SAGA: SystemC Acceleration on GPU Architectures *

Sara Vinco
Dip. Informatica
Università di Verona, Italy
sara.vinco@univr.it

Debapriya Chatterjee
EECS Department
University of Michigan, USA
dchatt@umich.edu

Valeria Bertacco
EECS Department
University of Michigan, USA
valeria@umich.edu

Franco Fummi
Dip. Informatica
Università di Verona, Italy
franco.fummi@univr.it

ABSTRACT

SystemC is a widespread language for HW/SW system simulation and design exploration, and thus a key development platform in embedded system design. However, the growing complexity of SoC designs is having an impact on simulation performance, leading to limited SoC exploration potential, which in turns affects development and verification schedules and time-to-market for new designs. Previous efforts have attempted to parallelize SystemC simulation, targeting both multiprocessors and GPUs. However, for practical designs, those approaches fall far short of satisfactory performance. This paper proposes *SAGA*, a novel simulation approach that fully exploits the intrinsic parallelism of RTL SystemC descriptions, targeting GPU platforms. By limiting synchronization events with ad-hoc static scheduling and separate independent dataflows, we shows that we can simulate complex SystemC descriptions up to 16 times faster than traditional simulators.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design, Performance

Keywords

Parallel SystemC, CUDA simulation acceleration

1. INTRODUCTION

Design simulation has traditionally been a key technique to validate digital systems and to conduct early performance and constraints evaluations. However, the increasing complexity of modern designs has been pushing the scalability limits of this technology: as of today its poor performance on complex systems has heavy impacts on the development timeline and ultimately on a system's time-to-market [2].

*This work has been partially supported by EU project FP7-ICT-2011-7-288166 (TOUCHMORE) and the Gigascale Systems Research Center

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

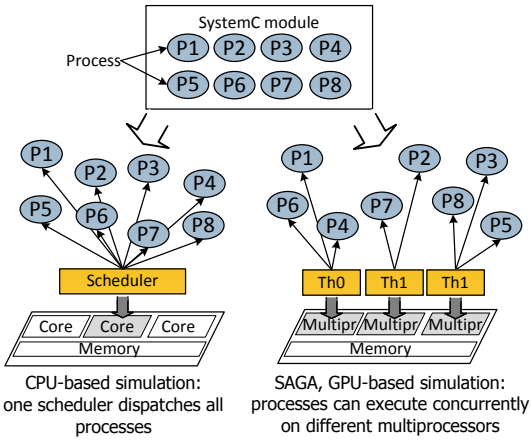

Figure 1: Methodology overview.

One of the most common languages for modeling many digital designs, and particularly embedded systems, is SystemC [8]. SystemC extends C/C++ with libraries to describe HW constructs. It is widely deployed in early-stage analyses and design-space explorations. However, its simulation performance is fairly slow, typically 10x slower than other RTL languages' simulations [2]. To make things worse, the most common SystemC simulation kernel (OSCI) uses application-level threading (co-operative threads), thus it is intrinsically sequential because the operating system cannot dispatch co-operative threads to different processing elements. When simulating transaction-level models (TLMs) these limitations do not have a major impact because the scheduler intervenes rarely and does not introduce heavy overhead. In contrast, RTL simulation requires frequent scheduler operations, leading to a heavy performance impact.

Several works in the literature have attempted to optimize and parallelize SystemC simulation, targeting heterogeneous architectures in order to reduce synchronization overheads and improve performance [1, 4, 10, 9]. Among them, the most promising direction targets GP-GPUs, highly parallel architectures designed for graphical applications and used in a wide range of scientific applications. Early solutions available in this space, such as [6], forego many performance benefits available when simulating SystemC designs on GP-GPU platforms. Looking ahead, a successful GP-GPU based solution for RTL SystemC simulation would bring additional benefits in integrated CPU-GPU architectures [5]. Indeed, the GPU could simulate embedded system's hardware while the CPU would remain available to execute embedded software applications, providing fast simulation of the entire system as a result.

Contributions. This paper proposes *SAGA*, a novel approach for concurrent SystemC simulation that leverages the massive parallelism available on GP-GPUs. Figure 1 overviews our approach

and compares it to a traditional SystemC simulation flow. The main contributions of our work are:

- A new concurrent simulation model for SystemC that exploits static scheduling to eliminate the need of frequent synchronization.
- A novel partitioning technique to carve independent dataflows; these are then mapped to distinct threads and multiprocessors to achieve concurrency in the execution.

