Parallel Processing Letters

Parallelization of Cycle-based Logic Simulation

Toni Mancini, Annalisa Massini, Enrico Tronci
Computer Science Department, Sapienza University of Rome
via Salaria 113 - 00198 Rome, Italy
{tmancini,massini,tronci}@di.uniroma1.it

Received July 2016
Revised March 2017
Communicated by C. Stirling

ABSTRACT

Verification of digital circuits by Cycle-based simulation can be performed in parallel. The parallel implementation requires two phases: the compilation phase, that sets up the data needed for the execution of the simulation, and the simulation phase, that consists in executing the parallel simulation of the considered circuit for a certain number of cycles. During the early phase of design, compilation phase has to be repeated each time a bug is found. Thus, if the time of the compilation phase is too high, the advantages stemming from the parallel approach may be lost. In this work we propose an effective version of the compilation phase and compute the corresponding execution time. We also analyze the percentage of execution time required by the different steps of the compilation phase for a set of literature benchmarks. Further, we implemented the simulation phase exploiting the GPU architecture, and we computed the execution times for a set of benchmarks obtaining values comparable with literature ones. Finally, we implemented the sequential version of the Cycle-based simulation in such a way that the execution time is optimized. We used the sequential values to compute the speedup of the parallel version for the considered set of benchmarks.

Keywords: Cycle based simulation, And Inverter Graph, GPU

1. Introduction

Design of digital circuits relies on logic simulation, that is part of the verification process. Logic simulation is used to verify the expected behavior of a new circuit design. For this reason, the simulation of a circuit is executed many times during the verification process, in particular in the early phase of design. The verification process is repeated until the new circuit design has been verified to implement the design objectives. Nevertheless, several execution scenarios can remain unverified, and circuit designs can be released with latent bugs.

When a circuit design consists of millions of gates, the logic simulation is a highly time consuming task, and can be the bottleneck in the design process.

Cycle-Based Simulation (CBS), or oblivious simulation, is a commonly used technique for the simulation of logic circuits, where each gate is evaluated during each simulation cycle. Instead, in Event-Based Simulation (EBS), output value of a gate is computed only

if at least one of its input values has changed. CBS provides a very simple scheduling and static data structures. Conversely, EBS requires dynamic analysis for scheduling gates producing a new output. CBS static policy of gate evaluation makes this kind of simulation more suitable for the parallelization.

One way to accelerate the logic simulation of digital circuits is to exploit Graphics Processing Units (GPUs) and general purpose programming models such as Compute Unified Device Architecture (CUDA), [1, 2], for the parallelization. Examples of parallel approaches to logic simulation have been proposed in [3, 4, 5, 6, 7, 8, 9, 10, 11]. Cycle-based simulation is considered in [3] and in [8]. In particular, both solutions describe a parallelization algorithm consisting of two phases: the compilation phase, devoted to prepare the data structures, and the simulation phase, that simulates the considered circuit in parallel.

During the early phase of digital circuit design, the simulation can be interrupted early because of bugs. Consequently, if the parallel version of the algorithm is used, the compilation phase can be repeated many times.

In this work, we want to answer to the question: *During the design process, is it more advantageous to use the (optimized) sequential implementation for the Cycle-based simulation of the circuit, or the parallel one?* This question arises from the observation that when using the parallel version, before running the simulation phase, we need to run the (preparatory) compilation phase. Then, we need to answer also to the question: *How much expensive is the compilation phase?*

To answer these questions, we analyze the compilation phase proposed in [8], and compute the execution time for a set of benchmarks. We also compute the execution time of the different steps of this phase and study which step takes the longest time in percentage. In order to evaluate our implementation of the compilation phase, we also implemented the simulation phase, following the approach described in [8], thus realizing the whole parallel algorithm. Our implementation gives execution time values comparable to those listed in [8]. Moreover, we realized an implementation of the sequential version of the simulation algorithm and computed the speedup of the parallel version.

In summary, the contribution of this paper is:

(1) Explicit computation of the execution time for the compilation phase, not given in [8]. This information is useful to understand which version, among sequential and parallel, is more convenient during the early phase of design, when compilation phase has to be repeated each time a bug is found.
(2) Effective exploitation of the AIG simulator features resulting in a substancial reduction of the sequential simulation time with respect to that given in [8].
(3) A tight estimation of the speedup of the parallel version, obtained using the optimized sequential execution time. It is worth noting that speedup values can appear worse with respect to those presented in [8]. This is due to the fact that our speedup time values refer to the optimized sequential implementation of the algorithm, that provides lower time values.

