On the Use of GP-GPUs for Accelerating Compute-intensive EDA Applications

Valeria Bertacco,
Debapriya Chatterjee
EECS Department
University of Michigan, USA
{valeria|dchatt}@umich.edu

Nicola Bombieri,
Franco Fummi, Sara Vinco
Dip. Informatica
Università di Verona, Italy
{name.surname}@univr.it

A. M. Kaushik,
Hiren D. Patel
ECE Department
University of Waterloo, CA
{amkaushi|hdpatel}@uwaterloo.ca

Abstract—**General purpose graphics processing units (GP-GPUs) have recently been explored as a new computing paradigm for accelerating compute-intensive EDA applications. Such massively parallel architectures have been applied in accelerating the simulation of digital designs during several phases of their development – corresponding to different abstraction levels, specifically: (i) gate-level netlist descriptions, (ii) register-transfer level and (iii) transaction-level descriptions. This embedded tutorial presents a comprehensive analysis of the best results obtained by adopting GP-GPUs in all these EDA applications.**

I. Introduction

Simulation plays an important role in the validation of digital hardware systems. It is heavily used to evaluate functional correctness, and to perform early design and performance tradeoffs. This entails a vast number of simulation runs to evaluate many design's trade-offs and validate as many execution scenarios as possible throughout the development of the design. As a result, simulation is one of the most time, effort and resource-consuming activities of the entire design cycle. Correspondingly, the performance of simulation affects the time-to-market of many digital designs today.

On the other hand, the continued increase in design's functionality has resulted in simulation models that are larger and more complex than ever. This trend further burdens the performance of simulation, leading to much longer completion times for many simulation runs, affecting all stages of the development: from early-stage high-level model simulations, to the vast efforts dedicated to RTL and gate-level validation. Ultimately, this issue has been preventing design team from meeting today's stringent time-to-market constraints [1]. As a result, there is considerable interest in developing techniques that expedite the simulation of large and complex digital hardware system models.

Gate-level simulation takes place at late development stages, once the design has undergone the first few synthesis iterations [2], [3], [4]. Its objective is that to evaluate the functional and electrical correctness of the netlist description of the system. Often, the reference model use to validate results of a gate-level simulation is either a high-level design model (such as a C or SystemC model) or an RTL specification. The major issue in gate-level simulation, is that it tackles a fairly detail description of the design, thus, usually, the corresponding model is extremely large. Consequently, the completion of these simulation, when even feasible, require many hours or days. Early effort to leverage concurrent processing resources to address this problem include dividing the processing of individual events across multiple machines with fine granularity. This fine granularity would generate a high communication overhead and, depending on the solution, the issue of deadlock avoidance could require specialized event handling [5]. Parallel logic simulation algorithms were also proposed for distributed systems [6], [7] and multiprocessors [8] with some success.

A widely used simulation environment for early design space exploration of digital hardware systems is SystemC [9]. SystemC is an open-source library of C++ classes that allows modelling at the register-transfer level (RTL) and transaction-level (TL) abstractions. SystemC is commonly deployed for early design trade-off evaluation and validation of high-level models. Even though SystemC simulation operates at a higher abstraction level than traditional RTL, SystemC simulation is known to suffer from long simulation times [1]. This aspect has motivated the research community to develop techniques to accelerate SystemC simulations. These techniques include model transformations [10], distributed simulation [11], process splitting [12], and static scheduling of processes [13]. Furthermore, the SystemC reference simulation kernel implements a conventional discrete-event semantics in its single-threaded implementation. This architecture prevents the kernel from being easily portable to traditional high-performance multiprocessing platforms (such as SMP). Overcoming this challenge has been the topic of much research [14], [15], [16], [17], [18].

Recently, another form of concurrent architecture has become widely available: general purpose graphics processing units (GP-GPUs). GP-GPUs are massively parallel architectures that support data-level and thread-level parallelism. While they were originally designed for graphics and scientific computing, researchers have recently been exploring the use of GP-GPUs to accelerate digital design simulation at several levels, particularly those discuss above, namely gate-level descriptions [19], [20], [21], [22] and SystemC descriptions, both when used for RTL and transction-level models [23], [24], [25], [26], [27].

On the logic simulation front, the key challenge lies in the high amount of unstructured interconnections among the

This work has been partially supported by EU project FP7-ICT-2011-7-288166 (TOUCHMORE) and by NSF grant #1217764

978-3-9815370-0-0/DATE13/©2013 EDAA

basic design components. On one hand, GP-GPUs provide structured and uniform communication patterns among their basic computation units (threads) in order to deliver the high performance they are designed for. On the other hand, a gate-level netlist presents an unpredictable web of connections among many millions of logic gates. Early efforts to solve this problem include the work by Perinkulam, et al. [28]. However, their solution could not complete solve the problem discussed above, presenting overall a performance cost when compared to a sequential gate-level simulator.

On the SystemC simulation front, executing SystemC models on GP-GPUs is non-trivial because of the underlying architecture of GP-GPUs. For example, GP-GPUs do not provide standard thread suspension and resumption capabilities that are utilized in conventional multi-threaded programs. This raises the challenge of efficiently parallelizing the simulation of SystemC models on GP-GPUs.

In this paper, we present an overview of recent research efforts that use GP-GPUs to accelerate design simulation. These efforts focus on logic simulation [19], [20], [21], SystemC RTL simulation [23], [24], mixed-abstraction RTL and TL simulation [26], [27], and on evaluating different GP-GPU programming frameworks [29].

II. Background

This section presents the basic concepts of the paper, including HDL simulation semantics (Section II-A) and an overview of the GP-GPU programming model (Section II-B).

A. HDL simulation scheduling

HDLs use an event-based architecture, where a centralized scheduler controls the execution of processes based on events (*e.g.*, synchronizations, time notifications or signal value changes).

Figure 1 depicts the main steps of a typical HDL simulator kernel. The flow is iterated until no event is left to be processed, indicating the end of the simulation. A *simulation cycle* completes at the end of each iteration through the complete flow. Within each cycle, there is first an evaluation phase, during which all runnable processes are executed. Signals are updated at the end of the execution of each process. If a signal value change occurs, all processes sensitive to that signal are added to the runnable queue (this is called signal and event update phase). Finally, during the time update phase, the time of the next simulation cycle is determined by setting it to the earliest of (i) the time at which the simulation ends, (ii) the next time at which an event occurs, or (iii) the next time at which a process is scheduled to resume. If simulation time is not increased, the next simulation cycle will be a delta cycle. When no new event is fired, simulation ends.

It is important to note that the order of process execution within a delta cycle does not affect the output of simulation since the simulator presents the same system status to all those processes. Thus, processes that are activated within the same delta cycle may be executed in parallel, either by using multiple threads or by designing a distributed scheduler. However, HDL schedulers are strictly sequential, thus not taking advantage of parallel architectures and frameworks.

