GPU BASED ACCELERATION OF SYSTEMC AND TRANSACTION LEVEL MODELS FOR MPSOC SIMULATION

A Thesis

by

ROHIT GANGRADE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Jiang Hu
Committee Members,	Paul Gratz
	Rabi Mahapatra
Head of Department,	Miroslav M. Begovic

May 2016

Major Subject: Computer Engineering

Copyright 2016 Rohit Gangrade

ABSTRACT

With increasing number of cores on a chip, the complexity of modeling hardware using virtual prototype is increasing rapidly. Typical SOCs today have multiprocessors connected through a bus or NOC architecture which can be modeled using SystemC framework. SystemC is a popular language used for early design exploration and performance analysis of complex embedded systems. TLM2.0, an extension of SystemC, is increasingly used in MPSOC designs for simulating loosely and approximately timed transaction level models. The OSCI reference kernel which implements SystemC library runs on a single thread, slowing up the simulation speed to a large extent. Previous works have used the computational power of multi-core systems and GPUs which can run multiple threads simultaneously, speeding up the simulation. Multi-core simulations are not as effective in cases where thread runtime is low, because synchronization overhead becomes comparable to thread runtime. Modern GPUs can run thousands of threads at a time and have shown good results for synthesizable designs in recent efforts. However, development in these works are limited to synthesizable subsets of SystemC models, not supporting timed events for process communication. In this research work, a methodology is proposed for accelerating timed event based SystemC TLM2.0 model to GPU based kernel, which maps SystemC processes to CUDA threads in GPU, providing high data level parallelism. This work aims to provide a scalable solution for simulating large MPSOC designs, facilitating early design exploration and performance analysis. Experiments have shown that the proposed technique provides a speed-up of the order of 100x for typical MPSOC designs.

DEDICATION

To my parents

ACKNOWLEDGEMENTS

I would take this opportunity to thank my thesis advisor Dr. Jiang Hu for always motivating me and providing support whenever I faced difficulty. I would also like to thank Dr. Rabi Mahapatra and Dr. Paul Gratz for serving on my committee and providing useful feedback on my work.

I would like to thank all my friends who have made last two years of my life memorable. I am also grateful to the faculty at Texas A&M University who helped my research through different courses which I took over past two years. Last but not the least, I would like to thank my family for providing constant support and motivation throughout my work.

NOMENCLATURE

ESL	Electronic System Level
MPSOC	Multi Processor System on Chip
TLM	Transaction Level Modelling
GPU	Graphics Processing Unit
ISA	Instruction Set Architecture
AMBA	Advanced Microcontroller Bus Architecture
OSCI	Open SystemC Initiative
CUDA	Compute Unified Device Architecture
SIMD	Single Instruction Multiple Data
DMI	Direct Memory Interface

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGEMENTS iv

NOMENCLATURE v

TABLE OF CONTENTS vi

LIST OF FIGURES ix

LIST OF TABLES xii

1. INTRODUCTION 1

   1.1 Brief History 1
   1.2 Overview 5
   1.3 Motivation 8

2. PREVIOUS WORK 11

   2.1 Multiprocessor Work 12
   2.2 GPU Work 13

3. BACKGROUND 14

   3.1 SystemC 14
      3.1.1 Hardware Constructs 15
   3.2 TLM2.0 21
      3.2.1 Components 21
      3.2.2 Interfaces 22
      3.2.3 Models 24
   3.3 GPU 26
      3.3.1 Overview 26
      3.3.2 Architecture 27
      3.3.3 Programming Model 27
      3.3.4 Memory Hierarchy 29

