Efficient Fault Simulation on Many-Core Processors

Michael A. Kochte, Marcel Schaal, Hans-Joachim Wunderlich, Christian G. Zoellin
Institut fuer Technische Informatik, Universitaet Stuttgart
Pfaffenwaldring 47
70569 Stuttgart, Germany
{kochte,schaal,zoellin}@iti.uni-stuttgart.de, wu@informatik.uni-stuttgart.de

ABSTRACT

Fault simulation is essential in test generation, design for test and reliability assessment of integrated circuits. Reliability analysis and the simulation of self-test structures are particularly computationally expensive as a large number of patterns has to be evaluated.

In this work, we propose to map a fault simulation algorithm based on the parallel-pattern single-fault propagation (PPSFP) paradigm to many-core architectures and describe the involved algorithmic optimizations. Many-core architectures are characterized by a high number of simple execution units with small local memory. The proposed fault simulation algorithm exploits the parallelism of these architectures by use of parallel data structures. The algorithm is implemented for the NVIDIA GT200 Graphics Processing Unit (GPU) architecture and achieves a speed-up of up to 17x compared to an existing GPU fault-simulation algorithm and up to 16x compared to state-of-the-art algorithms on conventional processor architectures.

Categories and Subject Descriptors

B.6.2 [**Hardware**]: Logic Design–Reliability and Testing

General Terms

Algorithms

Keywords

Parallel Fault Simulation, Many-Core Processors, PPSFP

1. INTRODUCTION

Fault simulation determines for a given circuit and a given set of input patterns all the faults of a structural fault model that are observable at the outputs. It is an essential step in the verification of design for test methods and in assessing the quality of test sets and fault tolerance techniques. In particular, applications in built-in self-test and in reliability assessment require the computationally expensive evaluation of a large number of patterns.

Many-core processor architectures maximize the number of execution units in a given chip area. These architectures provide extremely favorable cost-performance trade-offs for signal processing, computer graphics or in linear algebra computations [1]. For all these applications, there is a beneficial ratio between computation and communication. Furthermore, the memory accesses are easily predicted to take advantage of the small local memories.

In contrast, fault simulation is performed on a circuit graph and the numerous fault injections are implemented as traversals of the graph. Graph traversal has a low ratio between computation and communication and it is difficult to predict the memory accesses [2]. In general, graph-based algorithms are difficult to map to many-core processors and often only a moderate speed-up compared to large conventional processors is achieved [3–7]. Since the circuit graph is directed and acyclic, it is possible to use partitioning and topological ordering to parallelize the traversal. Logic simulation implemented in this way yields reasonable speed-ups compared to logic simulation on conventional processors [8].

However, fault simulation has to consider each fault and requires many more graph traversals than regular logic simulation. Hence, the fault simulation techniques used for conventional processors focus on algorithmic optimizations that avoid most of the graph traversals entirely. Common techniques for combinational circuits are the simulation of multiple patterns in parallel (Parallel-pattern single-fault propagation, PPSFP) [9], structural analysis [10, 11], and the use of signal dominators and stem regions [12, 13]. Efficient data structures can further decrease simulation time [14, 15].

Parallel implementations of the above techniques either target classical vector computers [16–19] or use coarse grained parallelization for symmetric multi-processors or clusters [20–24]. But all the techniques assume a complex conventional processor architecture with significant memory hierarchy.

Existing evaluation of the suitability of many-core or General Purpose GPU (GPGPU) architectures for fault simulation has indicated that it is difficult to implement the algorithmic optimizations above and that a brute-force approach already yields high utilization of the hardware [25]. However, the speed-up by the brute-force method is very small compared to simulators such as [14].

For the first time, this paper presents how algorithmic optimizations can be implemented on many core processors, similar to those in state-of-the-art fault simulators for conventional processors. The approach simulates multiple patterns and multiple faults in parallel. Structural analysis takes into account local sensitization and can take advantage of already detected faults to significantly reduce the number of graph traversals. Efficient data structures enable very high data-parallelism with thousands of execution threads, make best use of available memory and use single-instruction multiple-data (SIMD) execution units effectively.