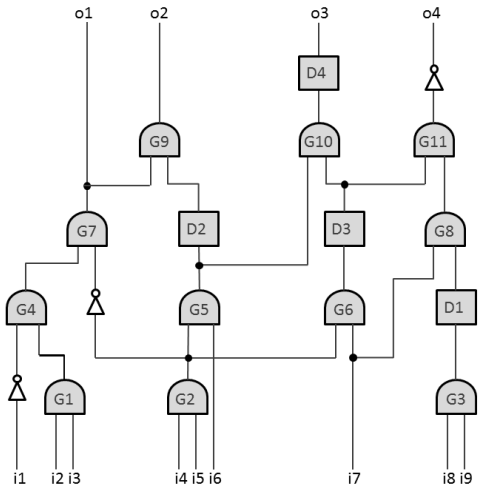

Fig. 1. A circuit consisting of AND gates and latches

2. Sequential and Parallel Cycle-based simulation

In this work we consider circuits consisting of AND gates with two inputs, NOT gates and D-type latches having one input. To represent logic circuits, we consider And Inverter Graphs (AIGs), since they provide an efficient representation for manipulating boolean functions. AIGs consist of two input nodes representing AND gates, whereas the NOT gates are represented as attributes (logical negation) of the edges. An example of circuit is shown in Figure 1.

The sequential Cycle-based simulation of a circuit is based on the following steps: simulation of combinational elements, simulation of sequential elements, generation of output values using input values and present state latch values, according to the scheme shown in Algorithm 1 (see also [8]).

1 **Input**: circuit, number of cycles of the simulation, input values
2 **Output**: output values
3 **For each** cycle
4 simulation of combinational elements
5 simulation of sequential elements
6 generation of output values using input values and present state latch values

Algorithm 1: Sequential Cycle-Based simulation algorithm

To parallelize the circuit simulation, the circuit can be partitioned in such a way that each group of gates can be simulated independently. The parallelized algorithm consists of

two phases, as described in [3] and [8]: namely, the compilation phase and the simulation phase. A similar approach is also used in [12] and [13].

During the compilation phase, data for the parallel execution of the simulation are prepared. To this end the levelization of gates and clustering operations are performed. Levelization is used to determine the gate dependencies; gates on the same level can be simulated at the same time. Clustering is used to partition the circuit in sub-circuits, namely clusters, each of which can be simulated independently. Each cluster will be simulated by a CUDA block. Clustering operation consists of three steps: logic cones construction, clustering of cones and cluster balancing. In particular, cluster balancing is used to obtain clusters whose levels have more or less the same number of gates, that is clusters having a rectangular shape. The aim of this step is to make the use of CUDA threads as efficient as possible. In fact, during the simulation phase, thousands of CUDA threads will simulate the circuit using the clusters produced by means of the compilation phase.

A scheme of the parallel algorithm is shown in Algorithm 2 (see also [8]).

```
1  Input: Circuit, Cycle number of the simulation
2  Output: Output values
3  Compilation Phase
4     Levelization of gates
5     Clustering
6        Logic cones construction
7        Clustering of cones
8        Cluster Balancing
9  Simulation phase
10    Data transfer from host to device (cluster representation)
11    Random input generation on the GPU
12    Parallel simulation of all clusters on the device and computation of output values
13    Output transfer from device to host
```

Algorithm 2: Cycle-based Parallel Simulation algorithm by using a GPU

3. Analysis and Implementation of the Compilation Phase

In this Section, we describe the compilation phase. In [8] the execution time values of the compilation phase are not given. On the contrary, only values of the parallel simulation phase are considered both for the comparison with the sequential version and for the computation of the speedup values.

In the following, we describe our implementation of the two stages of the compilation phase: Levelization and Clustering.

3.1. *Levelization*

This stage is devoted to arrange gates in levels. The level where primary inputs and present state values of sequential elements are located is denoted as Level 0 (see the example in Figure 2). Computing the level to which a gate belongs to is equivalent to compute the maximum distance between the gate and level zero. In general the computation of the longest path in a graph is an NP-hard problem. But, in the case of a direct acyclic graph, a linear algorithm can be used. Since we use AIGs to represent circuits, we have adopted the algorithm described in [14] and sketched in Algorithm 3.

1 **Find** a topological order on the given DAG
2 **For** each vertex v of the DAG, by using the topological order, compute the length of the longest path ending in v by adding 1 to the length of the longest path of its adjacent neighbors.
3 **If** v has not adjacent neighbours, set to zero the length of the longest path ending in v.

