Parallel Simulation of Mixed-abstraction SystemC Models on GPUs and Multicore CPUs

Rohit Sinha, Aayush Prakash, and Hiren D. Patel

Electrical and Computer Engineering
University of Waterloo, Waterloo, Canada
e-mail: {rsinha, aayush.prakash, hdpatel}@uwaterloo.ca

Abstract—This work presents a methodology that parallelizes the simulation of mixed-abstraction level SystemC models across multicore CPUs, and graphics processing units (GPUs) for improved simulation performance. Given a SystemC model, we partition it into processes suitable for GPU execution and CPU execution. We convert the processes identified for GPU execution into GPU kernels with additional SystemC wrapper processes that invoke these kernels. The wrappers enable seamless communication of events in all directions between the GPUs and CPUs. We alter the OSCI SystemC simulation kernel to allow parallel execution of processes. Hence, we co-simulate in parallel, the SystemC processes on multiple CPUs, and the GPU kernels on the GPUs; exploit both the CPUs, and GPUs for faster simulation. We experiment with synthetic benchmarks and a set-top box case study.

I. Introduction

SystemC [1] is an electronic system-level design language that supports modeling and simulation of designs at register-transfer level (RTL), and at abstractions higher than RTL such as at the transaction-level (TL). It also enables the simulation of mixed-abstraction level models, and hardware/software models. The OSCI reference implementation provides a discrete-event (DE) simulation kernel that executes models specified in the SystemC language. In contrast to traditional hardware description languages such as VHDL and Verilog, SystemC's ability for higher abstraction modeling promises shorter design cycle times allowing designers to meet stringent time-to-market deadlines.

With a continued increase in the complexity and size of modern SystemC designs (both at RTL and TL), we find that SystemC has struggled to deliver on its promise for faster design cycle times. The central reason for this is the inability of SystemC's reference implementation to exploit parallel processing architectures that are commonplace in today's computing platforms. The DE simulation kernel uses co-operative multithreading resulting in a simulation kernel that cannot execute SystemC processes in parallel. Consequently, there is considerable research interest in discovering methods to expedite the simulation of SystemC models [2, 3, 4, 5, 6, 7].

These efforts parallelize the DE simulation kernel using either conservative or opportunistic methods. Conservative approaches process the events in the event queue in causal order, and opportunistic approaches allow violating this causality constraint such that processes can proceed executing events past the current simulation time. Opportunistic methods require support for rollback or an agreement that such causality violations are acceptable given a certain error bound. Chopard et al. [2], Schumacher et al. [3], and Chandran et al. [4] take the conservative approach, and they focus on the parallel simulation of synchronous SystemC models on multicore CPUs.

Alternatively, Mello et al. [5] and Jones et al. [6] use opportunistic methods where the focus is on accelerating the simulation of loosely-timed TL models. However, Mello et al. [5] do not support mixed-abstraction models. Instead, they only support a subset of TL models that do not use timed or immediate events. This prohibits mixed-abstraction simulation, and the reuse of existing TL models using other SystemC events. On the other hand, Jones et al. [6] do enable parallel simulation of mixed-abstraction RTL and TL models with full support of all SystemC events, but they require the user to explicitly describe temporal constraints to limit the temporal decoupling. This retains an acceptable degree of functional correctness. We find this to be an involved task for a large and complex SystemC model where ensuring functional correctness with tweaks on temporal constraints and quanta can become cumbersome.

Notice that these efforts do not identify nor leverage graphics processing units (GPUs) as a potential platform for parallel computation. GPUs are commodity components on general purpose computing platforms, and clearly an alternative platform for accelerating SystemC simulation as shown by Nanjundappa et al. [7]. However, they only support GPU execution of SystemC models at the RTL abstraction. More importantly, they do not co-simulate the SystemC models across both the multiple central processing units (CPUs), and GPUs.