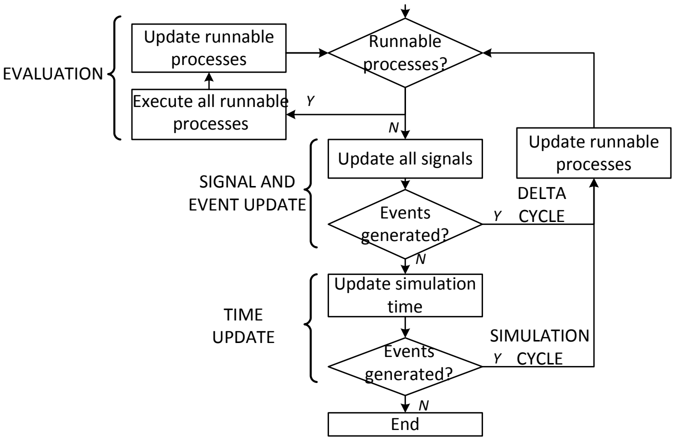


Fig. 1. Outline of the standard HDL scheduling semantics

B. General purpose GPU programming

GP-GPUs are increasingly being used as application accelerators as they offer low-cost, and high powered computing resources. CUDA has been introduced by NVIDIA in 2006 as the first framework for supporting the development of general purpose applications for GP-GPUs, and, since then, it has been the reference framework for GP-GPU programming [30]. CUDA is restricted to NVIDIA devices, thus in 2008 the Kronos group founded OpenCL [31], a standard alternative to CUDA that targets a wider range of architectures, ranging from CPUs to GP-GPUs.

Despite architectural differences between OpenCL and CUDA technologies, the fundamental idea of using GPUs for general purpose programming is the same: to utilize the multiprocessors on the GPUs for accelerating program execution. As a result, OpenCL and CUDA have common platform models, memory models, execution models and programming models.

The following of this section introduces GP-GPU programming and the main characteristics of GP-GPU architectures. Any time that keywords are different for CUDA and OpenCL, the CUDA version is used in the text and the OpenCL keyword is reported in brackets for completeness.

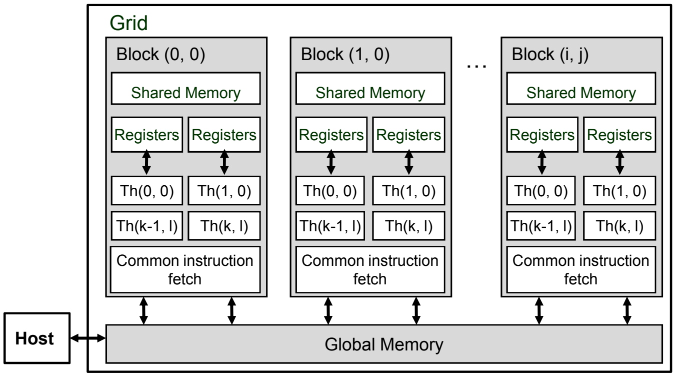


Fig. 2. The GP-GPU architecture

In GP-GPU computing, the GP-GPU is a co-processor capable of executing many threads (or work-items) in parallel, following the single instruction multiple data (SIMD) model of execution. A data parallel computation process, known as a kernel, can be offloaded to the GP-GPU for execution. The collection of threads represented by a kernel is divided into a grid of thread-blocks (or work-groups).

The GP-GPU architecture (Figure 2) consists of a number of multiprocessors within a single GP-GPU chip. Multiprocessors are responsible for the execution of the thread-blocks mapped to each of them. Each multiprocessor comprises multiple stream processors with common instruction fetching and support for a large number of concurrent threads. Thus, each multiprocessor executes several groups of threads at a time (known as a warp) in a time-multiplexed fashion, with frequent context-switches from one warp to another. Because of the shared fetch unit, execution path divergence between threads in a same warp is detrimental to performance as only one branch path can be executed at a time. Thus, if threads in a same multiprocessors must execute different code paths, the least penalizing solution is to map them to different warps.

Each multiprocessor has access to low latency scratchpad memory, divided between local registers (or private memory) and shared memory (or local memory). All multiprocessors also have access to a region of global memory, which has higher access latency.

The main difference between OpenCL and CUDA is that OpenCL targets a wider range of architectures, while CUDA is restricted to NVIDIA GP-GPUs. This implies that OpenCL is more flexible and it can not take into account specific properties of the underlying architecture, such as the availability of read-only memory. As a result, OpenCL requires environmental setup before launching kernel execution. When offloading code to the device, the host must define a context. Each context is made of a set of devices where execution occurs and by the kernel that must be executed. Furthermore, a context contains a reference to the program source code and to the memory objects that are visible from both the host and the devices. Compilation of OpenCL code is ended at runtime, once that the target device has been selected.

III. GP-GPU BASED SIMULATION

The following sections present three approaches that have been recently proposed for accelerating compute-intensive EDA applications at different abstraction levels. Section IV focuses on gate-level logic simulation of HDL designs. Section V presents the most recent results in accelerating such designs described in RTL subset of SystemC, while Section VI shows transaction level simulation on GP-GPUs. Section VII compares the performance of the proposed SystemC simulation techniques on CUDA vs. OpenCL platforms, with the goal of investigating advantages and drawbacks that the two thread management libraries offer to concurrent SystemC simulation.

For sake of clarity, all sections have the same structure: overview of the key contributions to the state of the art, detailed overview of the approach presented, and discussion of it's effectiveness. A common example is used to illustrate the application of the techniques to a case study. The example is represented in Figure 3. The nodes and edges represent different primitives and actions, depending on the abstraction level and the solution under discussion. The goal is to support the reader with a simple, common example that is developed and analyzed throughout the paper.

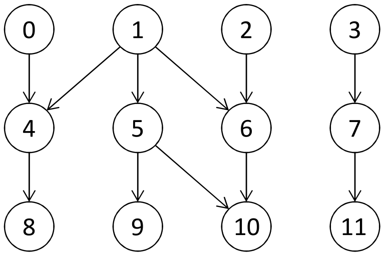


Fig. 3. Compute flow diagram to illustrate the application of all proposed methodologies. Each nodes may represent a logic gate, or a SystemC process depending on the abstraction level. Correspondingly, edges may represent wires or value dependencies through variables strictly depending on the adopted methodology.

IV. LOGIC SIMULATION

Logic simulation is the primary workhorse for functional validation of digital designs. Unfortunately, the performance of logic simulation is far from adequate for modern complex digital designs. Simulation of gate-level netlist descriptions is especially slow, since the netlist often comprises tens of millions of structural logic elements, such as logic gates and flip-flops. Structural gate-level simulation is an inherently parallel problem as several gates can be simulated simultaneously while abiding internal computation flow ordering. However, the commercial simulators available today operate primarily on single threaded processors, thus they do not exploit this for concurrent computation potential. In contrast, GP-GPUs are an ideal candidate for such massive parallelism.