Algorithm 3: Length of the longest path on a DAG

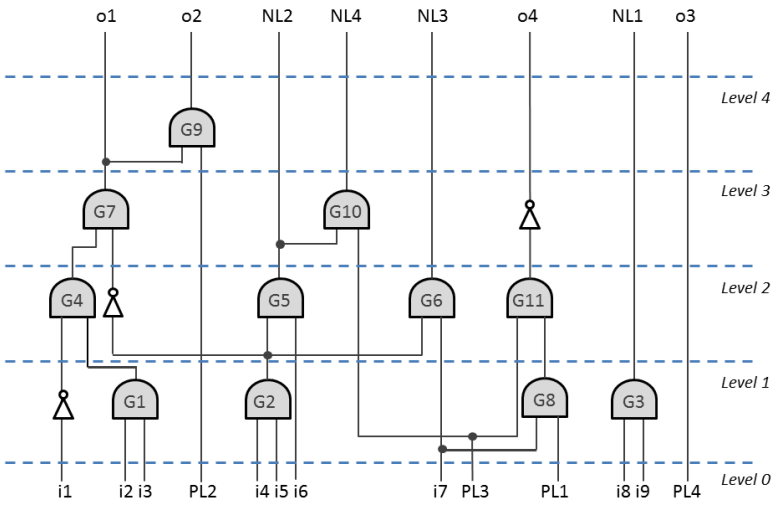

Fig. 2. Circuit levelization of circuit in Figure 1

The result of the levelization operation is shown in Figure 2. Note that latches are not shown because the levelization is performed only on combinational elements. Instead, the present values of latches are listed as inputs (PL_i), whereas the next values of latches are shown as outputs (NL_i). Note also that nodes on the lower level (Level 0) can receive only values from Level 0. For example, gate G8 receives two values coming from Level 0, and

its level is Level 0. Gate G11 receives one input value coming from Level 0, and the value produced as output by G8 at Level 1, then G11 level is Level 1.

3.2. *Clustering*

The goal of the clustering stage is to obtain balanced clusters. The clustering operation requires three steps, whose implementation is described in the following paragraphs.

3.2.1. *First Step – Logic cones construction*

The construction of the logic cones consists in creating cones of influence of circuit outputs [15, 16]. During this step, we build as many cones as the number of outputs and latches. The result of this step is shown in Figure 3.

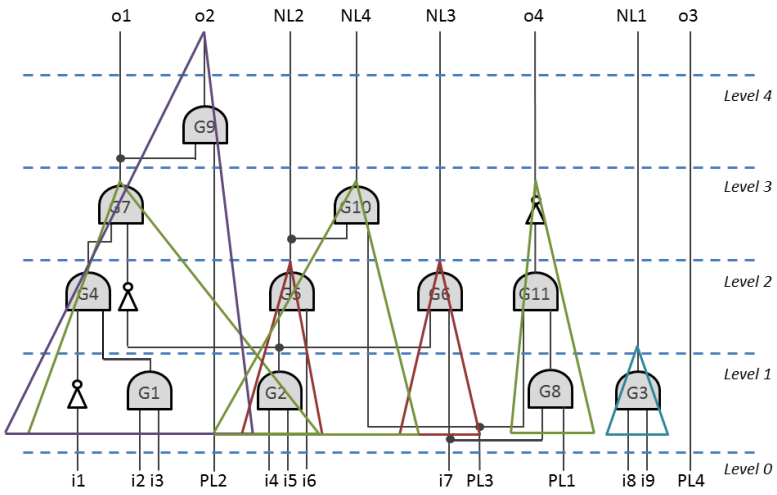

Fig. 3. Construction of logic cones for circuit in Figure 2.

3.2.2. *Second step – Clustering of cones*

The cones obtained in the previous step can have very different sizes, as shown in Figure 3. Hence, once the logic cones are built, they are merged to obtain clusters similar in size, with the characteristic of not overlapping with each other. This constraint can require the replication of a certain number of gates. To minimize the number of replicated gates, first the maximum number of gates that a cluster can contain is computed. Then, a cone is randomly chosen and it is merged with cones whose intersection with this random selected cone is maximum. Cone merging is executed by adding a new cone to the cluster until the maximum number of gates in the cluster is obtained. The maximum number of gates that a cluster can contain is obtained by dividing the number of gates (considering also the

replicated gates) by the number of CUDA blocks. This number depends on the considered GPU and on characteristics of the circuit.

Input: cones $S_1, \ldots, S_m$
Output: map M mapping a key-pair (i, j) (denoting the pair of cones S_i and S_j) to the cardinality of the intersection between cones S_i and S_j
for each cone S_i
  for each gate G_h **in** S_i
    $E[h] \leftarrow i$
for each element h (list) **in** E
  for each index i **in** $E[h]$
    for each index $j > i$ **in** $E[h]$
      $M[(i, j)] += 1$

Algorithm 4: Computation of intersections among all pairs of cones.

Computing the cardinality of the intersections among all pairs of cones is a time consuming task, that we executed by means of the technique described in Algorithm 4. In the pseudocode, we use the array of lists E of length the number of gates of the circuit, where element h is associated to gate G_h, and $E[h]$ is the list of cones containing gate G_h.

After this step, trapezoidal clusters containing almost the same number of gates are obtained, as shown in Figure 4.

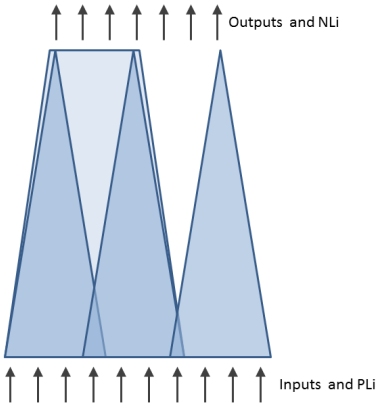

Fig. 4. A trapezoidal cluster obtained by merging triangular cones.

