SystemC Simulation on GP-GPUs: CUDA vs. OpenCL *

Nicola Bombieri
Dip. Informatica
Università di Verona, Italy
nicola.bombieri@univr.it

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

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

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

ABSTRACT

SystemC is a widespread language for developing SoC designs. Unfortunately, most SystemC simulators are based on a strictly sequential scheduler that heavily limits their performance, impacting verification schedules and time-to-market of new designs. Parallelizing SystemC simulation entails a complete re-design of the simulator kernel for the specific target parallel architectures. This paper proposes an automatic methodology to generate a parallel SystemC simulator kernel, exploiting the massive parallelism of GP-GPU architectures. Our solution leverages static scheduling to reduce synchronization overheads. The generated simulator code targets both CUDA and OpenCL libraries, to boost scalability and provide support for multiple GP-GPU architectures. Finally, the paper compares the performance of our solution on CUDA vs. OpenCL platforms, with the goal of investigating advantages and drawbacks that the two thread management libraries offer to concurrent SystemC simulation.

Categories and Subject Descriptors

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

General Terms

Design, Performance

Keywords

Parallel SystemC, simulation acceleration, CUDA, OpenCL

1. INTRODUCTION

SystemC is the de-facto standard language for system-level modeling, architectural exploration, performance analysis, software development, and functional verification of embedded systems [1].

*This work has been partially supported by EU project FP7-ICT-2011-7-288166 (TOUCHMORE) and the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

CODES+ISSS'12, October 7Ű12, 2012, Tampere, Finland.

It supports modeling and simulation of designs at different abstraction levels, from register transfer level (RTL) to electronic system level (ESL) through the transaction level modeling (TLM) [6].

Such a flexibility to model from the most accurate to the most abstracted description level simplifies system design, as well as verification, by providing a simpler, more powerful way to verify complex systems. Moreover, by exploiting reuse-based methodologies, IP cores, complete of their verification infrastructure, can be re-deployed across several abstraction levels, thus enabling shorter design cycles [2, 3, 11].

Nevertheless state-of-the-art SystemC simulation kernels rely on application-level threading (co-operative threads) forcing the execution to be intrinsically sequential, since the operating system cannot dispatch co-operative threads to different processing elements. Such inability of SystemC kernels to exploit parallel processing architectures has been addressed only recently. Several works in the literature have attempted to parallelize SystemC simulation, targeting heterogeneous architectures to reduce synchronization overheads [5, 9, 12].

Concurrently, general purpose graphics processing units (GP-GPUs) have been explored in the past few years as a new general purpose computing paradigm for accelerating computation-intensive EDA applications, such as gate-level fault simulation [6], fault table computation [7], and logic simulation [8]. SystemC simulation on GP-GPUs has been explored very recently, and it is proving to be a promising match between the most widespread system-level design language and these up-and-coming low cost devices [13, 19]. Early experimental results have shown SystemC simulation speed-ups from 3x to 2,000x, depending on testbench design and GP-GPU board model. Nevertheless, all these works target CUDA [14] GP-GPUs, thus limiting their applicability to only the NVIDIA family of many-core devices.

To allow execution of parallel programs across various platforms, the Khronos Group [17] has established an open standard, known as OpenCL. OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processing elements. As a result, OpenCL provides software developers with portable and efficient access to the power of diverse processing platforms.

An open question on this front is whether the portability in OpenCL impacts application's perfomance, as it is often the case for languages and middlewares that target portability across different platforms [18]. If performance were to suffer significantly on OpenCL platforms, their suitability would be limited.

To investigate the tradeoff between performance and portability of OpenCL, we present a detailed analysis using a range of Sys-

temC designs and compare simulation performance of CUDA vs. OpenCL running on several NVIDIA's devices.

The remainder of this paper is organized as follows. Section 2 reports the related work. Section 3 summarizes and compares CUDA and the OpenCL programming models. Section 4 discusses the implementation of a C++ SystemC kernel for CUDA and OpenCL, by comparing different scheduling solutions. Section 5 presents compares simulation performance on CUDA and OpenCL. Finally, experimental results are reported in Section 6, while Section 7 concludes our work.

2. RELATED WORK

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).

Within each simulation 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.

In SystemC, the order of process execution within an evaluation phase does not affect simulation output since the simulator presents the same system status to all those processes. Thus, processes activated within a same evaluation phase can be naturally executed in parallel, either by using multiple threads or by designing a distributed scheduler.

Several works in the literature suggest taking advantage of this inherent parallelism to speedup simulation [7, 9, 15, 20]. In [15], SystemC processes are executed as distinct threads on multiple CPUs. However, the overall simulation platform (ArchSim) introduces heavy overhead, thus making this approach ineffective overall. In [7], each processing node includes a copy of the scheduler and it simulates a subset of the application modules. All scheduler copies must synchronize at each signal and event update phase, 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 [15]. The methodology analyzes SystemC modules and it divides all processes into blocks of operations that are executed consecutively. Then, blocks are scheduled according to their data and control dependencies, parallelizing those blocks that can be executed during the same simulator phase In summary, all these solutions rely on SystemC source code modifications or introduce overhead.

SystemC simulation on CUDA GP-GPUs has been proposed for the first time in [13]. In that work, independent SystemC processes are mapped into parallel threads that synchronize at each iteration of a delta cycle, through a barrier synchronization, to maintain the correct producer-consumer relation among threads. The same authors then proposed [16], a methodology that parallelizes the simulation of mixed-abstraction level SystemC models across multicore CPUs and GPUs. Given a SystemC description, the methodology partitions the model into processes suitable for GPU and CPU execution. Then, it converts the processes identified for GPU execution into GPU kernels with additional wrapper processes to invoke those kernels. The wrappers enable seamless bi-directional communication of events between GPUs and CPUs. The authors reworked the OSCI SystemC simulation kernel to allow parallel execution of processes.

