Accelerating Hardware Simulation on Multi-cores

Mahesh Nanjundappa

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Sandeep K. Shukla, Chair
Dong Ha, Member
Patrick Schaumont, Member

May 4, 2010
Blacksburg, Virginia

Keywords: CUDA, SystemC Simulation, GPGPU, Multi-core simulation, Threading building blocks, POSIX threads, Design for Testing, Discrete Event Simulation (DES)

Accelerating Hardware Simulation on Multi-cores

Mahesh Nanjundappa

Abstract

Electronic design automation (EDA) tools play a central role in bridging the productivity gap for designing complex hardware systems. However, with an increase in the size and complexity of today's design requirements, current methodologies and EDA tools are unable to effectively mitigate the further widening of productivity gap. It is estimated that testing and verification takes $(\frac{2}{3})^{rd}$ of the total development time of complex hardware systems. Functional simulation forms the main stay of testing and verification process and is the most widely used technique for testing and verification. Most of the simulation algorithms and their implementations are designed for uniprocessor systems that cannot easily leverage the parallelism in multi-core and GPU platforms. For example, logic simulation often uses levelized sequential algorithms, whereas the discrete-event simulation frameworks for Verilog, VHDL and SystemC employ concurrency in the form of multi-threading to given an illusion of the inherent parallelism present in circuits. However, the discrete-event model of computation requires a global notion of an event-queue, which makes improving its simulation performance via parallelization even more challenging. This work investigates automatic parallelization of simulation algorithms used to simulate hardware models. In particular, we focus on parallelizing the simulation of hardware designs described at the RTL using SystemC/HDL with examples to clearly describe the parallelization. Even though multi-cores and GPUs offer parallelism, efficiently exploiting this parallelism with their programming models is not straightforward. To overcome this, we also focus our research on building intelligent translators to map simulation applications onto multi-cores and GPUs such that the complexity of the low-level programming models is hidden from the designers.

This work was funded by a NSF-CPA funding.

Acknowledgements

Firstly I would thank the almighty for my life. May your name be honored and glorified.

There are three souls in this world, without them in my life, this thesis was a mirage. I sincerely bow to all the three stalwarts I've come across in my life and thank them for being part of my life.

Graduate students are like new horses in an race field. Unless a jockey trusts a horse, puts faith in it, trains it and shows the right path to success, the horse will never achieve glory. I'm extremely grateful to have an advisor like Dr. Sandeep K. Shukla who trusted, trained and gave a complete turn to my life. We also acknowledge the support of NSF which provided the funding for the work reported in this Thesis.

Apart from a jockey, a caretaker is a person who constantly takes care of routine problems selflessly. I was lucky enough to have Dr. Hiren D. Patel as a guide and a friend who provided a helping hand which I can bank on anytime.

For the horses to compete at the top level, they need to be provided with peer support, motivation and trained in a competitive environment. I'm highly indebted to mayor, cheta, zombie, lazy man, slim shady and goobey for supporting me through the thick and thin of my Master's journey.

All the training, caretaking and motivating goes in vain unless there are spectators who cheer up the horses. I thank all my friends, family members, relatives who cheered me up for this voyage.

Lastly I offer my heart felt regards to everyone who supported me in any respect during this exciting phase of my life.

Dedication

I dedicate this thesis to....

Amma

&

Appa

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Functional Validation

The ever increasing urge for higher performance has caused a rapid increase the complexity of hardware systems. But designers face a tough challenge in designing, testing and verifying these complex systems under the ever reducing time-to-market constraints. With the drastic increase in complexity of designs, the current verification tools are lagging a long way behind the requirements. This fact can be observed from Figure 1.1 [1] which shows a larger verification gap as compared to design gap over the years and the gap keeps widening.

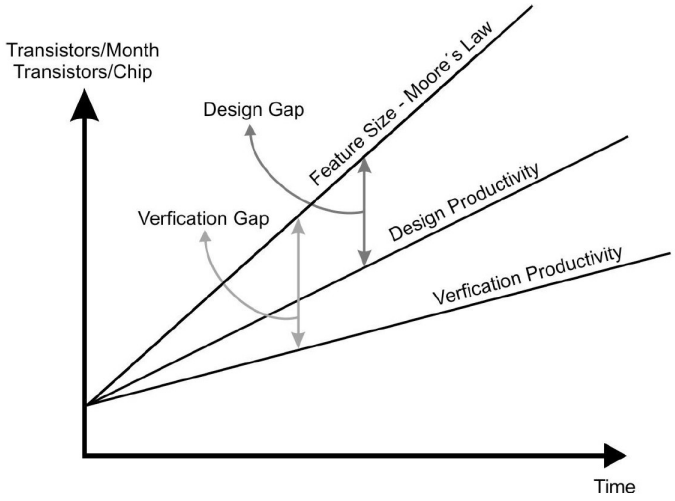

Figure 1.1: Design gap and Verification gap [1]

Functional validation forms the main stay of hardware testing and verification [1]. It's estimated that two-thirds of the overall design development time of integrated chips is spent

on functional validation [1]. Mainly functional validation is done through simulation. Figure 1.2 shows the increasing number of engineer years needed to simulate complex designs. For example, an increase in design complexity from 1 million gates to 10 million gates caused a multi-fold increase in number of simulation vectors from 100 million to 10 billion which takes almost 200 engineer years to complete.

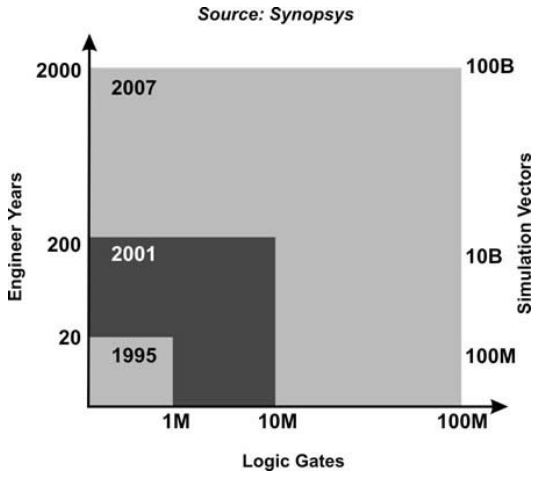

Figure 1.2: Simulation complexity [2]

Ample amount of research has been done in both academia and industry to reduce this widening verification gap. Most of it is focussed on improving simulation algorithms and efficient implementation of EDA tools. This approach was successful to a certain extent in reducing the widening verification gap. Also, faster microprocessors contributed to the performance improvement of EDA tools by executing the instructions faster. However, in the recent past, the improvements to traditional simulation algorithms have ceased and also the clock frequency of microprocessors is not increasing as earlier, which has in-turn caused the productivity gains obtained from EDA tools to cease. This has caused a verification bottleneck in the hardware design cycle. Development of novel techniques and methodologies to address this bottleneck are very important. In this thesis, we try to address this bottleneck by improving the functional simulation performance using multi-core platforms.

1.2 Why multi-core?

Moore's law states that *number of transistors on a chip would double almost every 18-24 months* [7]. In the computing world, this means that computing capacity of chips can be doubled almost every 18-24 months. Figure 1.3 shows Moore's law prediction. So far, this prediction has been fulfilled and the semiconductor industry uses it for planning their road maps. To further sustain the growth predicted by Moore's law, other physical parameters

such as voltage requirements and power dissipation of chips should also decrease at the same rate. But from Figure 1.4, we see that the semiconductor material technology has not been improving at the same rate as required to match Moore's law because of which the operating voltage is not dropping fast. Another physical parameter is power dissipation of circuits. From Figure 1.4 we also see that power dissipation of chips is increasing at a much higher rate than what is required to match Moore's law.

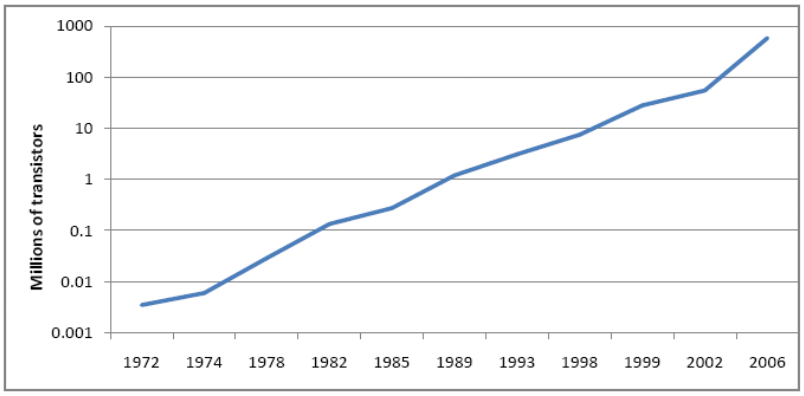

Figure 1.3: Variation in number of transistors on a chip over the years [3]

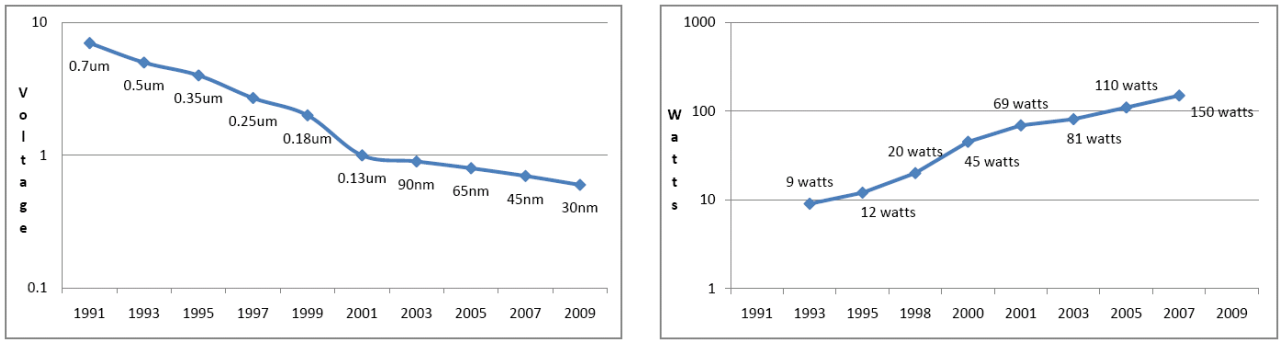

Figure 1.4: Variation in Voltage and Power over the years [3]

All these restrictions on physical parameters have threatened to cause the trend predicted by Moore's law to be off trajectory. This has impacted the growth of clock frequency of single core microprocessors. This can be observed from the Figure 1.5. The increase in clock frequency of single core microprocessors has not been sustainable after the year 2004 as compared to earlier. Supporting this claim, in May 2004 [8], Intel announced that it had hit a "thermal wall" on its microprocessor line and has not been able to increase the clock frequency of uniprocessor systems. Also in August 2004 [9], Advanced Micro Devices (AMD) announced that it changed its approach to design processors and will be demonstrating its

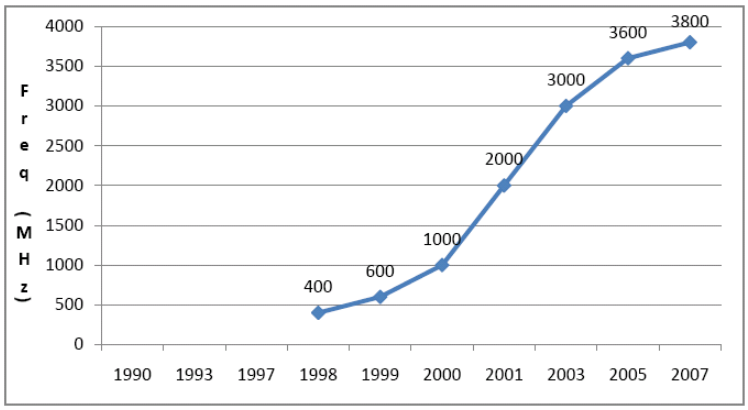

Figure 1.5: Variation in Clock speed over the years [3]

new chip with multiple processing units as increasing clock frequency is no longer useful. Later on, the semiconductor industry has focussed its force to deliver higher computing power in two major areas, multi-core CPUs and GPUs.

1.2.1 Multi-core Computing

Over the past few years, the computing world has embraced multi-cores for increased performance, power efficiency and compute capacity. This is because multi-cores effectively consume less power by running at lower clock rates but still increasing the throughput because of parallel processing. From a white paper released by Intel Corporation [10], based on the experiments conducted in their lab, we see that addition of a second core to an existing single core system, allows the clock speed to be lowered by 20%, at the same time delivering a 73% increase in performance. As expected, utilizing this fundamental relation between power and frequency, semiconductor companies have been rolling out chips with multiple cores. As per a study conducted by University of California, Berkeley [11], performance gains obtained from additional cores is not sustainable beyond 16 or 32 cores with the existing programming model and architectures. Novel architectures and programming models are essential to sustain the performance gains from highly parallel systems. Modern Graphics Processing Units (GPUs) make an attempt at improving the performance gains by means of providing massively parallel platforms.

Graphics Processing Unit (GPU) is a special purpose processor designed to offload graphics rendering work from CPU. Earlier GPUs were targeted to perform video rendering. Modern GPUs are seen as many-core processors which, not only can be used to do graphics rendering but also can be used to perform compute intensive tasks traditionally handled by the CPU. This technique of using GPUs to perform computations traditionally handled by CPUs is

known as General Purpose Computation on Graphics Processing Units (GPGPU). GPUs are increasingly seen as massively parallel platforms for executing compute intensive tasks because of the parallelism offered by them. On one side performance improvement of general-purpose microprocessors was slowing down, on the other side, GPUs relentlessly kept giving higher performance. Figure 1.6 shows the comparison between floating-point performance of Multi-core CPUs and GPUs.

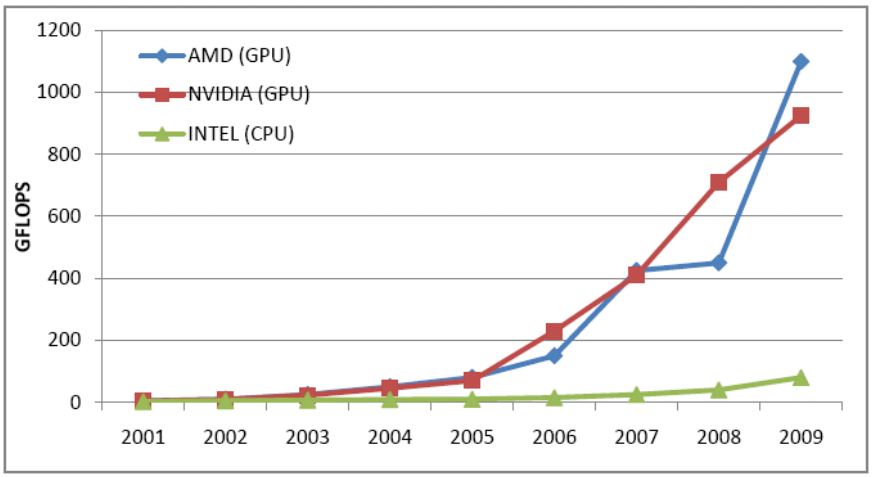

Figure 1.6: Comparison of GFLOPS of Multi-core CPU and GPU [4]

Currently we see that GPUs are achieving almost an order of magnitude higher performance. The cause of this difference in performance is due to the difference in architecture of CPUs and GPUs. A CPU architecture is optimized for sequential execution with a large number of branch and control statements, where as a GPU architecture is optimized to perform massive floating-point computations with less of branching and control. GPUs are well suited to perform computationally intensive tasks such as simulations, number crunching applications, encryption/decryption, signal processing applications and not suited for control intensive tasks. Apart from offering highly parallel platform for compute intensive applications, GPUs provide a affordable low cost alternative as compared to cluster based parallel platforms.

1.3 Multi-cores for simulation

With the computing trend shifting from single core to multi-core, software applications need to be re-written in order to take advantages of multiple cores. This is particularly very important in case of Electronic Design Automation (EDA) applications as growing complexity of designs has made performance improvement in EDA tools a necessity. A large amount of effort is put in both by industry and academia to improve the performance of EDA tools by leveraging the parallelism offered by distributed computing platforms. Recently research

is focussed on exploring low cost parallel platforms such as multi-core CPUs and GPGPUs to improve the productivity gains obtained from EDA tools. Given the intricacies of EDA tools, its not surprising, if one discovers that parallel version of tools perform slower than sequential versions. Efficient utilization of resources offered by multi-core platforms are a must to improve the performance.

In our research, we have focussed on improving performance of functional simulation, which has been one of the most compute intensive applications in EDA. We start our exploration first at netlist level by showing how the design description in EDIF netlists can be partitioned and then made amenable for parallel simulation on multi-cores. We present generic algorithms and methodologies that can be extended to other platforms than the ones we have experimented on. Further in our exploration we try to accelerate simulation of hardware models described using RTL subset of SystemC. We chose SystemC for our experiments as its the most widely used system-level description language for developing hardware models. We discuss techniques for mapping the SystemC simulation on to GPGPUs, in particular NVIDIA GPGPUs. We present the translator that can do source-to-source translation of SystemC to CUDA. Our initial investigation shows promising results and with further optimizations these techniques can yield further improvement in performance.

1.4 Main Contributions

Main contributions of this work can be summarized as following:

1. Methodology for parallel simulation of netlists
   (a) Approaches for partitioning EDIF netlists
   (b) Mapping of partitions to multi-cores using Threading Building Blocks (TBB) programming model
   (c) Generation of multi-threaded code for simulation
2. Methodology for simulating SystemC RTL models in parallel
   (a) Mapping of SystemC RTL processes to CUDA threads
   (b) Source-to-source translation of SystemC RTL models to CUDA programs
   (c) Intuitive equivalence proof of translation
3. Experimentally validating the methodologies and approaches used in parallel simulation techniques
4. Evaluation of GPUs and multi-cores for the hardware simulation purposes and comparing the performances

1.5 Organization of Thesis

This thesis is organized as follows:

In Chapter 2, we provide background material on algorithms used for SystemC and HDL simulation. We also introduce Compute Unified Device Architecture (CUDA), CUDA programming model, advantages and limitations of the CUDA. We then introduce multi-threading library, Threading Building Blocks (TBB) along with its advantages and limitations.

In Chapter 3, we provide detailed information on past and the ongoing work in the area of parallel simulation of SystemC and HDL. We also provide information regarding works that have tried to exploit parallelism offered by GPUs for simulation applications.

In Chapter 4, we describe another parallel simulation framework **PHDLSim** which includes the translator that takes EDIF netlists as input and generates TBB and Pthread designs for simulation purposes. We present the simulation times of the individual approaches and compare them.

In Chapter 5, we describe the **SCGPSim** [12] simulation framework and explain the design flow of our translator which does source-to-source translation of SystemC RTL designs to CUDA programs. We also describe the rules of the translator that preserve the simulation semantics and present a intuitive proof of the equivalence of SystemC simulation and CUDA program execution. We also show the mapping of CUDA programs to TBB designs, extending the functionality of translator to generate TBB designs. We present the simulation times of the individual approaches and compare them.

Chapter 6 concludes this thesis.

Chapter 2

Background

2.1 SystemC Simulation Kernel

SystemC is an open-source system-level design language based on C++ that has its own simulation kernel (Please refer SystemC Language Reference Manual [13] for details on describing hardware using SystemC). 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. Before proceeding into further details of scheduler, we explain certain basic terms using sample SystemC code shown in Listing 2.1.

Listing 2.1: Snippet of SystemC code

```
SC_MODULE(Module_name) {
  // Declare ports, internal data, etc.
  // Declare and/or define module functions
  SC_CTOR(Module_name) {
    // Body of the constructor
    // Process declarations and sensitivities

  SC_METHOD(function1);
  sensitive << input1 << input2;

  SC_THREAD(function2);
  sensitive << input1 << clk;
  }
  ...
  void entry();
};
```

1. **Module:** Module is the basic building block of SystemC model. Modules consist of processes which form the basic units of functionality. (Ex: Module_name)

2. **Process:** Processes are the basic units describing the functionality of the entire model. There can be two different types of processes namely SC_METHOD and SC_THREAD.