The implementation of Logic cones construction and Clustering of cones steps has been parallelized by using OpenMP.

3.2.3. *Third step – Cluster balancing*

To balance the clusters, the average width W of clusters is computed. When a level has a number of gates greater than W, the exceeding gates are moved to the upper level. To preserve the dependence among gates, also gates depending from a repositioned gate are moved up. To realize the move of a gate to the upper level, we insert a dummy gate in its place and move the gate and its depending gates to upper levels as shown in Figure 5.

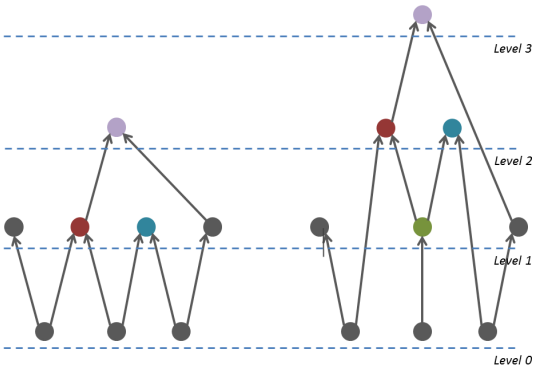

Fig. 5. Node move for cluster balancing.

Note that dummy nodes are added also when on a level there are less than W gates. Hence, dummy nodes are inserted both to increase the number of nodes on a level, and to reduce the number of nodes on levels having too large width. At the end of this step, we have a set of clusters having the same width, and presenting different heights. Each cluster will be assigned to a CUDA block. Figure 6 shows an unbalanced trapezoidal shaped cluster and a (higher) balanced rectangular shaped cluster.

An approach similar to the *Cluster balancing* is also used in [17, 24].

4. Simulation Phase

In this section we describe our implementation of the parallel simulation phase.

The simulation phase consists of several *for*-loops, namely: a loop to transfer the variables from the global memory to the shared memory, a loop to simulate all gates on a level, level by level, and a loop to update values in global memory, values that will be transferred during the next iteration.

Note that the next state of each latch is computed in its block, but at the end of the execution of the simulation of the whole circuit these values must be supplied to the clusters needing them as input. This makes necessary the synchronization among blocks, obtained by launching another kernel.

For the implementation of the simulation phase, we introduced a different way to generate inputs with respect to [8]. Our method consists in the generation of random inputs on the GPU and represents an optimization. In fact, the random inputs generation allows both

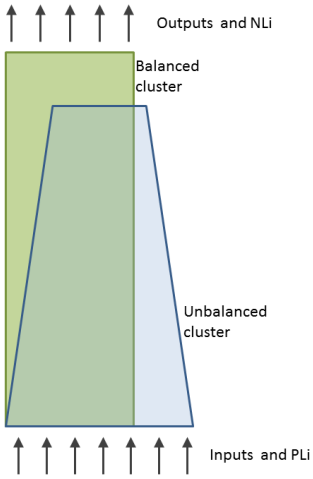

Fig. 6. A rectangular shaped cluster after the cluster balancing step.

a quicker execution and the elimination of communication between CPU (also called host) and GPU (also called device) to transfer the input values. For the implementation, the two libraries *cuRAND* and *CUDA Thrust* have been experimented: *cuRAND* is devoted only to the generation of random numbers, whereas *CUDA Thrust* provides also algorithms and data structures for the generation of parallel applications. Tests showed that, for generating less than ten millions random values, as in the case of our testbeds, *Thrust* is slightly quicker. On the contrary, *cuRAND* performs better for generating more than ten millions random values.

Furthermore, we adopted several optimization strategies to obtain the best possible performance from the CUDA architecture. In the following, we give a brief description of these optimization strategies, referring to the CPU as host, and to the GPU as device.

Use of the Pinned Memory For the transfers between host and device and vice versa, we used the asyncronous calls *cudaMemcpyAsync*, thus avoiding to block the computation during trasfers. Such calls require the use of the *pinned memory* by the host, that is allocated by means of the function *cudaMallocHost*, preventing the disk data swapping.

Coalesced access to the memory During the simulation on the GPU, input and output gate values must be accessed. To obtain efficient accesses, information are in global memory and are accessed in coalesced manner. In fact, the best performance in accessing the global memory is obtained when all 32 threads belonging to a *warp* access contigue locations. To obtain coalesced accesses, data structures for gates are modelled such that gates on a level are adjacent, and there is an array for the three indices of each gate (first input, second input and output).

Encoding of outputs of intermediate gates Shared memory is quite small in size. Blocks on the same multiprocessor share the same amount of shared memory. By assigning two

blocks to each multiprocessor, it follows that each block has half space available. If we use one byte for each output value of each gate, memory is not efficiently exploited, in particular when large circuits are simulated. Therefore, exploiting the fact that outputs can assume only 0 and 1 values, we encode 8 outputs on one byte. In our case, each multiprocessor has 48KB shared memory, then each block can use 24KB. By encoding 8 outputs on one byte, we can represent $24576 \times 8 = 196608$ variables per block (instead of 24576). This encoding allows to simulate circuits with over 2 millions of gates (having already considered a replication of gates of about 30%). It is worth noting that this encoding implicitly provides coalesced accesses to memory.

