Acceleration of Functional Validation using GPGPU

Lalith Suresh[1], Navaneeth Rameshan[2], M. S. Gaur[3], Mark Zwolinski[4], Vijay Laxmi[5]
[4]ESD Group, University of Southampton, UK
Malaviya National Institute of Technology, Jaipur, India.
{suresh.lalith[1], navaneeth.rameshan[2]}@gmail.com, {gaurms[3], vlaxmi[5]}@mnit.ac.in, mz@ecs.soton.ac.uk[4]

***Abstract*—Logic simulation of a VLSI chip is a computationally intensive process. There exists an urgent need to map functional validation algorithms onto parallel architectures to aid hardware designers in meeting time-to-market constraints. In this paper, we propose three novel methods for logic simulation of combinational circuits on GPGPUs. Initial experiments run on two methods using benchmark circuits using NVIDIA GPGPUs suggest that these methods can be used for accelerating the EDA design flow process.**

***Index Terms*—Logic Simulation, GPGPU, EDA**

I. Introduction

One of the biggest challenges faced by the Electronic Design Automation (EDA) industry is the design productivity gap. Various phases of design automation for today's complex hardware are computationally intensive. Existing EDA tools are not powerful enough to meet these computational demands within the time-to-market constraints. However, the emergence of parallel computing technologies on multi-core and multiprocessing systems offers opportunities to close the gap and keep pace with Moore's law and the ever increasing complexity of designs.

The ability to design at higher abstraction levels has permitted the design of complex circuits, which has resulted in an increase in design complexity and consequently, an increase in verification complexity. Functional verification accounts for around 70% [4] of the design time. We propose three methods to speed up this computationally intensive phase by employing the massively parallel processing capabilities of General Purpose Graphics Processing Units (GPGPUs). We chose NVIDIA's Compute Unified Device Architecture (CUDA) [10] for writing GPGPU code for evaluation of proposed methods. These three methods are – (a) data-parallel method, (b) pipelining method and (c) event driven method.

The first proposed method is 'Data-Parallel method' wherein a vertically levelised circuit is simulated one level at a time, for multiple input vectors in parallel. Since we are dealing with combinational circuits, the computation of the output of any gate is not dependent on that of any other gate at the same level. Functional verification procedure, in such a case, becomes "embarrassingly" parallel and is well suited for GPGPUs.

The second approach is the pipelining method, which involves taking a vertically levelised circuit and feeding input vectors to multiple levels simultaneously, with the input vectors forming a pipeline. As soon as the first of these input vectors propagates to the last level of the circuit, all threads in the GPGPU are active.

The third method uses an event-driven model for simulating digital circuits on a GPGPU. In this method, an activation list is maintained and only as many threads are launched as there are gates to be processed. Additionally, we made an attempt to port the pipeline-based simulation code from NVIDIA CUDA to OpenCL [1].

The paper is organised as follows: Section II gives an overview of Logic Simulation. Section III explains the CUDA programming model. Section IV outlines recent work done on circuit simulation using GPGPUs. In Section V, we explain the proposed methods and the OpenCL port in detail. Section VI provides details about the experiments run and the results. Section VII concludes the paper and discusses our future work.

II. Logic Simulation

VLSI circuits may be simulated at various levels of abstraction, ranging from the circuit level to the behavioural level. In logic simulation, the states represent the different signal values on the wires interconnecting the circuit elements. Logic simulation, an essential part of functional verification, is used to validate the behaviour of the design, and consists of levelising the circuit with the corresponding dependencies as specified in the netlist. Thus, it is possible to concurrently simulate gates at a single level since they do not have any dependencies on other gates in the same level. Before a gate is evaluated, all its fan-ins must be evaluated.