A framework for simulating SystemC code on CUDA GP-GPUs has been also proposed in [19]. The framework fully exploits the intrinsic parallelism of RTL SystemC descriptions, by limiting synchronization events with ad-hoc static scheduling and separate independent dataflows. Finally, SystemC simulation on CUDA has been proposed in [4], where the authors propose a framework for functional verification of RTL designs. The framework translates the RTL code into C code targeting NVIDIA GPUs, thus allowing fast parallel automatic test pattern generation and fault simulation.

All the previous works applied SystemC simulation to CUDA GP-GPUs, while, to the best of our knowledge, there is no work that has analyzed the performance of SystemC simulation by adopting OpenCL.

3. GP-GPU PROGRAMMING

The following Sections present CUDA and OpenCL (Section 3.1 and 3.2). Then, Section 3.3 compares platform models, memory models, execution models and programming models of the two frameworks.

3.1 CUDA

NVIDIA's Compute Unified Device Architecture (CUDA) [14] has been proposed to facilitate GP-GPU programming with a general purpose interface. In the CUDA execution model, the GP-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 GP-GPU for execution. The collection of threads represented by a kernel is divided into a grid of thread-blocks.

The CUDA architecture (Figure 1) consists of a number of multiprocessors within a single GP-GPU chip. Multiprocessors are responsible for the execution of the thread-blocks mapped to each of them. Each multiprocessor comprises multiple stream processors with common instruction fetching and support for a large number of concurrent threads. Thus, each multiprocessor executes several groups of threads at a time (known as a *warp*) in a time-multiplexed fashion, with frequent context-switches from one warp to another. Because of the shared fetch unit, execution path divergence between threads in a same warp is detrimental to performance as only one branch path can be executed at a time. Thus, if threads in a same multiprocessors must execute different code paths, the least penalizing solution is to map them to different warps.

Each multiprocessor has access to low latency scratchpad memory, divided between local registers 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 via lower bandwidth direct memory access (DMA) transfers. Thus, it is important to keep communication between the host and the GP-GPU to the bare minimum.

3.2 OpenCL

OpenCL is a standard for general purpose programming across heterogeneous processors developed by the Kronos group since 2008. OpenCL supports vendor-independent architectures, ranging from CPUs to GP-GPUs and matching the OpenCL platform structure.

An OpenCL program is divided into host code and device code. The host program prepares the device execution and it transfers data to the device. The device code is structured in *kernels*, pro-

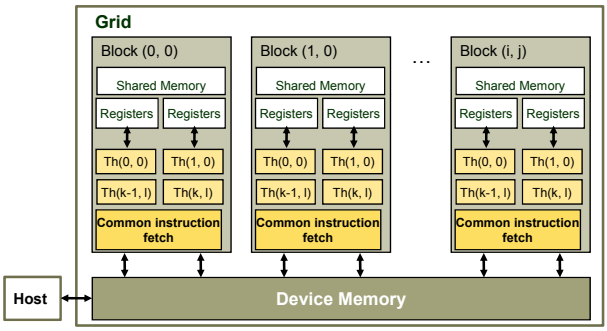

Figure 1: NVIDIA CUDA architecture

grams that perform the functionality of the target application. Each thread is associated with an instance of the kernel, called work-item, and is identified by an ID. Work-items are grouped into work-groups, that execute concurrently on the processing elements of a single compute unit. As a result, threads belonging to the same work-group execute the same instruction, in a SIMD flavor.

When offloading code to the device, the host must define a *context*. Each context is made of a set of devices where execution occurs and by the kernel that must be executed. Furthermore, a context contains a reference to the program source code and to the memory objects that are visible from both the host and the devices. A command queue is created to coordinate execution among contexts on the devices.

Each compute unit has access to low latency memory, that can be shared between the different processing elements. In addition, there is a region of memory, called global memory, that can be accessed by all multiprocessors. Communication with the host CPU's main memory is achieved by means of data transfers (added to the command queue) or shared pointers. However, data sharing with the host introduces a heavy overhead, and thus communication between host and device must be minimized.

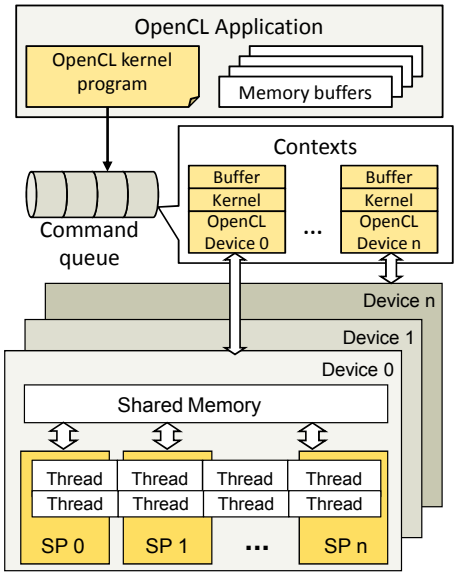

Figure 2: Typical OpenCL architecture

3.3 CUDA-OpenCL comparison

As outlined in the previous sections, OpenCL and CUDA share their core ideas as they have similar platform models, memory models, execution models and programming models. To the programmer, the computing system consists of a host (e.g., a typical CPU) and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units.