A. Key contribution

We investigate how the parallelism available in the problem structure can be mapped to that of the execution hardware of GP-GPUs. While the parallelism of netlists matches well with the parallel computational power available in GPUs, there are a number of problems that must be addressed to enable GPU-based logic simulation. First, a netlist must be partitioned into portions that can be mapped and simulated concurrently and efficiently on a GPU. The partitioning must be aware of the GP-GPU architecture and its memory model to exploit a GP-GPU's memory locality in an optimal manner. Our GPU-based simulation approaches leverage novel solutions to the problems of netlist partitioning and mapping, enabling large designs to be simulated on GPU hardware. Moreover, the GP-GPU execution model is most efficient when operating on regular data structures; to this end, we use novel algorithmic solutions to overcome the challenges posed by a netlist's structural irregularities as well.

Logic simulators come in two flavors: oblivious and event-driven. In an oblivious simulator all gates in the design are computed at every cycle. While the program's control flow for this approach has minimal overhead, computing the output values of every gate at every cycle can be time-consuming and, most importantly, unnecessary for all those gates whose inputs have not changed from the previous cycle. Event-driven simulation, on the other hand, takes advantage of precisely this fact: the output of a gate will not change unless its inputs have changed. Keeping GP-GPU nuances in mind, our solution is capable of leveraging this hardware platform for both oblivious and event-driven simulation to deliver high performance results. Indeed, both flavors of our simulators are capable of achieving over an order-of-magnitude speed-up over traditional simulation solutions – even when compared against commercial state-of-the-art solutions.

B. Approach

A logic simulator takes the netlist as input, then converts it to internal data structures: loops through the sequential storage elements in the design are cut open (see Figure 4(a)), creating two sets of values for all latches and flip-flops (present state and next state), which are maintained within the data structures of the simulator software. At this point the combinational portion of the logic is a directed acyclic graph (in absence of combinational loops) whose vertices are logic gates and edges are the wires connecting the input and output of the logic gates (see Figure 4(b)). On the input end of this graph are primary inputs and present state values, while the output side produces primary outputs and next state values. This graph is then levelized (*i.e.*, the vertices are topologically sorted) according to the dependencies implied by the gate's input-ouput connections. The simulation process can now begin: the simulator generates input values and then computes the outputs of the internal logic gates, one level at a time, until all output values are produced. In subsequent simulation cycles, the next state values are looped-back and used as the next cycle's present state values.

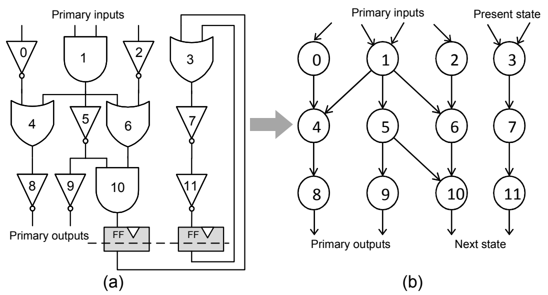


Fig. 4. Netlist to data structure mapping for gate-level simulation.

We developed both flavors of simulators (oblivious and event-driven) targeting the GP-GPU platform. The high level flow for both of these variants are shown in Figure 5. Both solutions attempt to expose the concurrency inherent in the netlist to the parallelism available in the hardware, and both operate as a compiled-code simulator, where first the combinational portion of the netlist is extracted, then levelized, and finally partitioned to produce the necessary internal data structures, which are then offloaded to the GP-GPU for the simulation process. We noted that in the CUDA architecture, there are two levels of parallelism, i) Thread-block level parallelism: separate thread blocks execute in separate multiprocessors which can only communicate throgh slow device memory and hence should minimize interactions, and ii) Thread-level parallelism: where several threads belonging to the same thread block execute in parallel while communicating through a small amount of low latency shared memory and are capable of fast barrier-type synchronization. Our primary objective is to morph the problem of gate-level simulation to leverage these two levels of parallelism.

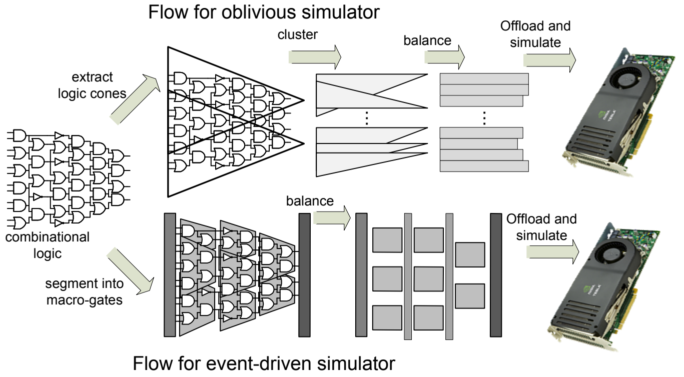


Fig. 5. Simulation flow for oblivious gate-level simulation (top) and for event-driven simulation (bottom).

1) Oblivious simulation: Parallelism is inherent in the structure of a levelized netlist, since all gates present in a single level of a levelized netlist can be simulated in parallel. However, such straightforward approach would require a large amount of communication among all threads after the completion of each level, a requirement that is not viable beyond individual thread blocks, because of the high latency of inter-block communication. In contrast, if we consider the fan-in logic cones of output signals in separation, we can simulate each of them independently, provided that we duplicate the shared logic components – hopefully just a small fraction of gates. This partitioning is a good fit for thread-block parallelism, since these cones can be independently simulated by different multiprocessors without requiring any communication between them. However, to avoid excessive duplication, it is beneficial to cluster logic cones that share a large amount of logic: to this end we developed an algorithm that collects several logic cones and combines them into a single cluster.

After partitioning the netlist into clusters of logic cones, we can delegate different clusters to different thread-blocks for simulation. The logic gates within each cluster can then be levelized, leading to several gates per level: an excellent fit for exploiting thread level parallelism, as parallel threads belonging to the same thread block can simulate the gates present in a same level concurrently, then synchronize through fast barrier synchronization primitives and move on to the next level of gates. It is beneficial to present a regular data structure to each thread block for the simulation process; however, clusters naturally tend to have more gates at the lower levels and fewer gates on the higher levels. As a result, within each thread block, more threads are needed at first, with a number of them becoming idle while moving towards the higher levels. To circumvent this issue we developed a balancing step (see Section IV-B3) to be performed after clustering, its goal is to generate a more regular structure to balance the use of threads within the block.

2) Event-driven simulation: Traditional event-driven simulators rely on a centralized event queue to schedule gates for simulation. In the context of the CUDA architecture, such a centralized queue would present a bottleneck as it would incur high latency delays – most probably higher than the computation of the gates themselves. However, a dynamic scheduling approach can still be viable if it operates at a much coarser granularity than individual gates and, as a result, invoked much more infrequently. As a result, we assign bundles of gates from the circuit's netlist to individual thread blocks. Each of these bundles are simulated by their corresponding thread block in an oblivious fashion, similar to the clusters of the oblivious approach discussed in the previous subsection and thus this solution exploits thread-level paralellism. At coarser granularities, we use an event-driven approach: we monitor the input values of the bundle, and schedule it for simulation only if one or more of its inputs have changed. We refer to these bundles of gates as 'macro-gates'. A high level schematic of this approach is presented in Figure 6, showing a pool of macro-gates for a netlist and a simulation cycle requiring to schedule only three of them for computation. To implement this design we must develop a segmentation algorithm to create macro-gates of appropriate size, striking a trade-off between performance and memory locality requirements. Note that macrogates are subjected to the balancing step as well (see Section IV-B3).