Traditionally, logic simulation has been classified into *compiled code* and *event driven* simulation [9]. In a compiled code simulator, all logic elements are evaluated at every cycle, regardless of whether there is a change in the inputs. The advantage lies in extremely fast evaluation and propagation. Such code would contain few loops and conditions, which is suitable for parallel simulation. The limitation in this case would be the unnecessary evaluation of gates which do not involve a change in state. In an event-driven simulation, evaluation only follows the occurrence of an event, that is, a change in any one of the inputs. The simulator creates internal data structures to represent the circuit, on which logic evaluation and propagation operations are performed. The advantages are that only the active components of the circuit are processed and that all circuit models can be handled. The limitations are the scheduling overhead and the time taken to evaluate and propagate due to the need for traversal of the data structures.

III. CUDA Programming Model

A GPU is a highly parallel, multi-threaded, multi-core dedicated processor capable of rendering 2D or 3D graphics more efficiently as compared to a CPU. General purpose operations on GPUs are made possible by the addition of higher precision arithmetic and programmable stages to the rendering pipelines, permitting the use of stream processing on non-graphics data. Owing to the parallel structure, high computational power and memory bandwidth, general purpose GPUs are more effective than general purpose CPUs for a variety of algorithms and floating-point arithmetic operations [11]. They are primarily suited for data parallel computations with high arithmetic intensity relative to the number of memory operations.

CUDA[10] is a parallel programming model that leverages the computational capacity of the GPUs for non-graphics applications. It is a set of extensions to the C programming language that offers heterogeneous support so that applications can use both the CPU and the GPU for serial and parallel areas of the code respectively, the GPU thus being a co-processor. CUDA, at its core, consists of three abstractions: a hierarchy of thread groups, shared memory and barrier synchronization. These provide fine-grained data and thread parallelism, nested within coarse-grained data and thread parallelism.

Programming in CUDA entails the distribution of the code for execution between the host (CPU) and the device (GPU). The accesses to the global, constant and texture variables by the GPU kernel are done using calls to the CUDA runtime. The instructions to be executed on the device are encapsulated within a kernel function and the GPU is capable of launching a large number of threads operating on identical instructions, but on different sets of data, in a Single Instruction Multiple Data (SIMD) manner. A thread block (or block) is a group of threads (up to 512 threads executing concurrently) that work together efficiently by sharing data and synchronization to coordinate memory accesses. While each thread has its own private memory, a block has shared memory, visible to all the threads within the block and within the lifetime of the block. During execution, each block is assigned to a multiprocessor, each of which consists of on-chip memory including a set of local 32-bit registers, a shared memory, a read-only constant cache and a read-only texture cache. For the threads in a multiprocessor, the memory accesses to the larger device memory have a high latency (300 - 400 cycles), but fewer for accesses to the shared memory (1 - 16 cycle(s)). Efficient memory utilization is key to GPGPU programming. In our experiments, we have used the NVIDIA Quadro 1700 FX and the NVIDIA Tesla C870 GPGPUs.

IV. Recent Work

Recent work demonstrated the use of NVIDIA's General-Purpose Graphics Processing Units (GPGPUs) in logic and fault simulation of circuit designs. Gulati, et al. [7], [6], exploited the inherent parallelism of fault simulation algorithms and employed GPGPUs to accelerate fault simulation, which showed promising results in terms of speedup. Perinkulam, et al. [12], proposed a solution to parallelizing logic simulation using GPUs, but the performance of the system was limited by the high communication overhead. In the work by Garland [5], solutions have been proposed to deal with irregular data structures, specifically Sparse-Matrix Vector Product (SMVP) and graph traversal, two important EDA computing patterns. Gulati et al., [6], discuss efforts to accelerate transistor model evaluations using GPUs for SPICE based circuit simulation. In the work of Chatterjee, et al. [3], a novel GPU-accelerated simulator was developed, optimized for large structural net lists. Further to this, [2], they created a GPGPU-based event-driven logic simulator, delivering significant speed-ups over other event-driven logic simulators.