We show the effectiveness of *SAGA* by applying it to a set of industrial SystemC designs and comparing its performance against that of simulating on chip multiprocessors (CMPs) and against that of previous GPU-based solutions.

The rest of this paper is organized as follows: Section 2 provides background, Section 3 highlights our contributions and the proposed simulation model. Finally, Section 4 presents our experimental evaluation and Section 5 concludes the presentation.

2. RELATED BACKGROUND

This section overviews a typical SystemC simulation flow and the CUDA programming model and architecture. It also discusses state-of-the-art solutions in concurrent SystemC simulation.

2.1 SystemC simulation

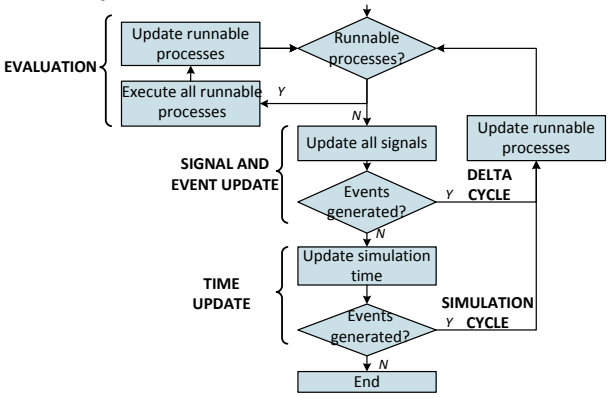

Figure 2: Traditional SystemC simulator scheduler.

SystemC uses an event-based architecture, where a centralized scheduler controls the execution of processes based on events (synchronizations, time notifications or signal value changes). Figure 2 depicts the execution flow of a typical SystemC 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.

The scheduler in a SystemC simulator coordinates the activation of all processes and manages both delta and simulation cycles. Because of this centralized approach, traditional SystemC simulators cannot take advantage of the concurrency of modern CMPs.

2.2 GP-GPU programming through CUDA

NVIDIA's Compute Unified Device Architecture (CUDA) [7] has been proposed to facilitate GP-GPU programming with a general purpose interface. In the CUDA execution model, the GPU is a co-processor capable of executing many threads in parallel, following the single instruction multiple data (SIMT) model of execution. A data parallel computation process, known as a kernel, can be offloaded to the GPU for execution. The collection of threads represented by a kernel is divided into a grid of thread-blocks.

The CUDA architecture (Figure 3) consists of a number of multiprocessors contained in a single GPU chip. Multiprocessors are responsible for the execution of the thread-blocks that can be mapped to each of them, as dictated by resource limits. Each multiprocessor is comprised of multiple stream processors that have common instruction fetch and support a large number of concurrent threads. Since all resident threads in a multiprocessor execute on a fixed number of stream processors with a common instruction fetch unit, each thread-block executes 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 of 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 and shared memory. All multiprocessors also have access to a region of global memory called device memory, which has higher access latency. Communication with the host CPU's main memory is achieved by means of direct memory access (DMA) transfers. Thus, it is important to keep communication between the host and the GPU to the bare minimum.

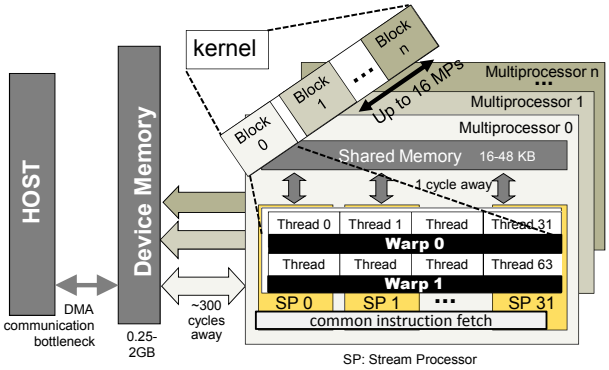

Figure 3: NVIDIA CUDA architecture.