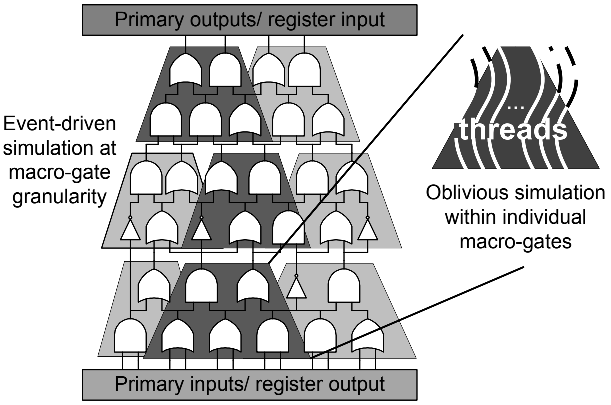

Primary outputs/ register input
Event-driven simulation at macro-gate granularity
...
threads
Oblivious simulation within individual macro-gates
Primary inputs/ register output

Fig. 6. The hybrid event-driven simulator is event-driven at the granularity of macro-gates, while the macro-gates themselves are simulated in an oblivious fashion. The macro-gates in a darker shade are the only ones scheduled for simulation during a possible simulation cycle.

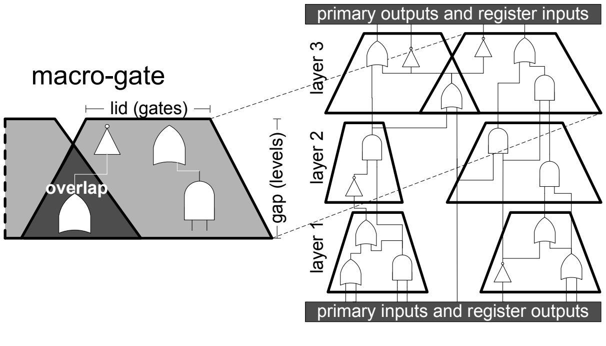

macro-gate
lid (gates)
overlap
gap (levels)
primary outputs and register inputs
layer 3
layer 2
layer 1
primary inputs and register outputs

Fig. 7. Macro-gate segmentation. The levelized netlist is partitioned into layers, each encompassing a fixed number of levels (gap). Macro-gates are then carved out by extracting the transitive fanin from a set of nets (lid) at the output of a layer, back to the layer's input. If an overlap occurs, the gates involved are replicated over all associated macro-gates.

Macro-gate segmentation is governed by three important factors: (i) since the objective of forming macro-gates is to perform event-driven simulation at a coarse granularity (compared to individual gates), the time required to simulate a given macro-gate should be substantially larger than the overhead to decide which macro-gates to activate. (ii) The multiprocessors in the GP-GPU can only communicate through high latency device memory, and thus, for best performance, there should be no communication among them. This can be ensured if the tasks executing on distinct multiprocessors are independent of each other. To this end, macro-gates that are simulated concurrently must be independent of each other (that is, they cannot share any logic gate). To achieve this goal, we duplicate small portions of logic that occasionally create overlap among macro-gates, eliminating the need of communication. (iii) Finally, we want to avoid cyclic dependencies between macro-gates, so that we can simulate each macro-gate at most once per cycle. To this end we levelize the netlist at the granularity of macro-gates as well. Segmentation begins by partitioning the netlist into *layers*: each layer encompasses a fixed number of the netlist's levels, as shown in Figure 7. Macro-gates are then defined by selecting a set of nets at the top boundary of a layer, and including their cone of influence back to the input nets of the layer. The number of nets used to generate each macro-gate is a parameter called *lid*, its value is selected so that the number of logic gates in each macro-gate is approximately the same. The number of levels within each layer is called *gap* and corresponds to the height of the macro-gate.

3) Balancing step: The cluster/macro-gate balancing algorithm exploits the slack available in a segment of a levelized netlist and reshapes it to have approximately the same number of logic gates in each level. The algorithm processes all the gates in a bottom-up fashion, filling every slot in the cluster/macro-gate with logic gates, with the goal of assigning each gate to the lowest level possible, while maintaining the restriction on maximum width. Note that this might also lead to an increase in the height of a particular cluster/macro-gate, and thus an increase in its simulation latency. Hence, there is an inherent trade-off between execution latency and thread utilization.

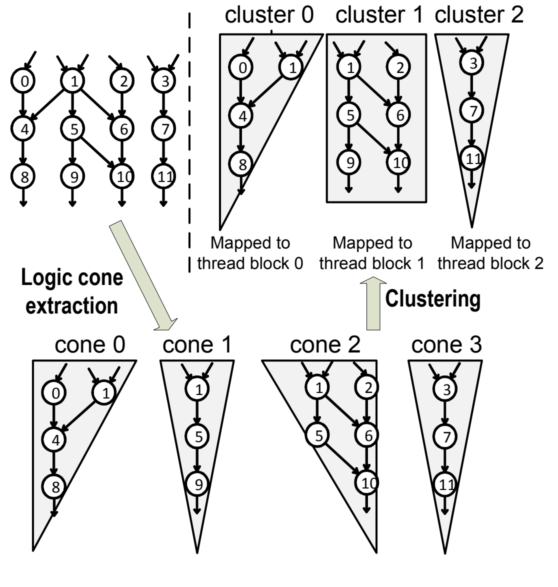


Fig. 8. Oblivious simulation example. First the levelized netlist is partitioned into logic cones, which are then clustered. Each cluster is delegated to an exclusive thread block for simulation.

C. Example

In this section we illustrate our methodologies on the example of Figure 4, where, for our solution, each node corresponds to a logic gate and each edge to a wire. Figure 8 illustrates the transformations applied to the netlist in our oblivious simulation solution. First the netlist is levelized and partitioned into distinct cones of logic. Then, cones are clustered so that each cluster is mapped to a single thread block. To minimize logic gate duplication, we devised an ad-hoc clustering algorithm: the largest cone is mapped to a new cluster, then we add cones to it, one by one, starting from the cone that shares the most logic with the cones already in the cluster. The process ends when we exhaust the resources for the cluster (shared memory allocated per thread block) or all cones are mapped. The next step entails balancing each cluster individually and mapping them to distinct thread blocks for simulation, as discussed in Section IV-C1.

Our event-driven simulator uses a different set of transformations. The netlist is first segmented into macro-gates as described in Section IV-B2. The outcome of this step is shown in Figure 9, with the parameters of lid and gap set to 2. We monitor the nets crossing macro-gate boundaries during simulation, since we must schedule a macro-gate for simulation only if a value change occurs on any of its monitored input nets. Scheduled macro-gates are then simulated in an oblivious fashion by distinct thread blocks.

