Des Autom Embed Syst (2012) 16:115–135
DOI 10.1007/s10617-012-9092-z

HDTLib: an efficient implementation of SystemC data types for fast simulation at different abstraction levels

Nicola Bombieri · Franco Fummi · Valerio Guarnieri · Francesco Stefanni · Sara Vinco

Received: 5 January 2012 / Accepted: 26 June 2012 / Published online: 18 July 2012

Abstract SystemC is the de-facto standard language for system-level modeling, architectural exploration, performance analysis, software development, and functional verification of embedded systems. Nevertheless, it has been proved that the performance of the SystemC implementation is typically less optimal than commercial VHDL/Verilog simulators when used for register transfer level (RTL) simulation. This is mainly due to the "slow" implementation of bit-accurate data types provided by the standard library. Such a problem limits the simulation performance even when SystemC designs are implemented at higher levels of abstraction (i.e., transaction-level modeling—TLM) and still make use of bit-accurate data types (e.g., for a more accurate verification, or in TLM descriptions automatically generated from RTL). This article presents *HDTLib*, a new bit-accurate data type library that increases the simulation speed up to 3.45× at RTL and up to 10× at TLM. In addition, when the level of abstraction rises from RTL and better simulation performance is required, accuracy of HW-dependent behaviors is no longer necessary. Thus, the article presents a type abstraction methodology to get rid of low level behaviors and how such a methodology can be combined with *HDTLib* for guaranteeing a sound tradeoff between accuracy and simulation speed. Finally, more recent works have proposed efficient and promising techniques to boost SystemC simulation through general purpose graphics processing unit (GP-GPU) architectures. In such parallel frameworks, the standard SystemC library for bit-accurate data types

This work has been partially supported by the European project FP7-ICT-2011-7-288166 (TOUCHMORE).

N. Bombieri (✉) · F. Fummi · V. Guarnieri · F. Stefanni · S. Vinco
Department of Computer Science, University of Verona, Strada le Grazie, 37134, Verona, Italy
e-mail: nicola.bombieri@univr.it

F. Fummi
e-mail: franco.fummi@univr.it

V. Guarnieri
e-mail: valerio.guarnieri@univr.it

F. Stefanni
e-mail: francesco.stefanni@univr.it

S. Vinco
e-mail: sara.vinco@univr.it

is not supported, with a consequent limitation of their application to actual designs. This article shows how *HDTLib* has been implemented for applying also to these today many-core architectures.

Keywords SystemC · TLM · Data types · Type abstraction · Simulation · GP-GPU

1 Introduction

The increasing complexity of modern designs makes simulation a key approach to support and accelerate verification in the design flow. Simulation allows functional validation and early evaluation of performance and requirements of the final system. As a consequence, design complexity has a huge impact on the efficiency of simulation during the design flow, and simulation time increases at least linearly with the circuit complexity [1]. Furthermore, simulation time impacts on the design and verification loop, as higher simulation speed shortens time until a regression runs and until results of a modification are made available. Since simulation heavily affects the overall verification process, fast simulation is a priority.

Many factors determine simulation efficiency: from the level of abstraction to the modeling style. Also the language used for modeling impacts on simulation performance [1].

One of the most common languages for embedded system design is SystemC [2]. SystemC is defined and promoted by Accellera Systems Initiative (ASI), the organization that has joined the Open SystemC initiative (OSCI) and Accellera. SystemC has been approved by the IEEE Standards Association as IEEE 1666-2005. SystemC extends the C/C++ language with libraries for describing HW constructs and thus it is familiar and easy to learn for C/C++ programmers. Nevertheless, SystemC is up to 10× slower at RTL level and 2.5× slower at behavioral level than the other HDLs (i.e., Verilog, VHDL and SystemVerilog) [1].

Most of this decrement of performance is due to the SystemC bit accurate data types (i.e., `sc_int`, `sc_lv`, `sc_logic`, etc.) [3]. Low level modeling of HW requires the adoption of bit-accurate and multi-valued data types, in order to recreate the correct behavior of the physical circuit. Bit-accuracy is required since each single bit impacts on the quality of the final product. Using native C/C++ types would allow to highly optimize performance. Nevertheless, native types do not support all the typical HW manipulation operators. Thus, it is necessary to define classes implementing accurate data types, that result in being slow.

SystemC also supports design modeling at levels of abstraction higher than RTL, such as electronic-system level (ESL) [4], with the aim of speeding up the design and verification process. Transaction-level modeling (TLM) is the standard modeling style at ESL proposed by ASI and has become the de-facto standard for embedded system designers [5]. TLM provides interfaces and primitives for implementing designs with more or less details according to the target use case (i.e., SW development, HW verification, architectural analysis, etc.) [6].

The most accurate SystemC TLM implementations make often use of bit-accurate data types (e.g., `sc_int`, `sc_uint`) [7]. As a consequence, the "slow" implementation of such data types also limits the potential simulation performance of the abstraction paradigm.

In addition, TLM models rely on bit-accurate data types even at the highest abstraction levels, when the models are automatically generated from an RTL-to-TLM abstraction process. Methodologies to abstract RTL implementations up to TLM exist in literature [8] and on the market [9], with the aim of reusing RTL IPs for faster TLM simulation. In these cases, besides the bit accuracy, low level HW-specific details implemented through multi-valued logic data types (i.e., `sc_logic`, `sc_lv`) are inevitably maintained during the abstraction process even though useless at such an abstraction level, thus resulting in a heavier simulation.

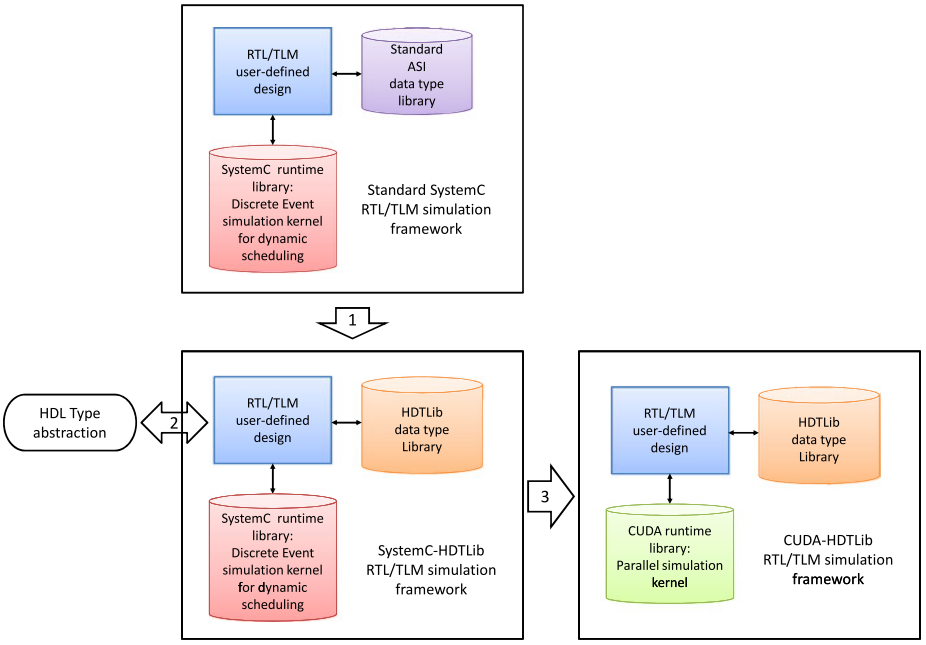


Fig. 1 Simulation frameworks with *HDTLib*

Finally, portability on many-core architectures is another major limitation of the data type standard library proposed by ASI. Recently, several research groups are investigating new solutions to reduce the simulation time of complex embedded systems by exploiting the high degree of parallelism afforded by today's general purpose graphics processing units (GP-GPUs) [10–12]. All these works present promising techniques that sensibly reduce the simulation time of SystemC models described at any level of abstraction and run on NVIDIA CUDA platforms [13]. Nevertheless, the proposed techniques do not support models implemented with bit-accurate data types, with a consequent limitation of their application to actual designs.