2.3 Concurrent SystemC simulation

Several works in the literature propose to take advantage of the inherent parallelism of SystemC processes to speedup simulation [1, 4, 10, 9]. In SystemC, the order of process execution within a delta cycle does not affect the simulation's output since the simulator presents the same system's status to all those processes. Thus, processes that are activated within the same delta cycle can be executed in parallel, either by using multiple threads or by designing a distributed scheduler. For instance, in [4] SystemC processes are executed as distinct threads on multiple CPUs. Simulation relies on a simulation platform (ArchSim) that introduces heavy overhead, thus making this approach ineffective. In [1], each processing node includes a copy of the scheduler and it simulates a subset of the application modules. All scheduler's copies must synchronize after each delta cycle to update the value of shared signals and of simulation time, thus generating many synchronization events among the separate processors. A different approach is proposed in [9], which transforms the modules' structure. The methodology analyzes SystemC modules and it identifies those blocks within processes that can be executed within one simulator's phase and can be scheduled according to their data and control dependencies. All these solutions rely on code modifications or introduce overhead, as they rely

on an existing simulator [1, 10].

A different approach is proposed by the authors of [6], who also target the massive parallelism offered by today's GP-GPUs. In their solution, independent SystemC processes are mapped into parallel threads that synchronize at each iteration of a delta cycle (Figure 2), through a barrier synchronization, to maintain the correct producer-consumer relation among threads. Since typical SystemC processes contain few word-level and arithmetic operations, this can lead to more time spent on synchronization than execution.

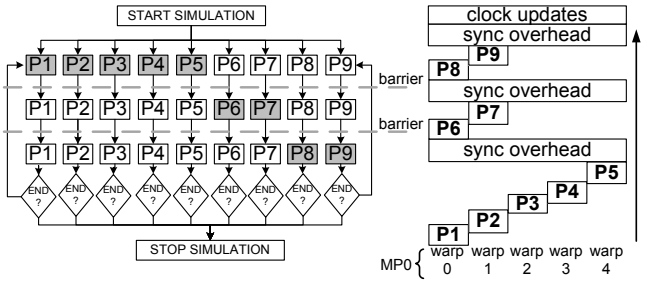

Figure 4: Scheduling example based on [6].

More importantly, the authors of [6] propose to map distinct processes to distinct threads in a same thread-block, so that they can leverage the fast intra-block synchronization mechanisms. However, this is unattainable for most practical SystemC descriptions, since different processes tend not to share the same code. The evaluation in [6] uses unusual designs, such as a 10-stage buffer, that do present lots of inter-process code similarity; however, this is not the common case. Our approach differs from [6] in that we do use distinct multiprocessors to map distinct processes. Then we propose a new scheduler design to minimize the number of synchronizations necessary. As a result we gain concurrent execution even in the common case of processes not sharing any code similarity. Figure 4 illustrate briefly their approach: on the left we show their planned scheduling of processes and on the right the timeline of computation for a same set of non-identical processes on a CUDA platform (*i.e.*, processes are serialized because they do not share the same SystemC code). The evaluation section compares their performance improvement with that of *SAGA*.

3. MAPPING SYSTEMC TO CUDA

Exposing parallelism in a SystemC simulator is not trivial, since the simulation is neither embarrassingly parallel, nor homogeneous. However, some parallelism can be extracted when treating processes active in a same delta cycle as concurrent tasks. *SAGA* exploits this aspect through three steps, as depicted in Figure 5:

1. construction of the *dependency graph* based on the signals read and written by each process, to build a static schedule for the SystemC processes (Section 3.1);
2. partitioning of the static schedule into *parallel dataflows*, that will be executed concurrently in different warps on the CUDA architecture (Section 3.2);
3. levelization of processes within each dataflow based on a *sequential order*. The resulting process blocks will be executed by concurrent thread-blocks in the GP-GPU (Section 3.3).