The technique is implemented for an NVIDIA graphics processor with 30 cores and 8 SIMD execution pipelines per core [26]. The architecture exhibits the typical properties of many-core architectures of simple in-order SIMD execution units, i.e. support for a large number of threads of execution, little

or no cache hierarchy and small local scratch-pad memories. In contrast to the existing approach, the technique presented here scales to large multi-million gate circuits. For a set of benchmark and industrial circuits, it achieves a considerable speed-up of 17x compared to the GPU-based approach in [25] and of 16x compared to a state-of-the-art fault simulator running on an Athlon X2 processor with conventional architecture.

The rest of the paper is organized as follows: After the introduction, known algorithmic optimizations for fault simulation are reviewed. The third section evaluates these optimizations regarding their suitability for many-core processors and introduces a set of algorithms and data structures that allow for efficient fault-simulation on many-core processors. Section four presents the implementation details as well as the evaluation for a set of benchmark circuits.

2. FAULT SIMULATION

For fault simulation, the circuit is modeled as an acyclic, directed graph $G(V, E)$ in which the gates are represented by the vertices V and the nets or signals of the circuit are the edges E (Fig. 1). The value of a signal edge is given by a mapping $v : E \rightarrow \{0, 1\}$. Fault simulation is performed with respect to an abstract fault model. The most common fault model is the stuck-at fault model, which assumes that a given vertex x has a constant value $v(x) \in \{0, 1\}$ independent of the input pattern. While the techniques below mainly target stuck-at faults, they can be extended to other fault models using the concept of conditional stuck-at faults [27].

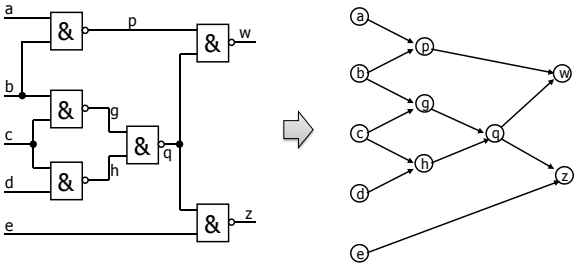

Figure 1: Example of a circuit graph

The two most common fault simulation techniques are the PPSFP technique [9] and the concurrent algorithm [28].

If only the combinational logic is considered (e.g. in full-scan circuits), the single fault propagation technique is more efficient than the concurrent approach [29]. This is due to the fact that in concurrent fault simulation multiple faults are simulated in groups and simulation of the fault propagation has to continue as long as at least one of the faults in the group is observable. This is particularly disadvantageous if the faults exhibit highly different sensitivities. In the PPSFP algorithm, fault propagation of only a single fault at a time is evaluated and multiple patterns are encoded into a machine word [9].

The fault propagation is usually computed with event-based simulation, since most faults cause only a small number of nets to change. The event-based simulation is implemented as a breadth-first traversal of the graph. The traversal stops at vertices that do not propagate an event. To efficiently implement the traversal, the vertices of the graph are stored in topological order (i.e. levelized) and vertices to be evaluated are kept in a priority queue or a level queue [14, 15]. This reduces the cost of traversal. But even then, the simulation of the fault propagation is the computationally most intensive part of fault simulation. Hence, additional algorithmic optimizations are employed to reduce the number of fault propagations as much as possible.

If a vertex of the graph has only a single successor (i.e. the represented gate in the circuit does not fan-out), the fault detection at the gate may also be determined locally using the critical path tracing method, which is based on the Boolean difference [10]. Combined with PPSFP, this technique substantially reduces the number of fault propagations to be explicitly computed by simulation to just faults at vertices with more than one successor (i.e. fan-out stems) [11] (Fig. 2).