5. Experimental Results

In this section we evaluate the effectiveness of our implementation and compare it with the solution proposed in [8].

Experiments were run on an istance G2 of Amazon EC2 for graphic applications and GPGPU, with an Intel Xeon E5-2670 processor (8 cores, 16 threads, 2.6 GHz frequency, 3.3 GHz turbo frequency), 16GB RAM and an NVIDIA grid K520 GPU with 4GB RAM and 8 multiprocessors, each with 192 cores.

Circuits considered for the evaluation were chosen from the collection of benchmarks IWLS [19], in particular from the subset of circuits OpenCores [20]. Our set consists of ten circuits. In particular, we included seven circuits considered also in [8] (*vga_lcd* - controller vga/lcd, *des_perf* - triple DES IP core optimized, *ethernet* - ethernet IP core, *wv_conmax* - interconnect matrix IP core, *pci_bridge32*, *aes_core* - AES IP core for encryption/decryption, *ac97_ctrl* - AC 97 Audio Codec controller core). We also included three circuits not included in the set considered in [8] (*usb_funct* - USB 1.1 slave/device IP core, *mem_ctrl* - memory controller for embedded applications, *systemcaes* - AES encryption/decryption core). For each considered circuit, the AIGER format (an implementation of the AIG corresponding to the circuit) [21] is generated by using the ABC tool [22, 23].

Table 2. Circuits used for the tests.

Design	Variables	Inputs	Latches	Outputs	Gates	L	C
vga_lcd	143879	89	17079	109	126711	23	16
des_perf	109308	17850	8808	9038	82650	19	16
ethernet	80326	98	10544	115	69684	31	16
wb_conmax	49753	1130	770	1416	47853	26	8
pci_bridge32	26305	162	3359	207	22784	29	8
aes_core	23371	1319	530	668	21522	25	8
usb_funct	19480	192	1746	87	17542	26	8
ac97_ctrl	12624	84	2199	48	10341	8	8
mem_ctrl	8044	23	1083	31	8021	35	4
systemcaes	6713	37	670	15	6676	45	4

Table 2 shows the list of circuits, specifying the number of variables, inputs, latches, outputs, AND gates, levels L and clusters C (note that each cluster is executed by a block). To exploit the available GPU, the number of blocks is determined according to the size

of the circuit, namely the number of gates. Taking into account that our GPU consists of 8 multiprocessors, each multiprocessor is equipped with 48KB of shared memory and supports 2048 threads, and each CUDA block can include up to 1024 threads, we generated 16 CUDA blocks for large circuits, whereas we used 8 or 4 blocks for smaller circuits (see Table 2).

Table 3. Execution times of steps of the compilation phase, in milliseconds.

Circuit Design	Cone construction	Clustering of cones	Cluster balancing	I/O op	Total
vga_lcd	2186	35382	2745	292	40605
des_perf	212	1134	59	188	1593
ethernet	506	4490	290	280	5566
wb_conmax	220	513	35	136	904
pci_bridge32	180	579	11	108	878
aes_core	117	303	28	115	563
usb_funct	73	199	6	78	356
ac97_ctrl	35	153	4	62	254
mem_ctrl	122	324	1	65	512
systemcaes	121	266	4	69	460

Table 3 shows execution times of the three different steps of the Clustering stage of the compilation phase, in milliseconds. Note that, time for the Levelization stage is not reported in Table 3 because is negligible, whereas time for I/O operations is considered.

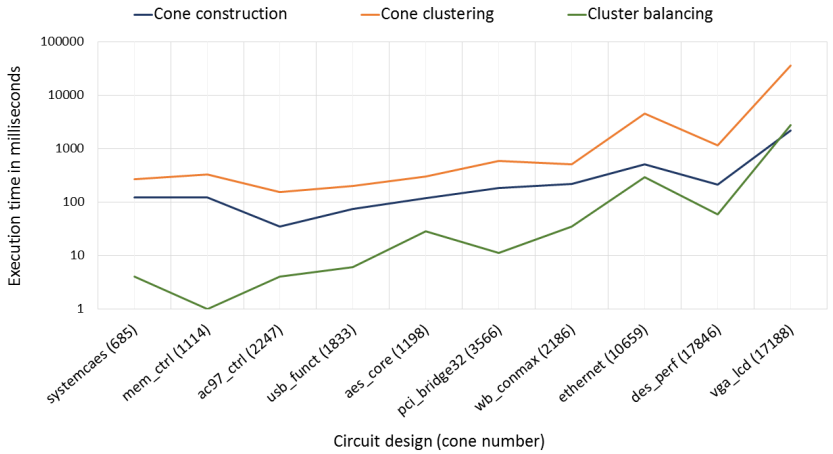

Fig. 7. Execution times of the three steps of the compilation phase.