SC_METHOD: These processes execute from first to end at once and cannot be suspended in-between. Usually these are used to describe combinatorial parts of the design. (Ex: function1)
SC_THREAD: These kind of processes can be suspended using *wait* statements. These processes implicitly preserve the state of execution and hence can be used to model sequential parts of circuits. (Ex: function2)
Both these processes will be executed in response to a change in signals present in the sensitivity list. (Ex: Process function1 is sensitive to signals input1 and input2)

3. **Event:** There are 3 different types of events namely immediate, delta and timed events. Event notification causes processes that are sensitive to it to be triggered and push to the execution queue. These processes are called *runnable* processes.
   Immediate events: These events are to be executed in the same delta cycle before updating the state variables.
   Delta events: These events are caused due to the updation of the state variables. These events will be processed in the next delta cycle.
   Timed events: These are the delayed time notifications which are processed if there exists no immediate and delta events.

4. **Runnable process:** A SystemC process will be made runnable process once an event occurs on a signal present in the sensitivity list of that particular process.

5. **Delta cycle:** A delta cycle is a small step of time within the simulation time that does not advance the simulation time.

6. **Entry function:** Entry function is used to indicate the entry at the first point into a process.

Now lets understand more details of the simulation kernel. 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. Figure 2.1 shows the execution semantics 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 till 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 **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.

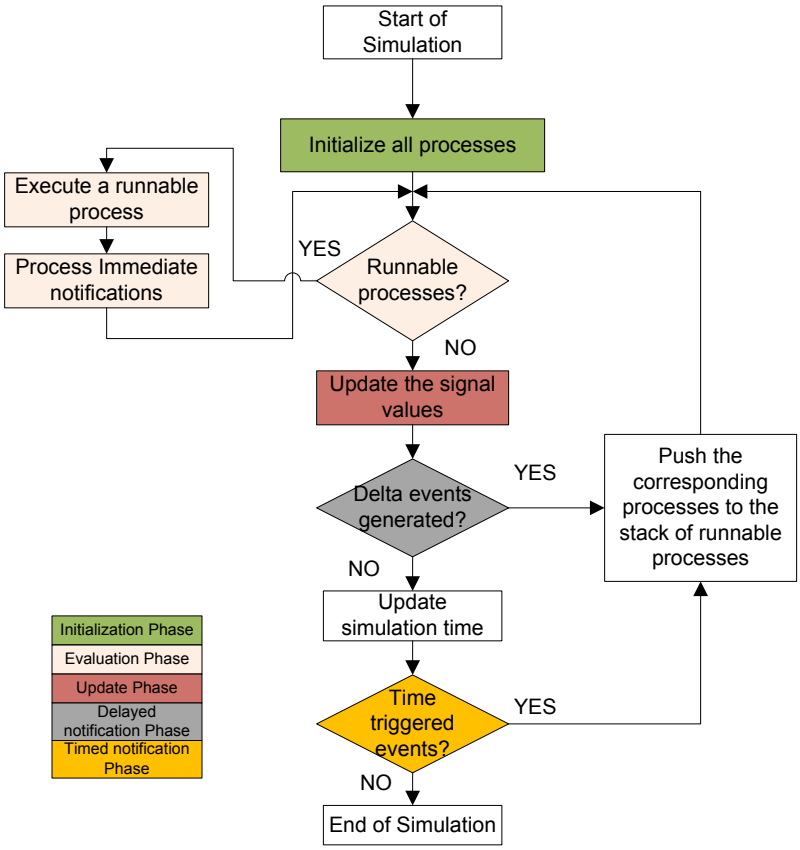

Figure 2.1: OSCI SystemC Simulation Kernel [5]

While this approach works perfectly fine for a single-core processor, it does not scale up and take advantage of multi-core/multi-processor environment.

2.2 HDL simulation

HDL simulators can be broadly divided into 2 types [14].
A. Sequential HDL Simulators: Sequential HDL simulators are the ones in which the simulation proceeds sequentially in terms of time or events. They can be further divided into 2 types. Time driven and event driven simulators. In case of time driven simulators, the simulation time advances in equivalent steps and for each step, all units are evaluated. It's also called *compile mode simulation* due to the fact that the entire simulation strategy is decided upfront during the compile time and no changes can be made during run time. Main disadvantage of these simulators are the idle time intervals. Event driven simulators avoid the idle time intervals by introducing the concept of Time-stamps. Instead of evaluating units every interval, discrete points in time, time-stamps indicated when exactly units were

to be evaluated. Of late, compiled code event driven simulators, that multiplexes both the types are the most commonly used ones.

B. Parallel HDL Simulators: There are 2 notable ways of doing parallel simulation. Synchronous and Asynchronous parallel simulation. Synchronous parallel simulation is an extension of time driven simulation in which the entire design is controlled by a global clock. For each interval of time, all the units are evaluated parallely, updated after each interval and the threads are usually synchronized using Barrier synchronization. Asynchronous parallel simulation is an extension of Event driven simulation where the design is partitioned into disjoint submodels and each of the submodels are executed parallely at discrete points in time and synchronized at certain point in time.

2.3 Compute Unified Device Architecture (CUDA)

CUDA (Compute Unified Device Architecture) [6] is a parallel programming model and software environment. It is an extension of 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 a maximum of 512 threads. These blocks are arranged in 2 dimensional matrix like structure to form a grid. Each grid consists of a maximum of 65536 blocks. Figure 2.2 illustrates the structure.

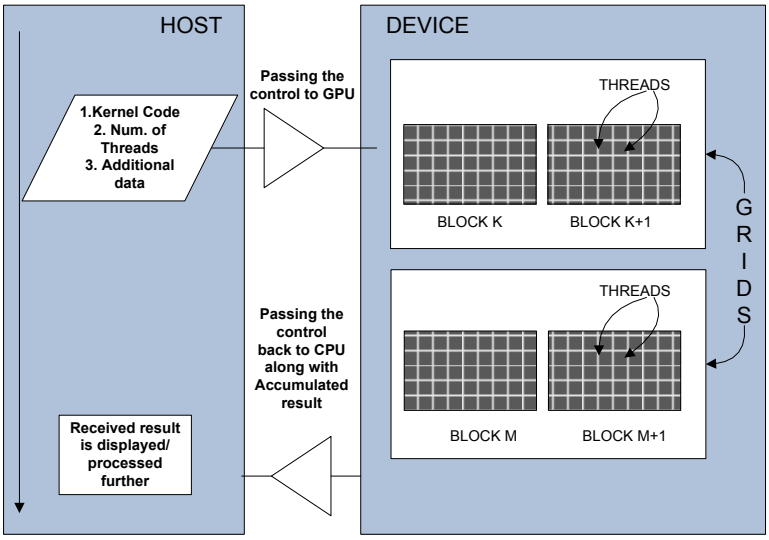

Figure 2.2: CUDA Threading Model [6]

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

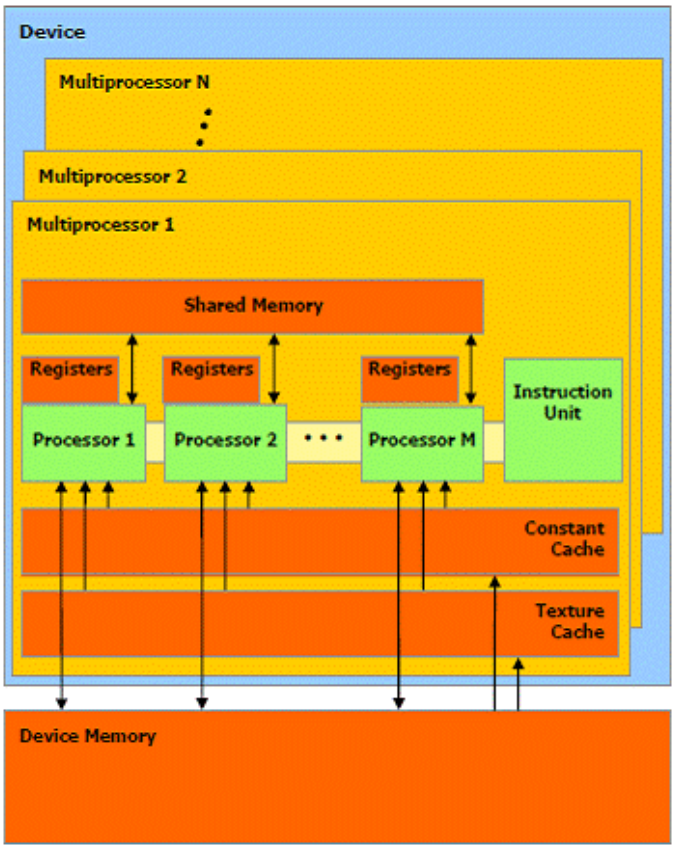

Figure 2.3: CUDA Hardware Model [6]

slower global memory. CUDA thread execution differs from that of CPU thread execution in terms of scheduling of threads. In CUDA, threads as grouped to form warps and these warps are executed in single-instruction multiple-thread (SIMT) way i.e., each processor of the multiprocessor executes the same instruction, but operates on different data. 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 make them run on different multi-processors.*

Figure 2.3 shows the hardware model of CUDA enabled GPUs. It's built of scalable array of streaming multiprocessors. A multiprocessor consists of 8 simple scalar processing cores, two special function units, a multi-threaded instruction unit and on-chip shared memory. Multi-processor is responsible for creating, managing and executing concurrent threads with minimal or zero overhead. All the threads in a multiprocessor can be synchronized using *__syncthreads()* function which implements barrier synchronization feature.

CUDA programming model necessitates that the program to be executed on GPU and the required data to be transferred from CPU to GPU via the connection bus between the two. In our case connection is via PCI-Xpress bus. The bottleneck in the CUDA programming model is the data transfer rate between the CPU and the GPU. Currently this is limited by the maximum speed of the PCIe bus. The overall performance of the applications depend significantly on the strategy of use of the memory model mentioned above. 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.

2.3.1 CUDA memory model

In CUDA, the host(CPU) and device(GPU) has different memory spaces. To execute a program on GPU, we first have to allocate memory on both CPU and GPU. Then we have to do a DMA(direct memory access) transfer to transfer input data from CPU to GPU and notify GPU to start computation. After the computation is complete, the results have to be transferred back from GPU to CPU. CUDA library provides functions to perform these memory transfers.

Figure 2.4 shows overview of memory model of CUDA enabled GPUs. There are 5 different types of device memories.

1. **Global memory**: This is non cached, read-write region of the device memory. All the multi-processors and scalar processors can access this memory. Its latency is highest among all types of memories and hence efficient utilization of this memory plays a key role in performance of the application. Since it is non cached, its important to follow the access pattern to get maximum memory bandwidth. Best performance is achieved when the global memory access is coalesced and the size can be 32 bytes, 64 bytes or 128 bytes.

2. **Constant memory**: This is a cached, read only memory which only host can write. Its latency is much lesser than global memory latency. When all threads of a half-warp are reading same memory location, its latency can be compared to that of registers. The latency cost increases linearly with the number of different addresses ready by the threads.

3. **Texture memory**: This is also a cached, read only memory which is only host write-able. This memory is optimized for 2D spatial locality and hence mostly used in graphics rendering programs.

4. **Shared Memory**: Each multiprocessor has a separate shared memory. Since its on-chip, its much faster than local, constant and global memory spaces. When all threads of a warp are accessing the shared memory, its latency is comparable to that of registers. Shared memory is divided into n multiple banks. Hence accessing memory

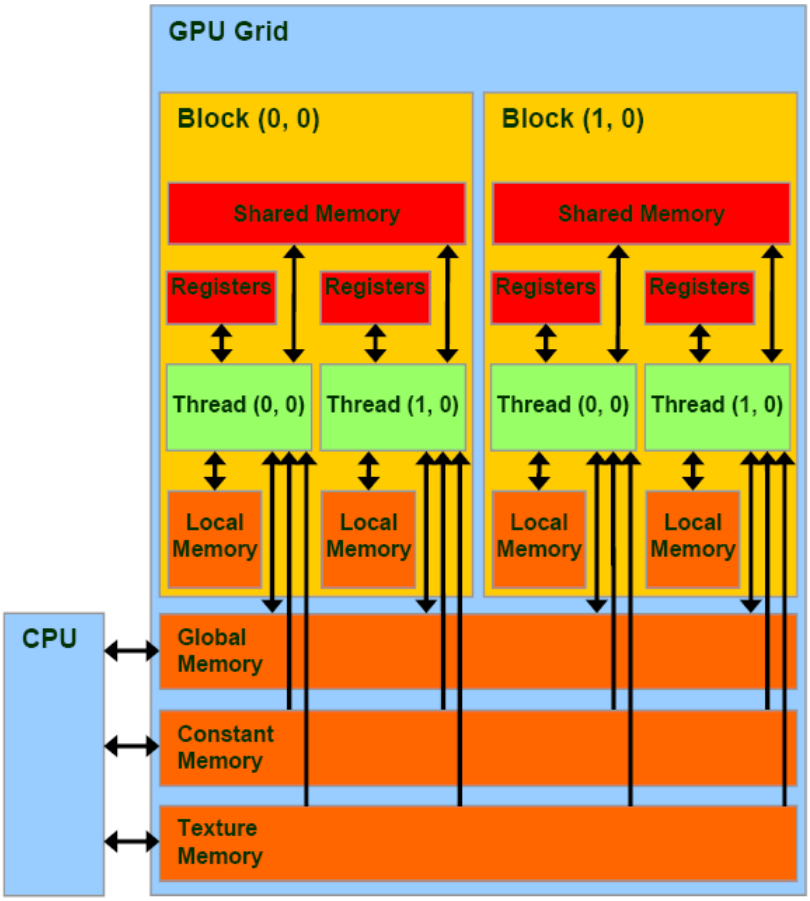

Figure 2.4: CUDA Memory Model [6]

locations in n different banks effectively yields n times the bandwidth. However if two memory locations are in same bank, then the accesses are serialized. The size of shared memory is 16KB and is divided into 16 banks.

5. **Registers**: These are specific to individual scalar processing units and are limited in number(one set of 32 registers per processing unit). They are not cached and hence accesses to local memory are as expensive as accesses to global memory.

2.3.2 Advantages of CUDA

1. Under the assumption that engineers know C, CUDA which is an extension to standard C language reduces the learning curve.

2. A 16KB of memory per multiprocessor shared between threads of same block can be used as cache as its latency is very less. This helps in speeding up applications which require frequent lookups.

3. PCI-Xpress facilitates efficient data transfers between host and device memory
4. CUDA hides the complexity of graphics APIs and simplifies the programming model to a great extent.
5. Linear memory addressing, gather and scatter, writing to arbitrary addresses.
6. Single precision floating point support, hardware support for integer and bit operations.

2.3.3 Limitations of CUDA

1. CUDA uses recursion free, pointer free subset of C.
2. Double precision floating point support deviates from IEEE specifications.
3. Branch divergence of threads in a warp will lead to a decrease in performance.
4. Learning curve for writing highly efficient CUDA programs is quite high.
5. The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.
6. CUDA is propriety closed architecture from NVIDIA.

2.4 Threading Building Blocks (TBB)

Threading Building Blocks (TBB) [15, 16] is an open-source C++ library that supports multi-threaded programming for parallel execution on CMPs.

2.4.1 TBB programming model

TBB enables auto scalable parallel programming, which means it possesses the ability to execute the same program on a single core system or on a multi-core system. This is achieved via the use of a programming model where the user specifies tasks instead of the conventional concurrency model with threads. The TBB scheduler maps these tasks onto threads at runtime and dispatches them to the available processing elements. The dispatch scheme is based on a task graph constructed from the specified program. Note that the TBB scheduler is an unfair task scheduler; however, it is able to identify loads on the processing elements and automatically load balance the task execution. TBB also reduces the wait time for threads by creating only as many number of threads as the cores in the system. In case of systems with hyper-threading capability, it creates twice the number of threads as the cores in the system.

TBB is a library based on C++ templates where the running application is partitioned into various tasks. The library offers several built-in patterns that can be used to describe the parallel program. For example, `parallel_for` and `task_group` are two patterns that we use in our research. These algorithms follow a divide and conquer approach to task execution and this research compares these two approaches with the conventional non-parallelized execution.

2.4.2 Constructs in TBB

`parallel_for`: Parallel iteration over a range

`parallel_for` is a pattern that allows parallel execution of loop iterations that are independent and safe to process in parallel. It breaks up the iteration space into smaller chunks recursively until a user-defined grain size is reached. Each of the chunks get mapped onto a separate thread. It is important to specifying an optimal grain size as it affects the efficiency of the parallel execution. The range of a `parallel_for` represents the total iteration space where as subranges are the smaller iteration spaces obtained by splitting the range. For a given range split into multiple subranges, `parallel_for` incurs an overhead cost for every subrange. If the subranges are too small, excessive overhead affects the performance. On the other hand, if the subranges are too large, we may not be optimally using the parallelism. Listing 2.2 shows the snippet of parallel program written using *parallelfor* construct.

Listing 2.2: Snippet of *Parallelfor* construct code

```
// FUNCTION POINTERS
typedef void (*pt2Function)();

// FUNCTION POINTERS
struct Body {
  void operator() (blocked_range<int>& range) const {
    for(int j=range.begin(), j_end=range.end(); j<j_end; ++j) {
      (*function[j])();
    }
  }
};

void function_0(){
...
}
//------
void function_1(){
...
}
//------
......
......
//------
void function_8(){
..
}
//------
/* MAIN FUNCTION */
```

```
int main()
{
  // Assigning function pointers
  function[0] = &function_0;
  ....
  function[8] = &function_8;

  parallel_for(blocked_range<int>(0,NUM_FUNCTIONS), Body(),auto_partitioner());
  ...
  return 0;
}
```

`task_group`: Parallel execution of groups of functions

The `task_group` pattern is a high level interface that can be used to create groups of parallel tasks and schedule them onto different threads without directly interacting with the task scheduler. During runtime of `task_group` object, threads in the thread pool start executing tasks as soon as they are created instead of waiting until all tasks are created. This reduces overhead and increases the efficiency. For each iteration, `task_group` has the overhead of adding tasks into the group, but since task spawning and execution happens in parallel overhead effect is minimized. We create groups of tasks from function pointers. The difference between `task_group` as compared to `parallel_for` is that in the case of `parallel_for` tasks to run are produced by multiple threads and hence stealing will be limited, where as in case of `task_group` all tasks are produced by the same thread and other threads in the thread pool constantly steal the tasks. Listing 2.3 shows the snippet of parallel program written using *taskgroup* construct.

Listing 2.3: Snippet of *Taskgroup* construct code

```
// FUNCTION POINTERS
typedef void (*pt2Function)();

void function_0(){
...
}
//------
void function_1(){
...
}
//------
......
......
//------
void function_8(){
..
}

//------
/* MAIN FUNCTION */
int main()
{

// Assigning function pointers
  function[0] = &function_0;
  ...
```

```
  function[8] = &function_8;

  task_group g;

  g.run(function[0]);
  ....
  g.run(function[8]);
  g.wait();

  return 0;
}
```

2.4.3 Advantages of TBB

1. Unlike CUDA, TBB is full fledged C++ and not extension to C.
2. TBB allows parallelism to be specified at a higher abstraction level and hence significantly reduces the number of lines of code required to develop multi-threaded applications.
3. Also TBB significantly reduces the programming complexity for specifying multi-threaded applications in terms of tasks rather than low level thread management.
4. TBB's task manager automatically analyzes the system programs are running on, chooses the optimal number of threads, and performs load balancing that spreads out the work evenly across all processor cores. This reduces context switching between threads.
5. TBB threaded applications automatically scale to fully utilize all available processing cores.