To deal with all these limitations of the SystemC data type standard library, this article presents *HDTLib*, an efficient library that provides a faster implementation of all the HDL-oriented data types. The library is bit-accurate and is semantically equivalent to the ASI standard. *HDTLib* modularly replaces the ASI data type library, thus applying to new as well as already existent SystemC designs (see Fig. 1). By adopting *HDTLib*, only few slight syntax modifications may be required to the original SystemC model implementation and the scheduling activities during simulation are left to the SystemC simulation kernel.

The article presents an analysis on SystemC simulation of both synthetic and industrial designs of different complexity and architectural characteristics to show:

- the impact of the different data types and corresponding operators on the overall simulation speed.
- the impact of the data type accuracy on the overall simulation by considering the abstraction level of the design implementation.
- a comparison of simulation speed obtained by adopting the standard ASI library, a commercial SystemC library, and *HDTLib*.

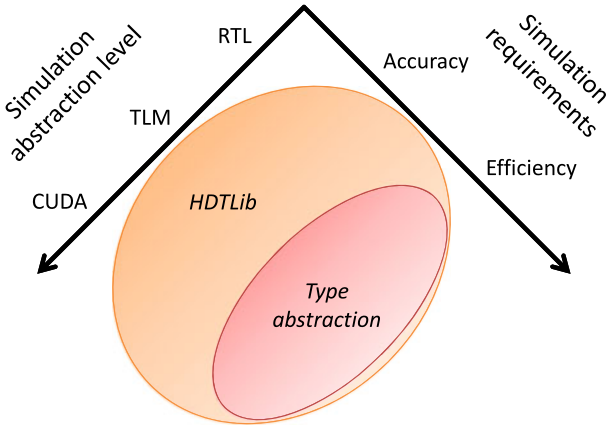


Fig. 2 *HDTLib* and type abstraction in RTL, TLM, and CUDA simulation

The experimental results show that, at RTL, *HDTLib* improves the simulation speed-up to 3.45× with regard to the standard ASI library.

The results also show that the optimized implementation of data types impacts even more at TLM, where the scheduling activity is minimal. Thus the article presents a type abstraction methodology that, combined with *HDTLib* (Fig. 2), improves the simulation performance at TLM by getting rid of low-level HW behaviors. The approach is analyzed to show how the type abstraction may influence the equivalence between the original and the abstracted model. In particular, the analysis shows what kind of HW-specific details (useless at TLM) can be lost during abstraction, and how this loss of details impacts on the simulation performance.

Finally, the article shows how *HDTLib* has been implemented for applying also to GP-GPU architectures (Fig. 3). Experimental results show that the simulation overhead introduced for supporting bitwise accuracy and the multi-value logic becomes negligible when considering the overall massive speed-up provided by such many-core architectures.

The article is organized as follows. Section 2 presents the related work. Section 3 presents *HDTLib*, deepening its implementation details, while Sect. 4 presents the proposed type abstraction methodology and the related issues. Section 5 presents the application of the proposed library to CUDA GP-GPU architectures. Section 6 applies the *HDTLib* library and the type abstraction methodology to a set of test cases to prove its efficiency with respect to the ASI SystemC implementation. Finally, Sect. 7 draws the conclusions.

2 Related work

Some works have been proposed in literature with the aim of defining a tradeoff between accuracy and simulation speed of SystemC data types.

In [14], the authors propose a C implementation of the main operators required by HW descriptions and not provided by standard C/C++ types. By using this approach, native types can be exploited for increasing performance. However, the work is limited to concatenation, bit selection and range operators. Furthermore, there is no support for multi-value logic and, thus, there is no compliance with the SystemC specifications.

Performance close to native C/C++ data types is reached in [3] and [15] with the definition of the *Algorithmic C* library (AC). AC provides arbitrary-length integer, fixed-point and complex data types. Data types are defined as templatized classes, such as integer arithmetic

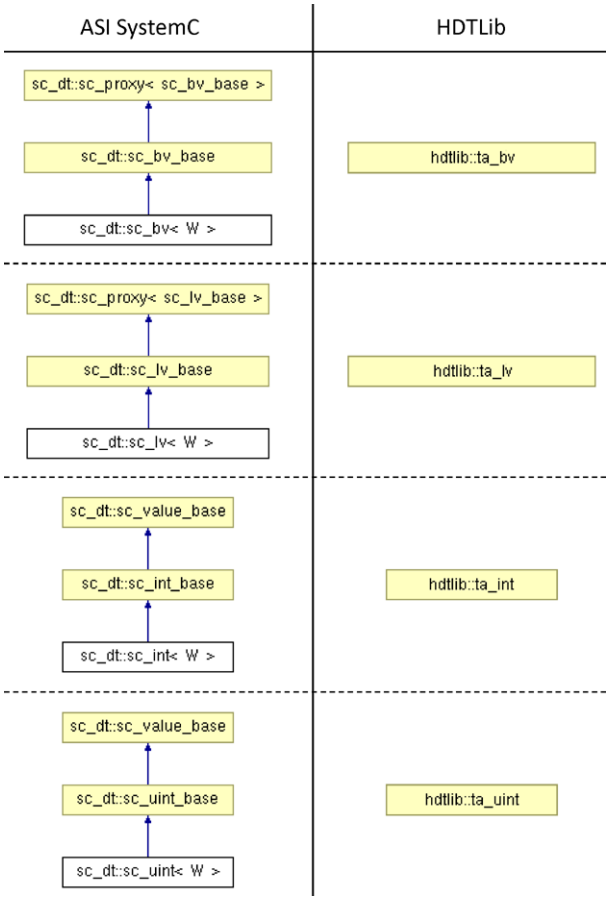


Fig. 3 Comparison between ASI SystemC data types and *HDTLib* class hierarchy

`ac_int` and fixed point numbers `ac_fixed`. Precision and signedness are specified as template parameters, and the mostly used operators are supported (i.e., arithmetic, logic, relational and bit or range selection). This approach allows to obtain a 2× simulation increase with respect to the corresponding SystemC data types. However, logic data types are not handled, thus preventing efficient simulation of low level designs.

In [16], the authors present an automatic technique for software generation from SystemC modules with an efficient class of data types, built to replace SystemC data types. Nevertheless, only the `uc_uint` data type is provided, to substitute the `sc_uint` SystemC type. Furthermore, only a very restricted range of operators is implemented (i.e., assignment, bit selection and read operations).