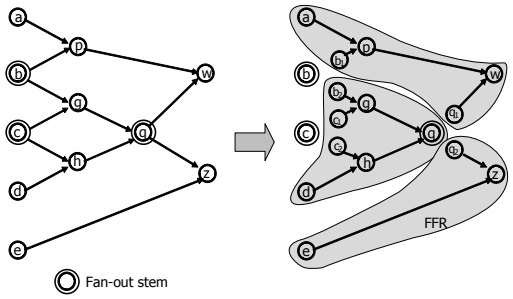

Figure 2: Fan-out stems and fan-out free regions (FFR) in the graph

The number of gate evaluations for a fault propagation analysis can be further reduced by taking advantage of signal dominators [12]. A vertex d is a dominator of vertex v if all forward paths from v go through d.

The aforementioned techniques are combined with fault-dropping: Faults that have been detected are not considered any more for critical path tracing or fault propagation. The effect of fault-dropping can be improved by analyzing a fan-out free region (FFR) with critical path tracing before performing an explicit fault propagation computation. If no fault in the FFR can be detected at the stem, the explicit fault propagation analysis for this stem is avoided entirely [13]. This technique is also called check-up [15].

Taking all of the optimizations into account, this leads to the following sequence of combinational fault simulation:

1. Simulation of the fault-free circuit for n patterns in parallel (good-simulation).
2. For each fan-out stem:
   (a) Backward traversal in the FFR of the stem to determine the local sensitivity of each fault at each vertex regarding the n patterns. Accumulation of the overall sensitivity at the fan-out stem regarding all the faults and vertices in the FFR.
   (b) For the patterns that sensitize faults at the fan-out stem, the observability of the stem is determined by explicit computation of the fault propagation. This explicit fault simulation stops at dominators.
3. The local sensitivities of step 2a) are combined with the results of the propagation analysis in step 2b) and the observabilities of the dominators to determine fault detection.

The algorithmic optimizations above improve the speed of fault simulation by several orders of magnitude. However, they are not directly suitable for implementation on SIMD many-core architectures.

3. PPSFP AND OPTIMIZATIONS ON MANY-CORE PROCESSORS

In this section, we discuss the basic properties of many-core architectures and evaluate the implications to the algorithm presented in the previous section. We show how the algorithmic optimizations and the fault propagation can be efficiently executed on these processors. For this, regularity is introduced to the algorithm and the problem is split into tasks that contain independent, data-parallel sub-problems.

Many-core architectures try to effectively use the transistor budget of recent process technologies. They contain a large number of simple compute cores with relatively little memory on the chip. Their architecture often allows for partially good chips to be used, such that manufacturing yield can be improved and they are available at relatively low cost.

Several architectures are commercially available today or in the near future that exhibit these properties. Amongst others, we find GPGPUs such as the NVIDIA GT200 and Fermi architectures, the AMD Evergreen architecture or Intel Larrabee. Similarly, the Cell processor, although providing only 8 cores per Chip, exhibits all the architectural properties of many-core processors. The Tilera TILE-Gx100 implements a MIPS-derived architecture with up to 100 VLIW cores. The ClearSpeed processors target scientific floating-point workloads and offer up to 192 compute cores with small local memory.

The common aspects of many-core architectures are:

- The execution units are very simple in-order pipelines. This requires static instruction scheduling to avoid stalls and the avoidance of difficult to predict branches to avoid pipeline flushes.
- The execution units are usually arranged in a SIMD micro-architecture to further reduce the overhead for control logic. This requires that the execution flow for parallel data matches exactly.
- The memory associated to each core is just the register file and a very small local memory that is directly addressed. Accesses to the local memory have very low latency.
- An on-chip network allows to exchange data between cores and with external main memory.
- No caching occurs when communicating with main memory. Hence accesses to main memory always suffer from extremely long latencies of several hundred clock cycles.
- The control logic and register file in each core provide hardware-support for multiple threads of execution. As a result, threads can be switched rather quickly on long-latency stalls caused by communication with other cores or with main memory.