V. Logic Simulation on GPGPUs

We chose to use the ISCAS '85 benchmark circuits [8] for our work. Each of the methods described below works in the following manner:

- A Python-based parser accepts an ISCAS '85 benchmark circuit as an input.
- The parser generates circuit specific information required for the specific simulation kernel being used.
- The parser also generates an input file which feeds per gate information to the simulation code.
- The simulation code receives the above inputs and launches the appropriate kernel and the circuit is evaluated.

A. Data-Parallel Simulation

GPGPUs are SIMD processors and are well suited for data parallel operations. In this method, we exploit the data parallel capabilities of the GPGPU to perform Functional Verification. The first step involves vertical levelisation of the circuit. Level of a gate is determined by its distance, in terms of links, from input. Larger the distance, higher is the level. Gates directly connected to input are at level-0, their neighbours are at level-1, and so on. For a combinational circuit, gates in a level are not dependent on each other. This property allows us to design our method such that there is no communication between CUDA threads in a single launch of a kernel. We first parse the net list to generate per-gate information required for the CUDA based simulation. During this phase, if a gate is to propagate an output to a gate, which is more than one level away, we add 'dummy gates' for each level in between these two gates. The dummy gates store the output of a gate for every iteration of the verification process until it is to be used by another gate. After this step we obtain information required for the `Gate` structure.

```
struct Gate {
  int type;
  int fanin;
  int fanout;
  int inputOffset;
  int outputsToSet[];
} // Struct used for Data Parallel code
```

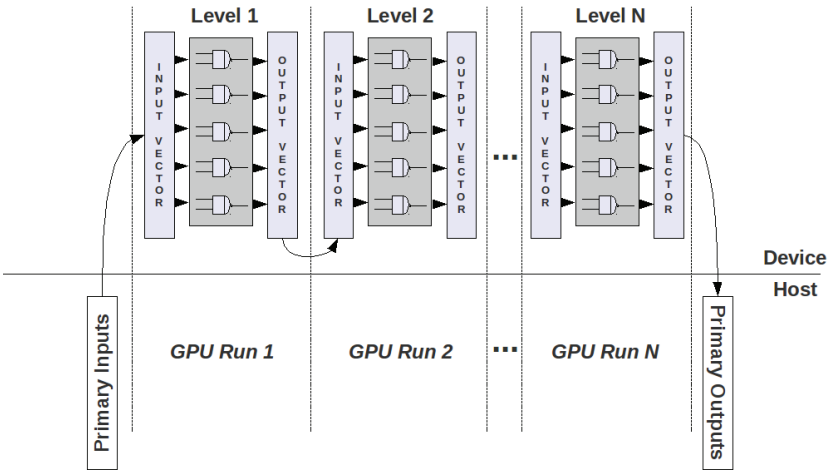

Figure 1. Data Parallel approach to Logic Simulation on a GPGPU

In the above, `type` indicates what kind of logic function the gate should perform. The logic functions are implemented using C macros. `fanin` and `fanout` respectively refer to the number of inputs the gate has and the number gates driven by this gate. `inputOffset` refers to the offset from the input vector from which the gate should consider the first `fanin` number of inputs. `outputsToSet` is the list of bits in the output vector that the gate's outputs are written to.

Figure 1 represents our method. In the first run, we take the first level of the circuit and the primary inputs (PIs). An input vector is constructed from a typical assignment of logical values (0 or 1) to PIs. This is done so that each CUDA thread, which is handling a gate, needs to refer only to a part of the same input vector if the offset for reading is known. The threads then perform the required operation for the gate. The output of the gate is written to the output vector in accordance with the gates to be activated (the fanout gates). This output vector forms the input vector for the next level. Threads within the same CUDA block perform the procedure. The parallelism attainable within a level is limited to the number of threads that can be launched within a block is 512. If the number of gates in a level exceeds 512, we perform the simulation of chunks. By launching multiple blocks, we can generate outputs for multiple input vectors in parallel. Note that there is no dependency between threads or blocks in this method.