3.1 Construction of process dependency graph

In our construction, we assume that there is no circular dependency loop between processes and we arrange them in a producer-consumer order based on the I/O direction of their connecting signals. To this end, we build a *process-graph PG = (V;E)* where each process is represented by a vertex V; a directed edge E from vertex v_1 to vertex v_2 represents a process dependency due to a

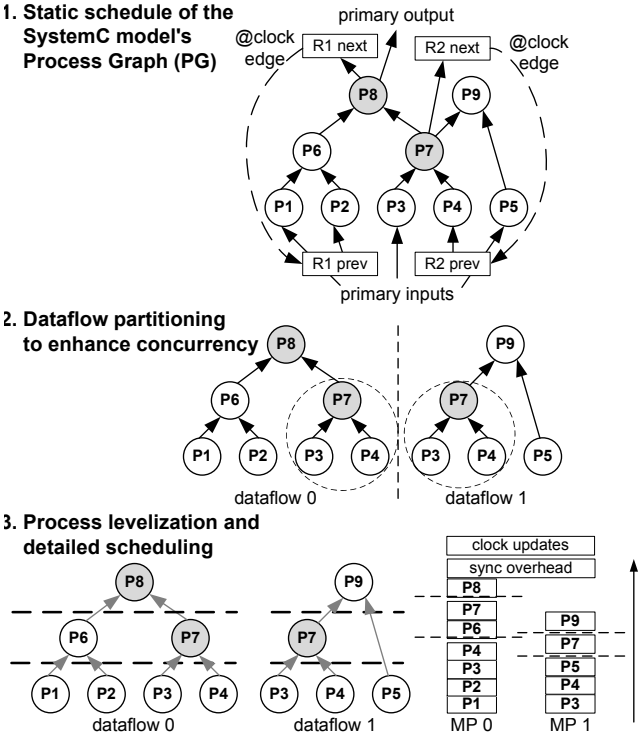

Figure 5: *SAGA* methodology steps.

signal generated by v_1 and consumed by v_2. We do not represent synchronous statements in the process-graph, since they create a dependency between present-state values and next-state values through time, while we focus on exploring the concurrency within a same time tick. The PG is a directed acyclic graph (DAG), and thus we can apply a topological sort to it. Processes dependent only on delta events at their primary inputs and at synchronous variables occupy the lowest level; the other levels are established by the edge connections.

Figure 5.1 shows an example of a process graph built for a SystemC module. Nodes in grey represent synchronous processes (*e.g.*, *P8*), while white corresponds to asynchronous ones (*e.g.*, *P6*). Signals *R1* and *R2* are written by synchronous statements, thus they have a current value (*R1 prev* and *R2 prev*) and a future value (*R1 next* and *R2 next*). Their current value will be updated once the dataflow execution has completed (as suggested by the dashed arrows). Steady-state values at the primary output signals and next state values for the synchronous signals can be obtained by executing the processes level-by-level. Because of how delta cycles operate in a traditional simulator, at stable state the PG-based simulator is guaranteed to provide the same results as a traditional one.

Moreover, our construction leveraging static scheduling presents an intrinsic advantage for parallel platforms, since there is no need for a central scheduler to manage events and to activate processes. Note that we can still benefit from the advantages of an event-driven simulation: if we only execute a process conditionally to a change at its inputs, then we are basically using an event-based approach. This optimization brings upon a 10% performance improvement on average over our baseline solution.

3.2 Partitioning into concurrent dataflows

There are several ways of partitioning the process graph obtained in the previous section: we select one based on the constraints of our target GPU platform. A straightforward approach would map different processes to distinct threads, one thread per process. We can then execute all processes in a same schedule level concurrently. However, this could lead to some of the same shortcomings

of the previous work discussed in Section 2.3, if the processes do not share the same source code. Thus, to extract as much parallelism as possible, we devise a novel scheme in which the static schedule of the process graph is partitioned into multiple independent dataflows. These are then mapped to distinct multiprocessors for concurrent execution since those have distinct fetch units. The dataflows we create are segments of the scheduled process graph that can be executed independently. When necessary we may replicate some portions of the process graph to attain independence among dataflows.

The partitioning algorithm is outlined Figure 6. First, we select processes in the static schedule that do not activate any other process asynchronously, that is, they are root processes in the PG graph (line 4) (*e.g.*, $P8$ and $P9$ in Figure 5.1). For each of these nodes, we select their fan-in cone in the PG (line 5–12), as illustrated in the second step in Figure 5. Processes that are common to multiple cones are replicated (*e.g.*, the processes in the dashed circles in the Figure) so to make the cones independent of each other and to enable concurrent execution. Even though we need to replicate some portions of the PG, thus increasing the amount of simulation required, replication ultimately eliminates the need of communicating values among dataflows, thus leading to an important reduction in communication cost through device memory.

```
 1: list queue;
 2: for each node n ∈ V do
 3:    list current_dataflow;
 4:    if n has no exiting edges then
 5:       queue.add(n);
 6:       while queue is not empty do
 7:          Node current_node = queue.pop();
 8:          current_dataflow.add(current_node);
 9:          for all incoming edges edge of current_node do
10:             queue.add(edge.getSource());
11:          end for
12:       end while
13:    end if
14: end for
15: dataflow_list.add(current_dataflow);
```

Figure 6: Dataflow partitioning algorithm for *SAGA*'s step 2.