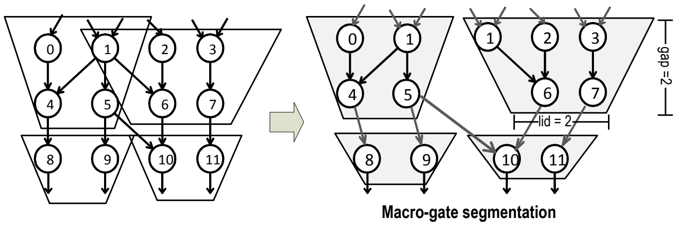


Fig. 9. Event-driven simulation example. The levelized netlist is partitioned into macro-gates, and then simulated in event-driven fashion at macro-gate granularity. The gray edges correspond to the nets to be monitored during simulation.

1) Oblivious simulation for clusters/macro-gates: A balanced macro-gate/cluster is analogous to a regular matrix-like structure, where the logic gates at each level correspond to rows of a matrix. The k-th thread in a thread block is responsible for the simulation of all the logic gates in the k-th column. The synchronization barriers at the completion of each level enforces read-after-write ordering in logic gate computations. The execution of a cluster is shown as an example in Figure 10. To leverage the benefits of the fast shared memory available within each multiprocessor, we store there the most frequently accessed internal wire values during simulation, in contrast, netlist topolgy and gate-type information is stored in the higher access latency device memory. Since adjacent threads in a thread-block access adjacent logic gates in each level, the fetch operations that retrieve topology information for each gate can be coalesced, resulting in better performance.

D. Discussion

Our oblivious simulation solution is presented in detail in [20], which delivers 14.4x speedup on average over a state-of-the-art event-driven commercial simulator. We note that the oblivious approach suffers from an important limitation, particularly for large designs: each cone of logic can only be as large as the amount of storage available in shared memory. In contrast, the event-driven solution is much more flexible, in that it is always possible to design macro-gates of sizes that can fit within the resources available in one thread block. As a result, our event-driven simulator can tackle netlist designs of any size. Our event-driven approach originally appeared in [19], [21]. It is capable of delivering a 13x speedup on average over the same commercial simulator.

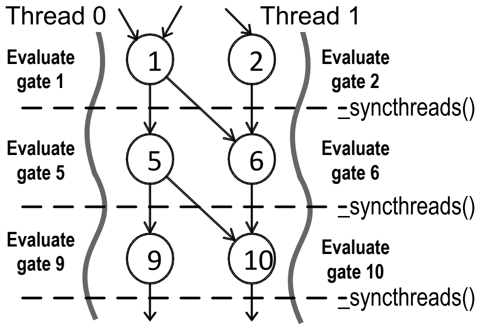


Fig. 10. Oblivious simulation of each cluster/macro-gate by an individual thread-block. In this example the thread-block contains two threads that are simulating cluster 1 of Figure 8.

Testbenches are an essential component of a logic simulator. Since the simulator is cycle-based, the task of the testbench is to read the outputs computed at the end of each cycle and provide suitable inputs for the next cycle. In our solution, testbenches are implemented as separate GP-GPU kernels: during simulation, the kernels for simulation proper and those for the testbench alternate execution on the GP-GPU device. The testbench has best performance if all associated data is stored in the memory on the device, eliminating communication with the host. At the completion of each simulation cycle, the outputs produced by the netlist are read from device memory, suitable inputs are computed and written back to be consumed during the following simulation cycle.

V. RTL SIMULATION

A. Key Contribution

Mapping of RTL designs to GP-GPU architectures is a non trivial task, since it requires scheduling support and the ability to preserve correct simulation outputs, even when the processes are executed in parallel.

SCGPSim is one of the first works targeting SystemC RTL simulation on GP-GPUs [23]. SCGPSim applies dynamic scheduling to the GP-GPU framework, by executing individual threads in parallel only when the testbench structure allows it and by synchronizing after each simulation step. This approach showed to be suitable for SystemC designs that exhibit a high degree of data parallelism. Any dependencies between processes increases the number of synchronizations necessary to mimic the original SystemC simulation kernel execution, which results in lost opportunity for performance improvements. Furthermore, SCGPSim maps processes to a thread block, and each thread block gets scheduled onto a single multi-processor. Thus, any branch divergence between the threads sequentializes the execution of all threads.

SAGA was proposed to overcome the limitations of SCG-PSim [24]. SAGA proposes a static scheduling approach that avoids frequent synchronization steps by partitioning the starting SystemC processes into independent sets that execute on different multiprocessors.

B. Approach

SAGA builds a static scheduling of RTL processes by analyzing the read-write dependencies between the processes to generate a dependency graph. The graph is then topologically sorted and partitioned into independent dataflows, i.e., portions of the process graph that can be executed independently (middle of Figure 11). Such dataflows are then mapped to distinct multiprocessors for concurrent execution. When necessary, some portions of the process graph may be replicated to attain independence among dataflows (as happened to processes $P1$ and $P5$ in Figure 11).

The dataflows built in the previous step are process dependency trees, that must be executed level-by-level to respect the internal dependency constraints. Thus, for each dataflow obtained in the previous step, a total serial order of processes must be built, with the goal of satisfying the level-to-level dependencies. Processes in each dataflow are serialized, starting from the lower levels up to the root processes of the dependency graph (processes at the same level can be executed in any sequential order). Such sequential order eliminates the need of frequent synchronization after each level. An example timeline is shown at the bottom of Figure 11.

GP-GPU execution is managed by a cycle that iteratively invokes two kernel functions. A simulation kernel manages dataflow execution, and it is generated by listing all the dataflows and predicating each by a thread-block ID condition, so that only a specific thread-block is responsible for executing a certain dataflow. The simulation kernel alternates execution with a value-update kernel, responsible for transferring next-state values into the corresponding present-state values and performing testbench actions.

Since device memory accesses are particularly slow, only variables written by synchronous processes are allocated in global memory, while all other variables can be declared as local variables mapped to registers.

C. Example

The graph in Figure 3 is reported on top of Figure 11, to highlight that nodes represent processes and edges their read-write dependencies. The graph is partitioned into four separate dataflows (middle of Figure 11). Dataflow 3 is completely independent from the others. On the other hand, the dependency trees of $P8$, $P9$ and $P10$ have some intersections, *i.e.* processes $P1$ and $P5$ are included in all of them. As such, without replication, the methodology may only identify one dataflow, including the dependency trees of $P8$, $P9$ and $P10$. This would reduce the level of parallelism. Thus, processes $P1$ and $P5$ are replicated, thus identifying three separate dataflows: dataflow 0, 1 and 2 (replicated processes are dashed in the figure). Finally, bottom of Figure 11 shows how processes belonging to each dataflow are sequentialized and mapped to distinct multi-processors to achieve parallel execution. The resulting simulation kernel code is outlined in Figure 12.

