SCGPSim: A Fast SystemC Simulator on GPUs

Mahesh Nanjundappa  
FERMAT Lab,  
Virginia Tech.,  
Blacksburg, VA, USA.  
knmahesh@vt.edu

Hiren D. Patel  
Dept of ECE,  
University of Waterloo,  
Waterloo, ON, Canada.  
hdpatel@uwaterloo.ca

Bijoy A. Jose  
FERMAT Lab,  
Virginia Tech.,  
Blacksburg, VA, USA.  
bijoy@vt.edu

Sandeep K. Shukla  
FERMAT Lab,  
Virginia Tech.,  
Blacksburg, VA, USA.  
shukla@vt.edu

Abstract—The main objective of this paper is to speed up the simulation performance of SystemC designs at the RTL abstraction level by exploiting the high degree of parallelism afforded by today's general purpose graphics processors (GPGPUs). Our approach parallelizes SystemC's discrete-event simulation (DES) on GPGPUs by transforming the model of computation of DES into a model of concurrent threads that synchronize as and when necessary. Unlike the cooperative threading model employed in the SystemC reference implementation, our threading model is capable of executing in parallel on the large number of simple processing units available on GPUs. Our simulation infrastructure is called SCGPSim and it includes a source-to-source (S2S) translator to transform synthesizable SystemC models into parallelly executable programs targeting an NVIDIA GPU. The translator retains the simulation semantics of the original designs by applying semantics preserving transformations. The resulting transformed models mapped onto the massively parallel architecture of GPUs improve simulation efficiency quite substantially. Preliminary experiments with varying-sized examples such as AES, ALU, and FIR have shown simulation speed-ups ranging from 30x to 100x. Considering that our transformations are not yet optimized, we believe that optimizing them will improve the simulation performance even further.

I INTRODUCTION

SystemC, a modeling and simulation language, has been used in making early trade-off analysis and design-space exploration. It has been shown that SystemC methodologies shorten design cycles allowing designers to meet the ever tightening time-to-market constraints. A key contributing factor to SystemC’s success is its ability to provide mechanisms for fast simulation. However, SystemC’s reference implementation makes no attempts to leverage the parallel architectures that are becoming increasing popular. The OSCI implementation of SystemC is built using application-level threading libraries (co-operative threads) that results in an unparallelizable simulation kernel, as co-operative threads cannot be scheduled by the operating system and hence cannot be dispatched to different processing elements. Clearly, this implementation is not designed to take advantage of multi-core processing systems.

A significant amount of effort has gone into improving the performance of SystemC simulations [9, 7, 6], but only a handful of attempts into parallelizing them [2, 8]. This paper presents a source-to-source (S2S) translator that transforms synthesizable SystemC designs into parallelly executable CUDA programs that run on NVIDIA Graphics Processing Units (GPUs). To convert the SystemC designs into CUDA designs, we use a programming model where co-operative threads can be mapped to parallel threads, and they can be efficiently simulated providing an order of magnitude decrease in simulation times under the constraint of preserving the existing SystemC simulation semantics. Note that we use semantic-preserving transformations to obtain the exact same behaviors as when using SystemC’s discrete-event scheduler. Our translator accepts synthesizable SystemC designs; however, we require modules in the designs to have registered outputs. As a result, we do not currently support combinatorial paths between modules. We use NVIDIA TESLA D870 GPU as an example target platform.

This paper is organized as follows, Section 2 presents related work. In Section 3 we present a brief introduction on the SystemC simulator. Section 4 presents the GPU architecture and the CUDA programming model. In Section 5 we explain the conversion procedure from SystemC to CUDA and the translator that does this automatic conversion. We present our results in Section 6 and conclude with future work in the last section.