All the previous works deal with simulation on standard architectures, where the SystemC discrete-event simulation (DES) is performed on a single CPU. More recently, many-core architectures have been applied for accelerating computation intensive EDA applications and, in particular, logic simulation [17–19] and gate-level fault simulation [20–22]. All these simulation frameworks yield reasonable speed-up compared to logic simulation on conventional processors. They meaningfully apply to logic and fault simulation of gate-level circuits since gate-level data types of each HDL are easily mapped into native C data types. Nevertheless, they apply neither to RTL nor to TLM simulation.

RTL-to-SystemC/C/C++ translation has been proposed in different works [23, 24] and tools [9, 25, 26] targeting simulation in conventional CPU architectures. The translation from RTL to C for GP-GPUs (i.e., NVIDIA CUDA) has been investigated only in most recent works [10–12]. All these works focus on improving simulation performance by parallelizing the SystemC kernel activity ([10, 11]) or by parallelizing instances of the design under verification [12]. Nevertheless, they do not face the problem of porting the bit-accurate multi-value data type into the CUDA framework.

3 *HDTLib*: an HDL-oriented data types library for SystemC

HDTLib is an accurate and efficient implementation of SystemC data types, which has the following characteristics:

- The implementation preserves semantics with the SystemC specifications and it supports all low-level and accurate behaviors.
- It has been implemented by exploiting advanced optimization techniques, like compile-time optimizations based on C++ templates, as explained in Sect. 3.1.
- It applies to standard "single core" architectures for discrete event simulation and GP-GPU architectures (e.g., CUDA) for massive parallel simulation, as explained in Sect. 5.
- It can be combined with a type abstraction (as explained in Sect. 4) to guarantee the best tradeoff between simulation performance and accuracy in each simulation paradigm (i.e., RTL, TLM, CUDA), as depicted in Fig. 2.
- It can be adopted on new as well as already existent SystemC designs, by exploiting a simple C++ preprocessor directive, namely a `#define`. In contrast, the type abstraction methodology requires code manipulations, which have been automated in a HIFSuite tool [27], as described in Sect. 4.

HDTLib has been acquired by EDALab [28] and it is available under licence as a component of the A2T tool of HIFSuite.

3.1 *HDTLib* implementation details

HDTLib consists of five data types: a 4-value logic vector class, a 2-value bit vector class, a single logic value class, a signed and an unsigned integer class. The fixed point data type is not supported by the current version of *HDTLib*. This extension will be part of future work.

All classes except the single logic value class are templatized, taking one integer parameter that indicates the bitwidth (i.e., the number of elements belonging to the vector).

In order to achieve a significant performance improvement, the following solutions have been adopted when implementing these data types:

1. *No heap memory allocation*. A statically allocated array of unsigned integers is employed as underlying data structure used to store vector elements. The size of such an array can be computed at compile time, since it depends only on the width template argument. For example, a bit vector having width `W` contains the following declarations:

```
// static constant that stores the number of chunks required
    to accommodate W bits
static const unsigned int CHUNKS_NUMBER = W / (sizeof(chunk_t)
     * 8) + (W % (sizeof(chunk_t) * 8) ? 1 : 0);
...
unsigned int _chunks[CHUNKS_NUMBER];
```

This solution allows us to avoid heap memory allocation, which causes additional overhead at runtime and prevents compiler optimizations, thus resulting in a performance hit. It is worth noting that this solution usually uses about the same size of stack memory w.r.t. the ASI implementation. Considering bitwidth up to 64 bits, the ASI implementation uses at least `sizeof(void*)` stack memory, while *HDTLib* uses `sizeof(int32_t)` or `sizeof(int64_t)`. Thus, *HDTLib* results in being more efficient than the ASI library on x86-64 machines, and nearly equivalent on x86-32 machines.

2. *Operations performed on words, instead of single bits*. The choice of unsigned integers as underlying data structure led to implementing operations on data types on a word as a whole, instead of repeating the same operation on each single vector element. For example, a snippet of the code that implements bitwise negation for bit vectors is the following:

```
for (register unsigned int i = 0; i < CHUNKS_NUMBER; ++i)
    result._chunks[i] = ~(_chunks[i]);
```

This is achieved by carefully implementing operations on architecture-dependent words by properly using bitwise operations and shifts. These are among the fastest instructions to be executed on any machine, since they take advantage of word-sized registers and optimized ALUs to be executed in a single CPU operation.

3. *Mapping of a logic value on two separate bits*. Logic vectors have been implemented by using two separate arrays of unsigned integers. Each logic value is associated with two bits, one per array, to represent the four possible values. For example, the logic vector class has the following instance variables:

```
unsigned int _lower_chunks[CHUNKS_NUMBER];
unsigned int _upper_chunks[CHUNKS_NUMBER];
```

In this way, it is still possible to implement operations on logic values in terms of bitwise and shift operations on architecture-dependent words.

4. *Replacement of lookup tables with Karnaugh maps*. Consistently with the previous choice, bitwise operations on logic values have been implemented by using Karnaugh maps, instead of lookup tables. Karnaugh maps are faster than lookup tables because they avoid accessing values that have not been fetched into cache. Implementation of logic operations has been achieved by rewriting the truth tables of such operators according to the two-bit encoding adopted for logic values into Boolean functions. These functions have been expressed in terms of their minimal sum of products form. For example, the implementation of the bitwise negation operator can be sketched as:

```
for (register unsigned int i = 0; i < CHUNKS_NUMBER; ++i) {
    result._lower_chunks[i] = _upper_chunks[i] &
       (~(_lower_chunks[i]));
    result._upper_chunks[i] = _upper_chunks[i];
}
```

5. *Minimal class hierarchy*. In order to reduce the impact of managing parent constructors and destructors at runtime, the class hierarchy has been kept to the bare minimum.

Table 1 Mapping of ASI SystemC types to *HDTLib*

SystemC	*HDTLib*	Notes
sc_int_base/sc_uint_base		Removed to optimize hierarchy.
sc_lv_base/sc_bv_base		Removed to optimize hierarchy.
sc_signed/sc_unsigned		Removed to optimize hierarchy.
sc_proxy/sc_subref/ sc_subref_r and other reference/proxy types		Returning reference to the actual type, when possible. Otherwise, code modification is required.
sc_int/sc_uint	ta_int/ta_uint	Implemented with optimized versions for most common bitwidths (8, 16, 32, 64).
sc_bigint/sc_biguint		Not implemented, since ta_int and ta_uint have no limits on the number of bits.
sc_bit		Deprecated, thus not implemented. Replaced by bool.
sc_bv	ta_bv	Implemented with optimized versions for most common bitwidths (8, 16, 32, 64).
sc_logic	ta_logic	
sc_lv	ta_lv	Implemented with optimized versions for most common bitwidths (8, 16, 32, 64).
All fixed point types		Not implemented. Future work.

In order to provide a better overview of *HDTLib* structure, Table 1 associates ASI SystemC data types with corresponding *HDTLib* data types. It is worth noting that most classes in the ASI hierarchy have been removed to shrink the class hierarchy as much as possible. For example, common base classes such as `sc_bv_base` and `sc_lv_base` have been omitted, since the end user of the library does not directly employ and access them. Figure 3 provides a visual representation of the class hierarchy reduction operated by *HDTLib*, thus underlying one of the main differences w.r.t. the ASI implementation. To improve performance, template specializations of the classes for bit and logic vectors and integer types have been developed, so that optimized definitions of these classes are provided for most common bitwidths (i.e., 8, 16, 32 and 64 bits).

