1. The data transferred from GPU to CPU is the the good circuit evaluations for all 32K patterns, for all gates in the circuit, and the array *Detect* (which is of type integer, and has length equal to the number of faults in the circuit). The runtimes for the commercial tool include the time taken to read the circuit netlist and 32K patterns. The speedup obtained using a single GPU card are listed in Column 9. The speedup obtained is on average 34.88×. By using the recently announced NVIDIA Tesla server housing up to eight GPUs [20], the available global memory increases by 8×. Hence we can potentially invoke T_S = 128M *fault_simulation_kernel*s in parallel. This allows for a 8× speedup in the processing time. However, the transfer times do not scale. Column 8 lists the projected runtimes for a 8 GPU system. The speedup obtained against the commercial tool in this case are listed in Column 10. The speedup obtained in this case is on average 238.18×. If a single thread were to perform the gate evaluation for a single gate, for a single pattern (as opposed to 2 patterns in our implementation), the speedup number obtained is ∼17.96× when using a single GPU and ∼87.18× for an 8 GPU Tesla server. Note that the commercial tool can be run on several CPUs using a distributed option. If each of these CPUs had a 8800 GTX GPU on

Circuit	# Gates	# Inputs	# Outputs	# Faults	Runtimes (in seconds)			Speedup	
					Comm. Tool	Single GPU	8 GPU	Single GPU	8 GPU
s9234_1	1261	168	155	2202	26.740	2.043	0.282	13.089	94.953
s713	277	56	40	498	6.900	0.253	0.032	27.237	217.899
s641	275	56	40	484	7.070	0.286	0.044	24.743	158.994
s5378	1682	180	180	3543	31.950	1.961	0.271	16.290	117.716
s526n	269	20	27	534	4.310	0.157	0.028	27.390	151.659
s526	282	20	27	526	4.100	0.147	0.018	27.977	223.811
s444	204	20	27	422	4.030	0.094	0.012	42.796	342.367
s400	203	20	27	440	4.100	0.105	0.013	38.893	311.148
s386	210	15	13	422	2.460	0.094	0.012	26.219	209.754
s38584	8995	1036	1305	20128	265.590	7.883	1.125	33.691	235.998
s38417	10821	1303	1283	23866	296.360	8.234	1.580	35.994	187.516
s382	202	20	27	430	4.190	0.112	0.014	37.465	299.714
s35932	10537	1765	2048	24256	455.380	5.434	0.802	83.795	567.941
s349	177	23	26	347	4.270	0.089	0.011	48.143	385.136
s344	187	23	26	361	4.090	0.089	0.011	45.978	367.839
s298	170	13	20	307	3.040	0.073	0.009	41.413	331.299
s208_1	97	20	9	210	2.340	0.044	0.005	53.241	425.919
s15850	984	152	229	1593	34.030	0.420	0.061	81.074	555.882
s13207	1594	281	311	3032	52.590	0.656	0.091	80.160	579.453
b08	323	28	25	596	4.480	0.229	0.037	19.591	119.995
b03	497	32	34	1069	5.530	0.271	0.034	20.393	163.141
b02	52	6	5	113	1.280	0.028	0.003	45.911	367.288
b01	98	7	7	204	1.640	0.058	0.007	28.218	225.740
b10	407	24	23	767	4.020	0.340	0.051	11.834	78.494
b09	334	30	29	708	4.900	0.277	0.043	17.675	112.893
b13	507	55	63	1121	9.980	0.322	0.049	30.992	203.661
b17_1	51340	1387	1512	120639	736.670	17.866	14.335	41.232	51.391
b17	51045	1387	1512	120626	712.970	19.028	14.375	37.469	49.599
b22	34060	736	724	55077	252.040	60.485	57.969	4.167	4.348
b21	22470	524	512	53863	153.330	46.583	38.259	3.292	4.008
Avg.								34.879	238.185

Table 2: Fault Simulation Results

board, then the GPU approach could also exploit a distributed option, and the above speedup numbers would be maintained.

6. CONCLUSIONS

In this paper, we have presented our implementation of fault simulation engine on a Graphics Processing Unit (GPU). Fault simulation is inherently parallelizable, and the large number of threads that can be computed in parallel on a GPU can be employed to perform a large number of gate evaluations in parallel. As a consequence, the GPU platform is a natural candidate for implementing fault simulation. In particular, we implement a pattern and fault parallel fault simulator. Our implementation fault-simulates a circuit in a levelized fashion. All threads of the GPU compute identical instructions, but on different data, as required by the Single Instruction Multiple Data (SIMD) programming semantics of the GPU. Fault injection is also done along with gate evaluation, with each thread using a different fault injection mask. Since GPUs have an extremely large memory bandwidth, we implement each of our fault simulation threads (which execute in parallel with no data dependencies) using memory lookup. Our experiments indicate that our approach, implemented on a single NVIDIA GeForce GTX 8800 GPU card, can simulate on average 35× faster when compared to a commercial fault simulator tool. With the NVIDIA Tesla server [20], our approach would be potentially 238× faster.