3.3 Parallel execution in CUDA

The cones 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, we now generate a total serial order of processes that satisfies the level-to-level dependencies.

First of all, we levelize the cones by following the algorithm in Figure 7. If the current node has no incoming edges (and thus it is not activated by any other process in the dataflow), then it belongs to the lowest scheduling level (lines 3–4). Otherwise, the node is scheduled at a level higher than that of all its fan-in processes (line 6-11). This step strengthens the dependency relation between processes (*e.g.*, in the example in Figure 5.3, not only *P3* and *P4* execute before *P7*, but also *P5* does). Then, processes in each dataflow are serialized, starting from the lower levels up to the root processes (processes at the same level can be executed in any sequential order). It is advantageous to create such sequential order for each dataflow, since it eliminates the need of frequent synchronization after each level. An example timeline obtained from this example is shown on the right hand side of Figure 5.3.

At this point *SAGA* generates the CUDA code corresponding to the generated process schedule. A *simulation kernel* manages dataflow execution, and it is constructed 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 body of each individual process is replaced by equivalent CUDA code, which might require translation of SystemC datatypes into native datatypes, as reported in Section 4.1. The simulation

```
 1: for each dataflow dataflow in dataflow_list do
 2:    for each node n in dataflow do
 3:       if n has no incoming edges then
 4:          n.setLevel(0);
 5:       else
 6:          n.setLevel(-1);
 7:       end if
 8:    end for
 9:    while at least one node has not been assigned a non-negative level
       do
10:       for each node n in dataflow do
11:          if for each incoming edge edge, the source node
             edge.getSource() has a non-negative level then
12:             for each incoming edge edge of n do
13:                if n.getLevel() ≤ edge.getSource().getLevel() then
14:                   n.setLevel(edge.getSource().getLevel() + 1);
15:                end if
16:             end for
17:          end if
18:       end for
19:    end while
20: end for
```

Figure 7: Dataflow levelization algorithm for *SAGA*'s step 3

kernel alternates execution with a *value-update kernel*, responsible for transferring the next-state values into the corresponding present-state values and performing testbench actions. A simulation cycle is completed by one execution of the simulation kernel followed by one execution of the update kernel.

Since device memory accesses are particularly slow, as indicated in Section 2.2, we allocate only variables written by synchronous processes in global memory, since their value must be persistent among different kernel executions. All other variables can be declared as local variables, and will consequently be mapped to registers with much faster access latency.

4. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of *SAGA*, provide insights on its intermediate data structure and compare it against other state-of-the art solutions in this space. Section 4.1 discusses our experimental setup; Section 4.2 compares *SAGA*'s performance against that of a sequential simulator, while Section 4.3 evaluates the performance of code compilation in *SAGA*. A comparison with other available solutions is provided in Section 4.4. Finally, Section 4.5 provides insights on the scalability of our solution.

4.1 Experimental setup

SAGA considers as input a SystemC design, it transforms it as discussed in Section 3, producing all the CUDA code necessary to run the corresponding simulation on a GPU as output. The code can then be off-loaded to a GPU platform and executed. All experiments discussed below were evaluated on a NVIDIA GTX480 GPU and a Intel quad core i7 operating at 2.8Ghz and running Linux RedHat 5.7. In addition, we leveraged the HIFSuite framework [3] to parse the SystemC code and generate an intermediate data structure that is used by *SAGA* for its internal transformations.

The first task in *SAGA* consists of considering a SystemC description and translating it into the HIFSuite's internal format (HIF) by using the HIFSuite *sc2hif* tool. The code generated at this point is a tree-structured XML-like representation of the original code, where semantic objects are represented with TAGS.