1) Optimisations: Since we initially worked with NVIDIA Quadro FX 1700 cards, we were limited to a global GPU memory of only 512MB. For a reasonably large circuit requiring about 32,768 input vectors in parallel, it was observed that `Gate` struct will consume too much memory, leading to a GPU kernel launch failure. To work around this, we modified the struct to the following:

```
struct Gate {
  unsigned int gateParams;
  unsigned int outputsToSet[MAXOPSTOSET];
} // Memory Optimised Gate Struct
  //for Data Parallel code
```

Within `gateParams`, which is 32 bits long, we pack the gate's `type`, `fanin`, `fanout` and `inputOffset` fields. For the ISCAS 85 benchmark circuits that we worked with, the `type`, `fanin` and `fanout` fields could be encoded with 4 bits each. This left us 20 bits for the `inputOffset` field, of which we needed only 10 bits. Similarly, for the `outputsToSet[]` member of the structure, we were able pack up to three indices per entry in the array. This allowed us to cut down our memory usage by a significant amount.

Secondly, instead of using two buffers in global memory for holding input and output vectors, we made use of a single buffer in the on-chip shared memory. Once the threads read the required input vectors from their respective buffers, they proceed to compute the outputs of the gates they are processing. The output vector (which becomes the input vector for the next level) is then written back into the shared memory buffer accordingly. To ensure that each thread has read its input vector before another thread writes onto the buffer, a barrier synchronization is performed before the output vectors are written. This approach not only reduced the global memory usage, but also improved the code's performance because shared memory accesses occur in 1-16 clock cycle(s) as opposed to nearly 400 clock cycles for global memory access.

B. Pipeline Based Simulation

In this method, we compute the output at each vertical level in a pipelined fashion to perform Functional Verification. The first step again involves vertical levelisation of the circuit. We parse the net list to generate per-gate information. Again, we add dummy gates, where needed. The `Gate` structure is a little different.

```
struct Gate{
  int type;
  int fanin;
  int fanout;
```

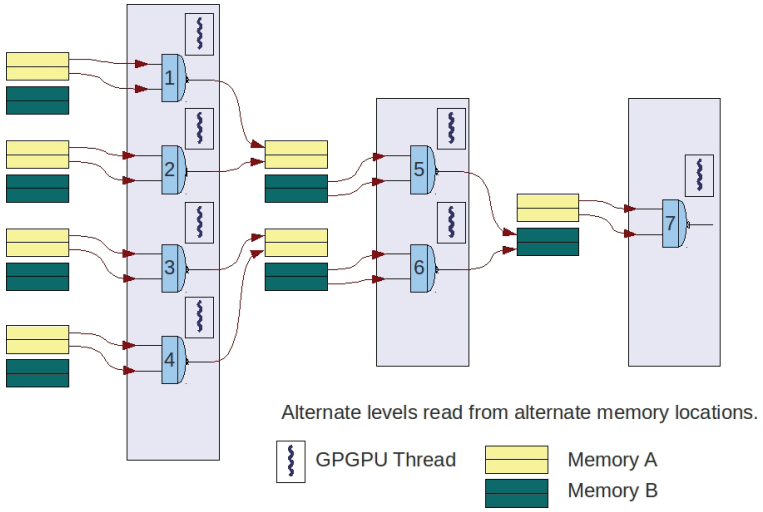

Figure 2. Pipeline based approach to Logic Simulation on a GPGPU

```
  int level;
  int InputValues[Fanin*2];
  int OutputPorts[Fanout*2];
} // Struct used for Pipelining Code
```

The first three fields are the same as in the data parallel method. `level` refers to the vertical level in which the gate is present. Input values contain the values received as input from the previous gate that triggers this gate. Input values for a gate are maintained as $2 \times$ `fanin` where one set of values are for one input vector and the other set for another input vector. Two sets of input values are used in order to avoid overlap during a write operation and at the same time run the simulation in a pipelined manner to achieve parallelism. Output ports are maintained as $2 \times$ `fanout` where one value contains the gate to which the output propagates and another contains the port of the gate to which the output propagates.