Thus, the following effects have to be taken into account regarding the algorithm from the previous section: The transfer of the graph-based data structures requires a significant amount of bandwidth. To achieve maximum data parallelism with the available memory, data structures must be space-efficient. The event-based traversal of the graph has very unpredictable memory access patterns, making it difficult to effectively use the memory hierarchy. The interleaving of forward and backward traversals further reduces the predictability of memory accesses and requires additional storage of backward edges. The level queue must be managed concurrently when multiple gates are evaluated in parallel using SIMD, causing considerable overhead. And finally, to take advantage of SIMD, simply extending pattern-parallel simulation does not yield sufficient benefits [9, 14] and reduces the number of signal values that can be kept in local memory.

Here we show how the algorithmic techniques can be mapped to many-core processors. This requires both modifications of the data structures and the algorithms involved. The underlying ideas of this mapping are:

- Retain the algorithmic optimization that reduce the number of required fault injections.
- Separate the forward and backward traversals in the algorithm to make memory accesses more regular.
- Extract subgraphs that allow all of the fault propagations to be evaluated in parallel.
- Encode the gate information so that the storage for the subgraph is minimized. Since latency and bandwidth to main memory are an issue in these architectures, some additional computations for decoding of data structures are acceptable.
- Implement fault propagation such that multiple vertices can be efficiently evaluated with a simple in-order SIMD-architecture.

Hence, the resulting flow of the mapped algorithm is:

1. Forward traversal of the graph: For each vertex, compute the following:
   (a) Evaluation of the Boolean function associated with the vertex for the fault-free simulation.
   (b) Computation of the local sensitivities taking into account already detected faults and propagation of the sensitivities up to the fan-out stems.
2. In parallel, evaluate fault propagation for all fan-out stems that observe sensitized faults.
3. Backward traversal of the graph: For each vertex compute the following:
   (a) For all vertices with only one successor (fan-out free gates), evaluate the sensitivity of the vertex based upon the sensitivity of the successor.
   (b) From the sensitivity of each vertex determine the observability of the faults and update the fault list.

The most computation effort is spent on step 2 of the algorithm and only this part is described in all details due to the limited space available.

3.1 Step 1: Fault-free simulation and local sensitivities

The netlist graph is topologically sorted (i.e. levelized) and avoids the expensive priority queue of gates. The number of patterns evaluated in parallel matches the machine word size. For the initial forward traversal, the following data structures are employed for the graph and the values:

```
type Vertex {
        int   input0, input1;
        int   function;
        short isFanout;
        short faultList;
}
Vertex[] graph;
int[]    faultFreeValues;
int[]    sensitivities;
```

The values *faultFreeValues* and the evaluation of local sensitivities are kept in separate arrays to facilitate efficient access to these values in step 2 of the algorithm. Only gates with two inputs can be mapped to a vertex in the graph. Supporting a variable number of inputs increases memory requirements and results in branching and divergent control flow during the graph traversal. The fault list is stored as a simple bit-field of the six stuck-at faults that can occur at the gate.

For step 1 of the algorithm, all vertices of identical topological rank can be evaluated in parallel. To improve the

efficiency of SIMD, all branching during the evaluation can be avoided by use of *select* instructions available in these architectures.

3.2 Step 2: Fault propagation from the fan-out stems

For evaluation of the fault propagation, the graph is partitioned such that each partition contains a limited number of fan-out stems including their transitive successors. These partitions may overlap. The maximum size of the partitions is determined by the size of the local memories and the address range of indices in the partition data structure.

Event-based simulation reduces the efficiency of SIMD computation and requires additional storage and clock cycles for management of the event queue. It turns out that evaluation of all gates of the partition starting from the fan-out stem is more efficient. Similarly, it did not prove efficient to check for signal dominators and the required merging of dominator sensitivities.