We can observe from Table 3 that the *Clustering of cones* step is the most time consuming step during the compilation phase. Note that the time for *Clustering of cones* is

particularly high in the case of *vga_lcd* circuit. In fact, due to the structure of the circuit, in that case we generate about 20000 cones, and we need to compute the cardinality of intersections of all possible pairs of cones.

In Figure 7 the values of the execution time of the steps of the Clustering stage of the compilation phase are shown. Note that in Figure 7 the cluster balancing value for *mem_ctrl* is on the horizontal axis visible (vertical logarithmic scale). In Figure 8 the percentage of time of each step of the compilation phase is shown for each considered circuit.

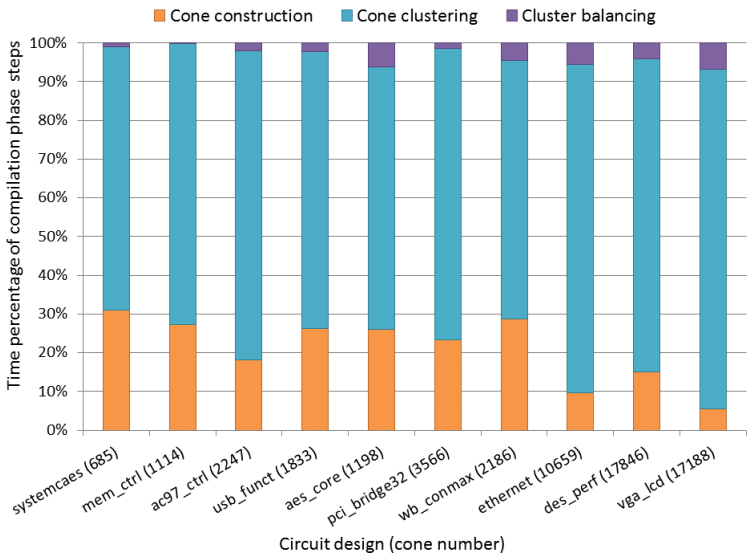

Fig. 8. Percentage of time of the three steps of the compilation phase.

After the compilation phase, the compiled circuit is saved, so that it is always available. This way the compiled circuit can be given to the simulator without repeating the compilation phase, for future executions.

In Table 4 and Table 5, we show the experimental results obtained by running 10^5 cycles and 10^6 cycles, respectively, with input randomly generated on the GPU. In particular, column *Sequential* shows the time of the sequential simulation computed with the tool *aigsim* available with AIGER; column *Parallel* shows the time of the parallel simulation, and column *Parallel + Comp.* shows the parallel simulation time plus the compilation time. Columns *Speedup* and *Speedup + Comp.* show the speedup values of the parallel version with respect to the sequential version, without and with the compilation phase, respectively. Times are in seconds.

Comparing speedup values in Table 4 and Table 5, we can observe that the time required by the compilation phase is quite short, and it is amortized when the number of executed cycles grows. Furthermore, results are strongly influenced by the structure of the

circuit design, and speedup values are quite different depending on the characteristics of the considered circuit.

We observe that the higher the number of variables, the more efficient the parallel approach. In fact, for the two smallest cases *mem_ctrl* and *systemcaes*, which represent circuits with less than 10^4 variables, the sequential version is more efficient, and the overhead due to the parallelization is not well amortized. Neverthless, the number of variables is not the only important parameter. In fact, a circuit having a smaller number of variables can require a short time, if it has a small number of levels.

Table 4. Experimental results for 10^5 cycles.

Circuit Design	**Sequential**	**Parallel**	**Parallel + Comp.**	**Speedup**	**Speedup + Comp.**
vga_lcd	106.968	20.688	61.293	5.2x	1.8x
des_perf	62.198	11.352	12.945	5.5x	4.8x
ethernet	69.226	14.573	20.139	4.8x	3.4x
wb_conmax	24.163	11.451	12.355	2.1x	2.0x
pci_bridge32	20.131	11.902	12.780	1.7x	1.6x
aes_core	10.958	9.185	9.748	1.2x	1.1x
usb_funct	12.192	9.964	10.320	1.2x	1.2x
ac97_ctrl	12.809	6.840	7.094	1.9x	1.8x
mem_ctrl	9.824	12.135	12.647	0.8x	0.8x
systemcaes	7.532	13.712	14.172	0.6x	0.5x

Table 5. Experimental results for 10^6 cycles.

Circuit Design	**Sequential**	**Parallel**	**Parallel + Comp.**	**Speedup**	**Speedup + Comp.**
vga_lcd	1067.136	201.152	241.394	5.3x	4.4x
des_perf	620.894	108.944	110.505	5.7x	5.6x
ethernet	690.082	140.960	146.416	4.9x	4.7x
wb_conmax	241.047	110.585	111.460	2.2x	2.2x
pci_bridge32	201.149	115.520	116.397	1.7x	1.7x
aes_core	109.341	87.980	88.527	1.2x	1.2x
usb_funct	121.683	96.288	96.647	1.3x	1.3x
ac97_ctrl	127.760	64.869	65.132	2.0x	2.0x
mem_ctrl	98.207	118.056	118.568	0.8x	0.8x
systemcaes	75.412	133.731	134.208	0.6x	0.6x

The implementation of the sequential version realized for this work provides very good sequential execution times, and represents an optimized version with respect to version presented in [8]. This implementation has been obtained by means of an optimized utilization of the sequential simulator *aigsim*, available with AIGER. To complete the results discussion, in Table 6 we compare the sequential version described in [8], our sequential version, and our parallel version (that show execution times comparable to the parallel version presented in [8]).