However, OpenCL targets a wider range of architectures, while CUDA is restricted to NVIDIA's GP-GPUs. This implies that OpenCL must be more flexible and it can not take into account specific properties of the architecture, such as the availability of read-only memory. This consideration impacts execution performance and thus a number of aspects should be taken into account to achieve a fair comparison:

- CUDA provides optimization pragma directives to automatically improve the code. An example is the `unroll` pragma, that applies unrolling to loops, thus reducing branch prediction failures and the presence of dynamic instructions. Once again, OpenCL can not exploit such optimizations. When removing pragma support, CUDA code may become slower than OpenCL [10];

- CUDA assumes the availability of fast-access read-only memory, such as texture memory and constant memory. Applications can exploit this architectural characteristics to reduce memory access overhead [10]. On the other hand, OpenCL can not make assumptions on the underlying platform and thus memory handling is less optimized;

- CUDA compilers are more mature than OpenCL ones. As a result, OpenCL compilers generate up to two times more arithmetic instructions than their CUDA counterpart [10]. This difference can not be eliminated and it is implied by the open source nature of OpenCL;

- Unlike CUDA, OpenCL requires environmental setup before launching the kernels on the GP-GPU. This process includes selecting the target device, by querying the available resources, and compiling the kernels at runtime. As a result, OpenCL's initialization cost includes compilation cost and lead to kernels setup taking longer than kernels execution [8]. However, OpenCL set up costs are fixed and thus it can be amortized on complex applications.

We took the above into consideration, and strove to create a fair comparison between the two platforms. We do not use pragma optimizations, nor read-only memory accesses in our analysis. However, the OpenCL simulator is still penalized by initialization overhead and less optimized compilers.

4. SYSTEMC TO C++ TRANSLATION TARGETING GP-GPUs

Mapping of SystemC applications to GP-GPU architectures is a non trivial task, since it requires scheduling support and the ability to preserve correct simulation outputs, even when the processes are executed in parallel. In the literature, two main approaches have been proposed to address this issue. [13] applies dynamic scheduling to the GP-GPU framework, by executing individual threads in parallel only when the testbench structure allows it and by synchronizing after each simulation step. On the other hand, [19] proposes a static scheduling approach that avoids frequent synchronization steps by partitioning the starting SystemC processes into independent sets that execute on different multiprocessors. The proposed techniques are outlined in Section 4.1 and Section 4.2 respectively, while Section 4.3 compares the two.

4.1 Dynamic scheduling

The work proposed in [13] maps SystemC processes to a model of concurrent threads that synchronizes after each simulation step. Each SystemC process is assigned to one thread and all threads belong to the same thread block, to gain fast synchronization. In this way, barriers can be exploited to synchronize threads, to update memory and to maintain the correct producer-consumer relation among threads. An example of this approach is shown in Figure 3. Grey-colored threads are those performing meaningful computation. White-colored threads are waiting on a barrier for the other threads to complete their job.

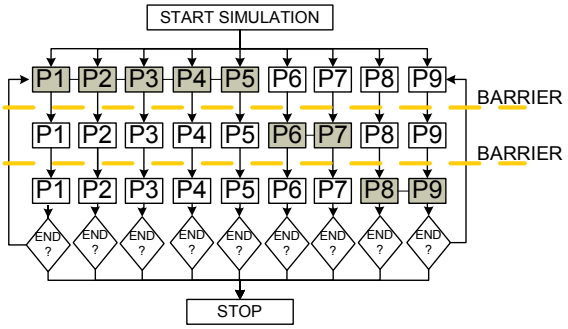

Figure 3: Scheduling example based on SCGPSim

4.2 Static scheduling

SAGA [19] proposes a different approach based on static scheduling. The read-write dependencies between the starting SystemC processes are analyzed to generate a dependency graph of the processes (left hand side of Figure 4). The graph is then topologically sorted and partitioned into independent dataflows, i.e., portions of the process graph that can be executed independently (right hand side of Figure 4). Such dataflows are then mapped to distinct multiprocessors for concurrent execution. When necessary, some portions of the process graph may be replicated to attain independence among dataflows (as happened to processes $P3$, $P4$ and $P7$ in Figure 4).

The dataflows built in the previous step are process dependency trees, that must be executed level-by-level to respect the internal dependency constraints. Thus, for each dataflow obtained in the previous step, a total serial order of processes must be built, with the goal of satisfying the level-to-level dependencies.

Processes in each dataflow are serialized, starting from the lower levels up to the root processes of the dependency graph (processes at the same level can be executed in any sequential order). Such sequential order eliminates the need of frequent synchronization after each level. An example timeline obtained from this example is shown on the right hand side of Figure 5.

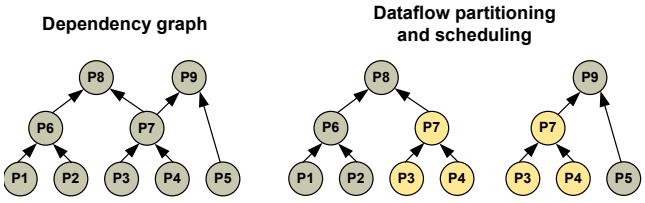

Figure 4: Application of the SAGA methodology to the same example testbench of Figure 3

4.3 Dynamic vs. static scheduling comparison

The proposed approaches to SystemC simulation on GP-GPU architectures have many differences that affect overall performance.

The dynamic scheduling approach of SCGPSim [13] implies an overhead for both handling of events and frequent synchronization barriers. The left hand side of Figure 5 highlights the frequency and the overhead induced by dynamic scheduling, with respect to the static approach (right hand side of Figure 5). Static scheduling has been adopted in SAGA to avoid such synchronization issues, while still preserving the dependencies between processes. The static approach was shown to be more efficient also on complex designs [19].