II RELATED WORK

Several attempts have been made in the past on parallelizing a SystemC simulator, though none of them are targeted at a massively parallel platform such as a GPGPU. In [9] Savoiu et al. show a static transformation algorithm for SystemC models. This work tries to improve simulation efficiency by reducing synchronization overheads by changing the structure of the model completely to fuse together the various modules and co-operative threads into a much higher granularity thread. In [2], Kaouane et al. have proposed a design flow for a CELL processor based SystemC simulation. With limited number of synergistic processing elements, it requires a lot of context switches that reduces the efficiency. This implementation also has several constraints; for example, the processes are sensitive to a single signal, which is very rare in hardware modeling. In addition, they manually divided SystemC code to extract compute intensive and control intensive parts. Although much work has been done in parallel discrete-event simulation, to the best of our knowledge no true parallel implementation of SystemC exists. In [8] Perumalla evaluated discrete-event simulation on GPUs, but it yielded no real speed up. In fact, the simulation time was more than what CPU would take without the GPUs. Naguib et al. [3] showed that by means of process splitting, one can speed up the SystemC simulations by automatically optimizing the model for simulation with results that showed...
15% improvement. Our approach shows a significant simulation performance improvement over existing approaches. We implement a GPU based cycle-accurate simulator by automatically carrying out a source-to-source transformation on SystemC designs to CUDA and executing it efficiently on the GPU.

III SYSTEMC SIMULATION KERNEL

SystemC is an open-source system-level design language based on C++ that has its own simulation kernel. The SystemC simulation kernel is a run-time scheduler that handles both the synchronization and scheduling of concurrent processes. It implements a discrete-event scheduler that executes processes in response to occurrence of events. When an event occurs, the corresponding process is pushed into the runnable processes queue for execution. The scheduler maintains a queue of runnable processes, and it keeps executing them until they complete execution or reach a suspension `wait()` statement.

otherwise, if there are timed notifications the simulator enters the `Timed notification phase` where it advances the current simulation time to the earliest pending timed notification and reenters the Evaluate phase. If there are no timed notifications, the simulation is finished. While this approach works perfectly fine for a single-core processor, it does not scale and take advantage of multi-core/multi-processor environment.

IV THE CUDA PROGRAMMING MODEL

CUDA (Compute Unified Device Architecture) is an extension to the C language that exploits the processing power of GPUs to solve complex compute-intensive problems efficiently. High performance is achieved by launching a number of threads and making each thread execute a part of the application in parallel. These threads are grouped and arranged as blocks. Each block contains maximum of 512 threads. These blocks are arranged in 2 dimensional matrix structures to form a grid. Each grid consists of a maximum of 65536 blocks. Figure 2 illustrates the structure.

Threads from the same block access fast shared on-chip memory and can be synchronized using built-in functions. Threads from different blocks can communicate data using the slower global memory. CUDA thread execution differs from that of CPU thread execution in terms of scheduling of threads. In CUDA, threads are grouped to form warps and these warps are executed in single-instruction multiple-thread (SIMT) way. All threads in a warp execute one instruction at a time. If threads that belong to a warp diverge at a data-dependent conditional branch, then they are executed sequentially and they all converge on to the same execution path in later instructions. Branch divergence applies only for threads of a particular warp; threads from different warps execute independently on different multiprocessors. We have to create threads that belong to different warps to get the maximum speed up.

A Extracting the Concurrency

A general SystemC design consists of producer/stimulus, consumer/response and channel/computation modules. The channel consists of numerous modules which model the behavior of the hardware. There may be one or many producers and consumers. Figure 3 shows the execution model of “N” runnable processes with one producer and one consumer process. Simulation of the design continues until all the processes get suspended, i.e. the end of simulation condition has been met. Threads created during SystemC simulation are cooperative threads and they effectively execute sequentially even in a multiprocessor environment. Figure 4 shows the proposed

Fig. 1.: OSCI SystemC Simulation Kernel [5]

Figure 1 shows the flow of SystemC simulation kernel. In the Initialization phase all processes are executed in an unspecified order. In the Evaluation phase, runnable processes are selected and executed until no runnable processes exist. This can cause additional processes to become ready to run in this phase due to immediate event notifications. When no such ready to run processes exist, the simulator enters the Update phase, where signal values are updated to the values computed in the evaluate phase. At this point, if there are any pending delayed notifications, the simulator enters Delayed notification phase that determines which processes are ready to run due to the delayed notifications and returns to the Evaluate phase. Otherwise, if there are timed notifications the simulator enters the Timed notification phase where it advances the current simulation time to the earliest pending timed notification and reenters the Evaluate phase. If there are no timed notifications, the simulation is finished. While this approach works perfectly fine for a single-core processor, it does not scale and take advantage of multi-core/multi-processor environment.