2.4.4 Limitations of TBB

1. Unlike CUDA which exploits massive parallelism in numerous cores of GPU, TBB only extracts limited parallelism in multi-core CPUs.
2. Task creation in TBB has a significant overhead of almost 600-1000 cycles. To nullify the effect of overhead of task creation on performance, each task should be at least 10 times the overhead.
3. Inability to preserve the tasks after execution. At present its not possible to preserve a task for later time, once its executed.

Chapter 3

Related Work

3.1 Distributed and Parallel Simulation of SystemC

Several attempts have been made in the past on parallelizing the SystemC simulator, though none of them are targeted at a massively parallel platform as a GPGPU. In [17] Savoiu et al. show a static transformation algorithm for SystemC models. The transformation modifies the threading structure of the models and produces efficient C/C++ code which can be utilized for parallel execution. 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 cooperative threads into a much higher granularity thread. In [18], Kaouane et al. have proposed design flow for a CELL based SystemC simulation. With limited number of synergistic processing elements, it requires a lot of context switches that reduces the efficiency. Also in this approach, 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 [19] 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. [20] showed that by means of process splitting, one can speed up the SystemC simulations by automatically optimizing the model for simulation. But the results showed that improvement achieved was around 15%. All these attempts have constraints in one or the other form and don't take the full advantage of highly parallel platforms. Our work is different from others in terms of targeted platform and the partitioning approach. GPUs provide large number of cores for executing many threads in parallel. This reduces the context switches necessary which will improve the performance.

3.2 Parallel HDL simulation approaches

In [21], Li et al. propose a simulation system based on the MPI library and Time Warp optimistic synchronous parallel algorithm. They compile Verilog designs to C++ designs, and link the C++ designs with MPI libraries to obtain a multi-threaded executable that executes parallely on a computing cluster. Lijun Li et al. [22] developed an object-oriented framework for distributed Verilog simulation. They parse the Verilog designs using ICARUS Verilog [23] which generates an efficient intermediate format for simulation purpose known as *vvp assembly code*. This assembly code is then parsed using VPP Parser [23] to obtain functor list and thread list representing the behavioral description model of the design. These lists are later translated, linked with MPI/PVM libraries and scheduled on cluster of systems to execute parallely. The primary limitation of these approaches is that the parallelization only works for Verilog designs. In addition, the authors use MPI, which is not best suited for CMPs because it is designed for computer architectures with a distributed-memory architectures whereas CMPs have shared-memory architectures.
In [24], Wang et al. propose a levelized, event-driven compiled code simulator in which they convert the gate-level netlist into a sequentially executing code. With the simulation being done at gate level and the code being sequential it doesn't scale up for larger designs. Also the speed up reported was not with respect to a commercial simulator. In [25] authors present a parallel gate-level simulation framework. They partition the design into disjoint submodels and each processor takes charge of the individual submodel. All these approaches are constrained. Sequential simulation approaches cannot scale up with larger designs, MPI based approaches have communication overheads and need clusters which are expensive and other approaches follow complex partitioning algorithms.

In [26], authors present an event-driven gate-level simulation approach on GPUs in which they segment the combinatorial portion of the netlist into various macro-gates and schedule groups of macro-gates on to different multiprocessors. This approach works at much lower level of granuality which yields speed up only CMP's which have lot of cores like GPU. For CMP's with lesser number of cores, there will be a lot of task overhead which reduces the overall efficiency. Our approach differs at the level of granularity we partition the designs and hence it reduces the task overhead. Our approach also avoids duplication of gates, hence increasing the efficiency of the framework.

Our work differs from the above works in terms of being a solution designed for CMPs that is scalable, works with both Verilog and VHDL hardware designs and completely automated. With minor changes in code generator, our simulation framework can be easily ported to SIMD/MIMD architectures such as Intel's Larrabee [27] and NVIDIA's GPUs.

3.3 Simulation on GPUs

Accelerators are specially designed processors used to increase the throughput of the overall system by offloading compute intensive tasks. Every application has its unique characteristics and distinct computational requirements. Hence no single accelerator architecture can improve performance for all applications. But use of the accelerators has shown improvement in performance for certain category of applications [28] among which simulation applications are on the top. Simulation is an act of enacting characteristics of a physical or abstract system. It usually requires fast processing of large volumes of data or running a compute intensive mathematical algorithm. Traditionally FPGAs were used to do Hardware-in-loop-simulation (HILS). In this work we investigate GPU as an accelerator for the simulation application. GPUs has a strategic advantage over other accelerators such as FPGAs in terms of the abstraction level of programming. A higher level of programming language reduces the development time drastically and software tools enable easier and faster debugging as compared to FPGA debugging. Another advantage of using GPUs for simulation is their high memory bandwidth. GPUs also have global and shared memory which on efficient programming can speed up applications drastically. All these advantages make GPUs strong contenders as accelerators for simulation purpose. A lot of reported work can be found in [29]

Chapter 4

Parallel netlist simulation

Simulation of complex hardware designs plays a critical role in meeting today's shrinking time-to-market requirements. It allows designers to verify their conceptual designs and to identify and debug incorrect behaviors early in the design cycle. It is clear that simulators for hardware description languages (HDL)s assist designers with fast and efficient simulation. However, the increasing complexity and size of today's hardware designs often results in simulation times in the order of days. This is because most simulation engines do not scale effectively with the complexity of the design and underlying execution platforms. For instance, most traditional HDL simulators implement the discrete-event (DE) model of computation for a uniprocessor. Consequently, most simulator implementations are sequential and not easily amenable to parallelization and execution on chip-multiprocessor (CMP) platforms. Therefore, existing HDL simulators fail to take advantage of the underlying parallelism of modern computer architectures.

To address this issue, current research has focused on transforming hardware design specifications into parallelly executable programs that can execute on multi-processor systems such as MPI programs on computing clusters [22, 21]. For CMPs, we find that MPI is not the best parallel target framework. This is because MPI is designed for distributed-memory architectures, whereas most CMPs employ shared-memory architectures. A distributed-memory architecture assumes that each processing core posses its own memory, and communication to another processing core requires transferring the data to a temporary buffer prior to it being sent. On the contrary, a pointer can simply be used to do the same in a shared-memory architecture. This has performance implications on the communication overhead incurred by MPI. Hence, we select Intel's TBB and POSIX threads(Pthread) as our framework because these libraries are designed for shared-memory architectures. In addition, existing approaches perform source-to-source transformations, which means they only work for a specific HDL.

Our approach exploits the parallelism in the underlying computer architectures to improve simulation performance. We achieve this by automatically converting netlists into parallelly

executable multi-threaded Intel Threading Building Blocks (TBB) code and Pthread code. We take different approach as compared to other works described in Chapter 3, by accepting an EDIF netlist as an input instead of an HDL program. This helps us to stay language independent as HDL designs written in both Verilog and VHDL can be synthesized to EDIF. We implement a translator that we call PHDLSim that implements the conversion process using semantic-preserving transformations. This process has three phases: 1) parsing a netlist and constructing an graph data structure, 2) partitioning the graph into subgraphs such that they can be executed in parallel, 3) generating TBB/Pthread code from the partitioned design.

The purposeful use of TBB/Pthreads for CMPs allows full utilization of the underlying architecture without intervention from the designer. This is because TBB/Pthreads executions automatically scale depending on the number of cores available in a hardware architecture.

In this chapter we describe PHDLSim, a multi-core netlist simulation framework to improve the simulation performance of synthesizable hardware designs by leveraging the parallelism available in chip-multiprocessor platforms. We show the design flow of the framework including the algorithms to partition the designs to smaller sub designs that can be simulated on multiple cores. A brief description of the intermediate format is also included. Finally we present the experimental results.

4.1 Design Flow of PHDLSim

Figure 4.1 shows the overall design flow of PHDLSim. We first start by taking a synthesizable hardware design and convert it to EDIF netlist format by using a synthesis tool such as Icarus Verilog or Synplify Pro. We then partition the netlist into clusters such that each cluster can be executed independently. After partitioning, we create tasks for executing clusters and run these tasks in parallel in a multi-core environment. Each phase is explained in detail in the following sections.

4.1.1 Design Phase

In this phase, we construct a hardware model that implements the design specifications. The model may be hand-written or automatically generated using high-level synthesis tools. We only require that the model is synthesizable.

4.1.2 EDIF Intermediate format

The resulting model from the Design phase is synthesized using any of the open-source or commercially available synthesis tools, which can produce *EDIF 2 0 0* (Electronic Design

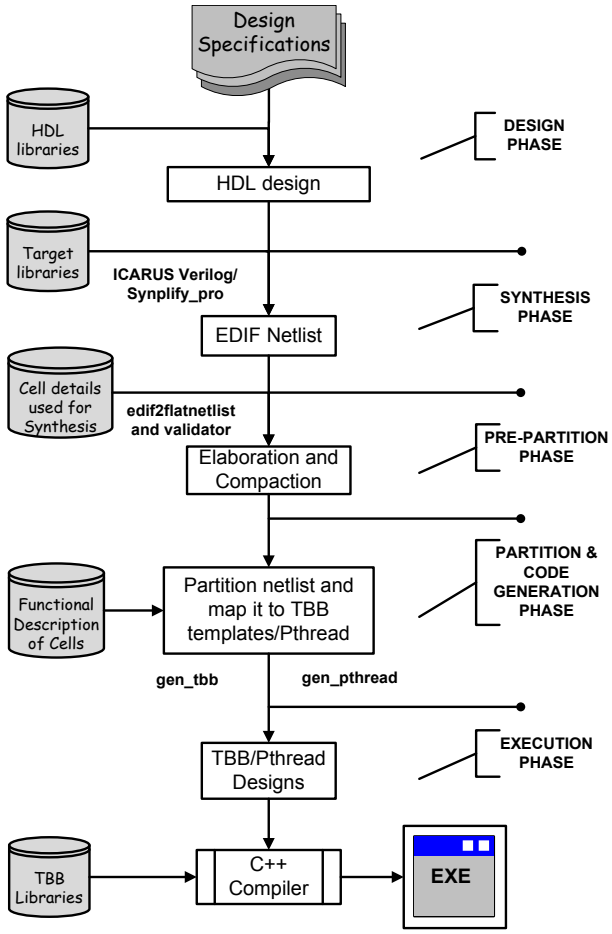

Figure 4.1: Design Flow of PHDLSim

Interchange Format Version 2) [30] netlists. Some examples are ICARUS Verilog (open-source tool that can synthesize only Verilog) and Synplify Pro (from Synopsys that can synthesize both Verilog and VHDL). We select EDIF as the intermediate format because it is a published standard for netlist descriptions. A sample EDIF file is shown in Listing B.1 of Appendix B.

4.1.3 Pre-Partitioning Phase

In the pre-partitioning phase, we parse the EDIF netlist, elaborate hierarchial dependencies and generate a flattened netlist as shown in Listing B.2 of Appendix B. Then, we convert this flattened netlist into a directed graph data structure $G = (V, E)$, where V is the set of nodes and E is the set of edges. If there exists strongly connected components in the graph G, to perform partitioning and code generation phase of the design flow, the feedback

paths in G have to be removed first. This is necessary for the code generation phase, which topologically sorts each of the partitions before generating the TBB/Pthread code. To better illustrate our transformations, we use a running example netlist as shown in Figure 4.2.

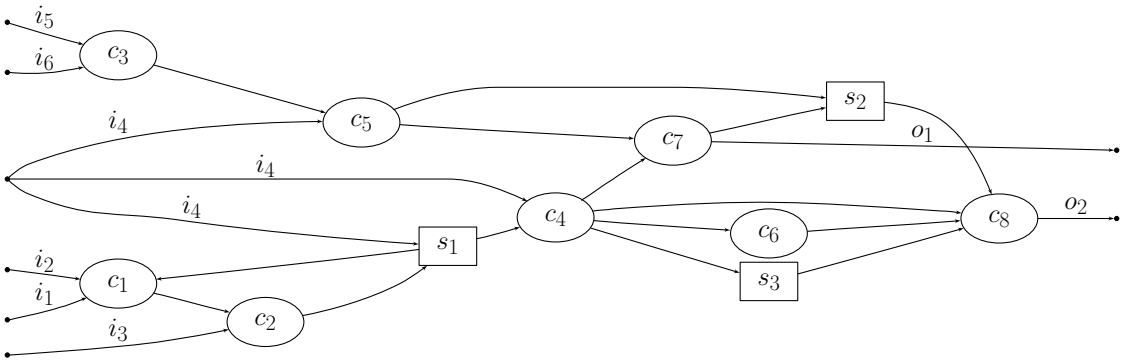

Figure 4.2: An illustrative example of G.

The set of nodes $V = C \cup S$ represents the cells in the EDIF netlist where C is the set of cells implementing combinatorial logic and S is the set of cells containing state elements. I and O represent input and output elements. We label each of the nodes to be either of combinatorial logic, state element type, inputs or outputs. The set of edges E represents the set of wires connecting the cells. Figure 4.2 shows an illustrative graph with C as ellipses, S as boxes, I and O as points.

The feedback separation procedure takes G and breaks any feedback loops. Algorithm 1 describes this procedure. We construct A_{scc} that identifies strongly connected components (SCC) in G. Every element in A_{scc} describes a different feedback path in the netlist that must be separated. We separate the feedback path by selecting an edge (v, v') in the feedback path that is an output of a state element ($v \in S$) and removing it. Then, we augment the netlist with a new node i_{new} and add an edge (i_{new}, v'). We keep track of these new nodes in V_{extra} because these are treated as special cases during the partition phase. These nodes act as primary inputs during partitioning and execution phases. The values of these nodes are updated with the values of the nodes whose outgoing edge was removed.

Notice that Figure 4.2 has one feedback path formed with the nodes $\{c_1, c_2, s_1\}$. Figure 4.3 shows the graph after the removal of feedback paths. The edge (s_1, c_1) is removed and replaced with (i_{new}, c_1). We add a new node i_{new} that feeds into c_1 and it holds the same value as s_1. The new node simply holds a copy of s_1's value, which we ensure is done correctly during code generation with double buffering technique.

The feedback paths are removed because of the need to perform a topological sort during the code generation phase. This is necessary to correctly order the execution of each partition in order to preserve the original semantics.

Algorithm 1: *feedback_separation*();

```
   /* Initial declarations                                          */
1  Let G = (V, E) be the original netlist.
2  Let V_extra be the set of extra nodes added to break feedback loops.
3  Let A_SCC = SCC(G) be the set of sets of edges that form strongly connected
   subgraphs in G.
4  for a ∈ A_scc do
5  |  Let (v, v') ∈ a where {v} ∩ S ≠ ∅.
6  |  V ← V ∪ {i_new}
7  |  V_extra ← V_extra ∪ {i_new}
8  |  E ← E ∪ {(i_new, v')} − {(v, v')}
9  end
```

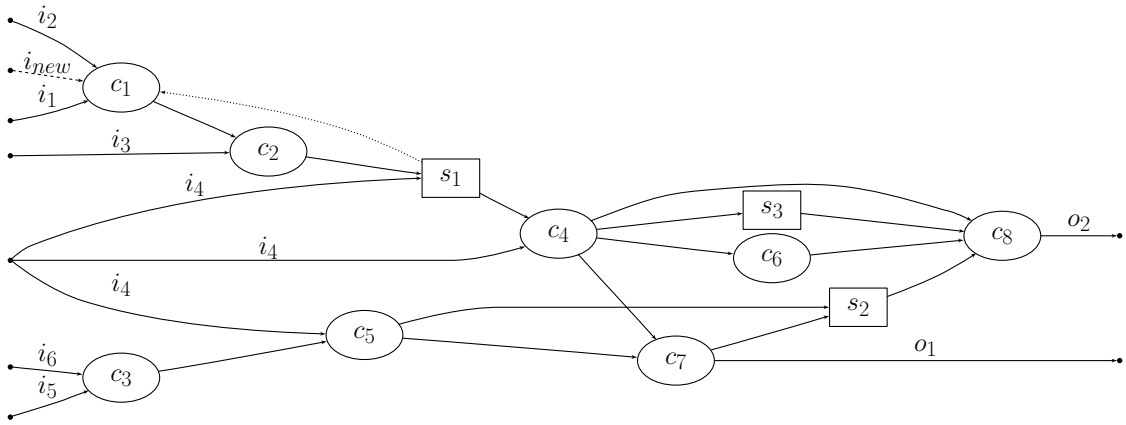

Figure 4.3: G after removing feedback edges.

4.1.4 Partition Phase

In this phase, we partition graph G into subgraphs that are amenable to parallel execution. The partitions must be made such that the original semantics of hardware simulation are preserved. We continue to use our running example to illustrate how the partitions are formed before presenting the algorithm that does this partitioning.

Notice in Figure 4.2 that we have identified combinatorial logic cells as C and state element cells as S. Our approach to parellelizing involves simulating all the combinatorial cells in parallel followed by updating the state elements with the results of the combinatorial computation. In order to allow such parallelization, we create a task for every state element and output, which contains all combinatorial logic that contributes to its inputs. These tasks will be later mapped to threads by the libraries. For the running example in Figure 4.2, we obtain five partitions consisting of the following nodes: $\{i_1, i_{new}, i_2, i_3, i_4, c_1, c_2, s_1\}$, $\{i_4, c_4, s_1, s_3\}$,

Algorithm 2: *partition*();

```
   /* Initial declarations                                          */
 1 Note that I is the set of all primary inputs.
 2 Let V_part ← ∅ be the set of all partitions.
 3 Let V_so ← S ∪ O be the set of state elements (S) & outputs.
   /* Reset partition number                                        */
 4 x ← 1;
 5 for m ∈ V_so do
 6 |   Let V_part_x ← ∅ be the set of nodes in partition x.
 7 |   Let V_trav ← ∅ be the set of nodes traversed.
   |   /* Compute fan-in cone of m                                  */
 8 |   compute_fanin(V_part_x, m, V_trav);
   |   /* Next partition.                                          */
 9 |   x ← x + 1;
10 end
11 return
```

$\{i_4, i_5, i_6, s_1, c_3, c_4, c_5, c_7, s_1, s_2\}$, $\{i_4, i_5, i_6, c_3, c_4, c_5, c_7, o_1\}$, $\{i_4, c_4, c_6, c_8, s_1, s_2, s_3, o_2\}$.

Algorithms 2 and 3 show the partitioning approach. We start with V_{so}, a set of state element and output nodes. For each of these nodes we obtain the fan-in cone using backtracking. Starting from a state element node or output node we backtrack until we reach primary input node or another state element node. V_{part_x} represents set of all nodes in the fan-in cone of the x^{th} element in V_{so}. V_{part}, the union of all V_{part_x}s contains the fan-in cone for each of the elements in V_{so}. Fan-in cone computation is done by depth first search method as indicated in Algorithm 3.