Both the scheduling approaches have also a heavy impact on thread management, as outlined in Figure 5. Dynamic scheduling requires fast synchronization primitives called barriers, available only when the synchronizing threads belong to the same thread block. However, if this is the case (as in SCGPSim), processes are serialized (as highlighed on the left hand side of Figure 5). Indeed, SystemC processes tend not to share the same code, generating a path divergence in the instructions executed by each thread. On the other hand, static scheduling can adopt heavier synchronization mechanisms, due to less frequent synchronization. Thus, the executing threads can belong to different blocks and still achieve concurrent execution (as highlighed on the right hand side of Figure 5).

As a result, static scheduling is a winning approach to speed up simulation, both for better thread management and reduced need for synchronization. Thus, we adopt static scheduling in our solution.

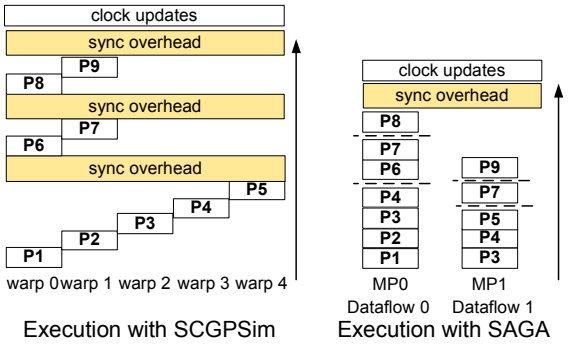

Figure 5: Comparison of actual execution schedule when using SCGPSim (on the left) vs. SAGA (on the right)

5. PARALLEL EXECUTION ON GP-GPUs

In this paper, we adopt SAGA to determine the static scheduling and to generate the target code. Both OpenCL and CUDA maintain the same code structure, with two separate kernels executed on the GP-GPU. A *simulation kernel* manages dataflow execution, and it is generated by listing all the dataflows and predicating each by a thread-block ID condition, so that only a specific thread-block is responsible for executing a certain dataflow. The simulation kernel alternates execution with a *value-update kernel*, responsible for transferring next-state values into the corresponding present-state values and performing testbench actions. A simulation cycle is completed by one execution of the simulation kernel followed by one execution of the value-update kernel.

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

The following sections discuss the differences in the code generated for OpenCL frameworks (Section 5.2) with respect to CUDA frameworks (Section 5.1). The main differences deal with memory handling and with kernel management on the device.

5.1 Execution on CUDA frameworks

A pseudo-code of the host code generated for CUDA frameworks is outlined in Figure 6. In CUDA, kernels are invoked by the host and they result in the activation of a high number of parallel threads that perform computation (line 8). Host and devices have separate memory spaces. Thus memory must be allocated separately on the host (line 2) and the device (line 4). When memory must be shared between the host and the device (e.g., to pass inputs or return the result of computation to the host), all memory transfers must be explicit (line 10). Since such transfers are time-consuming operations, they must be reduced to the bare minimum.

```
1: // allocation of memory on the host
2: host_m = malloc(size);
3: // allocation of memory on the device
4: cudaMalloc((void**)&device_m, size);
5: // memory transfer to provide inputs
6: cudaMemcpy(device_m,host_m,size,cudaMemcpyHostToDevice);
7: // kernel invocation
8: kernel_f«< block_number, thread_number »>(device_m);
9: // memory transfer to get results
10: cudaMemcpy(device_m,host_m,size,cudaMemcpyDeviceToHost);
```

Figure 6: Example of host code for CUDA frameworks

5.2 Execution on OpenCL frameworks

OpenCL management of GP-GPU execution requires an initialization phase to set up the GP-GPU and the code. Figure 7 outlines the code and the operations that are necessary for the setup. The host queries the system to get a reference to the device (line 2-3). Since OpenCL supports a wide range of devices, the device type must be provided (e.g., GP-GPU). Then, this information is used to setup the context (line 5) and the command queue used to communicate with the device (e.g., to transfer data or activate kernels, line 7). The command queue is necessary since kernel invocations and memory transfers are not directly invoked by the host. On the contrary, they are added to the command queue of each device.

Finally, the host must configure the program that runs on the GP-GPU, perform runtime compilation (line 9-10) and initialize kernel functions (line 12). All such operations are not necessary in CUDA, since CUDA targets exclusively NVIDIA GP-GPU.

```
1: // get information about the platform and the GP-GPU
2: status = clGetPlatformIDs(1, &platform, NULL);
3: status = clGetDeviceIDs(platform, device_type, 1, &device, NULL);
4: // initialize the context
5: context_properties = CL_CONTEXT_PLATFORM, platform, 0 ;
6: // create the command queue
7: clCreateCommandQueue(context, device, 0, &status);
8: // setup and compile the program
9: clCreateProgramWithSource(context, 1, &source, size, &status);
10: status = clBuildProgram(program,1,&device,options,NULL,NULL);
11: // initialize the kernel
12: initialize = clCreateKernel(program, "kernel_f", &status);
```

Figure 7: Example of host code to initialize execution on OpenCL frameworks

Once initialization and platform configuration have been executed, the host code must manage memory and kernel invocations. A simple outline of the host code generated for OpenCL frameworks is shown in Figure 8. In OpenCL, kernels and memory transfers are added to the command queue initialized during the initialization phase (Figure 7). When launching a kernel, parameters are explicitly set and then the kernel can be queued for execution (line 9-10), leading to parallel execution on the device.