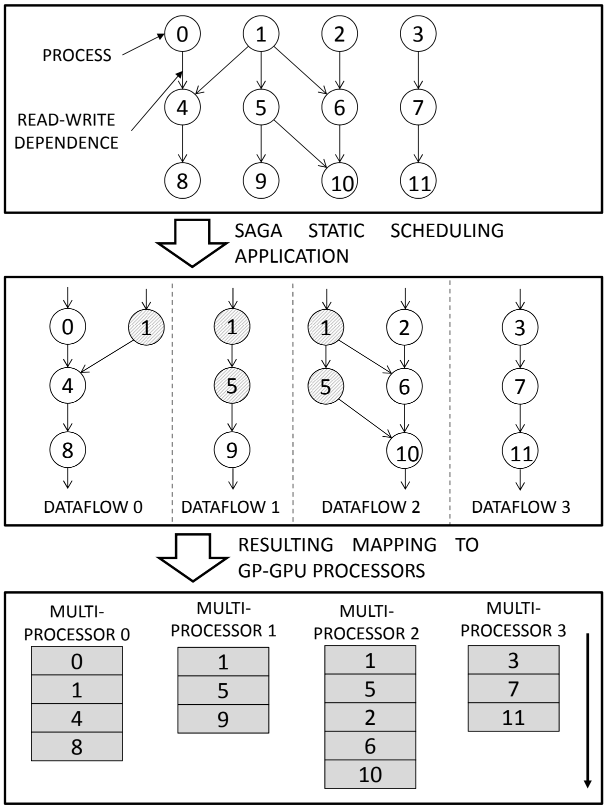


Fig. 11. Application of SAGA to the dependency graph in Figure 3. Processes are partitioned into independent dataflows, mapped to distinct multiprocessors.

```
__kernel void $simulate$(__global datastruct* $data$) {
// threads of a same block are instances of the same parallel item
**if** $block\_id == 0$ **then**
    execute_dataflow_0($data$);
**else if** $block\_id == 1$ **then**
    execute_dataflow_1($data$);
**else if** $block\_id == 2$ **then**
    execute_dataflow_2($data$);
**else**
    execute_dataflow_3($data$);
**end if**
}
```

Fig. 12. Kernel code generated for GP-GPU execution of the example in Figure 11

D. Discussion

SAGA advances the efforts on using GP-GPUs for simulating SystemC RTL models (achieving a maximum speed up of 16 times) and it may be easily extended to support models written in any RTL HDL. SAGA keeps branch divergence to the minimum by mapping independent process data-flows distinct thread blocks and by replicating paths of the data-flow that are shared amongst different data-flows, in order to eliminate synchronization between the different thread blocks. This replication of processes allows then to enhance parallelism or to extract parallelism from a model that otherwise may not exhibit parallelism. The disadvantage is that replication may increase the code size, which may then impact the simulation performance.

VI. Transaction-level simulation

A. Key Contribution

Sinha et al. [26] presented an approach with three main contributions. The first contribution is that their approach distributed processes of the same SystemC model across both the GP-GPU and the multicore CPUs for faster simulation. Second, they allowed parallel simulation of both processes deployed onto the CPUs and the GP-GPUs by extending SystemC's reference implementation to support synchronous parallelism. Third, Sinha et al [26] observed that SystemC models are often mixed-abstraction models where subsets of the processes may be at RTL whereas others may be at transaction-level (TL). Hence, they provided a method to allow the transferring of events between processes on the GP-GPU and CPU through a wrapper-based method. Notice that prior efforts on using GP-GPUs to accelerate SystemC simulations focused solely on either gate-level or RT-level SystemC models [20], [19], [21], [24], [27]. In addition, these efforts deployed the entire SystemC model onto the GP-GPU by translating the model, and scheduling their execution such that it honours SystemC's discrete-event semantics.

B. Approach

Sinha et al [26] accepted SystemC models specified at the RTL and TL abstract levels. These models underwent partitioning where the user identified processes suitable for GP-GPU execution, and those that are suited for CPU execution. The authors characterized processes suitable for GP-GPU execution as those that were compute-heavy and those that could exploit data-level parallelism. Processes that were suitable for CPU execution were those that had limited data-level parallelism, and those that performed control-dominated operations. The next step involved mapping the GPU-suitable processes into thread blocks. Depending on the amount of data-level parallelism, the processes were grouped into the same thread blocks or different thread blocks. Processes exhibiting high degree of data-level parallelism were grouped into the same thread block. The user performed this grouping with the goal of reducing the thread divergence suffered by threads within a thread block.

After the mapping and grouping, every GPU-suitable process was converted to its corresponding GP-GPU kernel. However, recall that GP-GPU kernels execute from start to completion. This is different than the manner in which SystemC processes execute. In particular, SystemC processes can suspend during execution using wait(), and resume from that point of suspension when there is a notify() on the suspending event. This was noted by Nanjundappa et al. [23], and they proposed an automata-based approach for translating the processes into GP-GPU kernels. However, in their work every suspension point amounted to a clock cycle delay because all models were at the RTL. In the work by Sinha et al [26], models can be at the TL, and a subset of the processes could be executing on the CPU. As a result, a mechanism to suspend on events, and resume from notifications of events, and inform all processes was developed. The proposed method used a SystemC wrapper process for each GP-GPU kernel that facilitated the communication between the GP-GPU kernel for that process, and for executing the appropriate suspension/resumption calls. This was essential because the main SystemC scheduler resided on the CPU. The key insight in translating SystemC processes to GP-GPU kernels was that every suspension point denoted by a wait() exited the GP-GPU kernel back to its calling SystemC wrapper process. The wrapper process retrieved information about the suspension call such as its type and duration, and invoked it from within the SystemC wrapper process; thereby, inserting the event in the SystemC scheduler. Notice that this required storing the state of the GP-GPU kernel such as any intermediate computation, a state identifier, and the information about the suspension/resumption calls in a data structure such that the SystemC wrapper process could invoke the correct calls. Furthermore, resuming from a suspension point amounted to starting the GP-GPU kernel at the state after the suspension point. This was accomplished by providing the intermediate state to the GP-GPU kernel, and the state identifier, which identified the point in the GP-GPU kernel to resume execution. The additional data structure to enable this interaction used the GP-GPU global memory, and it stored the state identifier, the suspension/resumption call information, and intermediate results.