3.4 Project Description 32
3.4.1 Approach/Methods 34

4. SYSTEMC/TLM2.0 BASED SIMULATOR 38

4.1 Microprocessor Design 39
4.1.1 Register File 40
4.1.2 ALU Design 41
4.1.3 Single Cycle Design 41
4.1.4 Pipelined Design 41
4.1.5 Forwarding Unit 43
4.1.6 Hazard Detection 43
4.2 Bus Modeling 43
4.2.1 Usage of Transaction Level Modeling 44
4.3 Memory 44
4.3.1 Data Memory 44
4.3.2 Instruction Memory 45
4.3.3 Cache Memory 45
4.4 Running the Simulator 45
4.5 Multi-core Execution Kernel 46

5. GPU BASED APPROACH FOR SYSTEM LEVEL SIMULATION 49

5.1 Overview 49
5.2 GPU Execution Flow 49
5.3 SystemC to CUDA Translation 53
5.3.1 Types of Events 53
5.3.2 Types of Variables 54
5.3.3 Algorithm 55
5.4 Different Code Structure Optimization 59
5.5 Shared Memory Optimization 59
5.6 GPU Execution of Non-Synthesizable Model 60
5.7 Support for Transaction Level Modeling 62
5.7.1 Blocking Interface 63
5.7.2 Non Blocking Interface 64
5.7.3 Direct Memory Interface 65
5.7.4 Debug Interface 66

6. TIME DECOUPLED GPU SIMULATION 67

6.1 Problems with Single SMP Framework 67
6.2 Trade-off in using Multiple SMP 68
6.3 Time Decoupled Parallel GPU Simulation 70

7. EXPERIMENTAL RESULTS 71
7.1 Experimental Setup 71
7.2 MPSOC Simulator 71
7.3 Synthetic Benchmarks for GPU 73
7.3.1 Memory Optimization 73
7.4 Typical SystemC Benchmarks 74
7.4.1 Packet Size Variation 75
7.4.2 Number of Instances Variation 77
7.4.3 Synchronous vs Asynchronous 78
7.4.4 Scalability 79
7.5 Multi-Initiator Target 80
7.6 Network on Chip 82
7.7 Time Decoupled Simulation 84
8. CONCLUSIONS AND FUTURE WORK 87
REFERENCES 88

LIST OF FIGURES

FIGURE | Page

1.1 System level design in the flow for IC Design . . . 3
1.2 ESL design cycle . . . 4
1.3 Order of execution in single core and multicore hosts . . . 8
1.4 Typical NOC based MPSOC schematic . . . 9
1.5 Simulation of a design using single core, multicore and GPU host . . . 10
3.1 Architecture of SystemC language . . . 15
3.2 SystemC execution kernel . . . 19
3.3 SystemC execution flow . . . 20
3.4 SystemC execution example . . . 20
3.5 Initiator, interconnect and target communicating through TLM sockets 21
3.6 TLM loosely timed model . . . 25
3.7 TLM approximately timed model . . . 25
3.8 Architecture of GPU . . . 28
3.9 Programming model of CUDA . . . 30
3.10 Memory hierarchy in GPU . . . 30
4.1 Schematic of simulation connected to a jpeg encoder . . . 40
4.2 Schematic of MIPS based microprocessor designed in SystemC . . . 42
4.3 Working of simulator with an external application . . . 47
4.4 Different types of scheduling in multiprocessor host . . . 48

5.1 SystemC to CUDA conversion 50
5.2 Execution of SystemC kernel in GPU 51
5.3 Timed event based communication in SystemC 54
5.4 Event mapping to GPU 55
5.5 Code mapping to multiple warps in GPU 59
5.6 Shared memory conflict in sensitive variables across multiple warps . 60
5.7 Execution mechanism of timed event communication in GPU using shared and register memory 61
5.8 TLM transformation of blocking interface 64
5.9 TLM transformation of non-blocking interface 65
6.1 Time decoupled simulation 69
6.2 Partitioning of a design for time decoupled simulation 70
7.1 List of benchmarks 74
7.2 Schematic of benchmark with different variations 76
7.3 GPU latency and speed-up variation with size of packet 77
7.4 GPU latency and speed-up variation with increasing number of instances 78
7.5 GPU latency and speed-up variation with increasing number of instances for asynchronous design 79
7.6 Runtime variation with number of instances across benchmarks . . . 80
7.7 GPU latency and speed-up variation with increasing number of initiators for multi-initiator target 81
7.8 Multi-initiator target simulation model for benchmark 82
7.9 Schematic of NOC Router 83
7.10 GPU latency and speed-up variation with increasing size of NOC . . 83
7.11 Handshake benchmark for time decoupled simulation 85