Memory management in OpenCL is more flexible. The programmer may decide to make memory locations visible only from the device or available to the host as well. Such configurations are set during memory allocation, by passing the desired configuration as a parameter (line 4). Performance of memory transfers between the device and the host are heavily affected by the selection of such parameters (line 6 and line 11).

```
1: // allocation of memory on the host
2: host_m = malloc(size);
3: // allocation of memory on the device
4: device_m = clCreateBuffer(context,OPENCL_CONFIG,size,NULL,
   &status);
5: // memory transfer to provide inputs
6: clEnqueueWriteBuffer(commandQueue, device_m, CL_TRUE, 0, size,
   host_m, 0, NULL, NULL);
7: // kernel setup and invocation
8: clSetKernelArg(kernel_f,n,size,parameter);
9: clEnqueueNDRangeKernel(commandQueue,kernel_f, 1, NULL,
   &thread_number, &grid_number, 0, NULL, NULL);
10: // memory transfer to get results
11: clEnqueueReadBuffer(commandQueue, device_m, CL_TRUE, 0, size,
   host_m, 0, NULL, NULL);
```

Figure 8: Example of host code for OpenCL frameworks

6. EXPERIMENTAL RESULTS

This section evaluates the performance of the generated code for SystemC simulation on GP-GPUs. Furthermore, we provide a detailed comparison between CUDA and OpenCL frameworks, to highlight their strengths and drawbacks when simulating SystemC RTL code.

6.1 Experimental setup

All the experiments have been carried out on two different GP-GPUs: NVIDIA GeForce GTX 460 and NVIDIA GeForce GTX 570. The main characteristics of such platforms are outlined in Table 1.

Characteristics	NVIDIA GTX460	NVIDIA GTX570
CUDA Version	4.2	4.2
OpenCL Version	1.1	1.1
Cores (#)	336	480
Core clock (MHz)	1350	1464
Memory clock (MHz)	1800	1900
Memory bandwidth (GB/s)	115.2	152.0
Compute capability	2.1	2.0

Table 1: Characteristics of the GP-GPU platforms

The designs used in the experiments are part of a complex embedded platform that was developed in the context of a European project together with silicon vendor industry partners:

- 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 encoding and decoding byte-wide data according to the 8b/10b protocol.

- Ecc is an error correction code device.
- System is a complex platform integrating all the designs previously discussed.

For each design, Table 2 reports the number of processes in the original SystemC description (*Processes (#)*), the lines of code (*SystemC (loc)*), the number of dataflows extracted (*Dataflows (#)*) while building the static scheduling and the number of thread blocks launched on the GP-GPU(*Thread block (#)*).

Design	Processes(#)		SystemC	Dataflows	Thread
	Synch.	Asynch.	(loc)	(#)	block (#)
ClockGen	6	15	741	12	12
ResGen	3	6	478	9	9
Sync	4	22	641	23	23
RegCtrl	18	32	2677	43	43
8b10b	7	30	799	7	7
ECC	4	7	582	4	4
System	42	112	5643	98	98

Table 2: Characteristics of the testbench designs

6.2 Compilation costs

Table 3 compares the cost of compilation for the generated code. Column *CUDA comp. (ms)* contains the compilation time for the CUDA version of each design, including compilation of the device code. Column *OpenCL comp. (ms)* details the time spent during the offline and runtime compilation phases by the OpenCL implementation.

Design	CUDA	OpenCL comp. (ms)	
	comp. (ms)	Offline	Runtime
ClockGen	5,170	270	8.42
ResGen	5,230	270	11.75
Sync	6,220	270	9.95
RegCtrl	4,920	280	11.27
8b10b	5,740	260	11.67
Ecc	4,500	290	7.09
System	16,080	290	17.37

Table 3: Comparison of compilation times for the sequential SystemC simulator and the CUDA and OpenCL versions executed on the NVIDIA GTX 570 GP-GPU.

OpenCL offline compilation cost is much smaller than CUDA compilation cost (avg. 270 ms vs 6,380 ms), since it consists only of the host code compilation and of the generation of an intermediate representation for the device code. On the other hand, OpenCL code initialization includes runtime compilation, consisting in the generation of the actual device code binary, starting from the intermediate representation (as explained in section 3.2). This last compilation step can be performed only at runtime, since it depends on the device chosen at initialization time for the execution. As a result, the runtime compilation cost for OpenCL is bigger and it depends on the size of the design (avg. 11.07 ms).

These GP-GPU experiments have been conducted on the NVIDIA GTX 570 GP-GPU. However, similar times apply to the NVIDIA GTX 460 GP-GPU.

6.3 Performance

Table 4 reports the execution time for the CUDA and OpenCL versions of each testbench example on the NVIDIA GTX 570 and NVIDIA GTX 460 GP-GPU, respectively. The table highlights that execution is faster on the GTX 460 architecture, even if the HW architecture of the NVIDIA GTX 570 GP-GPU is more optimized and contains more computing units. This behavior is strictly due to a more optimized version of the NVIDIA framework running on the NVIDIA GTX 460 architecture, including support SW and compilers. This shows that performance is heavily affected by both the underlying architecture and the support SW framework. Section 6.4 will show that the impact of each operation on the overall performance is similar on both the architectures, and thus the considerations made for the GTX 570 GP-GPU will apply also to the GTX 460 GP-GPU.

Design	NVIDIA GTX 570 (ms)		NVIDIA GTX 460 (ms)	
	CUDA	OpenCL	CUDA	OpenCL
ClockGen	1,011.0	1,317.0	210.2	420.3
ResGen	15,675.0	72,343.0	16,953.2	90,739.0
Sync	1,013.4	1,159.1	186.9	443.4
RegCtrl	1,046.8	1,450.0	235.6	637.1
8b10b	20,610.0	82,594.0	23,445.9	90,739.0
Ecc	1,055.8	1,181.1	183.3	402.1
System	61,995.7	227,142.0	71,720.4	201,379.0

Table 4: Execution time for the CUDA and OpenCL versions of each testbench example on the NVIDIA GTX 570 and NVIDIA GTX 460 GP-GPU

6.4 Performance analysis

Table 5 analyzes the performance of the generated code on the NVIDIA GTX570 GP-GPU, deepening the timing results reported in table 4. Column *(Real time (ms))* reports the execution time on CUDA and OpenCL frameworks, respectively. Then, the execution time is divided into the different operations:

- *initialization*: initialization phase that includes GP-GPU and memory setup and (for OpenCL frameworks) runtime compilation;
- *initialization kernel*: execution of the initialization kernel;
- *simulation kernel*: execution of the simulation kernel, to execute the dataflows;
- *value-update kernel*: execution of the value-update kernel;
- *DH memory transfer*: transfer of data from the device memory to the host memory;
- *HD memory transfer*: transfer of data from the host memory to the device memory.