Our work parallelizes and co-simulates mixed-abstraction SystemC models by supporting all SystemC events on both multiple CPUs and GPUs. We translate selected SystemC processes into GPU CUDA programs with the appropriate communication and event handling functionality. Our main contributions in this work are the following: 1) we parallelize the simulation of mixed-abstraction RTL and TLM (approximately and loosely-timed) SystemC models targeting multicore CPUs and GPUs while preserving the original DE semantics, and 2) we implement synchronization primitives in CUDA to enable co-simulation of RTL and TLM SystemC models on CPUs and GPUs.We experiment with synthetic benchmarks, and a case study that shows performance speedups of up to $41x$ without any compiler optimizations enabled, and $12x$ with compiler optimizations.

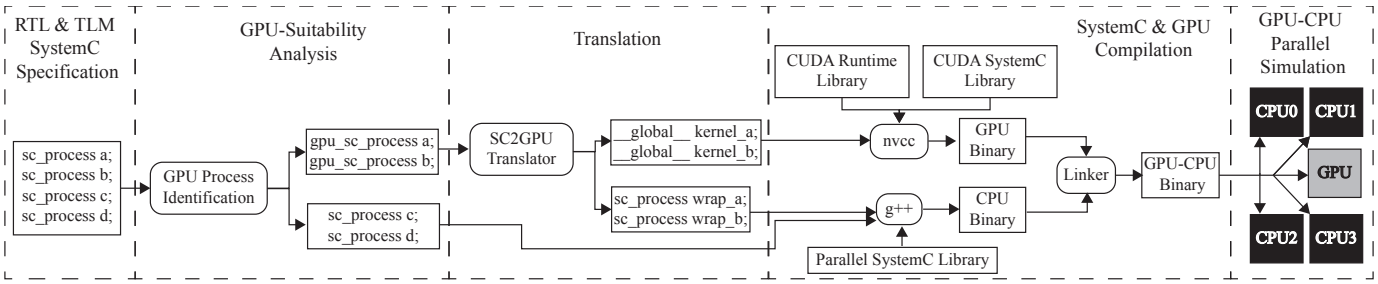


Fig. 1: Design flow overview of parallel GPU-CPU co-simulation of RTL & TLM SystemC models.

II. Related Work

Chopard et al. [2], Schumacher et al. [3], and Domer et al. [8] use a conservative approach for parallel simulation, but they primarily focus on synchronous SystemC models. Chopard et al. [2] execute a separate SystemC scheduler on a processing node, and synchronize at every delta cycle. Schumacher et al. [3], and Domer et al. [8] execute all processes made ready-to-run in parallel on multiple processing nodes. Chandran et al. [4] introduce a partitioning, and grouping of SystemC processes in addition to its parallel simulation on multicores.

Mello et al. [5] propose TLM distributed time (TLM-DT), and they provide a separate simulation kernel supporting temporal decoupling only supporting delta events. This approach, however, does not allow the simulation of mixed-abstraction RTL and loosely-timed TL models, which is an important component of the SystemC simulation methodology. Jones et al. [6] present an opportunistic approach to the simulation of RTL and TL models. Their approach instantiates multiple schedulers executing for a quantum called a time warp, and they provide interface extensions to limit the temporal decoupling caused by the loosely-timed execution. Determining the necessary temporal constraints and quanta for fast simulation is difficult; thus, we parallelize the execution conservatively.

Nanjundappa et al. [7] parallelize the simulation of SystemC RTL models on GPUs. They translate the SystemC model into CUDA code preserving the original DE semantics. However, their approach does not support TL models. We borrow concepts from Nanjundappa et al. [7]; however, we co-simulate mixed-abstraction RTL and TL models on multicore CPUs and GPUs.