Now the vertices are encoded as tightly as possible. The gate functions are mapped to the most common types which can be one-hot encoded in just four bits. The ordering and size of the subgraph allows to encode the predecessor indices with a reduced number of bits as described below. This information can be decoded with just a few machine instructions. For 32-bit machine words, the encoding is as follows:

31	30	29	28	27 .. 16	15 .. 0
AND	OR	XOR	INV	IN1	IN0

Table 1: Encoding of a gate for fault propagation

Table 2 lists the encoding of the supported Boolean functions.

Function	Code
AND	1000
NAND	1001
OR	0100
NOR	0101
XOR	0010
XNOR	0011
BUF	0000
NOT	0001

Table 2: Encoding of the Boolean function of a gate

Each partition is topologically sorted with all outputs placed at the end of the partition. Due to this ordering, at least one predecessor of a vertex is of the preceding rank. The input IN1 points to that predecessor and is stored relative to the current vertex index. That way, fewer bits are required for encoding. The input IN0 can point to any vertex in the partition.

To reduce the number of random memory transfers from main memory, only the fault-free simulation values at the boundary of the partition are copied. All fault-free simulation values internal to the partition are then computed again by the core. A partition can then be described by the following:

```
type Partition {
    int offPart[];   // indices to partition boundary
    int subgraph[]; // encoded gates
    int numOutputs; // number of outputs in part.
    Fanout fo[];     // fan-out stems in reverse
                     // order w.r.t. subgraph[]
}
type Fanout {
    int inPart; // index of fan-out stem in part.
    int global; // mapping to global netlist
}
```

In the event-based fault propagation, the consequences of a fault propagation need to be undone (e.g. the fault-free simulation values need to be restored). This step is avoided here as fan-out stems are evaluated in reverse topological order. Thereby, results of previous fault propagation computations are implicitly overwritten. Only copies of the fault-free output values have to be retained for final comparison. This results in the following algorithm for the fault propagation:

```
procedure evaluate_partition(Partition p)
    int values[];
    int outputs[];
    for i = 0 .. |p.offPart|-1
        values[i] = faultFreeValues[p.offPart[i]];

    // fault-free simulation by fwd. graph trav.
    for i = 0 .. |p.subgraph|-1
        evaluate(i);

    // retain fault-free outputs
    for i = 0 .. numOutputs-1
        outputs[i] = values[|p.subgraph|-i]

    for all f in p.fo[]
        continue if sensitivities[f.global] == 0;

        values[f.inPart] = not values[f.inPart];

        // fault propagation by partial fwd. trav.
        for i = f.inPart+1 .. |p.subgraph|-1
            values[i] = evaluate(i);

        //evaluate fan-out sensitivity
        int sens=0;
        for i = 0 .. p.numOutputs-1
            sens = sens | (outputs[i] xor
                           values[|p.subgraph|-i]);
        sensitivities[f.global] = sens;
```

3.3 Step 3: Computation of global fault observability

The computation of the global fault sensitivities is a backward traversal through the data structure used for step 1. To obtain the final observabilities for all vertices, the observabilities for fan-out stems as computed in step 2 are propagated to the stems' predecessors in the FFR. For each gate, the final detectability of the faults is computed and the fault list is updated. As in step 1, all the vertices with identical rank can be evaluated in parallel.

4. RESULTS

This section evaluates the implementation of the proposed algorithm for the NVIDIA GT200 GPGPU with 30 execution units. The implementation is evaluated on a set of the IWLS benchmark circuits [30] and industrial circuits (c.f. Table 3). The runtimes are compared to the existing GPGPU-based fault simulator of [25] which does not exploit any algorithmic optimizations. Instead, that approach tries to use the execution units to full capacity by regularity in the computations.

Furthermore, the results are compared to the event-based PPSFP algorithm in [14][1] which implements the discussed optimizations and yields runtimes comparable to state-of-the-art commercial fault simulators.

All runtime results have been obtained by simulation of 32768 random patterns on the circuit with a collapsed fault

[1]The implementation of [14] is openly available as source code.