As additional details, Table 2 compares syntactically equivalent methods for the logic vector class in the ASI SystemC library and *HDTLib*. Instead, Table 3 highlights the methods featuring syntax changes, as identified previously in the text. For space constraints, only the logic vector class is shown. However, the same applies to the classes for bit vector and integer types.

A limitation with respect to SystemC concerns portions of vectors. SystemC allows to create *references* to portions of vectors, by using the type `sc_dt::sc_subref`. Changing the values of a reference will also change the value in the referenced vector. *HDTLib* returns only values by copy, while reference support is part of our current work. Meanwhile, the SystemC behavior can be easily reproduced by using an explicit call to a range-setting method.

3.2 *HDTLib* usage

HDTLib can be adopted in two contexts:

Table 2 Comparison between ASI SystemC and *HDTLib* syntactically equivalent methods for the logic vector class

SystemC	*HDTLib*
`sc_lv<W>& operator = (const sc_lv<W>& a)`	`ta_lv<W>& operator = (const ta_lv_t<W>& a)`
`const sc_lv_base operator ~() const`	`const ta_lv<W> operator ~() const`
`sc_lv_base& operator &= (const sc_lv<W>& a)`	`ta_lv<W>& operator &= (const ta_lv_t<W>& a)`
`sc_lv_base& operator |= (const sc_lv<W>& a)`	`ta_lv<W>& operator |= (const ta_lv_t<W>& a)`
`sc_lv_base& operator ^= (const sc_lv<W>& a)`	`ta_lv<W>& operator ^= (const ta_lv_t<W>& a)`
`sc_lv_base& operator <<= (int n)`	`ta_lv<W>& operator <<= (int n)`
`sc_lv_base& operator >>= (int n)`	`ta_lv<W>& operator >>= (int n)`

Table 3 Comparison between ASI SystemC and *HDTLib* syntactically non-equivalent methods for the logic vector class

SystemC	*HDTLib*
`sc_subref_r<sc_lv_base> range(int hi, int lo) const`	`template <int WR> ta_lv<WR> range(int hi, int lo) const`
`sc_subref<sc_lv_base> range(int hi, int lo)`	`template <int WR> ta_lv<WR>& set_range(int hi, int lo, const ta_lv<WR>& rhs)`
`sc_bitref_r<sc_lv_base> operator[](int i) const`	`ta_logic operator[] (int i) const`
`sc_bitref<sc_lv_base> operator[](int i)`	`ta_logic set_bit (int i)`

1. implementation of SystemC designs from scratch;
2. replacement of the ASI SystemC data types with *HDTLib* data types on already existing SystemC designs.

In the first context, designers can easily use *HDTLib* data types according to the API and the documentation provided with the library. *HDTLib* largely preserves the same syntax as the ASI SystemC data types, except for operations on ranges and single bits that have a slightly different syntax. Such differences are due to the implementation choices adopted in order to improve *HDTLib* simulation performance. Specifically, *HDTLib* syntax differs with the ASI SystemC syntax only in the following cases:

1. range selections require the bitwidth of the selection to be provided as template argument, e.g. `a.range(15, 8)` becomes `a.range<8>(15, 8);`
2. assignments on ranges must be replaced with corresponding templatized function calls, e.g. `a.range(15, 8) = x` is replaced with `a.set_range<8>(15, 8, x);`

3. assignments on single bits must be replaced with corresponding function calls, e.g. `a[8] = x;` is replaced with `a.set_bit(8, x);`

In the second context, i.e., usage of *HDTLib* in existent designs, *HDTLib* data types replace the standard data types provided by the SystemC ASI library. In this case, some modifications to the source code may be required in order to take into account the previously stated syntax differences. However, it is worth noting that these changes have to be applied only if the existing SystemC description contains operations on ranges and single bits, as stated before. In order to make the transition to *HDTLib* smoother in these cases, these modifications have been automated by developing a tool on top of HIFSuite [27]. This tool performs all the syntax changes required by *HDTLib* on the existing SystemC description, according to the pseudocode described in Algorithm 1.

```
1  foreach expression do
2      if expression contains range selection then
3          add the range bitwidth as template argument
4          to the range function call;
5      end
6  end
7  foreach assignment do
8      if left-hand side is a range then
9          replace assignment with call to the set_range function
10         with the range bitwidth as template argument;
11     else if left-hand side is a single bit then
12         replace assignment with call to the set_bit function;
13     end
14 end
```

Algorithm 1: Pseudocode for the algorithm performing syntax changes required by *HDTLib*

The starting SystemC description is firstly parsed, in order to identify expressions and assignments, which are the only two constructs affected by the required syntax changes. In fact, the first syntax change targets read-only range selections, which can be found only in expressions, while the last two syntax changes impact assignments only.

Each expression is checked to determine whether it contains a range selection (lines 1–2). If this is the case, the range bitwidth is added as a template parameter to the function call that performs the range selection (lines 3–4). Then the left-hand side of each assignment is checked to determine whether it is a range (lines 7–8) or a single bit (line 11). If a range is on the left-hand side, the assignment is replaced with a call to the `set_range` function (lines 9–10). The bitwidth range is provided as template argument, while the range bounds and a reference to the left-hand side are passed as arguments to the function call. If the left-hand side consists of a single bit, the assignment is replaced with a call to the `set_bit` function (line 12). The bit index and a reference to the left-hand side are passed as arguments to the function call.

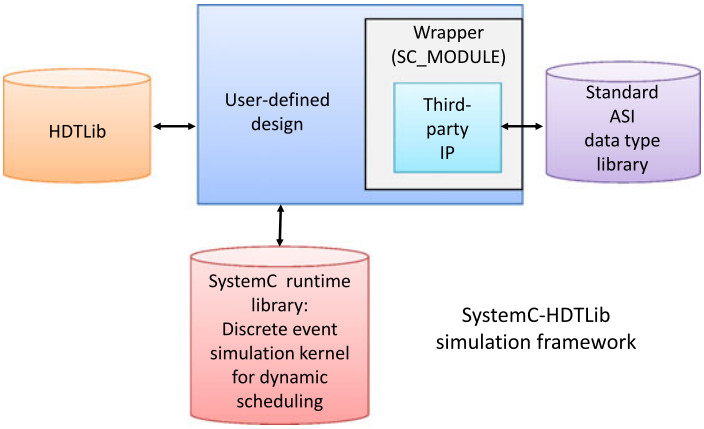


Fig. 4 Integration of third-party IPs with designs using *HDTLib*

3.3 Third-party IP integration

IP reuse is a key strategy in modern embedded system design flows to optimize the time-to-market and to minimize production costs. In many cases, third-party IPs are available only as *object files*, i.e. SystemC code pre-compiled by the vendors to protect the implementation details.

In these cases, the integration of such IPs, which are written by adopting standard SystemC types, with designs that adopt *HDTLib* is not immediate. The proposed solution is depicted in Fig. 4. Each IP is instantiated as a submodule of a wrapper (i.e., `SC_MODULE`). The wrapper module explicitly uses *HDTLib* types for the interface and performs the conversions between *HDTLib* and standard SystemC types. Therefore, open source (i.e., user-defined) designs can adopt *HDTLib* and interact with the third-party IPs without performing any type conversion.