III. Background

A. SystemC's Discrete-event Simulation Semantics

SystemC [1] implements the discrete-event semantics using the evaluate-update paradigm. The simulation kernel orchestrates the execution and synchronization of processes. The OSCI implementation is a single-threaded implementation. It starts simulation in an initialization phase where all SystemC processes are fired once. This causes other processes to become *ready-to-run*. After initialization, the evaluate phase begins where *ready-to-run* processes are executed until there are no remaining *ready-to-run* processes. This generates notifications on events, which makes other processes *ready-to-run*. For an immediate event, the process made *ready-to-run* executes within the same evaluate phase. When there are no immediate events, the simulation kernel enters the update phase where SystemC channels update computed values, which may again make other processes *ready-to-run*. The completion of an evaluate-update phase signifies the end of a delta cycle. If any of the events generated are delta events, then without forwarding the simulation time the kernel enters the evaluate phase. However, if there are no delta events, then the timed event with the earliest timestamp is chosen, and the simulation time advances to that timestamp. Then, the kernel enters the evaluate phase, and repeats this process until there are no remaining events.

B. CUDA C

Compute Unified Device Architecture (CUDA) is a computing engine developed by NVIDIA to perform general purpose computing on GPU architectures. The CUDA programming language is based on C, with extensions for single instruction, multiple data (SIMD) parallel computation and thread synchronization. The runtime framework offloads computation kernels to the GPU, and retrieves the results using memory copy operations. Our GPU-CPU co-simulation framework allows a SystemC developer to simulate data parallel computation on the GPU.

IV. GPU-CPU Co-simulation Methodology

A. Overview

Figure 1 presents an overview of the GPU-CPU co-simulation methodology. This consists of five stages. The first is the Specification stage where the designers use SystemC at RTL or TL abstractions to specify their design. The second stage is the GPU-Suitability Analysis stage where either through user-guided hints or through automatic analysis, a subset of the SystemC processes are identified as suitable for GPU execution. In Figure 1, processes a and b are chosen to execute on the GPU. The remaining processes c and d execute on the multicore CPUs. Currently, we use user-guided hints to identify this subset of processes. The next stage, the Translation stage, translates processes a and b into their equivalent GPU kernels and SystemC wrappers. The wrappers are central to enabling the communication between the SystemC processes on the CPUs and GPUs. Note that this requires us to provide synonymous implementations of SystemC's event paradigm supporting notifications of timed, immediate, and delta events for the GPU. The Translation stage also generates the necessary communication code for signalling on and across the GPU-CPU boundary,

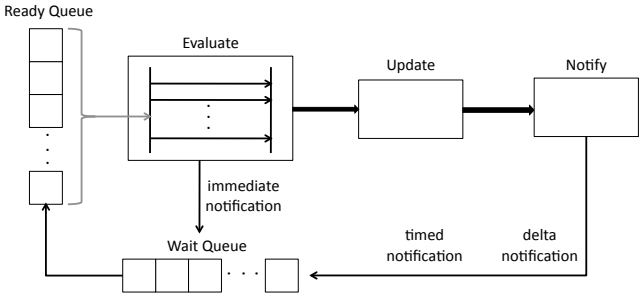


Fig. 2: Parallel SystemC simulation kernel.

and preserves the original semantics of the SystemC processes. Following translation, we enter the Compilation stage that compiles the processes and links them together into one GPU-CPU binary. Note that we compile the CPU processes with an altered parallel version of the SystemC library, and the GPU kernels with the CUDA SystemC library that contains the event handling mechanisms between the GPUs and CPUs, and the CUDA runtime library. The final Simulation stage takes this binary and co-simulates the SystemC design on multiple CPUs and GPUs. Although our framework implements co-simulation with multiple GPU devices, we present our methodology using only one GPU.

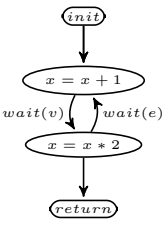
B. Parallel Simulation on Multicore CPUs

For the parallel simulation of SystemC processes on multicore CPUs, we take a conservative approach that uses the synchronous parallel DE systems approach [3]. During the evaluate-update cycles of SystemC's DE kernel, multiple processes are *ready-to-run* within a delta cycle. These *ready-to-run* processes can execute in any order. This means that they can execute in parallel on multiple CPUs. However, we ensure that all these processes complete their execution and wait at a barrier synchronization before entering the update phase. We also support parallel simulation of processes made *ready-to-run* via the notification of immediate events. Figure 2 illustrates the inclusion of immediate events in our parallel simulation kernel. Processes waiting on an event that are immediately notified get scheduled for execution within the same delta cycle. This amounts to adding the processes to the *ready-to-run* queue, and when all currently executing processes synchronize at the barrier, the newly readied processes start executing in parallel. During their execution, these processes generate events that are added into the event queue. We use shared locks to prevent race conditions on the shared event queue.