Circuit	Gates	Inputs	Outputs	Faults
s5378	1106	364	308	3182
s9234.1	5944	247	250	7274
s13207	1290	688	710	3732
s15850	596	287	320	1692
s35932	7152	3781	3747	24032
s38417	10125	3158	2985	27176
s38584	8571	2351	2024	23906
b17	37446	1452	1512	81330
b17_1	44544	1452	1512	92794
b18	130949	3357	3364	277976
b19	263547	6666	6713	560696
b20	22557	522	512	47376
b21	23100	522	512	48182
b22	33569	767	757	70464
p100k	84356	5902	5829	166960
p141k	152808	11290	10502	287552
p239k	224597	18692	18495	455992
p259k	298796	18713	18495	607536
p267k	238697	17332	16621	372140
p269k	239771	17333	16621	374296
p279k	257736	18074	17827	493744
p286k	332726	18351	17835	648044
p295k	249747	18508	18521	478996
p330k	312666	18010	17468	547808
p388k	433331	25005	24065	856678
p418k	382633	30430	29809	688808
p469k	96408	635	403	169364
p483k	444664	33264	32610	922950
p500k	431439	30768	30840	843286
p533k	586819	33373	32610	1221432
p874k	629723	61977	70863	1050534
p951k	816072	91994	104714	1590490
p1522k	1104085	71392	68035	1747416
p2927k	2408328	101844	95159	3593886

Table 3: Circuit characteristics

list. The proposed algorithm is implemented as an accelerator to [14]. The accelerator executes the algorithm on all partitions that can be mapped to the associated data structures. Fan-out stems in partitions that cannot be mapped to the data structures are evaluated with the event-based algorithm on the conventional processor.

To identify the impact of the algorithmic optimizations in [14], we first compare that approach to the existing GPGPU-based fault simulation [25] in table 4. The third column lists the runtime results of the event-based simulator in [14] run on an AMD Athlon 64 X2 processor with a frequency of 2.4 GHz. The results reported by [25] are given in column four. Column five gives the achieved speed-up of [25] w.r.t. the event-based simulator. Except for two circuits, the fault simulation on the conventional processor is faster by an order of magnitude compared to the brute-force GPGPU method. [25] did not report results for larger circuits, mentioning data transfer times as a scalability issue.

Due to space limitations, we evaluate the implementation of the proposed algorithm only with respect to the event-based fault simulator. We only take into account the circuits where the algorithmic optimizations in [14] alone did not outperform [25]. In addition we evaluate much larger industrial circuits with up to 2.4 million gates.

For the NVIDIA GT200 GPGPU, the algorithm is implemented in the CUDA (Compute Unified Device Architecture) language extension [26]. The experiments are run on a GTX285 graphics card, on which the GPGPU comprises 30 cores with 8-way SIMD units, hardware-support for multi-threading and 16kByte local memory per core.

Circuit	No. gates	[14] [s]	[25] [s]	Speedup
s5378	1106	0.06	1.96	0.03
s9234.1	5944	1.69	2.04	0.83
s13207	1290	0.08	0.66	0.12
s15850	596	0.04	0.42	0.10
s35932	7152	0.03	5.43	0.01
s38417	10125	2.41	8.23	0.29
s38584	8571	0.54	7.88	0.07
b17	37446	49.35	19.03	2.59
b17_1	44544	51.63	17.87	2.89
b21	23100	11.94	46.58	0.26
b22	33569	19.81	60.49	0.33

Table 4: Runtimes of [14] vs. [25]

The severe memory restrictions with this architecture as well as scalability considerations for large circuits require that the partitions are stored in main memory. However, due to the extensive parallelism the multi-threading is able to hide the additional memory access latencies. In this case, the size of a partition is limited by the address range of the partition data structure. Since future graphics processors are expected to contain larger local memory such as Intel Larrabee with 256kByte per core [31], additional speed-ups can be expected as high main memory access latencies can be avoided completely.