4.1.5 Code Generation using gen_tbb/gen_pthread

In the code generation phase, we generate TBB/Pthread code for the partitioned netlist. Each of the partition in T_{part} is traversed and sorted topologically. This is needed to correctly order the execution of nodes in a partition. The $tSort()$ function is used to perform this as indicated in Line 7 of Algorithm 4. Once the nodes in a partition are sorted, we check if there are any special nodes that were added to break the feedback loops. If there are such nodes, we must ensure that these nodes have the same values as the node from which the feedback path was removed. This is accomplished by copying the values to the new nodes as shown in Lines 8-10 of Algorithm 4. This preserves the simulation semantics. Then, we output the equivalent function code for each of the nodes in the partition. The function code of each node will check if its already evaluated in the current cycle by some other task. If yes, the evaluation will be skipped, else the node will be evaluated. In the next section we

Algorithm 3: $compute\_fanin()$;

```
   /* Initial declarations                                          */
 1 Note that I is set of primary inputs and S is set of state elements.
   /* Check for primary input                                       */
 2 if m ∈ I and m ∉ V_trav then
 3 |   V_part_x ← V_part_x ∪ {m};
 4 |   V_trav ← V_trav ∪ {m};
 5 else
   |   /* Check for state element                                  */
 6 |   if m ∈ S and m ∉ V_trav then
 7 |   |   V_part_x ← V_part_x ∪ {m};
 8 |   |   V_trav ← V_trav ∪ {m};
 9 |   else
   |   |   /* Recurse                                             */
10 |   |   V_next = {v1 | ∀(v1, v2) ∈ E where v2 = m}
11 |   |   for k ∈ V_next do
12 |   |   |   compute_fanin(V_part_x, k, V_trav);
13 |   |   end
14 |   |   if m ∉ V_trav then
15 |   |   |   V_part_x ← V_part_x ∪ {m};
16 |   |   |   V_trav ← V_trav ∪ {m};
17 |   |   end
18 |   end
19 end
20 return
```

explain more on execution model.

4.1.6 Execution Phase

In this phase the generated designs are all linked with libraries, compiled and executed. Figure 4.4 shows the pictorial representation of the execution model. The program initially starts in sequential mode and enters parallel mode as the tasks are being created. To start with during runtime, the scheduler creates a thread pool which consists of worker threads. The total number of worker threads depends on the number of cores present in the system. TBB runtime implements "task stealing" to balance a parallel workload across available processing cores in order to increase core utilization and therefore scaling. Initially, the workload is evenly divided among the available processor cores. If one core completes its work while other cores still have a significant amount of work in their queue, TBB reassigns some of the work from one of the busy cores to the idle core. This dynamic capability

Algorithm 4: *code_gen*();

```
   /* Initial declarations                                          */
 1 Note that V_part is the set of all partitions.
 2 Note that V_extra is the set of extra nodes added.
 3 Let V_part_i be i^th partition.
 4 Let c be a node.
 5 Let fn_i be the i^th function pointer.
   /* Iterate through partitions                                    */
 6 for V_part_i ∈ V_part do
 7 |   V_part_i ← tSort(V_part_i);
   |   /* Copy additional states values                            */
 8 |   for c ∈ V_part_i ⋂ V_extra do
 9 |   |   fn_i ← fn_i • copy_value(c.value);
10 |   end
   |   /* Add all nodes in a partition                            */
11 |   for c ∈ V_part_i do
12 |   |   fn_i ← fn_i • function_code{c.celltype};
13 |   end
14 end
15 return
```

decouples the programmer from the machine, allowing applications written using the library to scale to utilize the available processing cores with no changes to the source code or the executable program file. In Figure 4.4, TBB creates N threads. All these threads form a thread pool. At any given point of time N tasks are being executed in parallel. If there are more number of tasks than threads, after completion of a task, the worker thread picks up another task and starts executing. Once a task is complete it waits for other tasks to complete at the barrier and the thread that was in-charge of that task, now picks up another task to execute. When all the tasks are executed completely, the program returns to sequential mode wherein we update the states and save the outputs. In case of Pthreads, each of these N tasks are assigned to individual threads which may or may not get preempted during the execution. Barrier synchronization is implemented using conditional_wait broadcast and mutex constructs of Pthreads.

Execution is continued until the end condition is met. Outputs can be saved in 2 different formats, a binary file with all/necessary outputs or convert the output to VCD format using libraries using a script. At present we are verifying the output by comparing it with the output of commercially available simulators using a test case dependent script. We are working on automating this in future so that the output can be verified automatically if necessary. Listing 4.1 & 4.2 shows the sample TBB and Pthread code respectively for the simulation engine depicted in Figure 4.4.

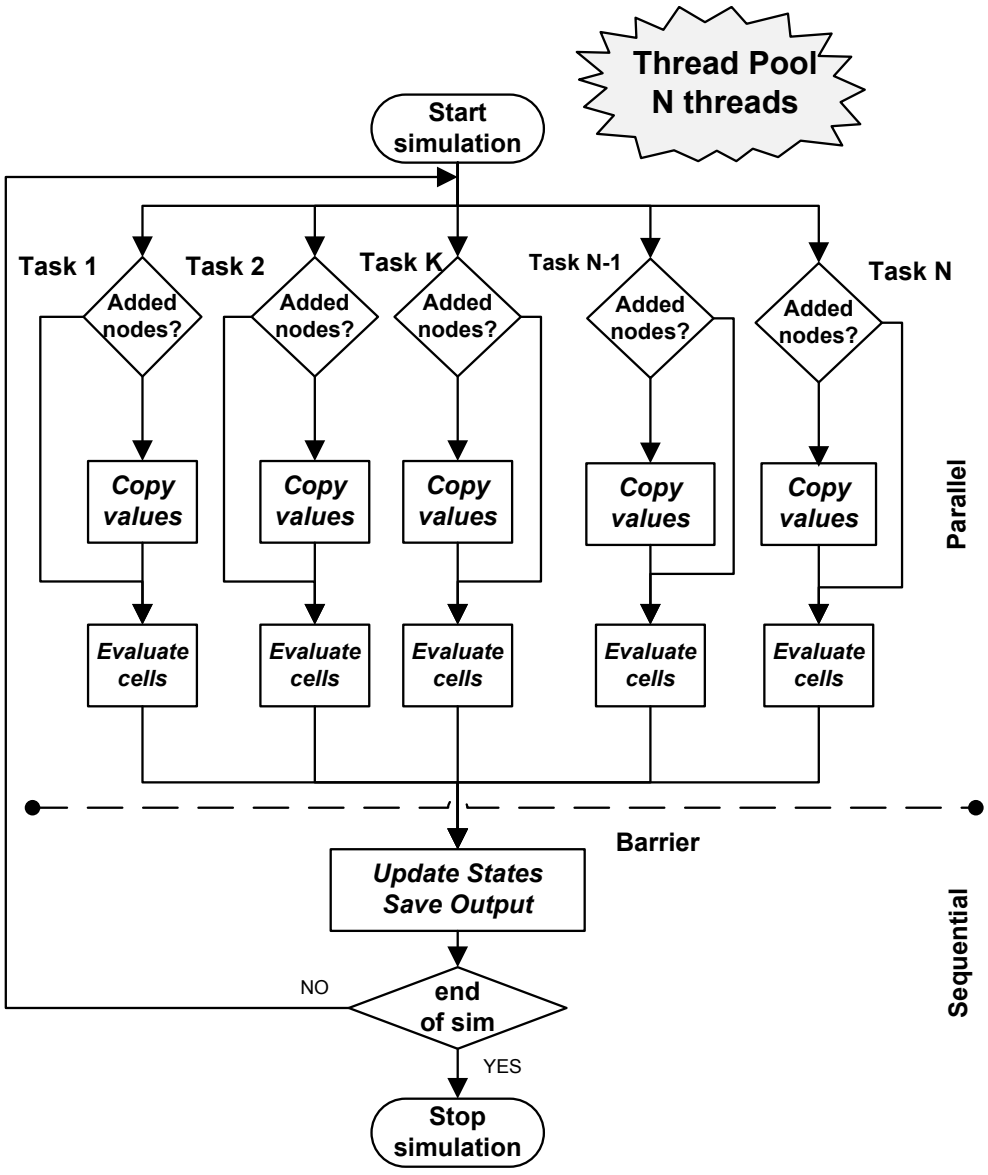

Figure 4.4: Execution model.

Listing 4.1: Sample TBB Simulation Engine Code

```
void task1(){

  //Check for extra added gates
  copy_extra_gate_values();

  //Partition 1 cells
  if(cell_1 not evaluated in current cycle)
    cell_1(args);
  ....
  if(cell_x not evaluated in current cycle)
    cell_x(args);

  return;
}
......
void taskN(){
  .....
}

/* MAIN FUNCTION */
int main()
{
  // Assigning function pointers
  function[1] = &task1;
  function[2] = &task2;
  ........
  function[N] = &taskN;

  task_group g;

  // Main simulation engine
  do{
    g.run(function[1]);
    .......
    g.run(function[N]);
    g.wait(); //Barrier Synchronization

    //Updating States
    update_states();

    //Saving Outputs
    save_output();

    n++;
  }while(n < MAX_CYCLES);

  return;
}
```

Listing 4.2: Sample Pthread Simulation Engine Code

```
pthread_mutex_t mutex1 = MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = MUTEX_INITIALIZER;
pthread_cond_t cond1 = COND_INITIALIZER;
pthread_cond_t cond2 = COND_INITIALIZER;

void task1(){

  //Check for extra added gates
  copy_extra_gate_values();
  //Partition 1 cells
  if(cell_1 not evaluated in current cycle)
    cell_1(args);
  ....
  if(cell_x not evaluated in current cycle)
    cell_x(args);

  pthread_mutex_lock(&mutex1);
  if(all_threads_finish_work)
    pthread_cond_signal(&cond1);
  return;
}
......

void taskN(){
  .....
}

/* MAIN FUNCTION */
int main()
{
  pthread_t handle[num_of_threads];

  // Assigning function pointers
  pthread_create(&handle[0],NULL,&task1,NULL);
  pthread_create(&handle[1],NULL,&task2,NULL);
  ........
  pthread_create(&handle[N],NULL,&taskN,NULL);

  // Main simulation engine
  while(true)
  {
    //Synchronization
    pthread_cond_wait(&cond1, &mutex1);

    //Updating states
    update_status();

    //Saving Outputs
    save_output();

    if(n<MAX_CYCLES)
    {
      n++;
      pthread_cond_broadcast( &cond2 );
    }
    else
      end_of_program;
  }

  return;
}
```

4.1.7 Experimental Results

The implemented framework is platform independent and was tested on Intel Xeon E5405 Quadcore processor with clock running at 2Ghz with 4GB RAM, 512KB L1 cache and 1MB L2 cache and the system was running Windows XP operating system. Preliminary experiments were conducted on small and medium sized examples and the results are tabulated in Table 4.1 & 4.2. Table 4.1 compares the results using *task_group* approach with Pthreads approach, whereas Table 4.2 compares *parallel_for* approach with Pthreads approach. Error free simulation was our prime motive. Each experiment was ran for 100,000 cycles and the outputs were compared using experiment dependent scripts.

Table 4.1: Experimental Results: HDL simulation using *Task_group*

Design	VCS(ms)	task_group(ms)	Pthreads(ms)
16 bit Adder	1120	1280	5845
8 bit Adder/ Subtracter	1090	1610	5782
Asynchronous Counter	1180	1670	3742
Linear to A-Law converter	1240	3620	3926
Ternary adder	1210	1320	3980

Table 4.2: Experimental Results: HDL simulation using *Parallel_for*

Design	VCS(ms)	parallel_for(ms)	Pthreads(ms)
16 bit Adder	1120	1740	5845
8 bit Adder/ Subtracter	1090	2050	5782
Asynchronous Counter	1180	2040	3742
Linear to A-Law converter	1240	4370	3926
Ternary adder	1210	1320	8300

We observed that *parallel_for* implementation is slower than *task_group* implementation. This is attributed to the fact that in *task_group*, task execution and spawning happens simultaneously whereas it's not the same in case of *parallel_for*. We also observed that TBB is faster than Pthread as the overhead of synchronization and context switching is more for Pthreads. Also we noticed that TBB task size should be large enough (approx.10,000 cycles) to subside the task creation overhead. Task switching in TBB is much faster than context switching in Pthreads. Given that, not all of our transformations and procedures are completely optimized, we expect a significant improvement in performance of the framework. Also given the fact that Larrabee platform supports multi-threaded C++ programming [27](Sec 6.1), the simulation framework can be ported on to Intel Larrabee platform.

Chapter 5

Parallel simulation of SystemC RTL models

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 increasingly popular. The OSCI implementation of SystemC is built using application-level threading libraries (co-operative threads) that results in 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.

In this chapter we describe SCGPSim, a framework to simulate hardware designs specified using synthesizable subset of SystemC on GPUs. The motivation to use GPUs comes from the fact that GPUs offer low cost massively parallel platforms for multi-core computing. We describe the source-to-source translation of synthesizable SystemC designs to CUDA language. To convert the SystemC designs into CUDA designs, we use a programming model where co-operative threads can be mapped to parallel threads and can be efficiently simulated providing an order of magnitude decrease in simulation times under the constraint of preserving the existing SystemC simulation semantics. We show the design flow of the framework including the functionality of the translator. A brief description of the intermediate format is also included. A intuitive proof of equivalence is included which shows the equivalence between OSCI SystemC simulation kernel execution with our execution technique. We also show the mapping of CUDA programs to TBB programs. Finally we present the experimental results and compare CUDA results with TBB results.

5.1 Design Flow of SCGPSim

Figure 5.1 shows the overall design flow of SCGPSim. We divide the entire process of SystemC simulation on CUDA into 5 different phases. Each phase is explained in detail in the following sections.

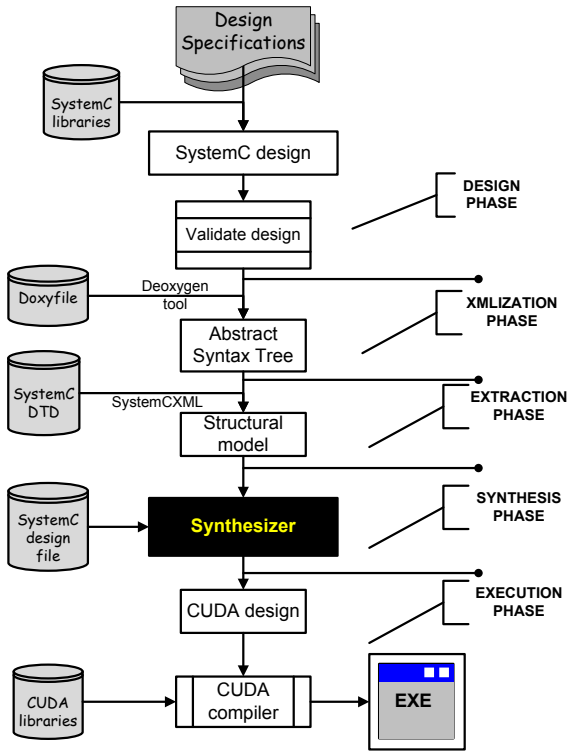

Figure 5.1: Design Flow of SCGPSim

5.1.1 Design Phase

This phase starts with the writing of a top module and identifying modules that are to be implemented as `SC_METHOD`s and `SC_THREAD`s. `SC_THREAD`s can have *while(true)* loops and multiple *wait()* statements. Currently sensitivity list is restricted to clock signal only. During writing of the program, instead of writing *printf()* statements, just accumulate the outputs in an array and display at the end. The reason for this restriction is that frequent memory transfer between CPU and GPU is costly. Hence, we accumulate the results in GPU and do block memory transfer at once. Other restrictions are the use of FILE handling and string compare functions. Currently these are not supported.

5.1.2 XMLization and Extraction Phase

Doxygen is a tool that does software reference documentation for software written in C/C++ and many languages. In the XMLization phase, the SystemC project is flattened and made into a single file and this file is processed by Doxygen tool. It captures declared classes, private members, public members and functions and represents them as Abstract Syntax Tree (AST) in XML format including the source. The reason for choosing XML as intermediate format is to take advantage of easier interpretation and manipulation.

Extraction using SystemCXML: A SystemC parser

In the Extraction phase we parse the output of Doxygen and extract port and signal names, their types and sizes, modules, sub-modules, and entry functions. This is done using SystemCXML [31] implemented by Berner et al. The input to this phase is the XML file and the Document Type Definition (DTD) file. We can extract more details from the flattened design by including more details in the DTD file. The output from SystemCXML represents structural information in XML format. We use this structural information in our synthesis phase. Listings A.1 and A.2 in Appendix A show a sample flattened source file and intermediate XML format.

5.1.3 Synthesis Phase

This is the most important phase where we synthesize CUDA code using the structural information and flattened design file. We parse the output of SystemCXML to obtain the set of all SystemC module instances M. A module instance $m \in M$ is a tuple (F_{met}, F_t, S). This tuple defines the set of entry functions $F = F_{met} \cup F_t$ as a union of the entry functions synchronously triggered in `SC_METHOD` processes F_{met}, 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. At this point, we require the user to specify combinational logic as a part of the top-level module's entry function. We also define S as the set of all `sc_signal` and `variable` declarations. An entry function $f \in F$ is captured as a sequence of computation blocks b_i^j, and suspension points w_k for some $i, k \in N$ and $j \in \{0, 1\}$, where $j = 1$ denotes the `true` case, and $j = 0$ denotes the `false` case.

Listing 5.1: SystemC Example using `wait()`s

```
//proc1 is sensitive to variable1
void proc1() {
    //OTHER STATEMENTS
    while(true) {
        variable2++;
        wait();
        variable3++;
        wait();
        result = variable1+variable2+variable3;
    }
}
```

Listing 5.2: 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;
        }
    }
}
```

In a program with no conditional branches, f is simply a sequence of b_i^1 followed by an infinite loop computation block, then some more $b_{i'}^1$ and w_k and so on. We show an example that illustrates this encoding in Listing 5.1 and Listing 5.2.This example does not have any conditional branches, so its encoding is $< b_i^1, b_{i+1}^1, w_k^1, b_{i+1}^1, w_{k+1}, b_{i+2}^1, ... >$.

In our implementation each `SC_MODULE`/`SC_THREAD` is mapped to an independent `CUDA` thread. All the processes and functions that come under that particular module will be executed by the same thread. Race condition for signals/ports is taken care by using the double buffering mechanism. Algorithms 5, 6 and 7 explain the overall conversion procedure from SystemC to CUDA. In Algorithm 5, *Line* 1, we collect all `sc_signal`s in *(S_all)* and declare them in `CUDA KERNEL CODE` (c). Additionally we check, if the signals are shared or present in the sensitivity list. If they are, then we enable double buffering on them by creating a copy of the signal. This is shown in Algorithm 7. We then parse all the module instances (M), and check if they are `SC_THREAD`s (F_t) or `SC_METHOD`s (F_{met}). If they are methods, then conversion is straight forward. We check if the statements are CUDA compatible (*Algorithm* 6) 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 here.

On the other hand, if the module instances are threads, we have to apply semantic preserving transformations and convert them to CUDA compatible functions. In Algorithm 5, we start by extracting the first computation block ($head(f)$). If the block (b_n) does not contain the `while(true)` loop statement then, we check for compatibility and output that statement. For simplicity, we drop the j index when referring to a computation block in a sequence in the algorithms. 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 5.1 and Listing 5.2. 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

Algorithm 5: SCGPSim synthesizer: Translating SystemC Design to CUDA Design

```
   /* Collect all signals.                                          */
1  S_all ← ⋃_{m∈M} m.S
   /* Convert all signals.                                          */
2  writeVariableBuffer(c.S^c, S_all)
   /* Generate CUDA kernel.                                         */
3  for m ∈ M do
       /* Enter loop only if m ∈ F_t                                */
4      for f ∈ m.F_t do
5          b_n ← head(f), n ← 0, loc ← 0
           /* Generate thread code.                                 */
6          while isWhileTrue(b_n) ≠ true do
7              c.f^c ← c.f^c • checkCompatible(b_n)
8              n ← n + 1
9          end
10         c.f^c ← c.f^c • writeSwitchBeginBlock()
11         c.f^c ← c.f^c • writeFirstCaseBeginBlock(loc)
12         while n ≤ |f| − 1 do
13             if isWait(b_n) then
14                 c.f^c ← c.f^c • writeCaseEndBlock(loc)
15                 loc ← loc + 1
16                 c.f^c ← c.f^c • writeNewCaseBeginBlock(loc)
17             else
18                 c.f^c ← c.f^c • checkCompatible(b_n)
19             end
20             n ← n + 1
21         end
22         c.f^c ← c.f^c • writeSwitchEndBlock()
23     end
24     writeMethods(m.F_met, c)
25 end
   /* Output the CUDA program                                       */
26 output(c)
```

Algorithm 6: $writeMethods()$