The wrapper module relies on functions to convert HDL types to primitive C++ types. For example, bit vectors are converted to their corresponding representation as unsigned integers, or as strings if their bitwidths exceed the length of an unsigned integer. Logic vectors can be converted to their corresponding string representation. For example, the following statement of the wrapper converts a *HDTLib* bit vector `a` to a corresponding standard SystemC bit vector `b`:

```
b = a.to_uint();
```

Both *HDTLib* and the standard SystemC data library provide these conversion functions.

4 Type abstraction methodology

SystemC supports hardware modeling by providing a set of bit accurate and multi-valued data types to recreate low level behaviors of the target physical circuit.

However, when the goal is functional verification or the design is abstracted from RTL to TLM, simulation performance can be further improved by abstracting some low level details typical of HDL types. Maintaining the same accuracy after the abstraction would have a huge

impact on performance of the abstracted code. That is, the high level functionality and the lightened communication protocol would still rely on bit accurate data types to describe HW characteristics that are useless for the high-level functional verification.

The type abstraction methodology aims at abstracting the multi-valued logic data types (i.e., `'X'`, `'Z'`, `'0'`, `'1'`) into a two-valued logic type (i.e., `'0'`, `'1'`). In particular, the idea is to substitute logic types with an efficient implementation of bit vectors. In general, this substitution creates a non-equivalent design, and thus, there are some restrictions to its possible adoption.

Bit vectors and integers are unchanged, since for their efficient simulation *HDTlib* can be adopted, still preserving an equivalent semantics.

Logic types are described in SystemC as elements able to assume four values: `'0'` and `'1'`, just like standard bits, plus two special values, high-impedance `'Z'` and unknown `'X'`.

The *unknown value* `'X'` is the default initial value of logic types, and it means that the value of such an element is uncertain. It is not explicitly used for implementing the RTL model behavior since it does not map any actual circuit value. Rather, it is used for low-level debugging. If an unknown value is observed after initialization of the design (usually, the reset phase) or during execution, it means that the circuit most likely contains a bug, since a non-deterministic behavior has been introduced. Therefore, a RTL design can have an explicit use of `'X'` only in conditional statements introduced for verification. When applying the proposed type abstraction technique, such debugging checks can be safely removed, while still obtaining a functionally equivalent design.

The *high-impedance value* `'Z'` is used for tri-state signals (i.e., signals with more than one driver). When a driver writes a `'Z'` on a signal, it actually allows other drivers to set the value. Thus, the explicit use of `'Z'` in a RTL design can occur in two cases only: in conditional statements inserted for debugging or in write operations. Debugging statements can be removed (as done for unknown values) by obtaining a functionally equivalent design. Write operations can also be removed since a correct design will always have a driver that sets a value different from `'Z'`.

The `'X'` meta value can also be implicitly introduced during simulation by the SystemC *resolution functions*. A resolution function is a HDL-dependent function that handles the four-valued logic. In the most common case, a `'X'` is generated when more than one driver tries to set different values (which are not `'Z'`) on the same signal. The generation of `'X'` notifies a design error as it represents a non-deterministic behavior of the actual circuit (e.g., the SystemC kernel by default stops the simulation whenever such a concurrent assignment happens). When applying the type abstraction methodology to such non deterministic designs, the resulting design will generate `'0'` or `'1'` non-deterministically instead of `'X'`.

The explicit use of meta values such as `'X'` and `'Z'` is always removable. On the other hand, as a drawback of the proposed type abstraction methodology, the design debugging may become harder, since debugging features useful for checking low level concurrency are removed (such as, explicit and implicit checks on `'X'` and `'Z'` logic values). Moreover, a design that makes explicit use of write operations with `'Z'` cannot be synthesized after type abstraction, since synthesis tools require to know whether a component has multiple drivers.

Thus, it makes sense to abstract multi-value logic when simulation aims at verifying high-level functionality rather than debugging low level HW behavior. However, in case of debugging, *HDTLib* can be adopted without type abstraction with an accepted loss of performance.

The proposed type abstraction methodology is suitable for functional simulation, especially when the design is abstracted from RTL to TLM. When the design has been proven

to be functionally correct, the lower level details can be re-introduced, and a slower but still efficient simulation can be performed by using *HDTlib* instead of the ASI SystemC data type library.

Finally, the reported analysis holds supposing that the original design is written following usual conventions of RTL designs. A design could present explicit unknown and high-impedance values for reasons different from those reported in this work. Such occurrences can be simulated but not synthesized. In other words, such a design would be incorrect.

To summarize, the algorithm for abstracting the multi value types consists of the following steps:

1. Replace each single logic bit with a boolean.
2. Replace each logic vector with a bit vector.
3. Remove any assignment statement containing an explicit `'X'` or `'Z'`.
4. Remove any condition statement and the corresponding branches containing an explicit `'X'` or `'Z'`.

The type abstraction algorithm has been implemented in the HIFSuite tool [27] and applied to several designs to analyze the simulation speed-up provided by the type abstraction, as reported in Sect. 6.

5 *HDTLib* and GP-GPUs

In recent years, GP-GPUs have been investigated as a new general purpose computing platform. Compute Unified Device Architecture (CUDA) is a C library extension developed by NVIDIA to provide a programming interface to GPU devices [13]. The host CPU is responsible for starting the main program and for executing serial code, while delegating parallel execution of compute-intensive tasks to the GPU device. The CUDA programming requires the definition of C functions, called *kernels*, which are executed in parallel by multiple GPU *threads* when invoked. These threads run the same kernel concurrently, and each one is associated with a unique thread ID. A kernel is executed by a two-dimensional *grid* of thread *blocks*. Threads are arranged into three-dimensional thread blocks.

The high number of available cores in GP-GPUs makes these architectures ideal candidates to accelerate simulation of SystemC designs, as proposed in [10–12]. In [10, 11], the authors propose to speed-up the simulation of both RTL and TLM SystemC designs by automatically translating SystemC designs into CUDA designs that can be executed in parallel. This is achieved by transforming the model of computation of SystemC discrete-event simulation into a model of concurrent GPU threads that synchronize as and when necessary. In [12], the authors present a framework for functional verification of RTL designs, which is based on fault injection and parallel simulation on GP-GPUs. Given a fault model, the framework translates the RTL code into an injected C code targeting NVIDIA GPUs, thus allowing a very fast parallel automatic test pattern generation and fault simulation.

Both these works focus on re-implementing the SystemC process scheduler in order to allow parallel execution on GP-GPUs. In fact, the main issue for simulating SystemC on CUDA is porting the process scheduling. Simulation of a SystemC description is overseen by the SystemC process scheduler, which properly executes and awakens processes according to the events happening during simulation. In standard DES simulators, this function is performed by the SystemC simulation kernel, which is external to the user-defined design. In contrast, when targeting GP-GPUs, such a scheduling function must be integrated within the kernel function (see Fig. 5) and can be parallel ([10, 11]) or single thread-based ([12]).