Circuit	[14]	GPGPU	Speedup
b17	49.35	4.95	9.97
b17_1	51.63	3.89	13.27
b18	235.97	18.35	12.86
b19	498.44	30.72	16.23
b20	11.29	2.65	4.26
b21	11.94	2.65	4.51
b22	19.81	3.4	5.83
p100k	44.07	6.22	7.09
p141k	93.39	15.84	5.90
p239k	94.43	13.79	6.85
p259k	118.71	18.97	6.26
p267k	133.13	12.46	10.68
p269k	139.19	12.57	11.07
p279k	173.99	19.34	9.00
p286k	274.53	28.7	9.57
p295k	225.96	14.38	15.71
p330k	214.19	18.71	11.45
p388k	223.73	29.57	7.57
p418k	258.93	29.84	8.68
p469k	2212.71	2212.71	1.00
p483k	249.33	25.65	9.72
p500k	389.03	36.39	10.69
p533k	333.74	37.67	8.86
p874k	476.98	46.22	10.32
p951k	327.31	59.49	5.50
p1522k	912.07	76.46	11.93
p2927k	2340.14	369.79	6.33

Table 5: Runtimes of [14] vs. proposed algorithm on GPGPU

Table 5 lists the speed-up obtained with the proposed algorithm on the GPGPU. Compared to the approach in [25], the method presented here achieves speedups of 3.8x(b17), 4.6x(b17_1), 16.4x(b21) and 17.6x(b22).

An average speed-up of 8.93x is achieved in comparison to the event-based fault simulator. The circuit topology has a strong impact on the achievable speed-up. For example, p469k has very unusual topology with just 635 inputs but about 96 thousand gates. Almost none of the fan-out stems can be mapped to the data structures with the given parti-

tion size and thus are evaluated on the conventional processor without any speed-up.

The Cell processor cores have 16x larger local memories than those on the GTX285 and are thus able to store the entire data structure for substantial circuit partitions locally. First experiments on the Cell architecture showed that the resulting throughput per core is much higher than on the GTX285.

5. CONCLUSIONS

This work proposes an efficient mapping of algorithmic optimizations found in PPSFP fault simulators to many-core processors. The adaptation of data structures and algorithms allows to maximize the number of cores used in parallel. The implementation for a GPGPU achieves a speed-up of up to 17x compared to a recently proposed GPGPU-based fault simulator and up to 16x compared to the serial event-based PPSFP algorithm on a conventional processor.

Future many-core processors will feature more execution units and an increased amount of local memory due to continued technology scaling and three-dimensional integration of memory. The presented approach can favorably exploit this trend and will offer further speed-up.

Acknowledgment

This work has been supported by the German Research Foundation (DFG) in the Cluster of Excellence in Simulation Technology (EXC 310/1).

References

[1] J. Owens, D. Luebke *et al.*, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[2] J.-S. Park, M. Penner, and V. K. Prasanna, "Optimizing graph algorithms for improved cache performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 9, pp. 769–782, 2004.

[3] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. International Conference on High Performance Computing (HiPC)*, 2007, pp. 197–208.

[4] Y. Frishman and A. Tal, "Multi-level graph layout on the GPU," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 6, pp. 1310–1319, 2007.

[5] C. N. Vasconcelos and B. Rosenhahn, "Bipartite graph matching computation on GPU," in *Proc. Intl. Conference Energy Minimization Methods in Computer Vision and Pattern Recognition*, 2009, pp. 42–55.

[6] G. Flach, M. de Oliveira Johann *et al.*, "Cell placement on graphics processing units," in *Proc. 20th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2007, pp. 87–92.

[7] Y. Liu and J. Hu, "GPU-based parallelization for fast circuit optimization," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, 2009, pp. 943–946.

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

[9] J. Waicukauski, E. Eichelberger *et al.*, "Fault simulation for structured VLSI," *VLSI Systems Design*, vol. 6, no. 12, pp. 20–32, 1985.