```
    /* Generate code for methods.                                    */
 1  for f ∈ F_met do
 2  |  if isStatementCudaCompatible(f) then
 3  |  |  c.f^c ← c.f^c • f
 4  |  else
 5  |  |  c.f^c ← c.f^c • makeCompatible(f)
 6  |  end
 7  end
```

Algorithm 7: $writeVariableBuffer()$

```
    /* Declaring variables.                                          */
 1  S^c' ← ∅
 2  for s ∈ S_all do
 3  |  S^c' ← S^c' • s
 4  |  if isInSensitivityList(s) or isShared(s) then
 5  |  |  S^c' ← S^c' • s_copy
 6  |  end
 7  end
```

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 $m.F_t$. After all the module instances are parsed, we finally output the `CUDA KERNEL`. A sample CUDA kernel synthesized using the above approach is shown in Listing A.4 of Appendix A.

5.1.4 Execution Phase

In this phase, we compile the synthesized `CUDA KERNEL CODE` (c) by adding other features such as 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. Figure 5.2 shows the simulation technique followed in SCGPSim. The execution of threads happen in vertically. Threads execute is two phases. For the phases colored in Figure 5.2, they do meaningful computation otherwise they wait at the barrier for synchronization. Barrier-1 ensures that inputs are generated before the computation starts where as Barrier-2 ensures that correct outputs are saved for the current cycle of computation.

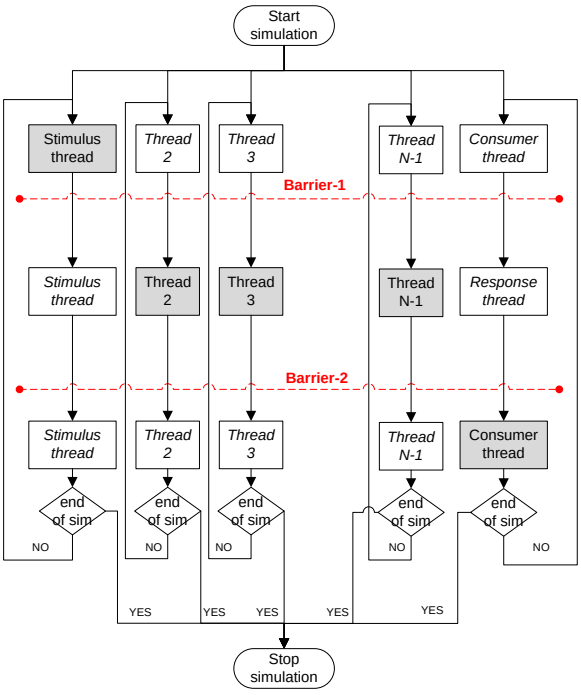

Figure 5.2: Simulation Technique followed in SCGPSim

5.1.5 Experimental Results

In this section we present some of the results of SystemC simulation on GPUs. SystemC CPU simulation was done on Intel Xeon E5405 Quadcore running at 2Ghz with 4GB RAM, 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 projects 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 time of CPU and CPU+GPU are shown in Table 5.1. Execution time includes even memory transfer time in case of CPU+GPU. The inputs to the experiments were randomized.

5.1.6 Intuitive Proof of Equivalence

Let us consider a SystemC model with threads T_1, T_2, ..., T_n. The authors in [32] 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. The basic idea is quite simple. Normally k `wait()` statements in an

Table 5.1: Experimental Results: SystemC simulation using *CUDA*

Design	CPU (sec)	CPU + GPU(sec)	SPEED UP
PIPELINE AES	120.1	3.916	30.66
FIR	51.9	1.37	37.88
10 STAGE BUFFER	28	0.277	101.083
3 STAGE BUFFER	11	0.276	39.85
SIMPLE ALU	13	0.146	89.041

`SC_THREAD` would be like s_1; $wait()$; s_2; $wait()$; ...; s_k; $wait()$, where s_i are the states. In such a case, the `SC_METHOD` results in:

```
static int i = 1;
switch(i)
  case 1: s_1; i++;
  break;
  case 2: s_2; i++;
  break;
  ...
  case k; s_k; i = 1;
```

The structure of thread T may be slightly different, and one can take care of such differences in a similar fashion. We call a thread T which runs in an infinite loop, and has k `wait()` statements, as a k-periodic thread. Every k cycle, its behavior repeats. However, the basic idea is that, each thread can be thought of as 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_is. 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 till 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 straight forward 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.

By formalizing this in a formal parallel hierarchical state machine model, one can easily prove that the CUDA simulation preserves the cycle by cycle behavior of the original SystemC simulation. However, here, we just provide this intuitive argument, and a future version of the paper will contain the rigorous formalization and proof. The intuitive proof has to take into account various other scenarios, such as `wait()/notify()` communications between threads, which would lead to asynchronous execution across the barriers, as well as more general thread structures in the SystemC model. However, one can see from the above semiformal argument that such cases can be taken care of in the formalization.

5.1.7 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. All the stages are sensitive to clock and can be pipelined. The SystemC implementation is done by implementing each stage as a `SC_THREAD`. Figure 5.3 shows the implementation hierarchy of the AES in SystemC with 12 `SC_THREAD`s modules. OSCI SystemC simulator executes each of these `SC_THREAD`s one after the other sequentially for a change in clock signal.

In CUDA implementation we create 12 threads; one thread for each stage, one for input stage and the other one for output stage. The *threadId*s are selected such that threads belong to different warps. All these threads execute in parallel, independent of one another and barrier synchronization technique is used for synching them. The CUDA code format is as shown in Listing 5.3. We present a few useful conversion rules. As shown in Listing 2 and 3 *wait()* statements and *while(true)* loops get converted to switch statements. Sensitivity lists get

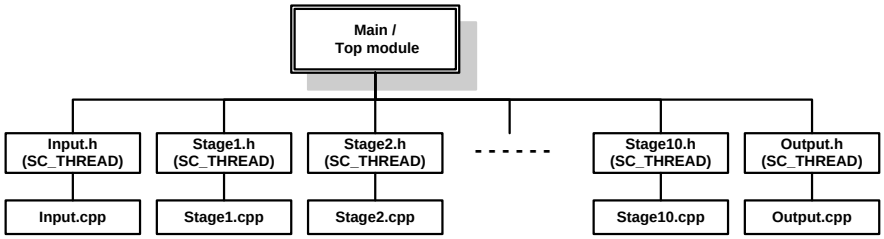

Figure 5.3: Hierarchical representation of pipelined AES implementation in SystemC

converted to *if* based conditional statements. All variables that are used in sensitivity lists are double buffered. 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_signal`s 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 data types `sc_uint`, `sc_int`, `sc_bit`, etc., in SystemC get converted to `unsigned int`, `int`, `bool`, etc., data types in CUDA.

Listing 5.3: AES CUDA Kernel

```
//Shared variable declarations
__global__ static void AES_kernel(int *result){
   int idx = threadIdx.x;
      //Any local variable declarations
  for(i=0;i<MAX_CYCLES;i++){
    if(idx == 352) {
      if(clock!=clock_copy)
      //INPUT STAGE CODE
      }
    ////////////////////////////////////
  __syncthreads();  // Barrier Synchronization
    else if(idx == 320) {
      if(clock!=clock_copy)
      // STAGE  1 CODE
    }
    else if(idx == 288){
      if(clock!=clock_copy)
      // STAGE  2 CODE
    }
    else if(idx == 256) {
      if(clock!=clock_copy)
      // STAGE  3 CODE
    }
    else if(idx == 224) {
      if(clock!=clock_copy)
      // STAGE  4 CODE
    }
    else if(idx == 192){
      if(clock!=clock_copy)
      // STAGE  5 CODE
    }
    else if(idx == 160){
      if(clock!=clock_copy)
```

```
      // STAGE  6 CODE
    }
    else if(idx == 128) {
      if(clock!=clock_copy)
      // STAGE  7 CODE
    }
    else if(idx == 96) {
      if(clock!=clock_copy)
      // STAGE  8 CODE
    }
    else if(idx == 64){
      if(clock!=clock_copy)
      // STAGE  9 CODE
    }
    else if(idx == 32){
      if(clock!=clock_copy)
      // STAGE  10 CODE
    }
    ////////////////////////////
  __syncthreads();  // Barrier Synchronization
    else if(idx == 0) {
      if(clock!=clock_copy){
      //OUTPUT STAGE CODE
      //DOUBLE BUFFERED VARIABLES DATA RESTORED
        // BY MAKING THE VALUES SAME
      }
    }
  }
}
```

The data type sizes are random. Hence instead of just assigning values to the variables, we check for overflow and then assign. 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.

5.2 Mapping of CUDA designs to TBB designs

In this section we present a systematic approach of converting CUDA designs to TBB designs. A CUDA program consists of main function from where the CUDA kernel is called in. A sample main function and CUDA kernel is listed in Appendix A.3 and A.4. In our implementation, main function does kernel call and output display functionalities. Actual simulation is taken care by CUDA kernel. Inside CUDA kernel, it starts with global variables and shared variables declaration. Then the partitions of the design to be simulated are listed in a for loop. Each partition is independent of each other and can be parallely simulated. Each of these partitions are made to execute by separate CUDA threads by distinguishing the individual partition code using $threadID$ as shown in Figure 5.4. Since we are implementing a cycle accurate simulator, we have to synchronize after each cycle of simulation. At the end of each cycle, save the outputs if necessary and save states too. If double buffering is implemented on certain variables, then make sure to copy the respective values into respective variables.

We replicate this programming model in TBB using *taskgroup* construct. Conversion of CUDA threads to TBB tasks directly can incur a lot of overhead as CPUs don't have numerous cores like GPUs. Threads that TBB creates is usually limited to the total number of cores on the system but even if TBB creates a large amount of threads, the throughput will be limited by the total number of cores on the system. To avoid the context switching overhead, TBB creates only as many number of working threads as the number of cores in the system. Keeping this in mind, we have to combine few CUDA threads to TBB tasks. One performance parameter to be kept in consideration is, each TBB task has to run at least for 10000 cycles to nullify the task creation overhead which is about 600 cycles. Combining the CUDA threads to make bigger TBB task can be tricky if shared variable and local variable declarations are not done carefully. In CUDA, global variables are variables which can be accessed by all the threads and hence they are not duplicated. Hence global variables remain as global variables in TBB as well. Shared variables in CUDA are duplicated in each multi-processor. All the threads in a block have access to these shared variables. One simple rule would be to combine all the CUDA threads in a block to make a task in TBB. This would also mean that shared variables get duplicated only once inside each task. In case if CUDA threads belonging to different blocks are combined to make a single task in TBB, care has to be taken while declaring shared variables by declaring shared variables of both the blocks in the TBB task. Also no shared variable in the different blocks should have same name. All the local variables of the CUDA threads being combined have to renamed and declared inside each task. Finally after all the variable declarations are taken care, tasks created have to be added to a task object and run. Synchronization in TBB while using *taskgroup* construct is done using *task_group_object.wait*() function which is very similar to *__syncthreads*() in CUDA.

Here we list few of the prominent rules needed in converting CUDA programs to TBB *taskgroup* construct. Similarly we can convert CUDA designs to other TBB constructs. Figure 5.4 shows the implementation of the simulation technique described in Section 5.1.4 and Figure 5.5 shows the mapping of a generic CUDA program to TBB.

Mapping Rules:

1. Global variables in CUDA remain global variables in TBB.
2. Shared variables in CUDA have to be handled in a different way in TBB. Threads belonging to single block only have access to shared variables. It's a good idea to combine threads belong to single block into a *task* in TBB. In that case all the shared variables should be declared inside the description of the *task*.
3. Local variables used in each thread of CUDA has to be declared carefully when merging CUDA threads to *tasks* in TBB. If we are not combining threads to *tasks*, then local variables of CUDA threads can be declared as local variables of *tasks* in TBB.

4. When creating a *task*, care should be taken that weight of each task in terms of cycle is 10000 cycles minimum.

5. Synchronization in TBB is done using *task_group_object.wait*(), which is similar to *__sync__threads*() in CUDA.

6. Device memory allocations have to be removed.

5.2.1 Extending the translator

As explained in Section 5.2, CUDA programs can be converted easily to TBB programs with the help of mapping rules described in Section 5.2. This extends the functionality of the SCGPSim translator to convert SystemC RTL designs to TBB designs too. At present this conversion is done manually, but using the mapping rules described earlier, the conversion can be done automatically using a parser and converter tool. First the CUDA kernel is parsed and global, shared and local variables are collected and using the rules described earlier. Then part of the code of CUDA threads belonging to same block are combined by detecting the *threadID* and made into a single task. Similarly all tasks are created and added to the task group object and run the task group object for MAX_CYCLES number of cycles.

5.2.2 Experimental Results

This framework is platform independent as the synthesized TBB code can run on any system with TBB libraries. The experiments were conducted on the Intel Xeon E5405 Quadcore processor with clock running at 2Ghz with 4GB RAM, 512KB L1 cache and 1MB L2 cache and the system is running Windows XP operating system. The libraries used for experiments are TBB library version 2.2. Preliminary experiments were conducted on small and medium sized examples and the results are tabulated in Table 5.2. First column represents the designs on which experiments were conducted and subsequent columns represents performance using TBB, CUDA and comparison of performance in terms of speed up. Each experiment was ran for 100000 cycles and the outputs were compared for accuracy using experiment dependent scripts.

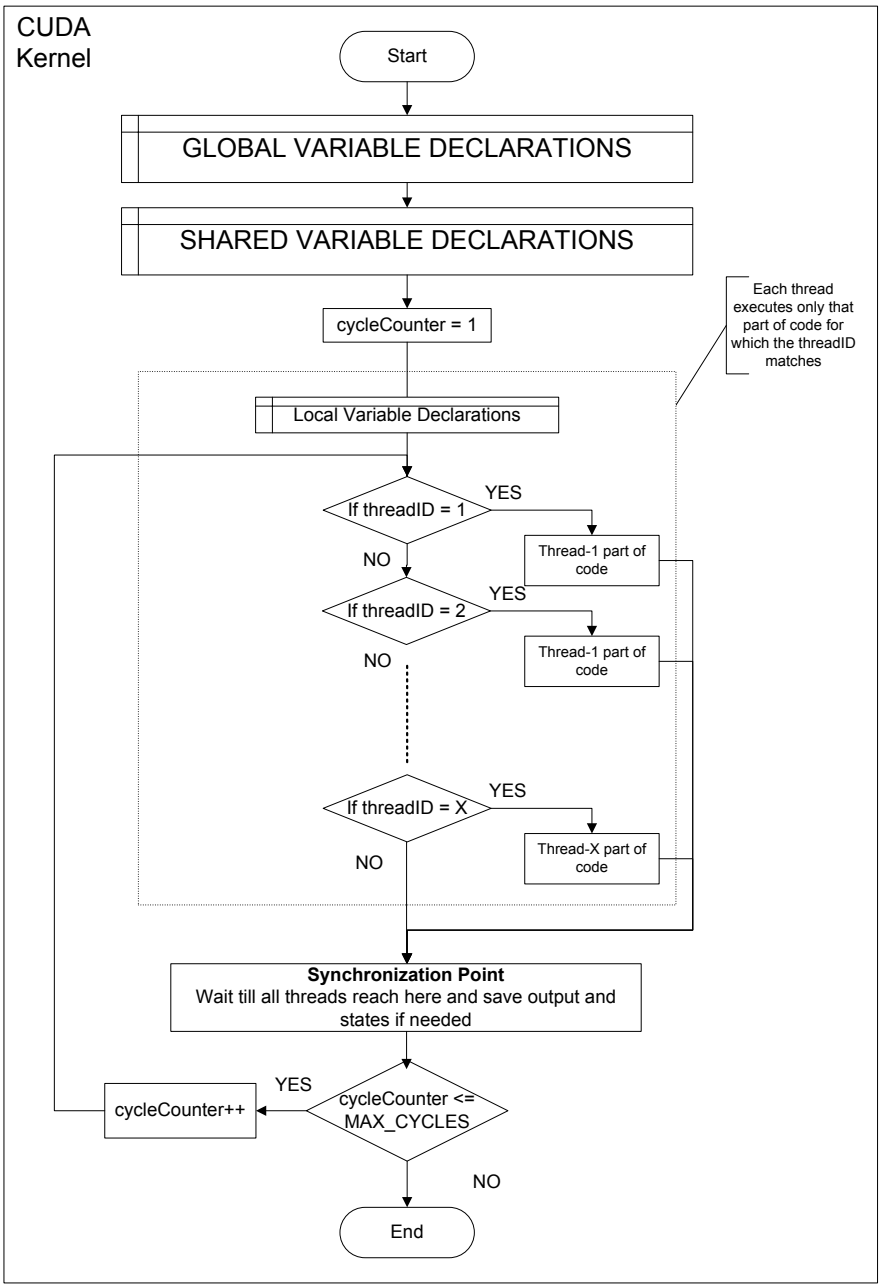

Figure 5.4: Implementation of simulation technique(Figure 5.2) in CUDA

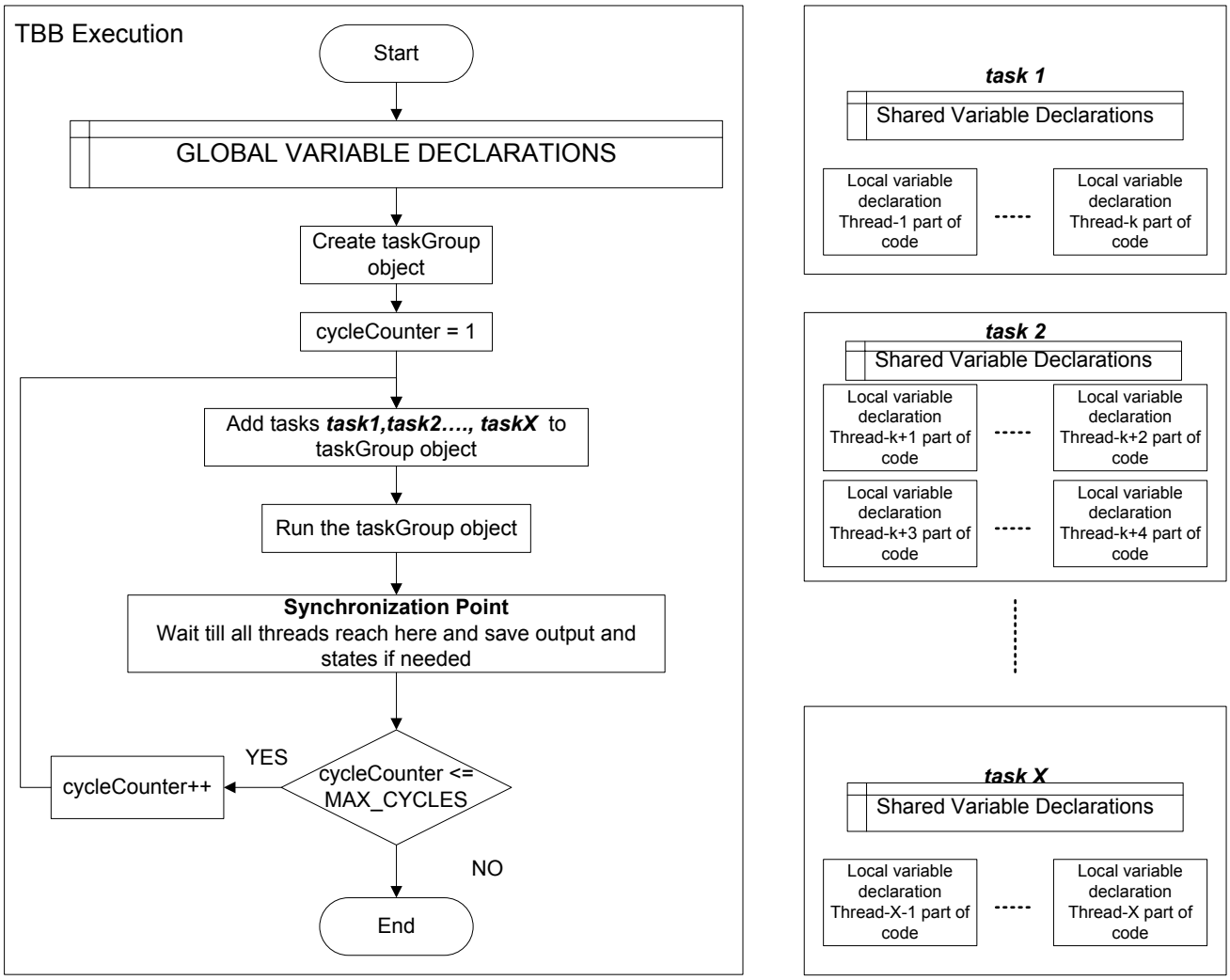

Figure 5.5: Mapping of the CUDA program to TBB *taskgroup* construct

Table 5.2: Experimental Results: SystemC simulation using TBB

Design	CPU(TBB) (sec)	CPU + GPU(CUDA) (sec)
PIPELINE AES	1.35	3.916
FIR	0.36	1.37
10 STAGE BUFFER	0.97	0.277
3 STAGE BUFFER	0.37	0.276
SIMPLE ALU	0.01	0.146

Chapter 6

Conclusion

With the diminishing gain from uniprocessor systems, in computing world there is a gradual shift towards complex multi-core and many-core systems. Electronic design automation EDA tools play a central role in bridging the productivity gap for designing complex hardware systems. They assist designers in simulation, verification and synthesis of these complex systems. Current EDA tools are unable to match the design requirements of these complex systems. One prominent reason for this being, most of the EDA algorithms and their implementations are designed for uniprocessor systems that cannot easily leverage the parallelism in multi-core and many-core systems. Most of the hardware simulation techniques followed, try to mimic the inherent parallelism of circuits using sequential programming languages. These techniques suit well on uniprocessor systems but are inefficient on multiprocessor systems as mimicking doesn't untilize the raw computational power of multiprocessor systems. Our research investigated automatic parallelization of simulation techniques used in designing hardware systems on GPUs and multi-cores. EDA on GPUs is an upcoming domain and simulation being a very compute intensive application, catches researchers interest. Most of the work in this domain is done in academia until concrete evidence showing advantage over traditional methods is clearly shown. Here our main aim was to explore the use of GPUs for hardware simulation and from the experiments we are successful to a certain extent.

SCGPSim:

In this work we present a method to automatically translate synthesizable SystemC designs into parallely executable CUDA designs. We believe synthesizable SystemC designs are a perfect fit for initial exploration of simulation using CUDA. The synthesis approach implemented employs semantic-preserving transformations such that the discrete-event semantics of SystemC's reference implementation is preserved. We present the core of our algorithm that can be employed for transformation. We also provide an intuitive argument

for the equivalence of the parallel CUDA design with the original SystemC design. In our future work, we would be presenting the formal proof of equivalence between the CUDA and SystemC designs. The simulation results show that we can obtain speedups ranging from 30x-100x. Considering that transformations are not optimized, we believe that optimizing them will improve the simulation performance even more. Our future work involves enabling the synthesizer with translations of designs beyond the synthesizable subset of SystemC and automation of verification of results. With a lot of high-level power estimation and reduction techniques being developed[33, 34, 35, 36, 37, 38] for faster power profiling, faster simulation techniques can further improve the performance of these estimation and reduction tools.

PHDLSim:

In this research we concentrated on developing a framework that can simulate hardware designs specified in Verilog/VHDL on multi-cores. We first synthesized the hardware designs to obtain netlists and used translator that automatically partitioned efficiently such that communication overhead is low. The partitioned netlists are transformed into TBB/Pthread programs, which can leverage multiple processors on CMP systems. These transformations retain the semantics of the original hardware design. While a formal proof is hard to develop at this time, we verify the correctness of the semantics through rigorous testing. Our experiments show that we were successful in preserving the simulation semantics and effectively utilized the parallelism in CMPs using threading libraries. We explained in detail the design flow, conversion procedures, code generation and results collected from experimental simulations. We also presented the synchronization, task creation, context switching overheads and performance of different threading libraries for the simulation purpose.

For future work, we will provide a formal proof of the equivalence between the original hardware design and the generated code. We have yet to optimize our partitioning algorithms, test on real large designs; hence, the potential for further improvements in performance are high. Moreover, we will explore statically identifying certain types of designs that may perform better using particular programming patterns in TBB such as pipelines. Our approach will again be to identify such patterns and transform them automatically. We plan to extend this framework for targets such as Intel's Larabee.

Appendix A

SCGPSim: Sample FIR design

A.1 Flattened source file

Listing A.1: Flattened Source file