Figure 2 represents our method. In the first run, only the first level is evaluated and the output is propagated to the first memory location (shown in yellow in Figure 2) in accordance with the fan outs of the gate. In the next run, the first level is evaluated for the next set of input-vectors and propagated to the second memory location (shown in blue in Figure 2), while at the same time the second level is evaluated on the inputs (yellow) obtained from previous run. Alternate levels read from alternate memory locations with the input vectors forming a pipeline as shown in the figure. A thread is launched for every gate and all the threads become active when the first set of input vector propagates to the last level.

C. Event Driven Simulation

An obvious issue with the Data-Parallel and Pipeline based methods was the introduction of dummy gates into the circuit net list. For a reasonably large circuit, introduction of dummy gates would increase the total number of gates in the circuit manifold. For instance, the *c7552* ISCAS '85 benchmark circuit has 3512 gates. After introduction of dummy gates, the total number of gates shot up to 13114 gates. This resulted in increased computational complexity on GPU platforms. A possible efficient alternative could be slack based optimization.

Results of the previous experiments suggested that inclusion of dummy gates increase size of circuit and offset the advantages of parallelization of logic simulation. This was evident from the fact that parallel simulation of the circuit after inclusion of dummy gates was no faster than the sequential simulation of the same circuit without the dummy gates. The need to work around dummy gate injection was necessitated to exploit parallel GPGPU architecture for better performance. To achieve this, we opted to experiment with an event-driven simulation method. We have been able to develop a prototype for an event-driven logic simulator running on a GPGPU.

As explained in Section II, one cycle of an event driven simulation processes only those events that are ready to fire in that cycle. In the case of logic simulation, only those gates are activated that are ready to compute their output. Preparing an activation list that describes the set of gates to be activated does this. In the next paragraph, we explain the details of our implementation on a GPGPU.

The important data structures used are:

```
uint32_t *count_inputs;
uint32_t *activation_list;

struct Gate_t {
  uint32_t gateParams;
  uint32_t outputsToSet[MAXOUTPUTSTOSET];
} Gate;

Gate *Gates;
```

Let the highest number of inputs to a single gate in the circuit be N. The first $\log_2 N$ bits of `count_inputs` hold

the number of input lines of the gate that have been activated, and the next $\log_2 N$ bits hold the number of 1's in the input line. Since logic operations are associative, and assuming that we have only 1's and 0's in a logic circuit, it will suffice to compute the output of the gate if we merely know how many of the inputs are 1's.

The `activation_list` maintains the activation list for the simulation. Let the total number of gates in the simulation be `T`. The first `T` least significant bits of a particular entry in the `activation_list` refer to the gate index which is ready for activation. The following one bit refers to the output of the gate, which is to be propagated to all the fanouts.

The elements of `Gate` type are described as follows. `gateParams` encodes `fanin`, `fanout` and `type` of the gate and a flag indicating whether the gate is a primary output gate. The `outputsToSet` field holds the indices of the fanout gates that the gate drives. To save memory, multiple indices are encoded within one array element of `outputsToSet`.

In the first run of the simulation, all primary input gates are activated with the input vectors. All the gates that have enough inputs to compute their outputs, after this run, are added to the `activation_list` as described above. This happens under two circumstances:

- when a gate observes that the number of activated inputs of the gate (as inferred from the first part of `count_inputs[fanout_gate]`) is just one less than the total number of input lines of the fanout gate;
- when a gate activates an input line of a fanout gate with the respective controlling input.