SAGA then applies a number of pre-processing steps to the HIF description. First it extracts all the processes and builds an initial dependency graph, according to signal dependencies among processes. It then applies the 3-step transformation described in Section 3. At this stage SystemC data types are substituted with native C/C++ data types and all corresponding data structures are built. Finally, *SAGA* generates the code for the kernel functions, and outputs the generated HIF description representing the detailed scheduled dataflows obtained with our algorithm. As a last step,

the resulting HIF code is converted into C code by means of the HIFSuite *hif2c* tool. This representation is ready to be compiled for the target CUDA architecture.

Table 1 presents our testbench designs. The designs are part of a complex embedded platform that was developed in the context of a European project together with silicon vendor industry partners:

- ECC is an error correction code device.
- ClockGen, ResGen, Sync and RegCtrl are part of a complex DSPI system. ClockGen is a multiple clock generator. ResGen transforms and outputs the computed results in the specified format. Sync is a specialized synchronization function among a number of components. RegCtrl is a register controller for a set of registers.
- 8b10b is a module performing encoding and decoding byte-wide data according to the 8b/10b protocol.

We evaluated *SAGA* on the individual designs and on two more complex SoC design assemblies: Half Platform, comprising ECC, ClockGen, ResGen and Sync; and Platform integrating all the designs previously discussed. For each design, Table 1 reports the number of processes in the original SystemC description (*Processes (#)*), the lines of code (*SystemC (loc)*) and the number of dataflows extracted (*Dataflows (#)*). Column *Replic. proc. (#)* reports the amount of code replication due to our step 2 (see Section 3) as number of replicated processes and the maximum amount of replication for these processes.

Design	Processes(#)		SystemC	Dataflows	Replic.
	Synch.	Asynch.	(loc)	(#)	proc. (#)
ECC	4	7	582	4	4 / 3
ClockGen	6	15	741	12	7 / 3
ResGen	3	6	478	9	0 / 0
Sync	4	22	641	23	0 / 0
RegCtrl	18	32	2677	43	17 / 8
8b10b	7	30	799	7	9 / 3
Half Platform	18	51	2355	48	11 / 3
Platform	42	112	5643	98	37 / 8

Table 1: Characteristics of the designs.

4.2 Performance

Table 2 compares *SAGA*'s performance with that of a SystemC sequential execution as discussed in Section 2. For each design, Table 2 reports simulation time of the SystemC simulation (Column *SystemC simul. (ms)*) and of the *SAGA*-generated CUDA code (Column *SAGA simul. (ms)*). It then reports their comparative performance in terms of *SAGA*'s speedup over sequential execution (Column *Speedup (x)*). The results show that the *SAGA* simulation is always faster than its corresponding SystemC sequential simulation. However, the speedup is moderate when comparing the small, individual component designs, leading to up to a 3.89 times improvement. Note, however, that even in presence of highly heterogeneous and complex processes *SAGA* achieves a respectable performance improvement. In addition the speedup achieved with the two more complex designs is much higher, ranging from 10 to almost 16x. This result suggests that *SAGA* is a promising solution that can extract even more concurrency from the more complex designs, where there are more processes available, leading to a better utilization of the parallel resources available on the GP-GPU.

The speedup achieved by *SAGA* is bounded by the amount of concurrency that can be extracted from each module and by the amount of computation they require. When both these factors are high, the generated code greatly outperforms sequential SystemC simulation. A low level of parallelism (ECC and ClockGen) or non-intensive computation (ResGen and Sync) lead to lower speedups, due to a heavier contribution of synchronization not balanced by computation, or because the limited concurrency is not offset by its setup overhead.

Our performance results also indicate that the benefits of replication far outweigh the costs. Indeed, as indicated in Table 2, even designs with the heaviest replication maintain a good speedup over a sequential simulator, since replication reduces communication by reducing the need of synchronization.

Design	SystemC simul. (ms)	*SAGA* simul. (ms)	Speedup (x)
ECC	11.99	5.05	2.37
ClockGen	18.00	7.13	2.52
ResGen	8.97	5.22	1.71
Sync	9.98	5.73	1.74
RegCtrl	41.97	13.05	3.21
8b10b	15.99	4.11	3.89
Half Platform	83.98	8.143	10.31
Platform	228.96	14.34	15.97

Table 2: Performance of *SAGA* vs. sequential simulation.