To address immediate notifications, we collect the notifications in the shared event queue, and allow all the processes executing in parallel to synchronize at the barrier. Note that we enforce mutual exclusion when all the immediate notifications access the shared event queue. Once this synchronization occurs, we allow the processes readied by the immediate events to execute in parallel. The kernel undergoes multiple iterations of dispatching multiple SystemC processes and joining them at the barrier synchronization before completing a delta cycle. The simulation proceeds to the *update* phase when there are no more *ready-to-run* processes in that delta cycle. From here on, the simulation kernel proceeds according to the reference implementation. We use primitives from the POSIX standard

```
1 gpu_sc_process
2   init();
3   while (true) {
4     x = x + 1;
5     wait(v);
6     x = x * 2;
7     wait(e);
8   }
9   return;
```

(a)



(b)

```
1  __global__ kernel
2    switch(stateId) {
3    case 0: init();
4            stateId = 1;
5    case 1: x = x + 1;
6            stateId = 2;
7            gpu_timed_wait(v);
8    case 2: x = x * 2;
9            stateId = 1;
10           gpu_event_wait(e);
11   }
```

(c)

Fig. 3: CUDA implementation of wait.

library to parallelize the simulation of SystemC processes on multicore CPUs with full support for all event notifications.

C. GPU-CPU Co-simulation

Our framework supports wait and notify primitives in the parallelized GPU and CPU simulation kernel. The GPU kernel communicates with a host SystemC thread, hereon called the wrapper thread, generated in the Translation stage. A GPU kernel (invoked from a wrapper thread) executes in parallel with other SystemC processes on multicore CPUs as well as other GPU kernels. We exploit task-level parallelism on both the CPUs and GPU, and we also exploit data-level parallelism on the GPU.

C.1 Translation Algorithm

Sharad et al. [9] propose a technique that converts SC_THREADs to SC_METHODs by generating a finite state machine equivalent to the original SC_THREAD. We borrow their technique in our translation algorithm. Each state transition denotes a wait statement in the original SC_THREAD specification. An SC_THREAD containing n wait invocations translates to a finite state machine containing n states. Figure 3b illustrates the transformation for the gpu_sc_process in Figure 3a. Since SystemC SC_THREADs typically contain an infinite loop, the state machine contains a transition from the final state to the first state within the loop. Their translator synthesizes a counter variable (stateId in Figure 3c) that signifies the next state. The code snippet in Figure 3c shows the translation for the gpu_sc_process in Figure 3a.

Algorithm 1 describes the steps involved in the translation. The internals of code generation are encapsulated within the $codegen\_wrapper$ and $codegen\_gpu$ functions, which synthesize code for the wrapper process and the GPU kernel respectively. A gpu_sc_process contains a sequence of statements S, where each statement is one of three types: notify, wait, or any valid SystemC statement. Line 9 of Algorithm 1 synthesizes GPU kernel code for serializing the values of local variables into GPU global memory. Section C.3 describes the need for this step. In lines 10 – 14, the translation replaces each timed wait with gpu_timed_wait, and event wait with gpu_event_wait. In addition, the translation generates an $\langle e, v, t \rangle$ tuple in GPU global memory for each gpu_sc_process. The functions gpu_timed_wait and gpu_event_wait update this tuple, and their implementation exists within the CUDA SystemC library. The translation stage also replaces notify with