Threads that have activated an input line and do not observe one of the above conditions in the gate they are driving enter a value of *T+1* into the activation list, where *T* is the total number of gates in the circuit. Once the activation list is prepared, it is sent back to the host, which pushes all the gates to be activated to the left of the list in a continuous sequence, also maintaining a count of the number of activations to be performed in the next run of the GPGPU kernel. This allows us to launch only as many threads as required so that threads on the GPGPU do useful work. If in a particular run of the GPGPU, there are no activations to be performed, it indicates that the simulation is over.

In the event-driven simulation method, if we try and simulate the circuit for multiple input vectors in parallel, with each input vector being handled by one block, it is not guaranteed that all blocks will end the simulation process together. This is a consequence of the event-driven method's property that only those components of the circuit are processed, which will be active. If some thread blocks remain idle as a result, the GPGPUs throughput will be affected.

Event driven implementation is work in progress but we expect this method do provide us improved speed-up on the basis of the arguments. We expect to compile the results by the time of final submission, if the paper gets accepted.

TABLE I
SPECIFICATIONS OF THE CPU USED FOR THE EXPERIMENT

Processor	Intel Core 2 Duo CPU E6550@ 2.33GHZ
Memory	1GB DDR2 RAM
L2 Cache	4 MB

TABLE II
SPECIFICATIONS OF THE NVIDIA QUADROFX 1700 GPGPU

CUDA Parallel Processor Cores	32
Memory Size	512 MB
Memory Interface	128-bit
Graphics Memory Bandwidth	12.8 GB/sec

TABLE III
SPECIFICATIONS OF THE NVIDIA TESLA C870

CUDA Parallel Processor Core	240
Memory Size	4 GB
Memory Interface	512-bit
Graphics Memory Bandwidth	102 GB/sec

D. OpenCL Port

OpenCL (Open Computing Language)[1] is a framework for writing programs that can execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL is a low level API compared to native CUDA. Both compile down to PTX (Parallel Thread eXecution), which is the intermediate assembly language. However the biggest advantage of OpenCL is that it supports other GPU computing hardware than NVIDIA's.

In OpenCL we define an N-Dimensional computation domain. Each independent element of execution in N-D domain is called a work-item. Work-items can be grouped together as work-group. Work-items in group can communicate with each other and can synchronize execution among work-items in group to coordinate memory access. It is possible to execute multiple work-groups in parallel. There are five main steps to run an OpenCL calculation: Initialization, Allocate resources, Creating programs/kernels, Execution, Tear down. Selecting a device and creating a context in which to run the calculation amounts to initialization. Allocate memory/storage that will be used on the device and push it to the device. Programs and kernels are read in from source and compiled or loaded as binary. Arguments to the kernel are set and the kernel is executed on all data. As part of the process we read back the results to the host and clean up memory. Tasks that are queued are always queued in order but they may be executed in order or out of order. Presently only the pipeline based approach has been converted to OpenCL. The run time of the OpenCL code was almost the same as its CUDA counterpart.

VI. RESULTS

We tested the Data Parallel and Pipelining code on the ISCAS '85 benchmark circuits. Tables I, II and III describe our experimental set-up. The serial code we developed is run on the CPU specified in Table I, on a single core. The results obtained from the NVIDIA Quadro FX 1700 (Table II) are

TABLE IV
RESULTS OF THE EXPERIMENTS RUN ON AN NVIDIA QUADRO FX 1700 GPGPU.

Circuit	Sequential (s)	Data-Parallel (s)	Pipelining (s)	Speed-up Data-Parallel	Speed-up Pipelining
c1355	2.40	0.91	2.38	2.64	1.01
c1908	6.01	4.08	5.49	1.47	1.09
c2670	5.57	3.23	9.34	1.72	0.6
c3540	7.60	5.21	7.02	1.46	1.08
c5315	19.28	10.37	17.73	1.86	1.09
c6288	20.05	8.19	17.21	2.45	1.17

TABLE V
RESULTS OF THE EXPERIMENTS RUN ON AN NVIDIA TESLA GPGPU.