```
/*****************************************************************************
  display.h --
 *****************************************************************************/
SC_MODULE(display) {

  sc_in<bool>  output_data_ready;
  sc_in<int>   result;

  int i, tmp1;

  SC_CTOR(display)
    {
      SC_METHOD(entry);
      dont_initialize();
      sensitive_pos(output_data_ready);
      i = 0;
    }

  void entry();
};

/*****************************************************************************
  fir.h --
 *****************************************************************************/
SC_MODULE(fir) {

  sc_in<bool>  reset;
  sc_in<bool>  input_valid;
  sc_in<int>   sample;
  sc_out<bool> output_data_ready;
  sc_out<int>  result;
  sc_in_clk    CLK;

  sc_int<9> coefs[16];

  SC_CTOR(fir)
     {
```

```
      SC_CTHREAD(entry, CLK.pos());
      watching(reset.delayed() == true);
      #include "fir_const.h"
    }

  void entry();
};

/*****************************************************************************
  fir_const.h --
 *****************************************************************************/
coefs[0] = -6;
coefs[1] = -4;
coefs[2] = 13;
coefs[3] = 16;
coefs[4] = -18;
coefs[5] = -41;
coefs[6] = 23;
coefs[7] = 154;
coefs[8] = 222;
coefs[9] = 154;
coefs[10] = 23;
coefs[11] = -41;
coefs[12] = -18;
coefs[13] = 16;
coefs[14] = 13;
coefs[15] = -4;

/*****************************************************************************
  fir_data.h --
 *****************************************************************************/
SC_MODULE(fir_data) {

  sc_in<bool>      reset;
  sc_in<unsigned>  state_out;
  sc_in<int>       sample;
  sc_out<int>      result;
  sc_out<bool>     output_data_ready;

  sc_int<19> acc;
  sc_int<8> shift[16];
  sc_int<9> coefs[16];

  SC_CTOR(fir_data)
    {
      SC_METHOD(entry);
      dont_initialize();
      sensitive(reset);
      sensitive(state_out);
      sensitive(sample);
#include "fir_const.h"
    };
  void entry();
};

/*****************************************************************************
  fir_fsm.h --
 *****************************************************************************/
SC_MODULE(fir_fsm) {

  sc_in<bool>      clock;
  sc_in<bool>      reset;
  sc_in<bool>      in_valid;
  sc_out<unsigned> state_out;
```

```
  // defining the states of the ste machine
  enum {reset_s, first_s, second_s, third_s, output_s} state;

  SC_CTOR(fir_fsm)
    {
      SC_METHOD(entry);
      dont_initialize();
      sensitive_pos(clock);
    };
  void entry();
};

/*****************************************************************************
  fir_top.h --
 *****************************************************************************/
#include <systemc.h>
#include "fir_fsm.h"
#include "fir_data.h"

SC_MODULE(fir_top) {

  sc_in<bool>       CLK;
  sc_in<bool>       RESET;
  sc_in<bool>       IN_VALID;
  sc_in<int>     SAMPLE;
  sc_out<bool>      OUTPUT_DATA_READY;
  sc_out<int>       RESULT;

  sc_signal<unsigned> state_out;

  fir_fsm  *fir_fsm1;
  fir_data *fir_data1;

  SC_CTOR(fir_top) {

      fir_fsm1 = new fir_fsm("FirFSM");
      fir_fsm1->clock(CLK);
      fir_fsm1->reset(RESET);
      fir_fsm1->in_valid(IN_VALID);
      fir_fsm1->state_out(state_out);

      fir_data1 = new fir_data("FirData");
      fir_data1 -> reset(RESET);
      fir_data1 -> state_out(state_out);
      fir_data1 -> sample(SAMPLE);
      fir_data1 -> result(RESULT);
      fir_data1 -> output_data_ready(OUTPUT_DATA_READY);

    }
};

/*****************************************************************************
  stimulus.h --
 *****************************************************************************/
SC_MODULE(stimulus) {

  sc_out<bool> reset;
  sc_out<bool> input_valid;
  sc_out<int>  sample;
  sc_in<bool>  CLK;

  sc_int<8>     send_value1;
  unsigned cycle;
```

```
  SC_CTOR(stimulus)
  {
      SC_METHOD(entry);
      dont_initialize();
      sensitive_pos(CLK);
      send_value1 = 0;
      cycle       = 0;
  }
  void entry();
};

/*****************************************************************************
  display.cpp --
 *****************************************************************************/
#include <systemc.h>
#include "display.h"

void display::entry(){
  //  Reading Data when valid if high
  tmp1 = result.read();
  result_array[i] = tmp1;
  i++;
  if(i == MAX_CYCLES) {
    sc_stop();
  };
}

/*****************************************************************************
  fir.cpp --
 *****************************************************************************/
#include <systemc.h>
#include "fir.h"

void fir::entry() {

  sc_int<8>  sample_tmp;
  sc_int<17> pro;
  sc_int<19> acc;
  sc_int<8> shift[16];

  // reset watching
  /* this would be an unrolled loop */
  for (int i=0; i<=15; i++)
    shift[i] = 0;
  result.write(0);
  output_data_ready.write(false);
  wait();

  // main functionality
  while(1) {
    output_data_ready.write(false);
    wait_until(input_valid.delayed() == true);
    sample_tmp = sample.read();
    acc = sample_tmp*coefs[0];

    for(int i=14; i>=0; i--) {
      /* this would be an unrolled loop */
      pro = shift[i]*coefs[i+1];
      acc += pro;
    };

    for(int i=14; i>=0; i--) {
      /* this would be an unrolled loop */
```

```
      shift[i+1] = shift[i];
    };

    shift[0] = sample_tmp;
    // write output values
    result.write((int)acc);
    output_data_ready.write(true);
    wait();
  };
}

/******************************************************************************
  fir_data.cpp --
 ******************************************************************************/
#include <systemc.h>
#include "fir_data.h"

void fir_data::entry()
{
  int state;
  sc_int<8> sample_tmp;

  // reset functionality
  if(reset.read()==true) {
    sample_tmp   = 0;
    acc = 0;
    for (int i=0; i<=15; i++)
      shift[i] = 0;
  }

  // default settings
  result.write(0);
  output_data_ready.write(false);
  state = state_out.read();

  // cycle behavior could be as well a case statement
  switch (state) {
  case 1 :
    sample_tmp = sample.read();
    acc = sample_tmp*coefs[0];
    acc += shift[14]* coefs[15];
    acc += shift[13]*coefs[14];
    acc += shift[12]*coefs[13];
    acc += shift[11]*coefs[12];
    break;
  case 2 :
    acc += shift[10]*coefs[11];
    acc += shift[9]*coefs[10];
    acc += shift[8]*coefs[9];
    acc += shift[7]*coefs[8];
    break;
  case 3 :
    acc += shift[6]*coefs[7];
    acc += shift[5]*coefs[6];
    acc += shift[4]*coefs[5];
    acc += shift[3]*coefs[4];
    break;
  case 4 :
    acc += shift[2]*coefs[3];
    acc += shift[1]*coefs[2];
    acc += shift[0]*coefs[1];
    for(int i=14; i>=0; i--) {
      shift[i+1] = shift[i];
    };
```

```
    shift[0] = sample.read();
    result.write((int)acc);
    output_data_ready.write(true);
    break;
  default :
    break;
  }
}

/******************************************************************************
  fir_fsm.cpp --
 ******************************************************************************/
#include <systemc.h>
#include "fir_fsm.h"

void fir_fsm::entry() {

  sc_uint<3> state_tmp;

  // reset behavior
  if(reset.read()==true) {
    state = reset_s;
  } else {
    // main state machine
    switch(state) {
    case reset_s:
      state = first_s;
      state_tmp = 0;
      state_out.write((unsigned)state_tmp);
      break;
    case first_s:
      if(in_valid.read()==true) {
  state = second_s;
      };
      state_tmp = 1;
      state_out.write((unsigned)state_tmp);
      break;
    case second_s:
      state = third_s;
      state_tmp = 2;
      state_out.write((unsigned)state_tmp);
      break;
    case third_s:
      state = output_s;
      state_tmp = 3;
      state_out.write((unsigned)state_tmp);
      break;
    default:
      state = first_s;
      state_tmp = 4;
      state_out.write((unsigned)state_tmp);
      break;
    }
  }
}

/******************************************************************************
  main.cpp --
 ******************************************************************************/

#include <systemc.h>
#include "stimulus.h"
#include "display.h"
#include "fir.h"
```

```
int sc_main (int argc , char *argv[]) {
  sc_clock         clock;
  sc_signal<bool>  reset;
  sc_signal<bool>  input_valid;
  sc_signal<int>   sample;
  sc_signal<bool>  output_data_ready;
  sc_signal<int>   result;

  stimulus stimulus1("stimulus_block");
  stimulus1.reset(reset);
  stimulus1.input_valid(input_valid);
  stimulus1.sample(sample);
  stimulus1.CLK(clock.signal());

  fir fir1( "process_body");
  fir1.reset(reset);
  fir1.input_valid(input_valid);
  fir1.sample(sample);
  fir1.output_data_ready(output_data_ready);
  fir1.result(result);
  fir1.CLK(clock);

  display  display1 ( "display");
  display1.output_data_ready(output_data_ready);
  display1.result(result);

  sc_start(clock, -1);
  return 0;
}

/*****************************************************************************
  stimulus.cpp --
 *****************************************************************************/
#include <systemc.h>
#include "stimulus.h"

void stimulus::entry() {

  cycle++;
  // sending some reset values
  if (cycle<4) {
    reset.write(true);
    input_valid.write(false);
  } else {
    reset.write(false);
    input_valid.write( false );
    // sending normal mode values
    if (cycle%10==0) {
      input_valid.write(true);
      sample.write( (int)send_value1 );
      send_value1++;
    };
  }
}
```

A.2 Intermediate XML format

Listing A.2: Intermediate XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE file SYSTEM "systemc.dtd">
<file name = "fir_flat.cc">
    <module type = "SC_MODULE" name = "display">
      <inport type = "bool" name = "output_data_ready"/>
      <inport type = "int" name = "result"/>
      <constructorof modulename  = "display">
      </constructorof>
    </module>
    <module type = "SC_MODULE" name = "fir">
      <inport type = "bool" name = "reset"/>
      <inport type = "bool" name = "input_valid"/>
      <inport type = "int" name = "sample"/>
      <outport type = "bool" name = "output_data_ready"/>
      <outport type = "int" name = "result"/>
      <inclk name = "CLK" />
      <constructorof modulename  = "fir">
      </constructorof>
    </module>
    <module type = "SC_MODULE" name = "fir_data">
      <inport type = "bool" name = "reset"/>
      <inport type = "unsigned" name = "state_out"/>
      <inport type = "int" name = "sample"/>
      <outport type = "int" name = "result"/>
      <outport type = "bool" name = "output_data_ready"/>
      <constructorof modulename  = "fir_data">
      </constructorof>
    </module>
    <module type = "SC_MODULE" name = "fir_fsm">
      <inport type = "bool" name = "clock"/>
      <inport type = "bool" name = "reset"/>
      <inport type = "bool" name = "in_valid"/>
      <outport type = "unsigned" name = "state_out"/>
      <constructorof modulename  = "fir_fsm">
      </constructorof>
    </module>
    <module type = "SC_MODULE" name = "fir_top">
      <inport type = "bool" name = "CLK"/>
      <inport type = "bool" name = "RESET"/>
      <inport type = "bool" name = "IN_VALID"/>
      <inport type = "int" name = "SAMPLE"/>
      <outport type = "bool" name = "OUTPUT_DATA_READY"/>
      <outport type = "int" name = "RESULT"/>
      <signal type = "unsigned" name = "state_out"/>
      <submodule module="fir_fsm" name="fir_fsm1"/>
      <submodule module="fir_data" name="fir_data1"/>
      <constructorof modulename  = "fir_top">
        <connection instance="fir_fsm1" member="clock" local_signal="CLK"/>
        <connection instance="fir_fsm1" member="reset" local_signal="RESET"/>
        <connection instance="fir_fsm1" member="in_valid" local_signal="IN_VALID"/>
        <connection instance="fir_fsm1" member="state_out" local_signal="state_out"/>
        <connection instance="fir_data1" member="reset" local_signal="RESET"/>
        <connection instance="fir_data1" member="state_out" local_signal="state_out"/>
        <connection instance="fir_data1" member="sample" local_signal="SAMPLE"/>
        <connection instance="fir_data1" member="result" local_signal="RESULT"/>
        <connection instance="fir_data1" member="output_data_ready" local_signal="
            OUTPUT_DATA_READY"/>
      </constructorof>
    </module>
```

```
    <module type = "SC_MODULE" name = "stimulus">
      <outport type = "bool" name = "reset"/>
      <outport type = "bool" name = "input_valid"/>
      <outport type = "int" name = "sample"/>
      <inport type = "bool" name = "CLK"/>
      <constructorof modulename  = "stimulus">
      </constructorof>
    </module>
    <module type = "sc_main" name = "sc_main">
      <signal type = "bool" name = "reset"/>
      <signal type = "bool" name = "input_valid"/>
      <signal type = "int" name = "sample"/>
      <signal type = "bool" name = "output_data_ready"/>
      <signal type = "int" name = "result"/>
    <submodule name="stimulus1" module="stimulus"/>
        <connection instance="stimulus1" member="reset" local_signal="reset"/>
        <connection instance="stimulus1" member="input_valid" local_signal="input_valid"/>
        <connection instance="stimulus1" member="sample" local_signal="sample"/>
        <connection instance="stimulus1" member="CLK" local_signal="clock.signal()"/>
    <submodule name="fir1" module="fir"/>
        <connection instance="fir1" member="reset" local_signal="reset"/>
        <connection instance="fir1" member="input_valid" local_signal="input_valid"/>
        <connection instance="fir1" member="sample" local_signal="sample"/>
        <connection instance="fir1" member="output_data_ready" local_signal="
            output_data_ready"/>
        <connection instance="fir1" member="result" local_signal="result"/>
        <connection instance="fir1" member="CLK" local_signal="clock"/>
    <submodule name="display1" module="display"/>
        <connection instance="display1" member="output_data_ready" local_signal="
            output_data_ready"/>
        <connection instance="display1" member="result" local_signal="result"/>
    </module>
</file>
```