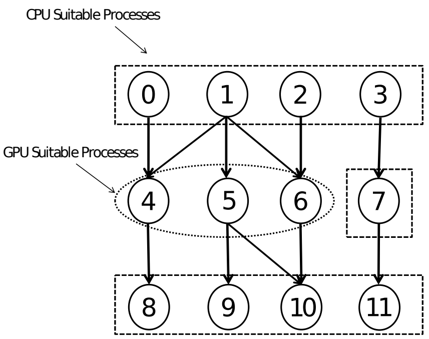
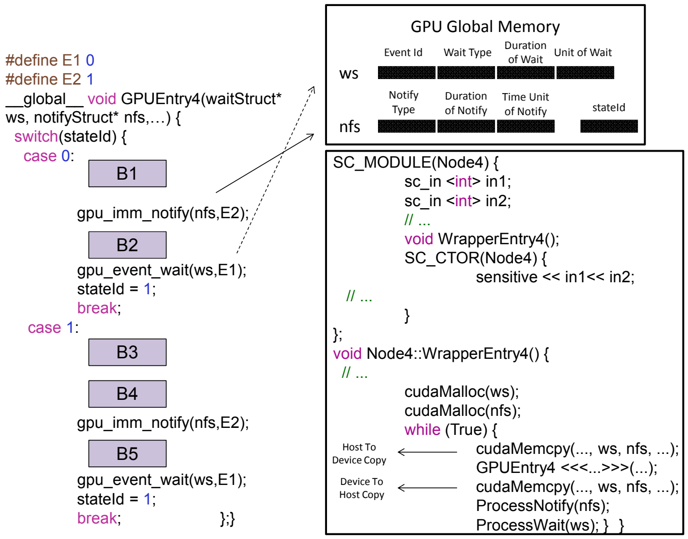


Fig. 13. Example mapping of SystemC processes to GP-GPU and CPU

Sinha et al [26] also extended SystemC's kernel to support synchronous parallelism as previously presented by Schumacher et al. [15]. The synchronous parallel simulation kernel exploits the fact that multiple SystemC processes may become ready-to-run, which the reference implementation executes sequentially. Instead, the synchronous parallel SystemC kernel executes these in parallel.

C. Example

Consider the illustrative example in Figure 13. Figure 13 shows processes enclosed in boxes as those that are suitable for CPU execution, and those in ellipses for GP-GPU execution. Assuming that these processes have distinct GP-GPU kernels, then there will be a unique SystemC wrapper process for each of these kernels. Figure 14 explains the interaction between the wrapper process and kernel for process 4, which is dependent on the inputs from processes 0 and 1. Given that process 4



```
#define E1 0
#define E2 1
__global__ void GPUEntry4(waitStruct*
ws, notifyStruct* nfs,...) {
 switch(stateId) {
  case 0:
          B1
        gpu_imm_notify(nfs,E2);
          B2
        gpu_event_wait(ws,E1);
        stateId = 1;
        break;
  case 1:
          B3
          B4
        gpu_imm_notify(nfs,E2);
          B5
        gpu_event_wait(ws,E1);
        stateId = 1;
        break;                };}
```

```
SC_MODULE(Node4) {
        sc_in <int> in1;
        sc_in <int> in2;
        // ...
        void WrapperEntry4();
        SC_CTOR(Node4) {
                sensitive << in1<< in2;
  // ...
        }
};
void Node4::WrapperEntry4() {
  // ...
        cudaMalloc(ws);
        cudaMalloc(nfs);
        while (True) {
                cudaMemcpy(..., ws, nfs, ...);
                GPUEntry4 <<<...>>>(...);
                cudaMemcpy(..., ws, nfs, ...);
                ProcessNotify(nfs);
                ProcessWait(ws); }  }
```

Fig. 14. Interaction between SystemC wrapper process and GP-GPU kernel

gets scheduled for execution when the inputs from 0 and 1 become available (inputs included in 4's sensitivity list), then the SystemC wrapper process has the same sensitivity of the original process. The wrapper process begins execution by first allocating space for the wait and notify data structures and other data structures required for computation on the device. This is done only once as the size of the data structures are fixed. The wrapper then copies information to be stored in the wait structure denoted by wsand notify structure nfs. This call refreshes the wsand nfsdata structures as on each invocation of the kernel, the kernel process modifies the contents of the data structures.

The entry function gpuEntry4 is process 4's GP-GPU kernel implements an FSM representing the behaviour of the original SystemC process. For this example, it contains code blocks $B1$ to $B5$ with a GP-GPU version of wait() and notify() interspersed between them. Notice that these GP-GPU versions of wait() and notify() are provided by Sinha et al [26] as an additional library. When the kernel is called for the first time, it executes $B1$, and encounters a gpu_imm_notify(), which is an immediate notification. The GP-GPU kernel saves the type of notify in the data structure nfs, and proceeds to execute $B2$. After executing $B2$ it encounters a call to gpu_event_wait(), which is a wait() on an event identified using $E1$. The kernel saves the event identifier of $E1$, and the type of wait in the data structure ws, updates a variable called state-id to and exits the kernel. The SystemC wrapper process copies the data structures nfsand wsfrom the device global memory, analyzes them, and issues corresponding notification and suspension calls via the ProcessNotify()andd ProcessWait()function call. Notice that this is a method to direct notifications and suspensions from the GP-GPU kernel to the main scheduler that resides on the CPU. When re-invoking the GP-GPU kernel, the state-id is passed, which provides the program point for the next state in the FSM. In our example, the kernel on re-invocation begins to execute code blocks $B3$.

D. Discussion

Sinha et al [26] presented a first effort on co-simulating mixed abstraction SystemC models across multi-core CPUs and the GP-GPUs. This approach utilized a SystemC wrapper process that served as an intermediary for communication between processes deployed on the GP-GPUs and the remaining processes on the CPUs. A key component of their approach used a synchronous parallel SystemC kernel. This allowed them to support multi-GPU platforms since it is possible to make parallel GP-GPU kernel invocations from multiple SystemC wrapper processes. The authors recognized that by requiring the simulation to make multiple invocations to the GP-GPU kernels, the number of memory transfers were significantly higher. Furthermore, memory transfers could inhibit performance improvement.

VII. Extension to OpenCL

CUDA has been introduced by NVIDIA in 2006 as the first framework for supporting the development of general purpose applications for GP-GPUs, and, since then, it has been the reference framework for GP-GPU programming. As such, all the works cited in this paper adopt CUDA as target language and architecture. CUDA is restricted to NVIDIA devices, thus resulting highly optimized for performance. Unfortunately, it results also in lacking portability across different vendors. For this reason, in 2008 the Kronos group founded OpenCL, a standard alternative to CUDA. OpenCL targets a wider range of architectures, ranging from CPUs to GP-GPUs. Thus, OpenCL is more portable but its implementation is not as optimized as the CUDA competitor.

[29] lately proposed a comparison between CUDA and OpenCL targeting the simulation of EDA descriptions. In detail, the work compares the performance of the adoption of SAGA to simulate SystemC designs by using either the CUDA or the OpenCL framework. As outlined in Section II-B, OpenCL and CUDA have common platform models, memory models, execution models and programming models. As a result, implementing a CUDA code for OpenCL requires very few modifications, related to the different APIs and to few thread organization points (i.e., the warp size is different for the frameworks). As such, this step can be automated or easily performed manually.

The main difference between the two frameworks is that OpenCL targets a wider range of architectures, while CUDA is restricted to NVIDIA GP-GPU. This implies that OpenCL is more flexible and it can not take into account specific properties of the underlying architecture, such as the availability of read-only memory. This consideration impacts execution performance. First of all, OpenCL requires environmental setup before launching kernel execution. This process includes selecting the target device, determining its characteristics and compiling the kernel at runtime. As a result, OpenCL has a heavy initialization cost, unnecessary in CUDA, where the features of the underlying architecture are statically determined. Furthermore, memory management and compilers are less optimized, since no assumption on the underlying architecture can be made at compilation time.