For each operation, the table shows time spent on the CPU (Column *CPU (ms)*) and the time spent on the GPU (Column *GPU (ms)*). CPU time includes the host overhead to launch functions, kernel setup and memory setup. GPU time includes kernel execution and memory transfer execution. Furthermore, Column *Operation overhead (%)* shows the overhead of each operation on overall execution, thus allowing to analyze the performance of each operation on both the frameworks.

Figures 9 to 13 highlight the most relevant aspects of the table. As shown in Figure 9, *real time* is always faster for the CUDA version of each testbench design. This is easily explained when considering the analysis in section 3.3. SystemC simulation on GP-GPUs does not exploit architecture peculiarities such as read-only memory or loop unrolling pragmas. However, CUDA is more mature and its compilers are more optimized for the NVIDIA GP-GPU architectures than the OpenCL counterpart. This speeds up execution of the CUDA versions, to the detriment of the OpenCL implementations.

Design	Real time (ms)		Operation	Invocations (#)	CPU (ms)		GPU (ms)		Operation overhead (%)	
	CUDA	OpenCL			CUDA	OpenCL	CUDA	OpenCL	CUDA	OpenCL
ClockGen	1,011	1,317	initialization	1	1.00	8.42	0.00	0.00	2.12	10.14
			initialization kernel	1	0.02	0.03	0.00	0.00	0.04	0.04
			simulation kernel	3,300	5.48	8.74	5.84	4.56	24.01	16.01
			value-update kernel	3,300	6.19	8.09	4.41	3.59	22.49	14.06
			DH memory transf.	3,300	15.32	32.98	2.06	4.90	36.86	45.61
			HD memory transf.	3,300	5.55	10.41	1.27	1.35	14.47	14.15
ResGen	15,675	72,343	initialization	1	0.80	11.75	0.00	0.00	0.00	0.04
			initialization kernel	1	0.02	0.03	0.00	0.00	0.00	0.00
			simulation kernel	1,000,000	3,675.34	6,625.11	2,594.11	2,296.28	23.66	31.28
			value-update kernel	1,000,000	3,381.12	5,305.48	1,953.26	2,002.04	20.13	25.62
			DH memory transf.	1,000,000	9,530.30	1,974.81	1,305.40	3,113.09	40.89	17.84
			HD memory transf.	1,000,000	3,245.82	6,354.84	813.44	835.60	15.32	25.21
Sync	1,013	1,159	initialization	1	0.99	9.95	0.00	0.00	2.57	11.67
			initialization kernel	1	0.01	0.04	0.01	0.00	0.05	0.05
			simulation kernel	1,300	4.85	14.07	3.76	3.30	22.45	20.36
			value-update kernel	1,300	4.47	9.00	4.58	3.85	23.57	15.07
			DH memory transf.	1,300	12.67	31.31	1.63	3.96	37.29	41.34
			HD memory transf.	1,300	4.38	8.77	1.01	1.05	14.07	11.51
RegCtrl	1,046	1,450	initialization	1	1.09	11.27	0.00	0.00	1.07	4.37
			initialization kernel	1	0.01	0.03	0.01	0.00	0.02	0.01
			simulation kernel	6,600	11.97	60.78	10.37	9.01	22.00	27.07
			value-update kernel	6,600	10.90	30.60	14.55	14.58	25.05	17.52
			DH memory transf.	6,600	35.00	90.66	4.25	9.96	38.64	39.03
			HD memory transf.	6,600	10.88	28.30	2.56	2.62	13.23	11.99
8b10b	20,610	82,594	initialization	1	0.98	11.67	0.00	0.00	0.00	0.02
			initialization kernel	1	0.01	0.03	0.00	0.00	0.00	0.00
			simulation kernel	1,000,000	3,602.43	5,904.27	5,904.27	5,203.65	26.44	21.52
			value-update kernel	1,000,000	3,269.76	4,356.19	4,217.16	3,579.21	20.82	16.28
			DH memory transf.	1,000,000	12,881.70	21,872.60	1,230.69	2,984.38	39.25	51.00
			HD memory transf.	1,000,000	3,186.54	4,646.12	1,659.49	798.13	13.48	11.17
Ecc	1,056	1,181	initialization	1	1.01	7.09	0.00	0.00	3.16	10.18
			initialize kernel	1	0.01	0.03	0.00	0.00	0.06	0.04
			simulation kernel	1,026	3.49	7.64	4.14	3.78	22.80	16.40
			value-update kernel	1,026	3.88	11.40	3.40	2.75	22.72	20.31
			DH memory transf.	1,026	10.63	25.42	1.28	3.06	37.18	40.86
			HD memory transf.	1,026	3.39	7.69	0.80	0.82	13.08	12.21
System	61,996	227,142	initialization	1	1.36	17.37	0,00	0,00	0,00	0,01
			initialization kernel	1	0.01	0.03	0.01	0.01	0,00	0,00
			simulation kernel	2,200,002	7,190.06	15,237.60	11,255.20	9,755.01	16.53	16.76
			update kernel	2,200,002	7,854.85	25,732.20	27,430.70	23,973.40	31.63	33.34
			DH memory transf.	2,200,002	45,820.00	50,841.90	2974.18	7,104.37	43.73	38.86
			HD memory transf.	2,200,002	7,348.70	14,517.30	1,697.00	1,917.47	8.11	11.02