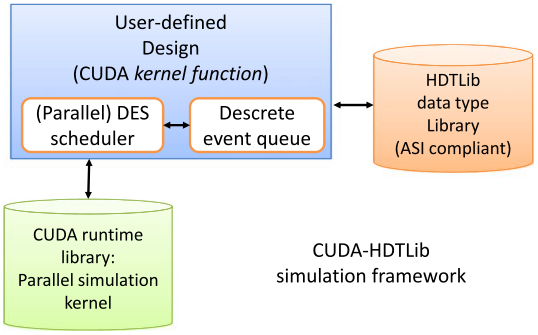


Fig. 5 SystemC simulation on CUDA GP-GPUs

The kernel function is executed by GPU threads in parallel, and is responsible for performing a simulation run of the description. By carefully structuring a design description in such a kernel function and other auxiliary functions, it is possible to exploit the computing power provided by the GPU device to perform parallel simulation runs of the description, thus achieving a significant simulation speed-up.

Nevertheless, no mention is made about data types. To allow parallel simulation runs, all the statements of the SystemC description must be executable by each single GPU thread, including data types operations. At the state of the art, to do that, designers have two alternatives:

1. To re-implement the design by using native C data types. Single bit or logic values can be mapped to the `bool` type, whereas bit or logic vectors can be mapped to the `unsigned int` type. By mapping to native data types, only built-in operators will be used, which can be natively executed by a single GPU thread. However, this mapping does not preserve multi-value logic accuracy, and requires significant manual changes in the code to properly re-implement all data type operations (e.g., range or bit selection, concatenation, etc.) performed in the description in terms of C primitive data types. Additionally, special care must be taken to ensure that bit accuracy is preserved, since primitive data types have architecture-dependent bit widths.
2. To modify the standard SystemC data type library source code in order to allow its execution on the GPU device. The CUDA framework requires for kernel and support functions that run on the GPU device to be explicitly marked in the code. Furthermore, a few limitations apply on the code, due to the NVIDIA CUDA architecture, as detailed in the following paragraphs.

Both the options require manual effort. The first option requires time-consuming and error-prone modifications to the user-defined design. The second option requires an amount of manual modifications simply too daunting due to the complex and huge class structure of the ASI data type library.

HDTlib allows designers to overcome this limitation thanks to its minimal class hierarchy. To apply to GP-GPU architectures, and in particular, to the CUDA frameworks, two categories of methods of the original library have been modified as follows:

1. *Output methods* (i.e., methods printing the representation of a data object on screen). GPU threads are not allowed to access the standard output (i.e., the output screen) and thus only code running on the host CPU can print on the standard output. As such, output data must be sent back from the GPU device to the host CPU in order to be printed. However, this is more a limitation on CUDA side, and does not interfere with the use of *HDTlib*.

2. *Methods operating on strings* (i.e., methods that initialize a vector from its string representation and methods that return the string representation of a vector). In fact, the `string` class belongs to the C++ standard library and thus its source code can not be modified.

In particular, the code modifications of the data type library have been the following:

1. *Adding the __device__ qualifier to methods*. This qualifier declares that a method can be executed and invoked only by the device. This allows to execute all operators on data types by GPU threads. All methods for each data type class are marked with this qualifier except for output methods and methods operating on strings.
2. *Removing output of error messages.* This modification is required because GPU threads cannot access the error (or the output) stream, as previously stated. Any operation on *HDTlib* data types generating errors is not supported in CUDA.
3. *Substituting string with arrays of characters in methods operating on strings*. This step allows to circumvent the string class of the C++ standard library, still retaining the functionalities related to string representations of data types. For example, this change allows designers to initialize a bit vector from its string representation (e.g. `ta_bv_t<8> a("00110011");`) also in the GPU device code.

After applying these changes, a SystemC description that adopts *HDTlib* data types can be automatically converted into a corresponding CUDA description, without requiring designers to modify or re-implement all the data type operations performed in the description.

6 Experimental results

Three different sets of experiments have been conducted to evaluate the impact of *HDTLib* on the overall simulation performance. The first set focuses on basic operations on bit and logic vectors, by applying the library to synthetic SystemC designs. This allows us to evaluate the different impact of the data types and the corresponding operators on the overall simulation speed.

In the second set, *HDTLib* has been applied to actual designs provided by industrial partners to analyze the performance improvement on real test cases with different architectural characteristics and coding styles.

In the third set, the industrial designs have been simulated on a GP-GPU device through the CUDA framework, in order to verify the correctness and performance of *HDTLib* on CUDA architectures.

The first two sets of experiments have been carried out on a 64-bit Linux server with four 2.83 GHz CPU cores and 4 GB RAM memory. The third set of experiments has been performed on a 64-bit Linux server with six 2.8 GHz CPU cores and equipped with a NVIDIA GeForce GTX 460 device.

6.1 Basic operation benchmarking

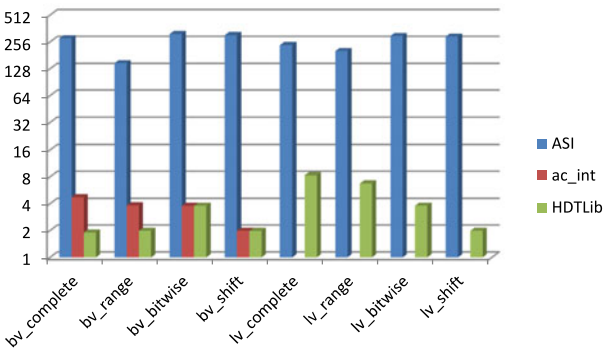
The first set of experiments consists of synthetic SystemC benchmarks containing a number of basic operations on bit and logic vectors repeated in a loop. The basic operations are divided in the following classes:

– initializations from unsigned integers;
– range selections and assignments;

Table 4 Simulation time (in seconds) of synthetic benchmarks

Benchmark	ASI	ac_int	*HDTLib*	Speed-up (×)
bv_complete	281.05	4.72	1.90	147.9
bv_range	147.82	3.83	1.99	74.3
bv_bitwise	313.89	3.79	3.79	82.8
bv_shift	305.52	1.99	1.98	154.3
lv_complete	236.41	–	8.33	28.4
lv_range	202.07	–	6.74	30.0
lv_bitwise	298.72	–	3.80	78.6
lv_shift	294.24	–	1.99	147.9

Fig. 6 Simulation time (in seconds) of synthetic benchmarks



– bitwise operations;
– shift operations;
– single bit selections and assignments.

Benchmarks `bv_complete` and `lv_complete` test all the operation classes within an iteration, while all the remaining benchmarks perform only a specific operation class.

Three data type libraries have been compared: ASI SystemC, *Algorithmic C* [15], and *HDTLib*. Although the Algorithmic C `ac_int` type implements an arbitrary bitwidth integer, it has been used as a surrogate for representing a bit vector, since it provides all bit-accurate operations required for this purpose. However, `ac_int` does not provide support for logic vectors. For this reason, it has been applied only to the first four benchmarks, which pertain to bit vectors.

Results are reported in Table 4 and represented in Fig. 6. Time needed to adapt the benchmarks to be compliant with the proposed technique is not reported, since the automatic tool makes this step instantaneous. All simulation times are in seconds and have been measured by using the `time` command (by considering both system time and user time). The speed-up reported in Table 4 concerns the simulation improvement of *HDTLib* with regard to the standard ASI. This set of experiments exposes the slowness of bit and logic vectors in the ASI implementation, as *HDTLib* provides a speed-up up to 154.3× with respect to the ASI implementation. Furthermore, such efficiency is not obtained to the detriment of accuracy, as *HDTLib* allows to keep multi-value logic accuracy, while Algorithmic C does not.