4.3 Compilation

Table 3 compares the costs of compilation for the target platform between a sequential simulator and our proposed *SAGA* flow. Column *SystemC comp. (ms)* reports the time needed to compile the original SystemC code. Column *Code generation* indicates the time needed to generate the target CUDA code in *SAGA*. For this component we report both the time spent in the execution of the *SAGA* algorithm presented in Section 3 (*SAGA (ms)*) and time required for the intermediate language transformations by the HIFSuite tools (*HIFSuite (ms)*). Finally, we show the time required to compile and generate the code to be off-loaded to the GPU.

The time spent in *SAGA* for code generation is a very small fraction of overall compilation time, which is dominated by the CUDA compiler and the HIFSuite transformations. Moreover, the total compilation time is within a factor of 2 of the compilation time of the sequential SystemC simulator. We expect that in a mature version of our solution, *SAGA* could parse and operate directly on the original SystemC source code, thus eliminating the need of resorting to the HIF intermediate format. Furthermore, since many simulations can be run for each model compilation, the value of *SAGA* does not lie in its compilation performance, but rather on the performance of the simulation generated.

Design	SystemC	Code generation		CUDA
	comp.(ms)	*SAGA*(ms)	HIFSuite (ms)	comp.(ms)
ECC	3,893	44	2,356	3,133
ClockGen	3,321	28	878	2,572
ResGen	2,863	16	864	2,457
Sync	3,027	24	180	2,608
RegCtrl	4,154	56	692	3,232
8b10b	3,354	40	3,284	2,936
Half Platform	965	101	3,850	6,431
Platform	10,960	187	7,428	6,824

Table 3: Comparison of compilation times for the sequential SystemC simulator and *SAGA*.

4.4 Architecture comparison

In order to show the effectiveness of the proposed methodology, we compared the performance of *SAGA* against two other concurrent solutions for SystemC simulation. For this study we report results on only two designs for sake of brevity. However, these two designs are representative of typical behavior and we found that the other designs lead to similar outcomes.

We first considered a concurrent SystemC simulator implementing the *SAGA* approach on a CMP architecture, where each dataflow is mapped to a different `pthread`. Furthermore, we compare the *SAGA* approach against the SCGPsim GPU-target simulation solution [6], which we implemented based on the authors' description, as outlined in Section 2. We report our findings in Table 4, where speedups are normalized to the performance of the sequential simulator.

The table indicates that *SAGA* is the fastest solution, providing a speedup of 2 to 4x over [6], and even more over the CMP design. Also note that the other solutions do not provide a performance improvement over the sequential simulation for ECC. Upon further inspection we found that the CMP solution does not achieve good concurrency because distinct processes are mapped to co-operative threads, as discussed in Section 1. SCGPsim's performance is limited because design processes do not share the same code, and thus they are executed sequentially when mapped on a same multiprocessor. We believe that the authors of [6] experienced much higher speedups because they evaluated their solution only on SystemC descriptions where processes had identical code. However, this is a very rare situation for any practical design.

Implementation	ECC		ClockGen	
	Time (ms)	Speedup	Time (ms)	Speedup
SystemC	11.99	1x	18.00	1x
Multiprocessor	94.00	0.13x	20.00	0.9x
SCGPsim [6]	20.08	0.59x	14.77	1.22x
SAGA	5.05	2.37x	7.13	2.52x

Table 4: Performance comparison of *SAGA* vs. a CMP concurrent simulator and the SCGPSim simulator.

In order to evaluate the speedup trends of *SAGA* against the solution in [6], we repeated a portion of functionality described in each process of the two designs, ECC and ClockGen, a number of times and then compared the simulation times achieved by all three solutions on these variants of the two designs. Figure 8 plots our findings; on the X axis we report the number of times that the functionality was repeated, and on the Y axis we indicate the corresponding simulation time. It can be noted from the graphs that *SAGA* outperforms SCGPsim and the sequential simulator even in this artificially larger designs with repeated functionality. However, note that the sequential simulator follows an exponential trend, as expected, while SCGPsim appears to follow a fairly constant trend, although with a higher baseline than *SAGA*.