In particular, in Table 6 we show: in column *Sequential SAB2011* the sequential times presented in [8], in column *Optimized Sequential* our optimized sequential times, and in column *Parallel* our parallel times, all computed for 10^5 cycles (sequential times are ob-

tained on comparable hardware).

Table 6. Speedups obtained comparing the parallel execution time with respect to the sequential time reported in [8] and our optimized sequential time, for 10^5 cycles.

Circuit Design	**Sequential SAB2011**	**Optimized Sequential**	**Parallel**	**Speedup1 (SAB2011)**	**Speedup2 (Our)**
vga_lcd	223.15	106.97	20.69	10.79	5.17
des_perf	180.62	62.20	11.35	15.91	5.48
ethernet	155.51	69.23	14.57	10.67	4.75
wb_conmax	94.81	24.16	11.45	8.28	2.11
pci_bridge32	50.12	20.13	11.90	4.21	1.69
aes_core	83.64	10.96	9.19	9.10	1.19
ac97_ctrl	58.66	12.81	6.84	8.58	1.87

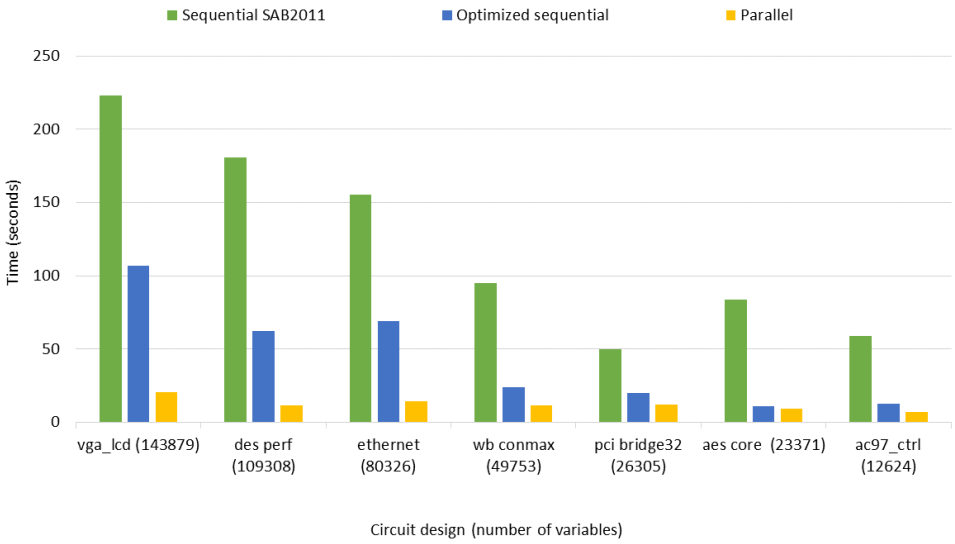

Fig. 9. Comparison among sequential times and parallel times for 10^5 cycles simulation.

In Table 6 we also show the speedup values obtained by using the times of our parallel version. In particular, values in column *Speedup1* are computed using the sequential times shown in [8], and values in column *Speedup2* are computed using our optimized sequential times. All time values are expressed in seconds. Note that the parallel execution times do not include the execution time of the compilation phase. Figure 9 plots times reported in Table 6.

Obviously, the speedup values with respect to the optimized sequential execution time are lower, and this provides a better estimation of the speedup obtainable when parallelizing a cycle-based simulation.

6. Conclusions

In this work we have studied and analyzed the parallelization of the cycle-based simulation of digital circuit. The parallel implementation consists of two phases: the compilation phase, running on the CPU and devoted to prepare the data structures, and the simulation phase, that is the parallel phase and runs on the GPU. In particular, we have analyzed the compilation phase. We propose an efficient implementation, and we study the execution time of the different steps of the compilation phase for a set of literature benchmarks. The computation of the execution time of the compilation phase is important to understand if, during the early phase of design, the parallel version for the Cycle-based simulation of the circuit under verification is more advantageous with respect to the (optimized) sequential implementation, since the parallelization entails the additional time due to the compilation phase.

In order to complete the evaluation of our implementation of the compilation phase, we also realized the parallel part of the algorithm, implementing the simulation phase, and obtaining execution times for the simulation phase similar to those listed in [8]. Moreover we have implemented the sequential version of the logic simulation algorithm by utilizing the circuit simulator in an optimized way, and we have used the corresponding execution time to give a tight estimation of the speedups with respect to those obtained with the sequential version presented in [8]. In fact, despite the fact that our speedup values are lower than those in the cited paper, they give a better estimation of the obtainable speedup, since the sequential time considered in [8] was obtained with a not optimized sequential version.