Table 5 RTL IP module characteristics

RTL IP module	PIs (#)	POs (#)	Gates (#)	FF (#)	SystemC (loc)
adpcm	66	35	24,412	364	305
dist	34	66	400	35	84
div	35	33	248	19	58
root	35	33	682	59	119
gcd	67	66	636	51	100
fft	92	114	87,397	1,359	3,335
ecc	25	32	993	79	175
crc	56	34	9,213	385	492

6.2 Industrial design benchmarking

In an actual industrial design, a number of factors come into play, such as simulation kernel overhead, synchronization between processes, etc. However, a fast implementation of data types is fundamental for increasing simulation speed.

HDTLib and the data type abstraction have been applied to six industrial case studies:

- DIST, DIV, ROOT, CRC and ECC have been provided by STMicroelectronics;
- FFT has been provided from CEA LETI;
- ADPCM and GCD have been downloaded from OpenCores.

Table 5 reports the structural characteristics of the RTL IP models: *PIs* (Primary Inputs), *POs* (Primary Outputs), *Gates*, *FF* (Flip Flops) and *SystemC (loc)* (number of lines of HDL code).

Six different implementations for each design have been considered:

- ASI SystemC RTL version;
- RTL version with *HDTLib*;
- RTL version with *HDTLib* and type abstraction;
- ASI SystemC TLM version;
- TLM version with *HDTLib*;
- TLM version with *HDTLib* and type abstraction.

The simulation results are reported in Table 6 and Table 7 for the RTL and TLM versions, respectively. Columns *ASI RTL*, *HDTLib RTL* and *HDTLib RTL + TA* in Table 6 indicate the elapsed simulation time in seconds for the ASI SystemC RTL version, the RTL version with *HDTLib* and the RTL version with *HDTLib* and type abstraction, respectively. Column *Speed-up RTL* shows the speed-up obtained by using *HDTLib* and the proposed type abstraction methodology at RTL. Results show that the advantage of using the accurate data type of *HDTLib* at RTL is limited by the impact of the SystemC scheduling activity, which affects simulation speed. This happens for example in designs ADPCM and GCD, which have a much lower speed-up with respect to the other designs since simulation overhead is higher than the actual functional simulation. On the other hand, the results show that *HDTLib* always provides at least a fair simulation speed-up, still preserving the simulation correctness. The type abstraction methodology further improves performance with a gain up to 6×.

Table 7 and Fig. 7 report and depict the simulation performance of the TLM versions of the same designs, obtained via RTL-to-TLM automatic abstraction. Columns *ASI TLM*,

Table 6 Simulation time and speed-up of RTL versions

Design	ASI RTL (s)	*HDTLib* RTL (s)	Speed-up RTL (×)	*HDTLib* RTL + TA (s)	Speed-up RTL + TA (×)
adpcm	482.49	323.86	1.49	320.18	1.51
dist	207.98	119.49	1.74	68.26	3.05
div	456.84	165.36	2.76	151.69	3.01
root	270.26	104.37	2.59	87.15	3.10
gcd	2,493.69	1,431.16	1.74	1,327.25	1.88
fft	950.58	275.11	3.45	154.51	6.15
ecc	158.82	99.21	1.60	73.78	2.15
crc	4,976.65	1,843.20	2.70	1,637.06	3.04

Table 7 Simulation time and speed-up of TLM versions

Design	ASI TLM (s)	*HDTLib* TLM (s)	Speed-up TLM (×)	*HDTLib* TLM + TA (s)	Speed-up TLM + TA (×)	Speed-up RTL-TLM (×)
adpcm	43.73	15.66	2.79	12.71	3.44	37.96
dist	102.32	14.06	7.28	5.22	19.60	39.84
div	74.04	7.40	10.00	6.35	11.66	71.94
root	136.21	18.47	7.37	15.46	8.81	17.48
gcd	14.31	12.20	1.18	6.09	2.35	409.47
fft	176.73	26.26	6.73	5.03	35.14	188.98
ecc	47.33	31.53	1.50	18.72	2.53	8.48
crc	8.77	8.02	1.09	2.17	4.03	2,287.03

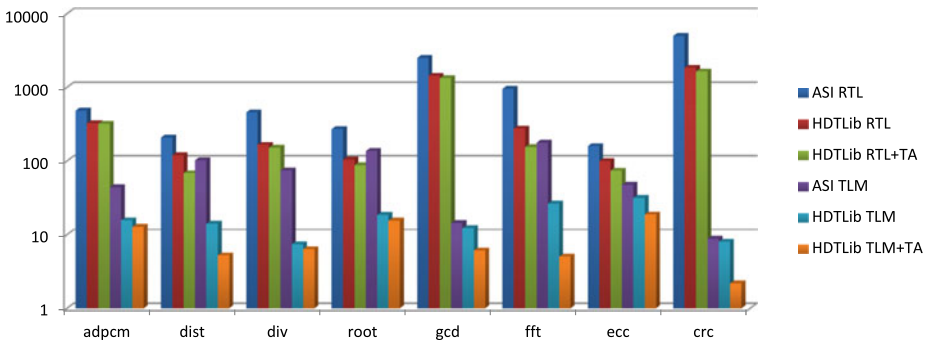


Fig. 7 Graph of experiment #2

HDTLib TLM and *HDTLib TLM + TA* indicate the elapsed simulation time in seconds for the ASI SystemC TLM version, the TLM version with *HDTLib* and the TLM version with *HDTLib* and type abstraction, respectively. Column *Speed-up TLM* shows the speed-up obtained by using *HDTLib* and the proposed type abstraction methodology at TLM. Simply abstracting the RTL design to a corresponding TLM implementation does not result in a significant speed-up if bit-accurate ASI data types are preserved. This is due to the slowness

Table 8 Simulation time comparison between single core and CUDA GP-GPU architectures

Design	Serial *HDTLib* (s)	Serial *HDTLib* + TA (s)	Serial C types (s)	CUDA *HDTLib* (s)	CUDA *HDTLib* + TA (s)	CUDA C types (s)
adpcm	803.1	645.7	616.9	7.78	7.01	6.99
dist	331.6	285.4	224.6	6.38	5.55	4.28
div	685.0	669.7	317.6	7.92	6.98	5.56
root	3,453.6	3,265.5	2,775.8	8.77	8.76	8.74
gcd	972.0	902.0	848.0	7.28	7.21	5.98
fft	–	–	–	–	–	–
ecc	60.3	57.3	54.1	1.92	1.83	1.35
crc	–	–	–	–	–	–

of the ASI implementation of such data types, that ends up absorbing the performance improvement obtained by moving up to a higher abstraction level. This is where *HDTLib* gives the highest benefit, by drastically lowering the simulation times thanks to an efficient and highly optimized implementation, gaining a 10× speed-up without the type abstraction and up to a 35× with type abstraction.

Column *Speed-up RTL-TLM* indicates the global speed-up obtained by abstracting the design from RTL to TLM and using *HDTLib* in conjunction with the proposed type abstraction methodology. The overall speed-up depends on a number of factors, such as the impact of accurate data types and how effective the abstraction algorithm is for handling the RTL functionality. For example, data type abstraction has a major impact (up to 35×) in designs such as DIST, DIV, ROOT and FFT, which use bitwise operations intensively. On the other hand, TLM abstraction has a huge impact on all designs. Thus, the best results are achieved when combining RTL-to-TLM abstraction with the use of *HDTLib*.