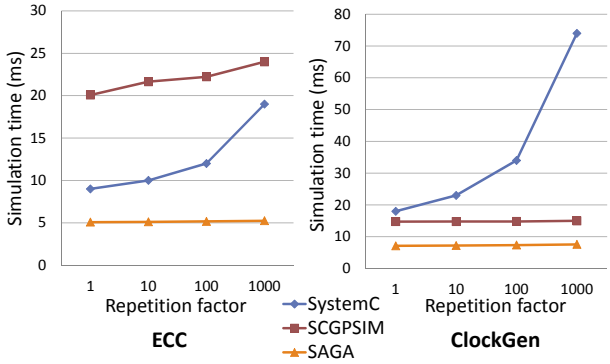

Figure 8: Simulation trends for SCGPsim and *SAGA*.

4.5 Scalability

To estimate the scalability of *SAGA*, we evaluated the GPU resource usage of our solution, so that we could determine when a SystemC design would reach a complexity that would exhaust the resources available on the GPU platform. A required resource in our solution is device memory; however, we only need to store there the values of the synchronous signals in the SystemC design and thus the usage of device memory is negligible, even for the largest designs (less than 1% of the available memory).

All other SystemC signals are stored as local variables and they exclusively require registers. As expected, the demand on registers is thus much more pressing, and these constitute a scarce resource that we need to consider in evaluating the scalability of *SAGA*. To this end we analyzed the register usage information for each of our testbench designs and used it to determine the maximum amount of dataflow concurrency that can be achieved. Figure 9 plots the fraction of concurrent resources used by our testbeds, based on this analysis. If a design reaches the limit of available concurrent resources, a portion of the computation will be serialized. Note, from Figure 9, that none of our designs reaches this limit, although the Platform testbench, our most complex design, was close, at 80%.

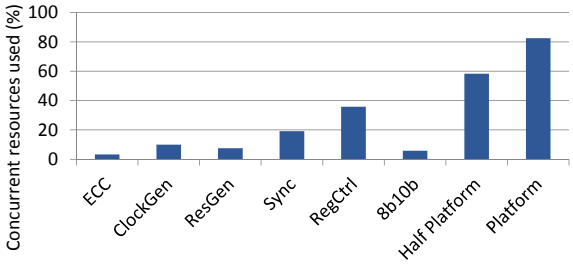

Figure 9: Percentage of available GPU concurrency required by our designs.

5. CONCLUSION

This paper presented *SAGA*, a novel solution for concurrent simulation of SystemC designs on GPU architectures. *SAGA* achieves its goal by extracting independent dataflows from a static schedule of SystemC designs, thus reducing synchronization overheads. As a result, the simulation is more efficient than both a sequential SystemC simulator and other state-of-the-art concurrent approaches. Experimental results show that we achieve the best speedups on complex designs, highlighting the effectiveness of the methodology when targeting complex industrial designs. Future work will focus on developing further optimizations for *SAGA* to further boost its performance and on evaluating the fitness of our solution for other hardware description languages.

6. REFERENCES

[1] P. Combes, E. Caron, F. Desprez, B.Chopard, and J. Zory. Relaxing synchronization in a parallel SystemC kernel. In *Proc. Of ISPA*, 2008.

[2] 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 DATE*, 2007.

[3] EDALab. *HIFSuite*, 2011. http://www.hifsuite.com/.

[4] P. Ezudheen, P. Chandran, J. Chandra, B. Simon, and D. Ravi. Parallelizing SystemC kernel for fast hardware simulation on SMP machines. In *Proc. of PADS*, 2009.

[5] L. Gwennap. Sandy bridge spans generations. *Microprocessor Report (www.MPRonline.com)*, September 2010.

[6] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. *Proc. of ASP-DAC*, 2010.

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

[8] Open SystemC Initiative. *SystemC Language Reference*, 2011. http://www.systemc.org/downloads/standards.

[9] N. Saviou, S. Shukla, and R. Gupta. *Design for Synthesis, Transform for Simulation: Automatic Transformation of Threading Structures in High Level System Models*. University of California at Irvine, 2008. Technical Report TR-01-58.

[10] H. Ziyu, Q. Lei, L. Hongliang, X. Xianghui, and Z. Kun. A parallel SystemC environment: ArchSC. In *Proc. of ICPADS*, 2009.