Fig. 2.: CUDA Threading Model [4]

The bottleneck in the CUDA programming model is the data transfer rate between the Central Processing Unit (CPU) and the Graphical Processing Unit (GPU). Currently this is limited by the maximum speed of the PCIe bus bandwidth. The overall performance of the applications depend significantly on the strategy of use of the on-chip GPU memory. Frequent memory transfers between CPU and GPU deteriorates the performance. This does not hamper efficiency in our case as we do not communicate with GPU frequently.

V SYSTEMC TO CUDA

A Extracting the Concurrency

A general SystemC design consists of producer/stimulus, consumer/response and channel/computation modules. The channel consists of numerous modules which model the behavior of the hardware. There may be one or many producers and consumers. Figure 3 shows the execution model of “N” runnable processes with one producer and one consumer process. Simulation of the design continues until all the processes get suspended, i.e. the end of simulation condition has been met. Threads created during SystemC simulation are cooperative threads and they effectively execute sequentially even in a multiprocessor environment. Figure 4 shows the proposed
parallel execution model of the SystemC design. Individual threads are executed in parallel and barriers are used for synchronization between all the threads. To prevent the race condition on shared variables we use “Double buffering” on such variables.

### B Program Structure

In this section we describe the structure of the CUDA program used in our implementation. In CUDA architecture all the threads can see the whole kernel code. The only way to make each thread execute a part of the whole code is by distinguishing that part of code with a threadIdx. Listing-1 shows a CUDA KERNEL CODE structure with four threads executing different parts of the whole code for MAX_CYCLES number of times. The in-built function __syncthreads() is used for synchronization between the various threads.

#### Listing 1: Sample CUDA Kernel Code

```c
// Shared variable declarations
__global__ static void sample_kernel(arguments) {
    int idx = threadIdx.x;
    // Any local variable declarations
    for(i=0;i<MAX_CYCLES;i++) {
        if(idx == 0) {
            // Code that has to be executed by thread 0
        }
        else if(idx == 1) {
            // Code that has to be executed by thread 1
        }
        ....
        else if(idx == 3) {
            // Code that has to be executed by thread 3
        }
        __syncthreads(); // Barrier Synchronization
    }
}
```

### C Design Flow

Figure 5 shows the overall design flow of SCGPSim. We divide the entire process of SystemC simulation on CUDA into 5 different phases as explained below in detail.

#### C.1 Design Phase

This phase starts with the writing of a top module, and identifying modules that are to be implemented as SC_MODULEs and
SC_THREADS. We do not place any restriction on design being implemented as SC_MODULE or SC_THREAD. We also do not put any restrictions on wait() statements or the sensitivity list or while(true) loops. Instead of writing printf() statements, we accumulate the outputs in an array and display them at the end. The reason for this is that frequent memory transfers between CPU and GPU is costly. Hence, we accumulate the results in GPU and do block memory transfers at defined points. Other restrictions are the use of FILE handling and string compare functions. Currently these are not supported. In addition we require the user to specify producer and consumer modules explicitly by following the nomenclature of having STIMULUS name for producer and OUTPUT name for consumer module.

C.2 XMLization and Extraction Phase

Doxygen tool is used to capture classes declared, private and public members and functions and represents them as an Abstract Syntax Tree (AST) in XML format including the source. We then parse the AST and extract port and signal names, their types and sizes, modules, sub-modules, and entry functions. This is done using SystemCXML [1].