7. REFERENCES

[1] P. Banerjee, *Parallel Algorithms for VLSI Computer-aided Design*. Prentice Hall, June 1994. ISBN-13: 978-0130158352 (ISBN-10: 0130158356).

[2] M. B. Amin and B. Vinnakota, “Workload distribution in fault simulation,” *J. Electron. Test.*, vol. 10, no. 3, pp. 277–282, 1997.

[3] A. Abramovici, Y. Levendel, and P. Menon, “A logic simulation engine,” in *IEEE Transactions on Computer-Aided Design*, vol. 2, pp. 82–94, April 1983.

[4] P. Agrawal, W. J. Dally, W. C. Fischer, H. V. Jagadish, A. S. Krishnakumar, and R. Tutundjian, “Mars: A multiprocessor-based programmable accelerator,” *IEEE Des. Test*, vol. 4, no. 5, pp. 28–36, 1987.

[5] V. Narayanan and V. Pitchumani, “Fault simulation on massively parallel simd machines: algorithms, implementations and results,” *J. Electron. Test.*, vol. 3, no. 1, pp. 79–92, 1992.

[6] S. Tai and D. Bhattacharya, “Pipelined fault simulation on parallel machines using the circuitflow graph,” in *Computer Design: VLSI in Computers and Processors*, pp. 564–567, Oct 1993.

[7] G. F. Pfister, “The yorktown simulation engine: Introduction,” in *DAC ’82: Proceedings of the 19th conference on Design automation*, (Piscataway, NJ, USA), pp. 51–54, IEEE Press, 1982.

[8] D. K. Beece, G. Deibert, G. Papp, and F. Villante, “The ibm engineering verification engine,” in *DAC ’88: Proceedings of the 25th ACM/IEEE conference on Design automation*, (Los Alamitos, CA, USA), pp. 218–224, IEEE Computer Society Press, 1988.

[9] F. Ozguner and R. Daoud, “Vectorized fault simulation on the cray x-mp supercomputer,” in *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pp. 198–201, Nov 1988.

[10] F. Ozguner, C. Aykanat, and O. Khalid, “Logic fault simulation on a vector hypercube multiprocessor,” in *Proceedings of the third conference on Hypercube concurrent computers and applications*, (New York, NY, USA), pp. 1108–1116, ACM, 1988.

[11] R. Raghavan, J. Hayes, and W. Martin, “Logic simulation on vector processors,” in *Computer-Aided Design, Digest of Technical Papers., IEEE International Conference on*, pp. 268–271, Nov 1988.

[12] N. Ishiura, M. Ito, and S. Yajima, “High-speed fault simulation using a vector processor,” in *Proceedings of the International Conference on Computer-Aided Design ICCAD*, Nov 1987.

[13] M. B. Amin and B. Vinnakota, “Data parallel fault simulation,” *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, vol. 7, no. 2, pp. 183–190, 1999.

[14] R. Mueller-Thuns, D. Saab, R. Damiano, and J. Abraham, “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, March 1993.

[15] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, “GPU cluster for high performance computing,” in *SC ’04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, (Washington, DC, USA), p. 47, IEEE Computer Society, 2004.

[16] J. Owens, “GPU architecture overview,” in *SIGGRAPH ’07: ACM SIGGRAPH 2007 courses*, (New York, NY, USA), p. 2, ACM, 2007.

[17] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, “GPGPU: general-purpose computation on graphics hardware,” in *SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 208, ACM, 2006.

[18] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, and T. Chiueh, “Graphic-card cluster for astrophysics (GraCCA) – performance tests,” in *Submitted to NewAstronomy*, July 2007.

[19] “IWLS 2005 Benchmarks.” `http://www.iwls.org/iwls2005/benchmarks.html`.

[20] “NVIDIA Tesla GPU Computing Processor.” `http://www.nvidia.com/object/IO_43499.html`.

[21] S. Parkes, P. Banerjee, and J. Patel, “A parallel algorithm for fault simulation based on proofs,” in *ICCD ’95: Proceedings of the 1995 International Conference on Computer Design*, (Washington, DC, USA), p. 616, IEEE Computer Society, 1995.

[22] S. Patil and P. Banerjee, “Performance trade-offs in a parallel test generation/fault simulation environment,” *IEEE Transactions on Computer-Aided Design*, pp. 1542–1558, Dec 1991.

[23] “NVIDIA CUDA Introduction.” `http://www.beyond3d.com/content/articles/12/1`.

[24] “NVIDIA CUDA Homepage.” `http://developer.nvidia.com/object/cuda.html`.