Table 5: Comparison of execution times for the CUDA and OpenCL versions executed on the NVIDIA GTX570 GP-GPU and analysis of the overhead of the single operations on overall performance

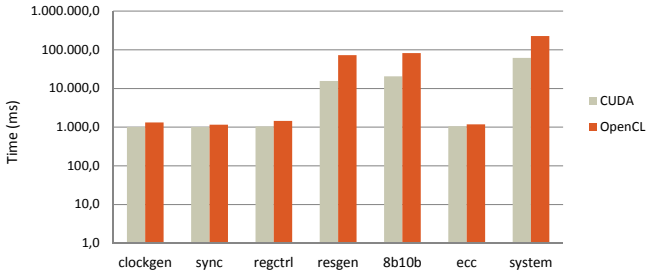

Figure 9: Comparison of simulation performance (real time) for the CUDA and OpenCL simulation of each design

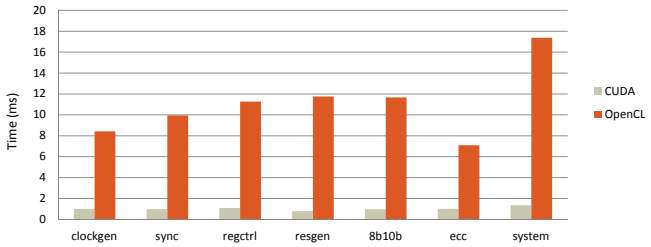

Figure 10: Initialization overhead of the CUDA and OpenCL implementations for all testbench designs

Figure 10 shows the time devoted to *initialization* in the CUDA and OpenCL code. The CUDA version has an almost constant initialization time, that results in being faster than the OpenCL version. Indeed, OpenCL initialization includes runtime compilation costs that strictly depend on the device code and on the kernel's size (as explained in section 3.2 and shown in section 6.2). As a result, a more optimized OpenCL compiler would enhance initialization and reduce its impact on performance.

Then, it is important to evaluate the overhead of each *operation* on the execution time. The analyzed operations are: initialization (including runtime compilation for OpenCL), device kernels (initialization kernel, simulation kernel and value-update kernel) and memory transfers (DH if the transfer is from device to host, HD for the other direction). Figures 11, 12 and 13 highlight the impact of each operation on the RegCtrl example. However, similar considerations apply to all testbench designs.

Figure 11 shows the average *CPU time* necessary to perform

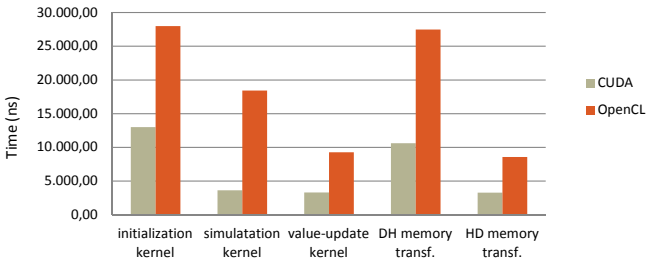

Figure 11: Comparison of average CPU time of each operation for CUDA and OpenCL implementations of the RegCtrl example

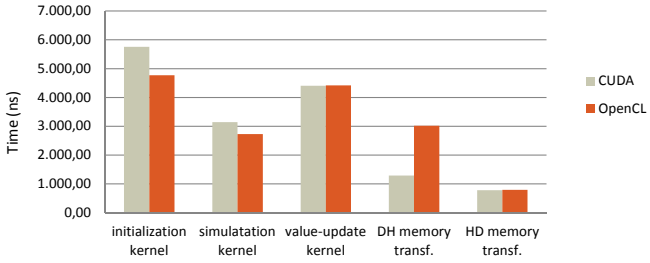

Figure 12: Comparison of average GPU time of each operation for CUDA and OpenCL implementations of the RegCtrl example

each operation. The figure highlights that CPU time is always higher for OpenCL versions. Indeed, kernel management and memory handling imply a heavier overhead for OpenCL. On the other hand, Figure 12 shows the average *GPU time* necessary to perform each operation. Kernel execution is faster on OpenCL, while memory transfers are more optimized and, thus, faster for the CUDA version. The difference on memory handling is explained by the different nature of CUDA and OpenCL. CUDA targets specific GP-GPU architectures, thus it can efficiently exploit the architecture characteristics (e.g., read-only memory). On the other hand, OpenCL must be more flexible and it can make no assumption on the underlying architecture, leading to an heavy impact on memory operations.

Finally, Figure 13 depicts the *impact of each operation* on execution performance for both the CUDA and the OpenCL versions of the examples. CUDA and OpenCL have similar behaviors. Initialization and the initialization kernels have a low impact on performance (0.02% and 1.07% on CUDA, 0.01% and 4.37% on OpenCL, respectively). Furthermore, the more complex funtionality execution is, the less impact initialization has.

A percentage comprised between the 40% and the 50% of the execution time is occupied by execution of the kernels that are necessary for simulation in both the CUDA and OpenCL versions. Indeed, the simulation and value-update kernels occupy the 22% and 27.07% on CUDA, and the 25.05% and 17.52% on OpenCL, respectively.