Algorithm 1: $translate(P)$

```
   /* Initial declarations                                   */
 1 Let P be the set of gpu_sc_processes.
 2 Let S be a sequence of program statements.
 3 Let ⟨e, v, t⟩ be a tuple where e is the event identifier, v is the
   time, and t is the event type.
 4 foreach p ∈ P do
 5     codegen_wrapper(gpu_kernel_init())
 6     S ← getStatements(p)
 7     foreach s ∈ S do
 8         if isWait(s) then
 9             codegen_gpu(gpu_save_reload_context(p))
10             if t = TIMED then
11                 codegen_gpu(gpu_timed_wait(e, v))
12             else if t = EVENT then
13                 codegen_gpu(gpu_event_wait(e))
14             end
15             codgen_wrapper(memcpyFromGPU())
16             codegen_wrapper(invokeNotify())
17             codegen_wrapper(invokeWait())
18         else if isNotify(s) then
19             Let ⟨e, v, t⟩ ← parseNotify(s)
20             if t = TIMED then
21                 codegen_gpu(gpu_timed_notify(e, v))
22             else if t = DELTA then
23                 codegen_gpu(gpu_delta_notify(e, v))
24             else if t = IMMEDIATE then
25                 codegen_gpu(gpu_immed_notify(e))
26             end
27         else
28             codegen_gpu(s)
29         end
30     end
31 end
32 return
```

their CUDA equivalent statement. In lines 20 - 26, we replace delta notifications with gpu_delta_notify, timed notifications with gpu_timed_notify, and immediate notifications with gpu_immed_notify. In addition, for each gpu_sc_process, the translation generates a set of $\langle e, v, t\rangle$ tuples, one for each sc_event in the SystemC model. *codegen_wrapper* synthesizes code for the wrapper process, which manages the interaction between the multicore CPU processes and the GPU kernel. Lines 15 - 17 of algorithm 1 generates code to initialize the GPU kernel and transfer the set of $\langle e, v, t\rangle$ tuples from GPU global memory. The details of this interaction are presented in section C.2. For statements within S that are neither notify nor wait, line 28 in the algorithm converts the System statement to a valid CUDA statement.

C.2 GPU-CPU Interaction

An SC_THREAD typically generates notifications prior to suspending itself using a wait (on an sc_event or for some period of simulation time). A GPU kernel, however, cannot suspend its execution; hence, we relinquish control to the host wrapper process on each wait. The wrapper process copies the result of the computation performed by the GPU kernel, and then invokes the SystemC version of wait. Whenever the wrapper process becomes *ready-to-run*, it resumes the execution of the GPU kernel from its next instruction as explained in section C.3. The preceding set of interactions occur for each instance of wait, until the gpu_sc_process invokes return.