We plan to investigate how to integrate the approach presented in this paper with the simulation based parallel verification techniques for hybrid systems described in [18, 25].

Acknowledgements This research has been partially supported by FP7 projects: *Energy Demand Aware Open Services for Smart Grid Intelligent Automation*, **SmartHG**, Project n. 317761, and *Model Driven Computation of Treatments for Infertility Related Endocrinological Diseases*, **Paeon**, Project n. 600773.

References

[1] NVIDIA CUDA, *http://www.nvidia.com/cuda*.

[2] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA Corporation, 2007.

[3] D. Chatterjee, A. DeOrio, and V. Bertacco, *GCS: High-performance Gate-level Simulation with GP-GPUs*, Proc. Conference on Design, Automation and Test in Europe, DATE '09, 1332–1337, 2009.

[4] B. Wang, Y. Zhu, and Y. Deng, *Distributed Time, Conservative Parallel Logic Simulation on GPUs*, Proc. IEEE/ACM Design Automation Conference, 2010.

[5] H. Qian and Y. Deng, *Accelerating RTL Simulation with GPUs*, Proc. IEEE/ACM Int. Conf. on ComputerAided Design, 2011.

[6] Y. Zhu, B. Wang, and Y. Deng, *Massively Parallel Logic Simulation with GPUs*, ACM Transaction on Design Automation of Electronics Systems, Vol.16, No.3, 2011.

[7] D. Chatterjee, A. De Orio, and V. Bertacco, *Gate-Level Simulation with GPU Computing*, ACM Trans. Des. Autom. Electron. Syst., 2011.
[8] A. Sen, B. Aksanli, and M. Bozkurt, *Speeding Up Cycle Based Logic Simulation Using Graphics Processing Units*, International Journal of Parallel Programming, 639–661, 2011.
[9] M. Chimeh, C.V. Hall, and J.T. O'Donnell, *Optimisation and parallelism in synchronous digital circuit simulators*, Int. Conf. Computational Science and Engineering (CSE), 94–101, 2012.
[10] V. Bertacco, D. Chatterjee, N. Bombieri, F. Fummi, S. Vinco, A. Kaushik, and H.D. Patel, *On the use of GP-GPUs for accelerating compute-intensive EDA applications*, Proc. Conference on Design, Automation and Test in Europe, DATE '13, 1357–1366, 2013.
[11] T. Hashiguchi, Y. Mori, M. Toyonaga, and M. Muraoka, *YAPSIM: Yet Another Parallel Logic Simulator using GP-GPU*, Proc. SASIMI 2015
[12] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci, *System level formal verification via model checking driven simulation*, In *Proc. CAV 2013*, LNCS 8044, Springer, 2013.
[13] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, *Anytime System Level Verification via Random Exhaustive Hardware In The Loop Simulation*, In Proc. 17th EuroMicro Conference on Digital System Design (DSD), 2014.
[14] R. Sedgewick, K. Wayne, *Algorithms*, Addison-Wesley, 661–666, 2011.
[15] S. Smith, W. Underwood, and M.R. Mercer, *An analysis of several approaches to circuit partitioning for parallel logic simulation*, In Proc. ICCD, 1987.
[16] K. Hering, R. Reilein, and S. Trautmann, *Cone clustering principles for parallel logic simulation*, Proc. Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 93-100, 2002.
[17] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, *System Level Formal Verification via Distributed Multi-Core Hardware in the Loop Simulation*, In Proc. Parallel, Distributed and Network-Based Processing (PDP), 2014.
[18] E. Tronci, T. Mancini, I. Salvo, F. Mari, I. Melatti, A. Massini, S. Sinisi, F. Daví, T. Dierkes, R. Ehrig et al. *Patient-Specific Models from Inter-Patient Biological Models and Clinical Records*, In Proc. Formal Methods in Computer-Aided Design (FMCAD), 2014.
[19] IWLS, *http://iwls.org/iwls2005/benchmarks.html*.
[20] OpenCores, *http://opencores.org*
[21] AIGER, *http://fmv.jku.at/aiger*.
[22] ABC, *http://www.eecs.berkeley.edu/~alanmi/abc*.
[23] R. Brayton and A. Mishchenko, *ABC: an academic industrial-strength verification tool*, Proc. Int. Conf. Computer-Aided Verification (CAV), 2010.
[24] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, *SyLVaaS: System Level Formal Verification as a Service*, In Proc. Parallel, Distributed and Network-Based Processing (PDP 2015), 2015.
[25] T. Mancini, E. Tronci, I. Salvo, F. Mari, A. Massini, and I. Melatti. *Computing Biological Model Parameters by Parallel Statistical Model Checking*, In Proc. Int. Work Conference on Bioinformatics and Biomedical Engineering (IWBBIO), pages 542-554, 2015.