Finally, memory management has a heavy impact on the performance. Copy of data from the device memory to the host memory is up to 3x slower than transfers in the opposite direction, for both the CUDA and the OpenCL version. Indeed, it requires an average of 38.83% of overall execution time (38.64% on CUDA and 39.03% on OpenCL), while memory transfers from the host memory to the device memory require an average of the 12.61% (13.23% on CUDA and 11.99% on OpenCL). As a result, memory transfers dominate both CUDA and OpenCL execution performance.

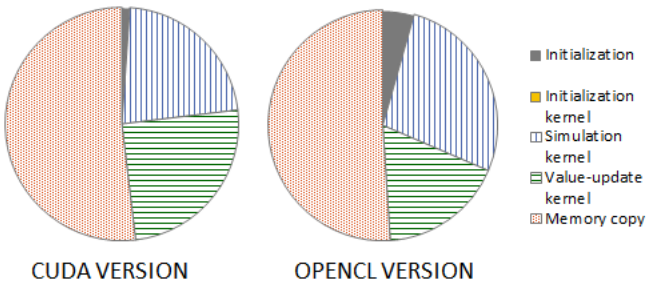

Figure 13: Overhead of the individual operations for the RegCtrl example, applied to the CUDA (on the left) and to the OpenCL version (on the right) on the NVIDIA GTX570 GP-GPU.

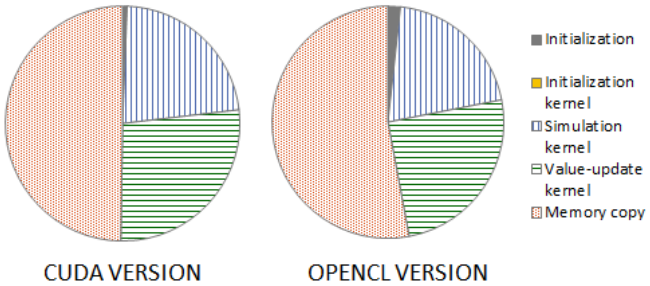

Figure 14: Overhead of the individual operations for the RegCtrl example, applied to the CUDA (on the left) and to the OpenCL version (on the right) on the NVIDIA GTX460 GP-GPU.

The above considerations justify the overall performance trends shown in Figure 9. OpenCL versions are faster only in kernel execution. On the contrary, all management tasks (kernel activation and synchronization and memory transfers) are faster on CUDA. Figure 14 highlights that the comments made for the GTX 570 architecture apply also to the GTX 460 GP-GPU. Indeed, each operation has the same impact on the execution for both the CUDA and the OpenCL version, despite of the more optimized compute capability support.

All the former considerations highlight how the OpenCL performance is affected by non optimized compilers. This is highlighted also by the comparison between the NVIDIA GTW 460 and 570 architectures, where a more optimized SW framework makes execution faster on the less performing architecture (the GTX 460). OpenCL is less mature that CUDA, thus affecting execution management and memory handling costs. However, former development of OpenCL may improve performance and management, reaching the performance of CUDA still preserving the higher degree of portability.

7. CONCLUSION

This paper presented an analysis of SystemC simulation on GP-GPU architectures, targeting both CUDA and OpenCL frameworks. The analysis is enriched with a detailed analysis of the performance differences between the two frameworks, highlighting strengths and drawbacks. We found that OpenCL is slower that the corresponding CUDA version, due to a less optimized management of the underlying architecture. However, OpenCL is less mature than CUDA and the analysis conducted in the paper showed that the SW framework has a heavy impact on performance. Thus, future development may lead to more optimized compilers and to overcome the limitations of OpenCL portability.

8. REFERENCES

[1] Accellera Systems Initiative. *SystemC*. `http://www.accellera.org`.

[2] F. Balarin and R. Passerone. Functional verification methodology based on formal interface specification and transactor generation. In *Proc. of IEEE/ACM DATE*, pages 1013–1018, 2006.

[3] N. Bombieri, A. Fedeli, F. Fummi, and G. Pravadelli. Hybrid incremental ABV for functional validation in TLM design flows. *IEEE Design and Test of Computer*, 24(2):140–152, 2007.

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

[5] R. Buchmann and A. Greiner. A fully static scheduling approach for fast cycle accurate SystemC simulation of MPSoCs. In *Proc. of IEEE ICM*, pages 101 –104, 2007.

[6] L. Cai and D. Gajski. Transaction level modeling: An overview. In *ACM/IEEE CODES+ISSS*, pages 19–24, 2003.

[7] P. Combes, E. Caron, F. Desprez, B.Chopard, and J. Zory. Relaxing synchronization in a parallel SystemC kernel. In *Proc. Of ISPA*, pages 180–187, 2008.

[8] P. Du, R. Weber, P. Luszczek, S. Tomov, G. D. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

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

[10] J. Fang, A. L. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Proc. of ICPP*, pages 216–225, 2011.

[11] R. Jindal and K. Jain. Verification of transaction-level SystemC models using RTL testbenches. In *Proc. of ACM/IEEE MEMOCODE*, pages 199–203, 2003.

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

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

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

[15] 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.

[16] R. Sinha, A. Prakash, and H. D. Patel. Parallel simulation of mixed-abstraction systemc models on GPUs and multicore CPUs. In *Proc. of ACM ASP-DAC*, pages 455–460, 2012.

[17] The Khronos OpenCL Working Group. *OpenCL - The open standard for parallel programming of heterogeneous systems*, 2011. http://www.khronos.org/opencl/.

[18] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki. *The OpenCL Programming Book*. Fixstars Corporation, 2010.

[19] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi. SAGA: SystemC Acceleration on GPU Architectures. In *Proc. of ACM/IEEE DAC*, 2012.

[20] H. Ziyu, Q. Lei, L. Hongliang, X. Xianghui, and Z. Kun. A parallel SystemC environment: ArchSC. In *Proc. of ICPADS*, pages 617–623, 2009.