C.3 GPU Implementation of wait

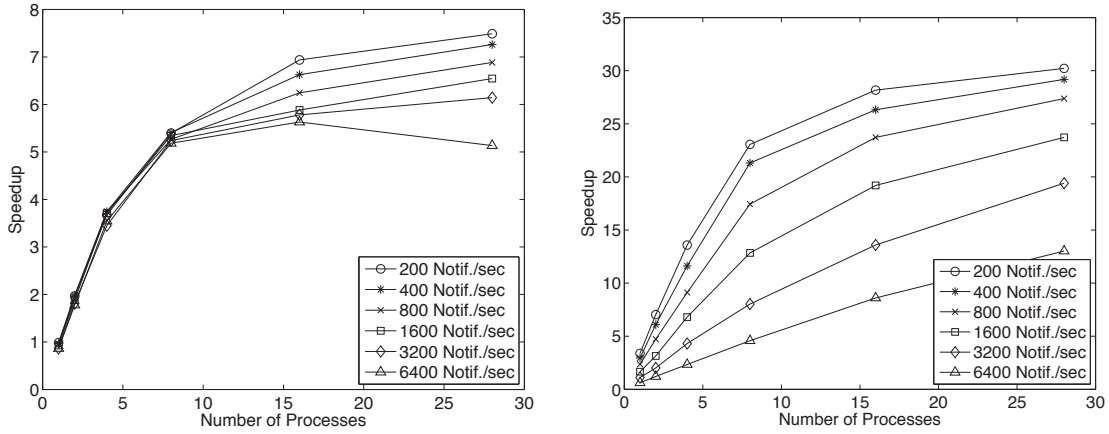
On invoking wait within a GPU kernel, the kernel invokes either gpu_timed_wait or gpu_event_wait. These functions store the event identifier e that is null for timed events, the wait type t that is either TIMED or EVENT, and the wait value v that is a double representing the time. After this, the kernel relinquishes control to the CPU wrapper process. The wrapper process first fetches $\langle e, v, t\rangle$ from the GPU global memory, and then invokes wait(v) if t is TIMED, and wait(e) if t is EVENT. While this wrapper process waits, other GPU kernels and processes on the CPUs continue execution. When the wrapper process resumes execution, $\langle e, v, t\rangle$ is reset in the GPU memory and the GPU kernel executes. However, this invocation of the GPU kernel transitions to the instruction after the gpu_timed_wait or gpu_event_wait. We use a program counter to identify the next instruction. The GPU kernel in Figure 3c uses stateId as a program counter. We serialize values of all local state variables updated during execution of the GPU kernel to a preallocated region of the GPU global memory before returning control to the wrapper process. Upon re-invocation, the GPU kernel reloads the execution state from global memory. In Figure 3c, x and stateId are saved and reloaded on each kernel invocation.

C.4 GPU Implementation of Delta and Timed Notifications

A notification in a GPU kernel invokes gpu_delta_notify, gpu_timed_notify or gpu_immed_notify. The gpu_delta_notify and the gpu_timed_notify store e, v and t in a data structure in GPU global memory. When there are multiple calls to notify before a single wait, we store multiple $\langle e, v, t\rangle$ tuples in the GPU global memory. An invocation of wait causes the CPU to copy these tuples from GPU global memory, and process the corresponding notifications. Multiple timed notifications to the same sc_event prior to a wait store the minimum time value amongst the notifications. After processing all notifications, the tuples signifying the notifications are reset in preparation for executing the next block of statements in the GPU kernel.

C.5 GPU Implementation of Immediate Notifications

An immediate notification causes any process waiting on the event to immediately become *ready-to-run*. During simulation, a GPU kernel records a notification by updating the entries in the tuple $\langle e, v, t\rangle$. The wrapper copies these tuples from GPU after the GPU kernel exits via gpu_timed_wait or gpu_event_wait, and then invokes notify. We allow fetching the notifications from the GPU memory before the kernel's exit. Recall that there is a set of tuples $\{\langle e, v, t\rangle\}$ for each gpu_sc_process. When any GPU kernel finishes executing, its wrapper process fetches the tuples for all other GPU kernels as well. That wrapper process then invokes the immediate notifications made by all gpu_sc_process. To prevent firing the same notification twice, the simulation kernel logs the immediate notifications made by each gpu_sc_process. When



(a) Multicore CPUs parallelization.

(b) GPU-CPU parallelization.

Fig. 4: Results for synthetic benchmarks.

a gpu_sc_process invokes wait, its tuple entries are compared with its log to prevent firing the same notification twice.

C.6 Optimization: Concurrent Kernel Execution

In addition to executing processes in parallel on multicore CPUs, we perform GPU operations concurrently using CUDA streams. The CUDA programming model defines a stream to be a sequence of commands, which execute in order. Furthermore, different streams may execute commands concurrently. For instance, a memory transfer associated with one GPU kernel can simultaneously occur with a different GPU kernel execution. The advantage is that memory transfers do not prevent other GPU kernels from executing. Moreover, a CUDA-enabled GPU can execute multiple kernels in parallel. For each GPU kernel, we allocate a distinct CUDA stream connecting the wrapper process and the GPU kernel. The performance boost is not only a result of parallel execution, but also the high memory bandwidth in NVIDIA GPUs. Moreover, NVIDIA GPUs implement hardware multithreading to enable cheap context switching, and thus hide the latency of global memory accesses.