C.3 Synthesis Phase

We generate CUDA code using the structural information and flattened design file. A flattened design file is obtained by concatenating all the design files including top module. We obtain the set of all SystemC module instances M by parsing the output from SystemCXML. A module instance \( m \in M \) is a tuple \((F_m, F_t, S)\). This tuple defines the set of entry functions \( F = F_m \cup F_t \) as a union of the entry functions synchronously triggered in SC_METHOD processes \( F_m \), and SC_THREAD processes \( F_t \) respectively. Note that we represent synchronously triggered sub-modules within \( M \) as they are flattened in the simulation kernel. Currently, we do not support combinational paths defined using the module hierarchy; however, allowing this is a trivial extension requiring flow analysis and appropriate ordering. We also define \( S \) as the set of all SC_signal and variable declarations. A function \( f \in F \) is captured as a sequence of computation blocks \( b^i \), and suspension points \( w_k \) for some \( i, k \in N \) and \( j \in \{0, 1\} \).

In a program with no conditional branches, \( f \) is simply a sequence of \( b^i \) followed by an infinite loop computation block, then some more \( b^i \), and \( w_k \) and so on. However, for conditional branches, we denote \( j = 1 \) as the true case, and \( j = 0 \) as the false case. Without further overburdening the reader with notation, we show an example that illustrates this encoding in Listing 2 and Listing 3. This example does not have any conditional branches, so its encoding is \( < b^1_1, b^1_2, \ldots, w^1_k, b^1_{k+1}, w_{k+1}, b^1_{k+2}, \ldots > \). We map each SC_MODULE/SC_THREAD to an independent CUDA thread. All processes and functions that belong to that module are executed by the same thread. Race conditions for signals and ports are handled by using the double buffering mechanism. Algorithms 1, 2 and 3 explain the overall conversion procedure. In Algorithm 1, Line 1, we collect all SC_signals in \( S_{all} \) and declare them in KERNEL CODE (c). Double buffering is enabled on signals that are shared or signals that are present in sensitivity list. This is shown in Algorithm 3. We then parse all the module instances (\( M \)), and check if they are SC_THREADS (\( F_t \)) or SC_METHODS (\( F_m \)). If they are methods, we check if the statements are CUDA compatible (Algorithm 2) or not, and if they are compatible, we reproduce the same statements. If not compatible, we make it compatible by applying further transformations not shown in this overview.

```
Algorithm 1: SCGPSim synthesizer: Translating SystemC Design to CUDA Design

/* Collect all signals. */
1 Sall ← \( \bigcup_{m \in M} m.S \)
/* Convert all signals. */
2 writeVariableBuffer(c.S, Sall)
/* Generate CUDA kernel. */
3 for \( m \in M \) do
   /* Enter loop only if \( m \in F_t \) */
   for \( f \in m.F_t \) do
      \( b_n ← head(f), n ← 0, loc ← 0 \)
      /* Generate thread code. */
      while isWhileTrue(b_n) \( \neq \) true do
         if isWait(b_n) then
            \( c.f^c ← c.f^c \cdot writeCaseEndBlock(loc) \)
            \( loc ← loc + 1 \)
         else
            \( c.f^c ← c.f^c \cdot writeNewCaseBeginBlock(loc) \)
         end
         \( n ← n + 1 \)
      end
      \( c.f^c ← c.f^c \cdot checkCompatible(b_n) \)
   end
end
Listing 2: SystemC Example using wait()s

//proc1 is sensitive to variaible!
void proc1() {
   //OTHER STATEMENTS
   while(true) {
      variable2++;
      wait();
      variable3++;
      wait();
      result = variable1+variable2+variable3;
   }
}
Listing 3: CUDA wait() Equivalent

if(thread.idx == 23) { //Random thread number
   if(variable1!=variable1_copy) {
      if(location_thread_23 == 0) {
         //OTHER STATEMENTS
         switch(location_thread_23) {
            case 0: variable2++;
                     location_thread_23++;
                     break;
            case 1: variable3++;
                     location_thread_23++;
                     break;
            case 2: result = variable1+variable2+variable3;
                     location_thread_23 = 0;
                     break;
         }
      }
   }
}
```
let us consider a SystemC model with threads \( T \), ... If an ... characters. We then proceed to the next statement, and similarly continue for all statements until we reach the infinite loop. Once we reach an infinite loop, we have to apply transformations such that the code gets transformed as described in listing 2 and 3. We start with writing a switch statement (line 10) and proceed. We keep writing the statements for the current case (line 19), until we encounter a suspension point (wait()). When we encounter a suspension point, we end the current case, increment the case number (loc) and start a new case (line 16). This process is repeated until there are no more statements left in \( f \). This entire thread to CUDA function conversion (lines 5 to 23) is repeated for all the functions in \( m.F_i \). After all the module instances are parsed, we finally output the CUDA kernel.