6.3 GP-GPU results

The third set of experiments has been carried out by simulating the industrial designs analyzed in the previous section on a GP-GPU device through the CUDA framework.

Six different versions for each design have been considered:

- Serial C++ version with C primitive data types;
- Serial C++ version with *HDTLib*;
- Serial C++ version with *HDTLib* and type abstraction;
- CUDA version with C primitives data types.
- CUDA version with *HDTLib*;
- CUDA version with *HDTLib* and type abstraction;

The C++ versions have been obtained from the starting SystemC descriptions by applying the methodology described in [24]. It is worth noting that using *HDTLib* did not require any change to the source code of the designs. On the contrary, the versions containing the C primitive data types have required manual modifications to the source code of the design in order to re-implement data type operations in terms of C primitive data types.

The results are reported in Table 8. Columns *Serial HDTLib*, *Serial HDTLib + TA* and *Serial C types* show the elapsed simulation time in seconds for the serial version with *HDTLib*, the serial version with *HDTLib* and type abstraction and the serial version with primitive C

data types, respectively. Columns *CUDA HDTLib*, *CUDA HDTLib + TA* and *CUDA C types* show the elapsed simulation time in seconds for the CUDA version with *HDTLib*, the serial version with *HDTLib* and type abstraction and the serial version with primitive C data types, respectively.

As expected and as already known by the literature on CUDA simulation, CUDA makes simulation faster than the starting SystemC code as well as any sequential version of the designs. The simulation speed-up greatly varies from design to design, according to the inherent features of each design, e.g., the ratio between control operations and operations on data types, the number of operations performed on data types for each iteration. The FFT and CRC code size was too large to successfully compile with the used CUDA framework (4.0). However, it is important to note that such designs may not have been executed on the GPU either by using native data types or the ASI SystemC data types, since the same memory limits apply. For all the other designs, it is worth noting that the simulation overhead introduced by *HDTLib* for supporting bitwise accuracy and multi-valued logic is negligible considering the overall massive speed-up provided by such many-core architectures.

7 Conclusions

This article presented *HDTLib*, a new library of data types for speeding up simulation of SystemC designs. The article also presented a methodology for abstracting data types to further increase the simulation performance when low level HW-specific details are not required in high-level descriptions like TLM designs. Then, the article showed how *HDTLib* has been implemented for applying to the today's many-core architectures, in particular to CUDA GP-GPUs. A set of experiments has been conducted to both synthetic and industrial designs to analyze (i) the impact of the different data types and corresponding operators on the overall simulation speed, (ii) the impact of the data type accuracy on the overall simulation by considering the abstraction level of the design implementation, and (iii) the difference of simulation speed obtained by adopting the standard ASI library, a commercial SystemC library, and *HDTLib*. Finally, a set of experimental results has been conducted by simulating SystemC design with *HDTLib* on a GP-GPU device. The results showed that the simulation overhead introduced by *HDTLib* for supporting bitwise accuracy and multi-valued logic is negligible considering the overall massive speed-up provided by such many-core architectures.

References

1. Ecker W, Esen V, Schonberg L, Steininger T, Velten M, Hull M (2007) Impact of description language, abstraction layer, and value representation on simulation performance. In: Proc of ACM/IEEE DATE, pp 1–6
2. Accellera Systems Initiative (2005) IEEE 1666-2005 Standard SystemC Language Reference Manual. http://www.accellera.org
3. Ecker W (2007) Impact of SystemC data types on execution speed. http://www-ti.informatik.uni-tuebingen.de/systemc/Documents/Presentation-15-UP2_ecker.pdf
4. Grotker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic, Norwell
5. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: ACM/IEEE CODES+ISSS, pp 19–24
6. TLM Transaction-Level Modeling Library (2009) OSCI TLM-2.0 Language Reference Manual. http://www.accellera.org
7. Ghenassia F et al (2003) Using transactional level models in a SoC design flow. Kluwer Academic, Dordrecht

8. Bombieri N, Fummi F, Pravadelli G (2011) Automatic abstraction of RTL IPs into equivalent TLM descriptions. IEEE Trans Comput 60(12):1730–1743
9. Carbon Design Systems (2012) Carbon design model studio. http://www.carbondesignsystems.com
10. Nanjundappa M, Patel HD, Bijoy AJ, Shukla SK (2010) SCGPSim: a fast SystemC simulator on GPUs. In: Proc of ACM/IEEE ASP-DAC, pp 149–154
11. Sinha R, Prakash A, Patel HD (2011) Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs. In: Proc of ACM/IEEE ASP-DAC, pp 1–6
12. Bombieri N, Fummi F, Guarnieri V (2012) FAST-GP: an RTL functional verification framework based on fault simulation on GP-GPUs. In: Proc of ACM/IEEE DATE, pp 1–6
13. NVIDIA (2012) Cuda home page. http://www.nvidia.com/object/cuda_home_new.html
14. Tumbush G, Hupp M (2007) Dramatically increase the performance of SystemC simulations. In: Proc of DVCon conference
15. Takach A, Gutberlet P, Waters S (2004) Fast bit-accurate C++ datatypes for functional system verification and synthesis. In: Proc of FDL conference, pp 337–345
16. Herrera F (2008) Heterogeneous specification and automatic software generation from SystemC for embedded systems. Ph.D. thesis, University of Cantabria
17. Chatterjee D, DeOrio A, Bertacco V (2009) Event-driven gate-level simulation with GP-GPUs. In: Proc of ACM/IEEE DAC, pp 557–562
18. Chatterjee D, DeOrio A, Bertacco V (2009) GCS: high-performance gate-level simulation with GP-GPUs. In: Proc of ACM/IEEE DATE, pp 1332–1337
19. Sen A, Aksanli B, Bozkurt M, Mert M (2010) Parallel cycle based logic simulation using graphics processing units. In: Proc of IEEE ISPDC, pp 71–78
20. Gulati K, Khatri SP (2008) Towards acceleration of fault simulation using graphics processing units. In: Proc of ACM/IEEE DAC, pp 822–827
21. Kochte MA, Schaal M, Wunderlich H-J, Zoellin CG (2010) Efficient fault simulation on many-core processors. In: Proc of ACM/IEEE DAC, pp 380–385
22. Li H, Xu D, Han Y, Cheng KT, Li X (2010) nGFSIM: a GPU-based fault simulator for 1-to-n detection and its applications. In: Proc of IEEE ITC, pp 1–10
23. Stoye W, Greaves D, Richards N, Green J (2003) Using RTL-to-C++ translation for large SoC concurrent engineering: a case study. IEEE/IET Electron Syst Softw 1(1):20–25
24. Bombieri N, Fummi F, Pravadelli G (2010) Abstraction of RTL IPs into embedded software. In: Proc of ACM/IEEE DAC, pp 24–29
25. Aldec DVM (2012) http://www.aldec.com
26. Snyder W, Wasson P, Galbi D (2012) Verilator—Convert Verilog code to C++/SystemC. http://www.veripool.org/wiki/verilator
27. Bombieri N, Di Guglielmo G, Ferrari M, Fummi F, Pravadelli G, Stefanni F, Venturelli A (2010) HIF-Suite: tools for HDL code conversion and manipulation. EURASIP J Embed Syst 2010:1–20
28. EDALab s.r.l. (2012) EDALab networked embedded systems. http://www.edalab.it/