V. Synthetic Benchmarks

We perform our benchmarks on an Intel Xeon E5645 processor (12 cores) and NVIDIA Tesla C2075 graphics card connected via PCI Express using Linux 2.6.32-33 operating system. The synthetic benchmarks allow us to evaluate our system with a set of generic SystemC processes. These processes communicate using wait and notify, and model computation using dummy computation. We vary the number of SystemC processes, and the number of generated notifications (measured in notifications per second). The simulation kernel measures the number of notifications.

Figure 4a shows the parallel execution of the synthetic benchmarks on the multicore CPUs with the baseline as the OSCI SystemC 2.2.0 reference implementation. Suppose the benchmark contains n SystemC processes. Then, half of the processes generate notifications, and wait. The other half of the processes wait on the notification of these events, and upon notification, they perform some dummy computation and wait again. This notification and waiting continues. Notice that initially the speedup increases as the number of processes increase. However, increasing the number of processes beyond the number of CPUs available (twelve in our platform) shows the tapering off the speedup improvements. In addition, increasing the number of notifications shows a reduction in the speedups, which is primarily because of the increased number of process waits.

Our GPU-CPU experiment uses a CUDA enabled NVIDIA GPU. Figure 4b uses the same experimental setup as before with half the processes mapped onto the CPUs and the other half onto the GPU. For the processes mapped onto the GPU, we scale the dummy computation by a factor k. This factor k shows the speedup of the computation part of the code. In our experiments, we set $k = 10$, which means that the computation code of a process executing on the GPU executes only ten times faster than on the CPU. Note that this is a conservative measure for the improvement offered by using GPUs, and it is possible to obtain greater speedups [7]. The speedups in Figure 4b are higher than those achieved in Figure 4a because of the increased task-level and data-parallel parallelism. Once again, as the number of notifications per second increase, the speedups decrease, which is primarily because of increased overhead of data transfer between CPU and GPU via PCI Express. This helps in efficient partitioning of the SystemC designs.

VI. Case Study: Multi-channel Set-top Box

We implement a set-top box case study modeled after the Scientific Atlanta's 8300 PVR with multiple video input channels as shown in Figure 5. It has three channels, two for viewing, and the third for recording to disk. Each channel corresponds to a different video stream. We partition this case study manually. We map the bilateral filtering to the GPU because it can exploit data-level parallelism. There are other blocks that we can also map to the GPU such as inverse DCT to further show improvements. MPEG2 decoding, on the other hand, is a highly control flow intensive operation, which is suitable for

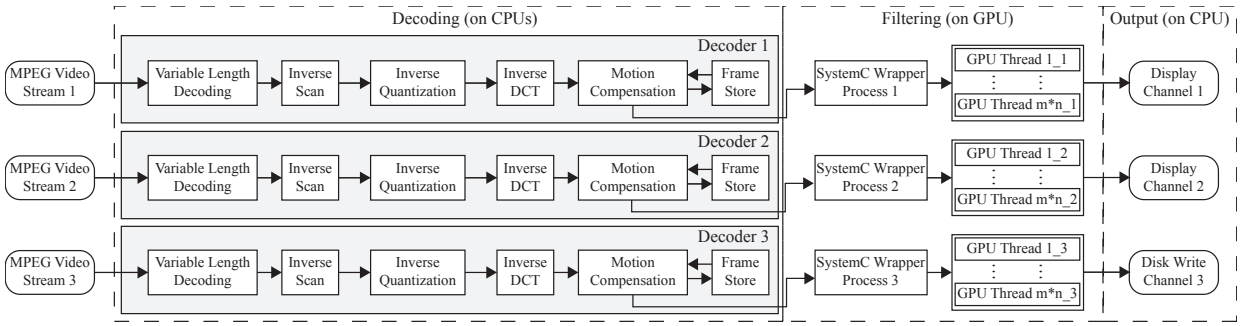


Fig. 5: Case study: Multi-channel set-top box.