A.3 Main CUDA file

Listing A.3: Main CUDA file

```
/*************************************************************
*  FIR.cu
*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil.h>
#include <conio.h>
#include <fir_kernel.cu>
#include <time.h>

/*************************************************************/
/* Main File                                                 */
/*************************************************************/
int main(int argc, char* argv[])
{
  time_t start, end;
  double dif;

  FILE *pfile = fopen("temp.log","wb");
```

```
  unsigned long int i;

  int *device_result  = 0;
  static int  host_result[3*MAX_CYCLES];

  time(&start);

  CUDA_SAFE_CALL( cudaMalloc((void**) &device_result, sizeof(int) *3* MAX_CYCLES));

  unsigned int timer = 0;
  CUT_SAFE_CALL( cutCreateTimer( &timer));
  CUT_SAFE_CALL( cutStartTimer( timer));

  dim3 dimGrid(1,1,1);
    dim3 dimBlock(128,1,1);

  fir_kernel<<<dimGrid, dimBlock>>>(device_result);

  CUDA_SAFE_CALL( cudaThreadSynchronize() );

  CUDA_SAFE_CALL( cudaMemcpy(&host_result, device_result, sizeof(int) *3* MAX_CYCLES,
      cudaMemcpyDeviceToHost));

  CUDA_SAFE_CALL( cudaFree(device_result));
  CUT_SAFE_CALL( cutStopTimer( timer));

  for(i=0;i<3*MAX_CYCLES;i+=3)
  {
      printf("%d  %d  %d\n",host_result[i],host_result[i+1],host_result[i+2]);
      fprintf(pfile,"%d  %d  %d\n",host_result[i],host_result[i+1],host_result[i+2]);

  }

  printf("Processing time: %f (ms)\n", cutGetTimerValue( timer));
  CUT_SAFE_CALL( cutDeleteTimer( timer));

  time(&end);

  dif = difftime (end,start);
  printf ("It took you %f seconds to execute full.\n", dif );

  getch();
  return 0;
}
```

A.4 CUDA Kernel file

Listing A.4: Converted CUDA source

```
#define MAX_CYCLES 100000

__shared__ int clk;
__shared__ int reset;
__shared__ int input_valid;
__shared__ long int sample;
__shared__ int output_data_ready;
__shared__ long int result;
```

```
__shared__ long int send_value;
__shared__ long int cycle;

__shared__ int long state_tmp;
__shared__ int long state_out;
__shared__ int long sample_tmp;
__shared__ int long acc;
__shared__ int long shift[16];
__shared__ int long state1;
__shared__ long int counter;

__shared__ enum {reset_s, first_s, second_s, third_s, output_s} state;

__constant__ int coefs[]={-6,-4,13,16,-18,-41,23,154,222,154,23,-41,-18,16,13,-4};


/*****************************************************************/
/* FIR KERNEL                                                    */
/*****************************************************************/
__global__ static void fir_kernel(int* end_result)
{
  int idx = threadIdx.x;
  unsigned long int i;
  int m;

  if(idx==0)
  {
    cycle = 0;
    send_value = 0;
    counter = 0;
  }

  for(i=0;i<MAX_CYCLES;i++)
  {
    ////////////////////////////////////////////////////////////////////////
    if(idx == 0)
    {
      cycle++;
      // sending some reset values
      if (cycle<4)
      {
        reset = 1;
        input_valid = 0;
      }
      else
      {
        reset = 0;
        input_valid = 0;
        // sending normal mode values
        if (cycle%10==0)
        {
          input_valid = 1;
          sample = send_value;
          send_value++;
        }
      }
    }
    __syncthreads();
    ////////////////////////////////////////////////////////////////////////
    if(idx == 33)
    {
      // reset functionality
      if(reset == 1)
      {
```

```
      sample_tmp = 0;
      acc = 0;
      for (m=0; m<=15; m++)
        shift[m] = 0;
    }

    // default settings
    result = 0;
    output_data_ready = 0;
    state1 = state_out;

    // cycle behavior could be as well a case statement
    switch (state1)
    {
    case 1 :
      sample_tmp = sample;
      acc = sample_tmp*coefs[0];
      acc += shift[14]*coefs[15];
      acc += shift[13]*coefs[14];
      acc += shift[12]*coefs[13];
      acc += shift[11]*coefs[12];
      break;
    case 2 :
      acc += shift[10]*coefs[11];
      acc += shift[9]*coefs[10];
      acc += shift[8]*coefs[9];
      acc += shift[7]*coefs[8];
      break;
    case 3 :
      acc += shift[6]*coefs[7];
      acc += shift[5]*coefs[6];
      acc += shift[4]*coefs[5];
      acc += shift[3]*coefs[4];
      break;
    case 4 :
      acc += shift[2]*coefs[3];
      acc += shift[1]*coefs[2];
      acc += shift[0]*coefs[1];
      for(m=14; m>=0; m--)
        shift[m+1] = shift[m];

      shift[0] = sample;
      result = acc;
      output_data_ready = 1;
      break;
    default :
      break;
    }
  }
  ////////////////////////////////////////////////////////////////////////
  if(idx == 66)
  {
      // reset behavior
      if(reset == 1)
      {
      state = reset_s;
      }
      else
      {
      // main state machine
      switch(state)
      {
      case reset_s:
        state = first_s;
```

```
          state_tmp = 0;
          state_out = state_tmp;
          break;
        case first_s:
          if(input_valid == 1)
              state = second_s;
          state_tmp = 1;
          state_out = state_tmp;
          break;
        case second_s:
          state = third_s;
          state_tmp = 2;
          state_out = state_tmp;
          break;
        case third_s:
          state = output_s;
          state_tmp = 3;
          state_out = state_tmp;
          break;
        default:
          state = first_s;
          state_tmp = 4;
          state_out = state_tmp;
          break;
        }
        }
    }
    ////////////////////////////////////////////////////////////////////////
    __syncthreads();
    ////////////////////////////////////////////////////////////////////////
    if(idx == 100)
    {
      if(output_data_ready == 1)
      {
        end_result[counter] = result;
        end_result[counter+1] = output_data_ready;
        end_result[counter+2] = sample;
        counter+=3;
      }
    }
    if(input_valid!=1)
      i--;
    ////////////////////////////////////////////////////////////////////////
  }
}
```

Appendix B

PHDLSim: Sample adder design

B.1 EDIF intermediate netlist

Listing B.1: EDIF netlist representation

```
(edif add
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
  (status
    (written
      (timeStamp 2009 8 12 10 2 20)
      (author "Synplicity, Inc.")
      (program "Synplify Pro" (version "Version 9.4, mapper 9.4.0, Build 086R"))
     )
   )
  (library VIRTEX
    (edifLevel 0)
    (technology (numberDefinition ))
    (cell IBUF (cellType GENERIC)
       (view PRIM (viewType NETLIST)
         (interface
           (port O (direction OUTPUT))
           (port I (direction INPUT))
         )
       )
    )
    (cell OBUF (cellType GENERIC)
       (view PRIM (viewType NETLIST)
         (interface
           (port O (direction OUTPUT))
           (port I (direction INPUT))
         )
       )
    )
    (cell LUT2 (cellType GENERIC)
       (view PRIM (viewType NETLIST)
         (interface
           (port I0 (direction INPUT))
           (port I1 (direction INPUT))
           (port O (direction OUTPUT))
         )
```

```
      )
   )
   (cell XORCY (cellType GENERIC)
      (view PRIM (viewType NETLIST)
        (interface
          (port LI (direction INPUT))
          (port CI (direction INPUT))
          (port O (direction OUTPUT))
        )
      )
   )
   (cell MUXCY_L (cellType GENERIC)
      (view PRIM (viewType NETLIST)
        (interface
          (port DI (direction INPUT))
          (port CI (direction INPUT))
          (port S (direction INPUT))
          (port LO (direction OUTPUT))
        )
      )
   )
   (cell MUXCY (cellType GENERIC)
      (view PRIM (viewType NETLIST)
        (interface
          (port DI (direction INPUT))
          (port CI (direction INPUT))
          (port S (direction INPUT))
          (port O (direction OUTPUT))
        )
      )
   )
)
(library UNILIB
   (edifLevel 0)
   (technology (numberDefinition ))
   (cell GND (cellType GENERIC)
      (view PRIM (viewType NETLIST)
        (interface
          (port G (direction OUTPUT))
        )
      )
   )
)
(library work
   (edifLevel 0)
   (technology (numberDefinition ))
   (cell add (cellType GENERIC)
      (view verilog (viewType NETLIST)
        (interface
          (port (array (rename a "a[7:0]") 8) (direction INPUT))
          (port (array (rename b "b[7:0]") 8) (direction INPUT))
          (port (array (rename result "result[8:0]") 9) (direction OUTPUT))
        )
        (contents
         (instance result_axb_1 (viewRef PRIM (cellRef LUT2 (libraryRef VIRTEX)))
          (property INIT (string "6"))
         )
         (instance result_axb_2 (viewRef PRIM (cellRef LUT2 (libraryRef VIRTEX)))
          (property INIT (string "6"))
         )
         (instance result_axb_3 (viewRef PRIM (cellRef LUT2 (libraryRef VIRTEX)))
          (property INIT (string "6"))
         )
         (instance result_axb_4 (viewRef PRIM (cellRef LUT2 (libraryRef VIRTEX)))
```

```
 (property INIT (string "6"))
)
(instance result_axb_5 (viewRef PRIM (cellRef LUT2 (libraryRef VIRTEX)))
 (property INIT (string "6"))
)
(instance result_axb_6 (viewRef PRIM (cellRef LUT2 (libraryRef VIRTEX)))
 (property INIT (string "6"))
)
(instance result_axb_7 (viewRef PRIM (cellRef LUT2 (libraryRef VIRTEX)))
 (property INIT (string "6"))
)
(instance result_axb_0 (viewRef PRIM (cellRef LUT2 (libraryRef VIRTEX)))
 (property INIT (string "6"))
)
(instance result_s_7 (viewRef PRIM (cellRef XORCY (libraryRef VIRTEX)))
)
(instance result_cry_7 (viewRef PRIM (cellRef MUXCY (libraryRef VIRTEX)))
)
(instance result_s_6 (viewRef PRIM (cellRef XORCY (libraryRef VIRTEX)))
)
(instance result_cry_6 (viewRef PRIM (cellRef MUXCY_L (libraryRef VIRTEX)))
)
(instance result_s_5 (viewRef PRIM (cellRef XORCY (libraryRef VIRTEX)))
)
(instance result_cry_5 (viewRef PRIM (cellRef MUXCY_L (libraryRef VIRTEX)))
)
(instance result_s_4 (viewRef PRIM (cellRef XORCY (libraryRef VIRTEX)))
)
(instance result_cry_4 (viewRef PRIM (cellRef MUXCY_L (libraryRef VIRTEX)))
)
(instance result_s_3 (viewRef PRIM (cellRef XORCY (libraryRef VIRTEX)))
)
(instance result_cry_3 (viewRef PRIM (cellRef MUXCY_L (libraryRef VIRTEX)))
)
(instance result_s_2 (viewRef PRIM (cellRef XORCY (libraryRef VIRTEX)))
)
(instance result_cry_2 (viewRef PRIM (cellRef MUXCY_L (libraryRef VIRTEX)))
)
(instance result_s_1 (viewRef PRIM (cellRef XORCY (libraryRef VIRTEX)))
)
(instance result_cry_1 (viewRef PRIM (cellRef MUXCY_L (libraryRef VIRTEX)))
)
(instance result_cry_0 (viewRef PRIM (cellRef MUXCY_L (libraryRef VIRTEX)))
)
(instance (rename result_obuf_8 "result_obuf[8]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
)
(instance (rename result_obuf_7 "result_obuf[7]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
)
(instance (rename result_obuf_6 "result_obuf[6]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
)
(instance (rename result_obuf_5 "result_obuf[5]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
)
(instance (rename result_obuf_4 "result_obuf[4]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
)
(instance (rename result_obuf_3 "result_obuf[3]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
)
(instance (rename result_obuf_2 "result_obuf[2]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
```

```
)
(instance (rename result_obuf_1 "result_obuf[1]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
)
(instance (rename result_obuf_0 "result_obuf[0]") (viewRef PRIM (cellRef OBUF (
    libraryRef VIRTEX)))
)
(instance (rename b_ibuf_7 "b_ibuf[7]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename b_ibuf_6 "b_ibuf[6]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename b_ibuf_5 "b_ibuf[5]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename b_ibuf_4 "b_ibuf[4]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename b_ibuf_3 "b_ibuf[3]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename b_ibuf_2 "b_ibuf[2]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename b_ibuf_1 "b_ibuf[1]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename b_ibuf_0 "b_ibuf[0]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename a_ibuf_7 "a_ibuf[7]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename a_ibuf_6 "a_ibuf[6]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename a_ibuf_5 "a_ibuf[5]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename a_ibuf_4 "a_ibuf[4]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename a_ibuf_3 "a_ibuf[3]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename a_ibuf_2 "a_ibuf[2]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename a_ibuf_1 "a_ibuf[1]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance (rename a_ibuf_0 "a_ibuf[0]") (viewRef PRIM (cellRef IBUF (libraryRef
    VIRTEX)))
)
(instance GND (viewRef PRIM (cellRef GND (libraryRef UNILIB)))          )
(net (rename GNDZ0 "GND") (joined
 (portRef G (instanceRef GND))
 (portRef CI (instanceRef result_cry_0))
))
(net (rename resultZ0 "result") (joined
 (portRef O (instanceRef a_ibuf_0))
 (portRef DI (instanceRef result_cry_0))
 (portRef I1 (instanceRef result_axb_0))
```

```
))
(net (rename a_0 "a[0]") (joined
 (portRef (member a 7))
 (portRef I (instanceRef a_ibuf_0))
))
(net (rename a_c_1 "a_c[1]") (joined
 (portRef O (instanceRef a_ibuf_1))
 (portRef DI (instanceRef result_cry_1))
 (portRef I0 (instanceRef result_axb_1))
))
(net (rename a_1 "a[1]") (joined
 (portRef (member a 6))
 (portRef I (instanceRef a_ibuf_1))
))
(net (rename a_c_2 "a_c[2]") (joined
 (portRef O (instanceRef a_ibuf_2))
 (portRef DI (instanceRef result_cry_2))
 (portRef I0 (instanceRef result_axb_2))
))
(net (rename a_2 "a[2]") (joined
 (portRef (member a 5))
 (portRef I (instanceRef a_ibuf_2))
))
(net (rename a_c_3 "a_c[3]") (joined
 (portRef O (instanceRef a_ibuf_3))
 (portRef DI (instanceRef result_cry_3))
 (portRef I0 (instanceRef result_axb_3))
))
(net (rename a_3 "a[3]") (joined
 (portRef (member a 4))
 (portRef I (instanceRef a_ibuf_3))
))
(net (rename a_c_4 "a_c[4]") (joined
 (portRef O (instanceRef a_ibuf_4))
 (portRef DI (instanceRef result_cry_4))
 (portRef I0 (instanceRef result_axb_4))
))
(net (rename a_4 "a[4]") (joined
 (portRef (member a 3))
 (portRef I (instanceRef a_ibuf_4))
))
(net (rename a_c_5 "a_c[5]") (joined
 (portRef O (instanceRef a_ibuf_5))
 (portRef DI (instanceRef result_cry_5))
 (portRef I0 (instanceRef result_axb_5))
))
(net (rename a_5 "a[5]") (joined
 (portRef (member a 2))
 (portRef I (instanceRef a_ibuf_5))
))
(net (rename a_c_6 "a_c[6]") (joined
 (portRef O (instanceRef a_ibuf_6))
 (portRef DI (instanceRef result_cry_6))
 (portRef I0 (instanceRef result_axb_6))
))
(net (rename a_6 "a[6]") (joined
 (portRef (member a 1))
 (portRef I (instanceRef a_ibuf_6))
))
(net (rename a_c_7 "a_c[7]") (joined
 (portRef O (instanceRef a_ibuf_7))
 (portRef DI (instanceRef result_cry_7))
 (portRef I0 (instanceRef result_axb_7))
))
```

```
(net (rename a_7 "a[7]") (joined
 (portRef (member a 0))
 (portRef I (instanceRef a_ibuf_7))
))
(net (rename b_c_0 "b_c[0]") (joined
 (portRef O (instanceRef b_ibuf_0))
 (portRef I0 (instanceRef result_axb_0))
))
(net (rename b_0 "b[0]") (joined
 (portRef (member b 7))
 (portRef I (instanceRef b_ibuf_0))
))
(net (rename b_c_1 "b_c[1]") (joined
 (portRef O (instanceRef b_ibuf_1))
 (portRef I1 (instanceRef result_axb_1))
))
(net (rename b_1 "b[1]") (joined
 (portRef (member b 6))
 (portRef I (instanceRef b_ibuf_1))
))
(net (rename b_c_2 "b_c[2]") (joined
 (portRef O (instanceRef b_ibuf_2))
 (portRef I1 (instanceRef result_axb_2))
))
(net (rename b_2 "b[2]") (joined
 (portRef (member b 5))
 (portRef I (instanceRef b_ibuf_2))
))
(net (rename b_c_3 "b_c[3]") (joined
 (portRef O (instanceRef b_ibuf_3))
 (portRef I1 (instanceRef result_axb_3))
))
(net (rename b_3 "b[3]") (joined
 (portRef (member b 4))
 (portRef I (instanceRef b_ibuf_3))
))
(net (rename b_c_4 "b_c[4]") (joined
 (portRef O (instanceRef b_ibuf_4))
 (portRef I1 (instanceRef result_axb_4))
))
(net (rename b_4 "b[4]") (joined
 (portRef (member b 3))
 (portRef I (instanceRef b_ibuf_4))
))
(net (rename b_c_5 "b_c[5]") (joined
 (portRef O (instanceRef b_ibuf_5))
 (portRef I1 (instanceRef result_axb_5))
))
(net (rename b_5 "b[5]") (joined
 (portRef (member b 2))
 (portRef I (instanceRef b_ibuf_5))
))
(net (rename b_c_6 "b_c[6]") (joined
 (portRef O (instanceRef b_ibuf_6))
 (portRef I1 (instanceRef result_axb_6))
))
(net (rename b_6 "b[6]") (joined
 (portRef (member b 1))
 (portRef I (instanceRef b_ibuf_6))
))
(net (rename b_c_7 "b_c[7]") (joined
 (portRef O (instanceRef b_ibuf_7))
 (portRef I1 (instanceRef result_axb_7))
))
```

```
(net (rename b_7 "b[7]") (joined
 (portRef (member b 0))
 (portRef I (instanceRef b_ibuf_7))
))
(net (rename result_0 "result[0]") (joined
 (portRef O (instanceRef result_obuf_0))
 (portRef (member result 8))
))
(net (rename result_c_1 "result_c[1]") (joined
 (portRef O (instanceRef result_s_1))
 (portRef I (instanceRef result_obuf_1))
))
(net (rename result_1 "result[1]") (joined
 (portRef O (instanceRef result_obuf_1))
 (portRef (member result 7))
))
(net (rename result_c_2 "result_c[2]") (joined
 (portRef O (instanceRef result_s_2))
 (portRef I (instanceRef result_obuf_2))
))
(net (rename result_2 "result[2]") (joined
 (portRef O (instanceRef result_obuf_2))
 (portRef (member result 6))
))
(net (rename result_c_3 "result_c[3]") (joined
 (portRef O (instanceRef result_s_3))
 (portRef I (instanceRef result_obuf_3))
))
(net (rename result_3 "result[3]") (joined
 (portRef O (instanceRef result_obuf_3))
 (portRef (member result 5))
))
(net (rename result_c_4 "result_c[4]") (joined
 (portRef O (instanceRef result_s_4))
 (portRef I (instanceRef result_obuf_4))
))
(net (rename result_4 "result[4]") (joined
 (portRef O (instanceRef result_obuf_4))
 (portRef (member result 4))
))
(net (rename result_c_5 "result_c[5]") (joined
 (portRef O (instanceRef result_s_5))
 (portRef I (instanceRef result_obuf_5))
))
(net (rename result_5 "result[5]") (joined
 (portRef O (instanceRef result_obuf_5))
 (portRef (member result 3))
))
(net (rename result_c_6 "result_c[6]") (joined
 (portRef O (instanceRef result_s_6))
 (portRef I (instanceRef result_obuf_6))
))
(net (rename result_6 "result[6]") (joined
 (portRef O (instanceRef result_obuf_6))
 (portRef (member result 2))
))
(net (rename result_c_7 "result_c[7]") (joined
 (portRef O (instanceRef result_s_7))
 (portRef I (instanceRef result_obuf_7))
))
(net (rename result_7 "result[7]") (joined
 (portRef O (instanceRef result_obuf_7))
 (portRef (member result 1))
))
```

```
(net (rename result_c_8 "result_c[8]") (joined
 (portRef O (instanceRef result_cry_7))
 (portRef I (instanceRef result_obuf_8))
))
(net (rename result_8 "result[8]") (joined
 (portRef O (instanceRef result_obuf_8))
 (portRef (member result 0))
))
(net (rename result_c_0 "result_c[0]") (joined
 (portRef O (instanceRef result_axb_0))
 (portRef I (instanceRef result_obuf_0))
 (portRef S (instanceRef result_cry_0))
))
(net (rename result_cryZ0Z_0 "result_cry_0") (joined
 (portRef LO (instanceRef result_cry_0))
 (portRef CI (instanceRef result_cry_1))
 (portRef CI (instanceRef result_s_1))
))
(net (rename result_axbZ0Z_1 "result_axb_1") (joined
 (portRef O (instanceRef result_axb_1))
 (portRef S (instanceRef result_cry_1))
 (portRef LI (instanceRef result_s_1))
))
(net (rename result_cryZ0Z_1 "result_cry_1") (joined
 (portRef LO (instanceRef result_cry_1))
 (portRef CI (instanceRef result_cry_2))
 (portRef CI (instanceRef result_s_2))
))
(net (rename result_axbZ0Z_2 "result_axb_2") (joined
 (portRef O (instanceRef result_axb_2))
 (portRef S (instanceRef result_cry_2))
 (portRef LI (instanceRef result_s_2))
))
(net (rename result_cryZ0Z_2 "result_cry_2") (joined
 (portRef LO (instanceRef result_cry_2))
 (portRef CI (instanceRef result_cry_3))
 (portRef CI (instanceRef result_s_3))
))
(net (rename result_axbZ0Z_3 "result_axb_3") (joined
 (portRef O (instanceRef result_axb_3))
 (portRef S (instanceRef result_cry_3))
 (portRef LI (instanceRef result_s_3))
))
(net (rename result_cryZ0Z_3 "result_cry_3") (joined
 (portRef LO (instanceRef result_cry_3))
 (portRef CI (instanceRef result_cry_4))
 (portRef CI (instanceRef result_s_4))
))
(net (rename result_axbZ0Z_4 "result_axb_4") (joined
 (portRef O (instanceRef result_axb_4))
 (portRef S (instanceRef result_cry_4))
 (portRef LI (instanceRef result_s_4))
))
(net (rename result_cryZ0Z_4 "result_cry_4") (joined
 (portRef LO (instanceRef result_cry_4))
 (portRef CI (instanceRef result_cry_5))
 (portRef CI (instanceRef result_s_5))
))
(net (rename result_axbZ0Z_5 "result_axb_5") (joined
 (portRef O (instanceRef result_axb_5))
 (portRef S (instanceRef result_cry_5))
 (portRef LI (instanceRef result_s_5))
))
(net (rename result_cryZ0Z_5 "result_cry_5") (joined
```

```
            (portRef LO (instanceRef result_cry_5))
            (portRef CI (instanceRef result_cry_6))
            (portRef CI (instanceRef result_s_6))
           ))
           (net (rename result_axbZ0Z_6 "result_axb_6") (joined
            (portRef O (instanceRef result_axb_6))
            (portRef S (instanceRef result_cry_6))
            (portRef LI (instanceRef result_s_6))
           ))
           (net (rename result_cryZ0Z_6 "result_cry_6") (joined
            (portRef LO (instanceRef result_cry_6))
            (portRef CI (instanceRef result_cry_7))
            (portRef CI (instanceRef result_s_7))
           ))
           (net (rename result_axbZ0Z_7 "result_axb_7") (joined
            (portRef O (instanceRef result_axb_7))
            (portRef S (instanceRef result_cry_7))
            (portRef LI (instanceRef result_s_7))
           ))
          )
        )
    )
  )
  (design add (cellRef add (libraryRef work))
  (property PART (string "xc4vlx15sf363-10") (owner "Xilinx")))
)
```