### C.4 Execution Phase

In this phase, we compile the synthesized CUDA kernel code (c) by adding other features like formatting the output to display on standard output device or to convert it into waveform. The simulation results are to be transferred from GPU memory to CPU memory before we output. This also can be done automatically by writing a script.

### D Reasoning about Equivalence

Let us consider a SystemC model with threads \( T_1, T_2, \ldots, T_n \). The authors in [10] have shown how to convert an SC_THREAD into an SC_METHOD by creating a state machine embedded inside the SC_METHOD. If an SC_THREAD \( T \) contains \( k \) wait() statements, then the state machine contains \( k \) states. Minor differences in structure of thread \( T \) can be taken care in a similar fashion. The transformation to SC_METHOD results in:

```c
static int i = 1;
switch(i)
    case 1: s.1; i++;
    break;
    case 2: s.2; i++;
    break;
    ...
    case k; s.k; i = 1;
```

We call a thread \( T \) which runs in an infinite loop, and has \( k \) wait() statements, as a k-periodic thread whose behavior repeats every \( k \) cycles. Each thread can be transformed to a hierarchical state machine, such that it has \( k \) macro states, and a transition of state \( k \) takes the state machine back to state 1. The state transitions happen at clock ticks of the global clock. Within each macro state, there is a micro state machine which is basically the state machine encoding the behaviors of the \( s_i.s \). When a macro state is reached in the state machine, the inner micro state machine gets to its initial state. The transitions of the micro state machine are taken based on the sensitivity list of various non-state signals and variables.

The overall behavior of simulation kernel can be described as follows: Initially, all macro states are in their state 1. At this point, each micro state machine is in its initial state. When input changes, or some other signal changes for which the threads are sensitive to, transitions that depend on those signals take place in the micro state machines in all the different macro states, in parallel. If such a change leads to an output during transition, and some threads are sensitive to those outputs, then some of the micro state machines may have to take more transitions. This way the delta cycles are simulated. When all the micro state transitions take place, and none are enabled, then the macro state gains control. At this time, at the global clock tick, each macro state machine changes their state to another state in lock step. During this step, they also update all the state variables. Then the control goes back to the microstate machines within each macro state, and they work as described before. This continues until the number of clock cycles being simulated gets over, or some other condition terminates the simulation. Such a condition is modeled by another state machine external to all these state machines.

Having this view of the SystemC simulation makes it straightforward for us to transform the regular SystemC model to CUDA thread model. Each thread \( T_1 \) is mapped to a CUDA thread belonging to a distinct warp. Then \( T_1 \) progresses by modeling the macro state transitions as Barrier synchronizations, and micro state transitions as computations in each thread in distinct PEs. The fact that the micro state machines require to transition based on changes in signals from outside (inputs) or on signals that are changed by microstate machines belonging to other threads, requires that the communication between the CUDA threads on these signals is fast. Therefore, these communicate on the block shared memory. However, when the barrier synchronization happens, and states are changed, they are done in the global memory. In this version of the paper, an intuitive argument is presented but a future version of the paper will contain the rigorous formalization and proof.