As a result, [29] showed that the execution of the same simulation code on OpenCL frameworks results in average 2.5 times slower than the corresponding CUDA implementation. This performance may improve with the delivery of more mature compilers for OpenCL and it is compensated by the high level of portability of the generated code.

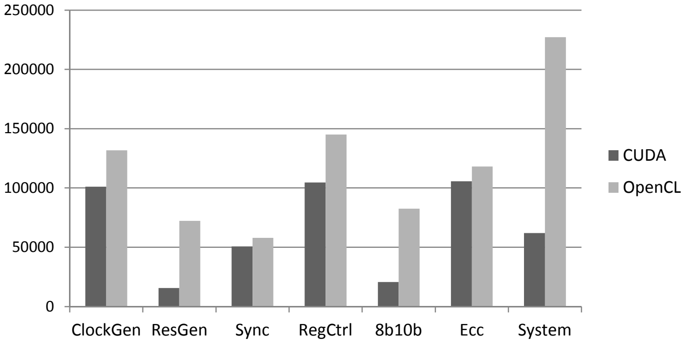


Fig. 15. Comparison of simulation of the same designs on CUDA and OpenCL frameworks, as presented in [29].

VIII. Conclusions

The paper proposed a comprehensive analysis of state of the art approaches for the exploitation of GP-GPUs in the context of computation intensive EDA applications. All the approaches have been outlined with a common example and by highlighting the contribution of GP-GPU architectures in increasing effectiveness and performance. Both CUDA and OpenCL have been investigated, to propose a balanced trade off between performance and portability.

References

[1] W. Ecker, V. Esen, L. Schonberg, T. Steininger, M. Velten, and M. Hull, "Impact of description language, abstraction layer, and value representation on simulation performance," in *Proc. of ACM/IEEE DATE*, 2007, pp. 767–772.

[2] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: a compiled simulator for MOS circuits," in *Proc. ACM/IEEE DAC*, 1987, pp. 9–16.

[3] Z. Barzilai, J. Carter, B. Rosen, and J. Rutledge, "HSS–a high-speed simulator," *IEEE Trans. on CAD*, vol. 6, no. 4, pp. 601–617, 1987.

[4] D. Lewis, "A hierarchical compiled code event-driven logic simulator," *IEEE Trans. on CAD*, vol. 10, no. 6, pp. 726–737, 1991.

[5] W. Baker, A. Mahmood, and B. Carlson, "Parallel event-driven logic simulation algorithms: tutorial and comparative evaluation," *IEEE Journal on Circuits, Devices and Systems*, vol. 143, no. 4, pp. 177–185, 1996.

[6] Y. Matsumoto and K. Taki, "Parallel logic simulation on a distributed memory machine," in *Proc. IEEE EDAC*, 1992, pp. 76–80.

[7] N. Manjikian and W. Loucks, "High performance parallel logic simulations on a network of workstations," in *Proc. of ACM PADS*, 1993, pp. 76–84.

[8] H. K. Kim and S. M. Chung, "Parallel logic simulation using time warp on shared-memory multiprocessors," in *Proc. IEEE International Symposium on Parallel Processing*, 1994, pp. 942–948.

[9] *SystemC 2.3.0*, Accellera Systems Initiative, 2012, http://www.systemc.org.

[10] S. A. Sharad and S. K. Shukla, *Optimizing system models for simulation efficiency*. Norwell, MA, USA: Kluwer Academic Publishers, 2004, pp. 317–330.

[11] D. R. Cox, "RITSim: distributed SystemC simulation," Ph.D. dissertation, Rochester Institute of Technology, 2005. [Online]. Available: http://hdl.handle.net/1850/1014

[12] Y. N. Naguib and R. S. Guindi, "Speeding up SystemC simulation through process splitting," in *Proc. of ACM/IEEE DATE*, 2007, pp. 111–116.

[13] R. Buchmann and A. Greiner, "A fully static scheduling approach for fast cycle accurate systemc simulation of mpsocs," in *Proc. of IEEE Microelectronics*, 2007, pp. 101–104.

[14] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory, "Relaxing synchronization in a parallel systemc kernel," in *Proc. of IEEE ISPA*, 2008, pp. 180–187.

[15] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: synchronous parallel SystemC simulation on multi-core host architectures," in *Proc. of ACM/IEEE CODES+ISSS*, 2010, pp. 241–246.

[16] P. Ezudheen, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC kernel for fast hardware simulation on SMP machines," in *Proc. of ACM/IEEE PADS*, 2009, pp. 80–87.

[17] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations," in *Proc. of ACM/IEEE DATE*, 2010, pp. 606–609.

[18] S. Jones, "Optimistic parallelisation of SystemC," Universite Joseph Fourier: MoSiG DEMIPS, Tech. Rep., 2011.

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

[20] ——, "GCS: High-performance gate-level simulation with GP-GPUs," in *Proc. ACM/IEEE DATE*, 2009, pp. 1332–1337.

[21] ——, "High Performance Gate-Level Simulation with GP-GPUs," in *GPU Computing Gems*. Morgan Kaufmann, 2011, ch. 23.

[22] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert, "Parallel cycle based logic simulation using graphics processing units," in *Proc. of IEEE ISPDC*, 2010, pp. 71–78.

[23] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: A fast SystemC simulator on GPUs," *Proc. of ACM/IEEE ASP-DAC*, pp. 149–154, 2010.

[24] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi, "SAGA: SystemC acceleration on GPU architectures," *Proc. of ACM/IEEE DAC*, pp. 115–120, 2012.

[25] N. Bombieri, F. Fummi, and V. Guarnieri, "FAST-GP: An RTL functional verification framework based on fault simulation on GP-GPUs," *Proc. of ACM/IEEE DATE*, pp. 562–565, 2012.

[26] R. Sinha, A. Prakash, and H. D. Patel, "Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs," *Proc. of ACM/IEEE ASP-DAC*, pp. 455–460, 2012.

[27] M. Nanjundappa, A. Kaushik, H. D. Patel, and S. K. Shukla, "Accelerating SystemC simulations on GPUs," *Proc. of IEEE HLDVT*, pp. 1–8, 2012.

[28] A. Perinkulam and S. Kundu, "Logic simulation using graphics processors," in *Proc. International Test Synthesis Workshop*, March 2007.

[29] N. Bombieri, S. Vinco, D. Chatterjee, and V. Bertacco, "SystemC Simulation on GP-GPUs: CUDA vs. OpenCL," *Proc. of ACM/IEEE CODES+ISSS*, pp. 343–352, 2012.

[30] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, NVIDIA, 2008, http://developer.download.nvidia.com.

[31] *OpenCL - The open standard for parallel programming of heterogeneous systems*, Khronos Group, http://www.khronos.org/opencl.