TABLE I: Results of the experimentation on the case study.

Case Study	CPU-seq	CPU-parallel		GPU-CPU	
	Time(sec)	Speedup	Time (sec)	Speedup	Time (sec)
No Opt.	350.37	2.7x	127.49	41.0x	8.49
With Opt.	108.01	2.8x	39.10	12.6x	8.59

execution on CPUs. Our implementation has three stages. In the first stage, we read an MPEG2 video from a streaming input channel and decode it using SystemC processes executed on the CPUs. This stage includes inverse scanning, quantization, discrete-cosine transform, and motion compensation. The second stage enhances the image via bilateral filtering, which we execute on the GPU as a separate GPU thread. We further parallelize the processing of each frame at the pixel-level to exploit the data-level parallelism of GPUs. The final stage displays the processed frames to the respective outputs via SystemC processes on the CPUs. We validate the results by comparing outputs of sequential and parallel simulation framework for each frame of the video stream. We show the performance results for this case study in Table I with (With Opt.) and without compiler optimizations (No Opt.). Note that optimizations in the synthetic benchmarks are ineffectual due to the modelled dummy computation; however, for the case study it is important. As a result, we report performance numbers without enabling optimizations, and those with -O3 optimization level enabled. The column with GPU-seq shows the execution times using OSCI's reference implementation of SystemC, and the column with CPU-parallel is our implementation of SystemC parallelization as described by Schumacher et al. [3], and the GPU-CPU column is our approach that allows co-simulation on both the multicore CPUs and the GPUs. The GPU-CPU co-simulation provides a speedup of up to approximately $12x$ for the multi-channel set-top box case study when compared to using OSCI's reference implementation of SystemC with all optimizations enabled. When compared to the CPU-parallel, the co-simulation provides $5x$ speedup. We find that the case study results in Table I are indicative of the case study not fully exploiting the parallelism offered by the GPU. We expect case studies with larger partitions of the SystemC processes mapped to GPUs to further leverage the benefits of GPU parallelism. Furthermore, notice this approach allows SystemC models with a high level of parallelism to use all the multicore CPUs on the platform, and the resources on the GPU together.

VII. Conclusion

We present a methodology that parallelizes mixed-abstraction SystemC models across multicore CPUs and GPUs. In doing this, we parallelize the SystemC kernel, and provide a synchronization library for handling SystemC events on the GPU. These events are transferred to the main SystemC scheduler so as to preserve the discrete-event semantics of SystemC. We also present a translation algorithm that converts selected SystemC processes to GPU CUDA kernels. For experimentation, we compare our approach against a previously proposed technique to parallelize the execution of SystemC models, and OSCI's reference implementation of SystemC. We also illustrate that co-simulating models across the multicore CPUs and GPUs can offer performance advantages by utilizing compute resources on the GPU in addition to the multicore CPUs. For future work, we intend to investigate automatic GPU suitability analysis.

References

[1] Open SystemC Initiative, "SystemC," http://www.systemc.org.

[2] B. Chopard, P. Combes, and J. Zory, "A conservative approach to systemc parallelization," in *Computational Science ICCS 2006*, ser. Lecture Notes in Computer Science, V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2006, vol. 3994, pp. 653–660.

[3] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: synchronous parallel systemc simulation on multi-core host architectures," in *Proceedings of ACM International Conference on Hardware/software Codesign and System Synthesis (CODES/ISSS)*, 2010, pp. 241–246.

[4] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing systemc kernel for fast hardware simulation on smp machines," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009, pp. 80–87.

[5] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 606–609.

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

[7] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: a fast SystemC simulator on GPUs," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, 2010, pp. 149–154.

[8] R. Dömer, W. Chen, X. Han, and A. Gerstlauer, "Multi-core parallel simulation of system-level description languages," in *Proceedings of Asia and South Pacific Design Automation Conference*, 2011, pp. 311–316.

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