[10] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing - an alternative to fault simulation," in *Proc. Design Automation Conference (DAC)*, 1983, pp. 214–220.

[11] K. Antreich and M. H. Schulz, "Accelerated fault simulation and fault grading in combinational circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 6, no. 5, pp. 704–712, 1987.

[12] D. Harel, R. Sheng, and J. Udell, "Efficient single fault propagation in combinational circuits," *Communications of the ACM*, vol. 29, no. 4, pp. 300–311, 1986.

[13] F. Maamari and J. Rajski, "The dynamic reduction of fault simulation," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 12, no. 1, pp. 137–148, 1993.

[14] H. K. Lee and D. S. Ha, "An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagation," in *Proc. IEEE International Test Conference*, 1991, pp. 946–955.

[15] B. Becker, R. Hahn, and R. Krieger, "Fast fault simulation in combinational circuits: an efficient data structure, dynamic dominators and refined check-up," in *Proc. Conf. on European design automation (EURO-DAC)*, 1992, pp. 436–441.

[16] N. Ishiura, M. Ito, and S. Yajima, "Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 9, no. 8, pp. 868–875, 1990.

[17] V. Narayanan and V. Pitchumani, "Fault simulation on massively parallel SIMD machines algorithms, implementations and results," *J. Electronic Testing*, vol. 3, no. 1, pp. 79–92, 1992.

[18] T. Nagumo, M. Nagai *et al.*, "VFSIM: Vectorized fault simulator using a reduction technique excluding temporarily unobservable faults," in *Proc. Conference on Design Automation (DAC)*, 1994, pp. 510–515.

[19] R. Daoud and F. Özgüner, "Highly vectorizable fault simulation on the Cray X-MP supercomputer," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 8, no. 12, pp. 1362–1365, 1989.

[20] P. Agrawal, V. D. Agrawal *et al.*, "Fault simulation in a pipelined multiprocessor system," in *Proc. International Test Conference (ITC)*, 1989, pp. 727–734.

[21] R. B. Mueller-Thuns, D. G. Saab *et al.*, "VLSI logic and fault simulation on general-purpose parallel computers," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 12, no. 3, pp. 446–460, 1993.

[22] S. Parkes, P. Banerjee, and J. H. Patel, "A parallel algorithm for fault simulation based on PROOFS," in *Proc. International Conference on Computer Design (ICCD)*, 1995, pp. 616–621.

[23] D. Krishnaswamy, E. M. Rudnick *et al.*, "SPITFIRE: scalable parallel algorithms for test set partitioned fault simulation," in *Proc. IEEE VLSI Test Symposium (VTS)*, 1997, pp. 274–281.

[24] M. B. Amin and B. Vinnakota, "Data parallel fault simulation," *IEEE Trans. VLSI Syst.*, vol. 7, no. 2, pp. 183–190, 1999.

[25] K. Gulati and S. P. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Proc. Design Automation Conference (DAC)*, 2008, pp. 822–827.

[26] E. Lindholm, J. Nickolls *et al.*, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[27] H. Wunderlich and S. Holst, "Generalized fault modeling for logic diagnosis," in *Models in Hardware Testing*. Springer, 2009, pp. 133–155.

[28] E. Ulrich and T. Baker, "The concurrent simulation of nearly identical digital networks," in *Proc. 10th Workshop on Design Automation*, 1973, pp. 145–150.

[29] W. Ke, S. Seth, and B. Bhattacharya, "A fast fault simulation algorithm for combinational circuits," in *IEEE Int'l Conf. on Computer-Aided Design (ICCAD)*, 1988, pp. 166–169.

[30] "IWLS 2005 Benchmarks," http://www.iwls.org/iwls2005/benchmarks.html.

[31] L. Seiler, D. Carmean *et al.*, "Larrabee: a many-core x86 architecture for visual computing," in *ACM SIGGRAPH*, 2008, pp. 1–15.