### E Example: AES

In this section, we explain the conversion of a fully pipelined implementation of Advanced Encryption Standard (AES) in
SystemC to CUDA. There are 10 stages of AES, added to that are input and output stages. The SystemC implementation is done by implementing each stage as a SC_THREAD. Figure 6 shows the implementation hierarchy of the AES in SystemC with 12 SC_THREAD modules.

Fig. 6: Hierarchy of pipelined AES implementation in SystemC

Each of the SC_THREADS execute sequentially and sensitive to change in clock. In CUDA implementation we create 12 threads; one thread for each stage, one for input stage and the other one for output stage. The threadIDs are selected such that threads belong to different warps. All these threads execute concurrently and synchronize by barrier synchronization.

It is out of the scope of the paper to present all the conversion rules. Hence we present a few useful rules. As shown in Listing 2 and 3 wait() statements and while(true) loops get converted to switch statements. Sensitivity lists get converted to if based conditional statements. If a process is sensitive to many variables, disjunction of change in values of all variables is checked in the if condition statement. All variables that are declared as sc_signals get declared as shared variables and reside in shared memory. The hard-coded values like coefficient arrays, scaling factors, etc., get declared as constant variables and reside in constant memory. The read() and write() functions in SystemC get converted to assignment operators with overflow check. For simplicity, these conversions are not shown in the example. Figure 7 shows the variation of execution time taken for AES execution by CPU and GPU for different number of cycles starting from 10,000 cycles to 100,000 cycles, and the performance gains of approximately 30x.

Fig. 7: Time taken versus Number of cycles for pipeline AES

VI EXPERIMENTAL RESULTS

In this section we present some of the results of SystemC simulation on GPUs. SystemC CPU simulation was done on Intel Core2Duo Windows XP laptop running at 1.83Ghz with 1.5GB RAM, 987Mhz, while GPU simulation was done on NVIDIA’s TESLA D870. Our primary goal was to maintain the simulation semantics of SystemC simulation kernel. For this purpose, we store the results obtained from CPU and GPU simulations and do a byte-to-byte comparison of the two files. We did SystemC to CUDA conversion of the few examples and compared simulation results for accuracy. Semantics of SystemC simulation kernel was preserved. We simulated each of the designs for 100,000 cycles. The execution times of CPU and CPU+GPU are shown in Table 1. Execution time includes even memory transfer time in case of CPU+GPU. The inputs to the experiments were randomized. Increased simulation speed up is observed due to the fact that concurrent parts of the design are being executed in parallel and there is no need of generation of delta events. The speed up greatly depends on memory access to computation ratio. In examples which have frequent global memory access (AES) tend to show lesser speed ups as compared to others which have lesser frequency of global memory access which is expected in GPU architecture.

<table>
<thead>
<tr>
<th>Design</th>
<th>CPU (sec)</th>
<th>CPU + GPU (sec)</th>
<th>SPEED UP</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIPELINE AES</td>
<td>120.1</td>
<td>3.916</td>
<td>30.66</td>
</tr>
<tr>
<td>FIR</td>
<td>51.9</td>
<td>1.37</td>
<td>37.88</td>
</tr>
<tr>
<td>10 STAGE BUFFER</td>
<td>28</td>
<td>0.277</td>
<td>101.083</td>
</tr>
<tr>
<td>3 STAGE BUFFER</td>
<td>11</td>
<td>0.276</td>
<td>39.85</td>
</tr>
<tr>
<td>SIMPLE ALU</td>
<td>13</td>
<td>0.146</td>
<td>89.041</td>
</tr>
</tbody>
</table>

VII CONCLUSIONS

This work presents a method to automatically translate synthesizable SystemC designs into parallely executable CUDA designs. The CUDA synthesis employs semantic-preserving transformations such that the discrete-event semantics of SystemC’s reference implementation is preserved. We present the core of our algorithm, and an intuitive argument for the equivalence of the parallel design with the original design. The simulation results show that we can obtain speedups ranging from 30x-100x. Our future work involves enabling the synthesizer with translations of designs beyond the synthesizable subset of SystemC, and presenting a formal proof of equivalence.

REFERENCES