Circuit	Sequential (s)	Data-Parallel (s)	Pipelining (s)	Speed-up Data-Parallel	Speed-up Pipelining
c1355	2.40	0.22	0.88	10.90	2.72
c1908	6.01	0.79	1.49	7.60	4.03
c2670	5.57	0.76	3.76	7.32	1.48
c3540	7.60	0.99	2.00	7.67	3.79
c5315	19.28	2.28	5.37	8.45	3.59
c6288	20.05	1.98	4.32	10.12	4.64

shown in Table IV. The speedups achieved on the NVIDIA Tesla C870 (Table III) are displayed in Table V.

Tables IV and V suggest that the optimizations play an important part in the speed-ups achieved. The Data-Parallel code, wherein we had used the optimizations described in Section V.A.1, runs faster that the unoptimized Pipelining code. Shared memory usage was the key factor in this difference since the access times for global memory are two orders of magnitude slower than that required for shared memory. It also demonstrates that the embarrassingly parallel approach is much better suited for GPGPUs.

It is also noted that the speed up is higher for the larger circuits. The speedups obtained with the NVIDIA Tesla C870 card is higher than that obtained with the NVIDIA Quadro FX 1700. This is because the Tesla has more number of processing cores (240) in comparison to the Quadro FX card (32). A higher number of processing cores provides room for a higher degree of parallelism. Thus, more blocks are executed in parallel and the overall number of context switches is reduced for the same number of blocks.

VII. CONCLUSIONS AND FUTURE WORK

We have explored the use of GPGPUs in the logic simulation of digital circuits. The three prototypes we developed for GPGPU kernels for simulating circuits suggest that it is possible to use GPGPUs in the EDA domain. The affirmative results indicate that GPGPUs can help towards shortening the design gap. There is also room for further optimization. We plan to evaluate these methods on larger circuits such as IWLS benchmark circuits. This will allow us to better analyze the effectiveness of the proposed methods.

In near future, we are working to generate simulation results using OpenCL. Porting all the code to OpenCL will help us experiment on different architectures and compare architecture specific code like CUDA and Brook+ with their OpenCL counterparts.

Though pipelining approach demonstrated a speed up for combinational designs, the method need extension for handling sequential designs, which is one of our future works.

As explained in Section V.C, our prototype for the event-driven simulation kernel is yet to be ready for extracting results. We plan to address this and perform comparisons with the other kernels.

REFERENCES

[1] Khronos Group OpenCL. www.khronos.org/opencl.

[2] D. Chatterjee, A. DeOrio, and V. Bertacco. Event-driven gate-level simulation with GPGPUs. In *DAC*, pages 557–562. ACM, 2009.

[3] D. Chatterjee, A. DeOrio, and V. Bertacco. GCS: High-performance gate-level simulation with GPGPUs. In *DATE*, pages 1332–1337. IEEE, 2009.

[4] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 286–291, New York, NY, USA, 2003. ACM.

[5] M. Garland. Sparse matrix computations on manycore gpu's. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 2–6, New York, NY, USA, 2008. ACM.

[6] K. Gulati, J. F. Croix, S. P. Khatri, and R. Shastry. Fast circuit simulation on graphics processing units. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 403–408, Piscataway, NJ, USA, 2009. IEEE Press.

[7] K. Gulati and S. P. Khatri. Towards acceleration of fault simulation using graphics processing units. In L. Fix, editor, *DAC*, pages 822–827. ACM, 2008.

[8] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design and Test*, 16(3):72–80, 1999.

[9] G. Meister. A survey on parallel logic simulation. Technical report, University of Saarland, Department of Computer Science, Misra J, 1993.

[10] Nvidia. Compute unified device architecture. www.nvidia.com/object/cuda_home.html.

[11] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.

[12] A. Perinkulam and S. Kundu. Logic simulation using graphics processors. In *Proc. ITSW*, 2007.