B.2 Levelized netlist

Listing B.2: Levelized netlist before partitioning

```
%Gate_Number  Gate_level  Gate_type Num_inputs  Input_list  Num_outputs Output_list
1 2 41  1 37  2 27  26
2 2 43  3 37  45  53  2 27  26
3 2 43  3 52  37  44  2 25  24
4 2 43  3 51  37  43  2 23  22
5 2 43  3 50  37  42  2 21  20
6 2 43  3 49  37  41  2 19  18
7 2 43  3 48  37  40  2 17  16
8 2 43  3 47  37  39  2 15  14
9 2 43  3 46  37  38  2 13  12
10  2 41  1 37  1 11
11  11  100 2 10  13  1 28
12  10  100 2 9 15  1 29
13  10  56  3 46  15  9 1 11
14  9 100 2 8 17  1 30
15  9 56  3 47  17  8 2 13  12
16  8 100 2 7 19  1 31
17  8 56  3 48  19  7 2 15  14
18  7 100 2 6 21  1 32
19  7 56  3 49  21  6 2 17  16
20  6 100 2 5 23  1 33
21  6 56  3 50  23  5 2 19  18
22  5 100 2 4 25  1 34
23  5 56  3 51  25  4 2 21  20
24  4 100 2 3 27  1 35
25  4 56  3 52  27  3 2 23  22
26  3 100 2 2 1 1 36
27  3 56  3 53  1 2 2 25  24
```

```
28   12   69   1  11   1  79
29   11   69   1  12   1  78
30   10   69   1  14   1  77
31   9  69   1  16   1  76
32   8  69   1  18   1  75
33   7  69   1  20   1  74
34   6  69   1  22   1  73
35   5  69   1  24   1  72
36   4  69   1  26   1  71
37   1  27   1  70   10   10   9  8  7  6  5  4  3  2  1
38   1  27   1  69   1  9
39   1  27   1  68   1  8
40   1  27   1  67   1  7
41   1  27   1  66   1  6
42   1  27   1  65   1  5
43   1  27   1  64   1  4
44   1  27   1  63   1  3
45   1  27   1  62   1  2
46   1  27   1  61   2  13   9
47   1  27   1  60   2  15   8
48   1  27   1  59   2  17   7
49   1  27   1  58   2  19   6
50   1  27   1  57   2  21   5
51   1  27   1  56   2  23   4
52   1  27   1  55   2  25   3
53   1  27   1  54   2  27   2
54   0  113  0  1  53
55   0  113  0  1  52
56   0  113  0  1  51
57   0  113  0  1  50
58   0  113  0  1  49
59   0  113  0  1  48
60   0  113  0  1  47
61   0  113  0  1  46
62   0  113  0  1  45
63   0  113  0  1  44
64   0  113  0  1  43
65   0  113  0  1  42
66   0  113  0  1  41
67   0  113  0  1  40
68   0  113  0  1  39
69   0  113  0  1  38
70   0  113  0  1  37
71   5  114  1  36   0
72   6  114  1  35   0
73   7  114  1  34   0
74   8  114  1  33   0
75   9  114  1  32   0
76   10   114  1  31   0
77   11   114  1  30   0
78   12   114  1  29   0
79   13   114  1  28   0
```

B.3 Generated TBB code

Listing B.3: Generated TBB code

```
#include "tbb/tick_count.h"
#include "tbb/task.h"
#include "tbb/task_group.h"
#include "tbb/task_scheduler_init.h"
#include <cstdio>
#include "default_cell_library.h"

using namespace tbb;
#define MAX_CYCLES 4000000
#define MAX_GATES 75
bool output[MAX_GATES];
static long int n;

// FUNCTION POINTERS
typedef void (*pt2Function)();

void function_0()
{
output[50] = false;
output[58] = false;
output[48] = IBUF( output[50] );
output[40] = IBUF( output[58] );
output[8] = LUT2( output[40], output[48], 6 );
output[32] = OBUF( output[8] );
output[66] = OUTPUT( output[32] );

}
//------
void function_1()
{
output[59] = false;
output[51] = false;
output[49] = false;
output[48] = IBUF( output[50] );
output[39] = IBUF( output[59] );
output[47] = IBUF( output[51] );
output[8] = LUT2( output[40], output[48], 6 );
output[1] = LUT2( output[47], output[39], 6 );
output[23] = MUXCY_L( output[48], output[49], output[8] );
output[21] = XORCY( output[1], output[23] );
output[31] = OBUF( output[21] );
output[67] = OUTPUT( output[31] );

}
//------
void function_2()
{
output[51] = false;
output[60] = false;
output[52] = false;
output[47] = IBUF( output[51] );
output[38] = IBUF( output[60] );
output[46] = IBUF( output[52] );
output[1] = LUT2( output[47], output[39], 6 );
output[2] = LUT2( output[46], output[38], 6 );
output[23] = MUXCY_L( output[48], output[49], output[8] );
output[22] = MUXCY_L( output[47], output[23], output[1] );
output[19] = XORCY( output[2], output[22] );
output[30] = OBUF( output[19] );
```

```
output[68] = OUTPUT( output[30] );

}
//------
void function_3()
{
output[52] = false;
output[61] = false;
output[53] = false;
output[46] = IBUF( output[52] );
output[37] = IBUF( output[61] );
output[45] = IBUF( output[53] );
output[2] = LUT2( output[46], output[38], 6 );
output[3] = LUT2( output[45], output[37], 6 );
output[22] = MUXCY_L( output[47], output[23], output[1] );
output[20] = MUXCY_L( output[46], output[22], output[2] );
output[17] = XORCY( output[3], output[20] );
output[29] = OBUF( output[17] );
output[69] = OUTPUT( output[29] );

}
//------
void function_4()
{
output[53] = false;
output[62] = false;
output[54] = false;
output[45] = IBUF( output[53] );
output[36] = IBUF( output[62] );
output[44] = IBUF( output[54] );
output[3] = LUT2( output[45], output[37], 6 );
output[4] = LUT2( output[44], output[36], 6 );
output[20] = MUXCY_L( output[46], output[22], output[2] );
output[18] = MUXCY_L( output[45], output[20], output[3] );
output[15] = XORCY( output[4], output[18] );
output[28] = OBUF( output[15] );
output[70] = OUTPUT( output[28] );

}
//------
void function_5()
{
output[54] = false;
output[63] = false;
output[55] = false;
output[44] = IBUF( output[54] );
output[35] = IBUF( output[63] );
output[43] = IBUF( output[55] );
output[4] = LUT2( output[44], output[36], 6 );
output[5] = LUT2( output[43], output[35], 6 );
output[18] = MUXCY_L( output[45], output[20], output[3] );
output[16] = MUXCY_L( output[44], output[18], output[4] );
output[13] = XORCY( output[5], output[16] );
output[27] = OBUF( output[13] );
output[71] = OUTPUT( output[27] );

}
//------
void function_6()
{
output[55] = false;
output[64] = false;
output[56] = false;
output[43] = IBUF( output[55] );
```

```
output[34] = IBUF( output[64] );
output[42] = IBUF( output[56] );
output[5] = LUT2( output[43], output[35], 6 );
output[6] = LUT2( output[42], output[34], 6 );
output[16] = MUXCY_L( output[44], output[18], output[4] );
output[14] = MUXCY_L( output[43], output[16], output[5] );
output[11] = XORCY( output[6], output[14] );
output[26] = OBUF( output[11] );
output[72] = OUTPUT( output[26] );

}
//------
void function_7()
{
output[56] = false;
output[65] = false;
output[57] = false;
output[42] = IBUF( output[56] );
output[33] = IBUF( output[65] );
output[41] = IBUF( output[57] );
output[6] = LUT2( output[42], output[34], 6 );
output[7] = LUT2( output[41], output[33], 6 );
output[14] = MUXCY_L( output[43], output[16], output[5] );
output[12] = MUXCY_L( output[42], output[14], output[6] );
output[9] = XORCY( output[7], output[12] );
output[25] = OBUF( output[9] );
output[73] = OUTPUT( output[25] );

}
//------
void function_8()
{
output[65] = false;
output[57] = false;
output[56] = false;
output[57] = false;
output[34] = IBUF( output[64] );
output[42] = IBUF( output[56] );
output[43] = IBUF( output[55] );
output[33] = IBUF( output[65] );
output[41] = IBUF( output[57] );
output[42] = IBUF( output[56] );
output[41] = IBUF( output[57] );
output[5] = LUT2( output[43], output[35], 6 );
output[6] = LUT2( output[42], output[34], 6 );
output[7] = LUT2( output[41], output[33], 6 );
output[16] = MUXCY_L( output[44], output[18], output[4] );
output[14] = MUXCY_L( output[43], output[16], output[5] );
output[12] = MUXCY_L( output[42], output[14], output[6] );
output[10] = MUXCY( output[41], output[12], output[7] );
output[24] = OBUF( output[10] );
output[74] = OUTPUT( output[24] );

}
//------
/* MAIN FUNCTION */
int main()
{
        n = 0;
        pt2Function function[9] = {NULL};

    // Assigning function pointers
        function[0] = &function_0;
        function[1] = &function_1;
```

```
        function[2] = &function_2;
        function[3] = &function_3;
        function[4] = &function_4;
        function[5] = &function_5;
        function[6] = &function_6;
        function[7] = &function_7;
        function[8] = &function_8;

    // Set all values to zero
        for(int j=0; j< MAX_GATES; j++)
                output[j] = false;

        task_scheduler_init init;
        task_group g;

        do{
                g.run(function[0]);
                g.run(function[1]);
                g.run(function[2]);
                g.run(function[3]);
                g.run(function[4]);
                g.run(function[5]);
                g.run(function[6]);
                g.run(function[7]);
                g.run(function[8]);
                g.wait();
                n++;
        }while(n < MAX_CYCLES);

        return 0;
}
```

Bibliography

[1] Shai Fine and Avi Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC '03: Proceedings of the 40th annual Design Automation Conference*, New York, NY, USA, 2003, pp. 286–291, ACM.

[2] A Molina and O Cadenas, "Functional verification: Approaches and challenges," pp. 65–69, 2007.

[3] PatHelland, "The Irresistible Forces Meet the Movable Objects," http://blogs.msdn.com/pathelland/attachment/6523760.ashx.

[4] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[5] OSCI, "SystemC," http://www.systemc.org/.

[6] NVIDIA Corp., "NVIDIA CUDA," http://www.nvidia.com/cuda.

[7] Gordon E. Moore, "Cramming more components onto integrated circuits," pp. 56–59, 2000.

[8] NY times, "Intelś Big Shift After Hitting Technical Wall," http://www.nytimes.com/2004/05/17/business/17intel.html.

[9] AMD, "Multicore processors - The next evolution in computing," AMD white paper.

[10] Intel Corporation, "Intel Multi-Core Processors: Making the Move to Quad-Core and Beyond," http://www.intel.com/technology/architecture/downloads/quad-core-06.pdf.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick, "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[12] Mahesh Nanjundappa, Hiren D. Patel, Bijoy A. Jose, and Sandeep K. Shukla, "SCGPSim: A fast SystemC simulator on GPUs," in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*. January 2010, IEEE, Taipei, Taiwan - **Best Paper Award.**

[13] OSCI, "SystemC Language Reference Manual," http://www.systemc.org/downloads/standards/.

[14] Gerd Meister, "A survey on parallel logic simulation," Tech. Rep., University of Saarland, Department of Computer Science, Misra J, 1993.

[15] James Reinders, *Intel threading building blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[16] TBB, "Intel Threading Building Blocks," http://www.threadingbuildingblocks.org/.

[17] Nick Savoiu, Sandeep K. Shukla, and Rajesh K. Gupta, ""design for synthesis, transform for simulation: Automatic transformation of threading structures in high-level system models"," *UC Irvine Technical Report*, 2001.

[18] Linda Kaouane, Dominique Houzet, and Sylvain Huet, "SysCellC: SystemC on Cell," *Computational Science and its Applications, International Conference*, vol. 0, pp. 234–244, 2008.

[19] Kalyan S. Perumalla, "Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)," in *PADS '06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, Washington, DC, USA, 2006, pp. 74–81, IEEE Computer Society.

[20] Youssef N. Naguib and Rafik S. Guindi, "Speeding up systemc simulation through process splitting.," in *DATE*, Rudy Lauwereins and Jan Madsen, Eds. 2007, pp. 111–116, ACM.

[21] Tun Li, Yang Guo, and Si-Kun Li, "Design and implementation of a parallel verilog simulator: Pvsim," *VLSI Design, International Conference on*, vol. 0, pp. 329, 2004.

[22] Lijun Li, Hai Huang, and Carl Tropper, "Dvs: An object-oriented framework for distributed verilog simulation," in *PADS '03: Proceedings of the seventeenth workshop on Parallel and distributed simulation*, Washington, DC, USA, 2003, p. 173, IEEE Computer Society.

[23] Stephen Williams, "Icarus Verilog," http://www.icarus.com/eda/verilog/.

[24] Zhicheng Wang and Peter M. Maurer, "Lecsim: a levelized event driven compiled logic simulation," in *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, New York, NY, USA, 1990, pp. 491–496, ACM.

[25] L. Zhu, G. Chen, B.K. Szymanski, C. Tropper, and T. Zhang, "Parallel logic simulation of million-gate vlsi circuits," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, Sept. 2005, pp. 521–524.

[26] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco, "Event-driven gate-level simulation with gp-gpus," in *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, New York, NY, USA, 2009, pp. 557–562, ACM.

[27] Seiler et. al, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 1–15, 2008.

[28] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach, "Accelerating compute-intensive applications with gpus and fpgas," *Application Specific Processors, Symposium on*, vol. 0, pp. 101–107, 2008.

[29] Researchers, "CUDA Research," http://www.nvidia.com/object/cuda_home_new.html.

[30] Electronic Industries Association, "Electronic design interchange format version 2 0 0," 1988.

[31] David Berner, Jean-Pierre Talpin, Hiren D. Patel, Deepak Mathaikutty, and Sandeep K. Shukla, "Systemcxml: An exstensible systemc front end using xml.," in *FDL*. 2005, pp. 405–409, ECSI.

[32] Shekhar A. Sharad and Sandeep Kumar Shukla, *Optimizing system models for simulation efficiency*, Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[33] Sumit Ahuja, Deepak A. Mathaikutty, Sandeep Shukla, and Ajit Dingankar, "Assertion-based modal power estimation," *8th International workshop on Microprocessor test and Verification (MTV)*, pp. 3–7, 2007.

[34] Sumit Ahuja, Deepak A. Mathaikutty, and Sandeep Shukla, "Applying verification collaterals for accurate power estimation," *9th International workshop on Microprocessor test and Verification (MTV)*, pp. 61–66, 2008.

[35] Sumit Ahuja, Deepak A. Mathaikutty, Avinash Lakshminarayana, and Sandeep Shukla, "Statistical regression based power models for co-processors for faster and accurate power estimation," *22nd IEEE International SOC Conference*, pp. 399–402, 2009.

[36] Sumit Ahuja, Deepak A. Mathaikutty, Gaurav Singh, Joe Stetzer, Sandeep Shukla, and Ajit Dingankar, "Power estimation methodology for a high-level synthesis framework," *10th International Symposium on Quality Electronics Design (ISQED)*, pp. 541–546, 2009.

[37] Sumit Ahuja, Deepak A. Mathaikutty, Avinash Lakshminarayana, and Sandeep K. Shukla, "Scope: Statistical regression based power models for co-processors power estimation," .

[38] Sumit Ahuja, Wei Zhang, and Sandeep K. Shukla, "A Methodology for Power Aware High-Level Synthesis of Co-Processors from Software Algorithms," .