7.12 Time decoupled parallel GPU simulation for handshake benchmark . 85
7.13 Time decoupling partition for NOC 86
7.14 Time decoupled parallel GPU simulation for NOC 86

LIST OF TABLES

TABLE Page

4.1 Simulation speed comparison of this simulator with other known simulators . . . 39

7.1 Variation of latency(in sec) with different types of bus modeling in a host with 4 cores . . . 71

7.2 Variation of latency(in sec) with increasing number of cores for encoding 128X128 image . . . 72

7.3 Variation of latency(in sec) across different types of scheduling in a host with 4 cores . . . 72

7.4 GPU_Sh is the scgp[24] implementation of benchmarks which sensitive variables are mapped to shared memory and GPU_Reg is our approach where sensitive variables are mapped to register memory. Simulation time(in ms) comparison across benchmarks . . . 74

7.5 Simulation time(in s) of benchmarks with increasing packet size . . . 77

7.6 Simulation time(in s) of benchmarks with increasing number of instances 78

7.7 Simulation time(in s) of benchmarks with increasing number of instances for asynchronous design . . . 79

7.8 Simulation time(in ms) of multi-initiator socket with increasing number of initiators across different platforms . . . 81

7.9 Simulation time(in ms) of NOC with increasing size across different platforms . . . 82

1. INTRODUCTION

1.1 Brief History

The growth of the semiconductor industry in a past few decades has been tremendous. Any system can be categorized into two parts namely hardware and software, which collaborate to perform a set of tasks desired by the end user. In the early years, majority of the research effort was spent on trying to develop a flow to deliver an IC in a given period of time. Performance and power are two aspects which are critical to any electronic system performing a set of functions. Lots of effort has thereby spent on optimizing the design to achieve good performance with low power.

Transistor is the basic element of any Integrated Circuit(IC), which combines in large numbers to form a hardware with a particular specification. Some of the hardware components which are most commonly used in ICs are microprocessors, memories, PLL, PMIC, and amplifiers. Software is a layer which is built upon hardware and is comprised of a series of tasks which can be performed by a microprocessor. Software is like a driver of an integrated circuit guiding it to perform tasks in an order desired by the user. In order to utilize the best of any system, it is essential that both hardware and software components are utilized to their maximum.

In the industry, hardware and software are typically handled by different teams working in parallel with timely interaction in order to ensure that the specifications are met. First phase of the project is of the architecture team to lay down the specifications of the design and decide what functionalities are to be supported. The hardware team then uses a Register Transfer Level(RTL) modeling language such as Verilog or VHDL to realise the behaviour of hardware. It is simulated multiple times with all the possible test cases to ensure that the specifications are met with no

bugs in the design. Verified design is synthesized into a gate level netlist with actual gates and cells. Final step is to place and route the design efficiently to achieve the functionality with minimum area.

The software team in parallel starts the development of embedded software which can support basic functionalities such as booting, memory operations, etc. Since, no actual hardware is present initially, verification of software involves running the embedded software on hardware emulators which is generally very slow. System level models can also be used to create a virtual prototype which can emulate hardware in a software environment.

The very first step involving key architectural decisions in the design flow as shown in figure 1.1 is very important part of the whole process. It is in this phase that the decision has to be taken to make sure that the system specifications can be met in a given time. Another important aspect is the optimization of performance and power in this stage. It is only after this stage, specifications are finalized and hardware and software teams can begin their work. It is therefore crucial to make sure that this step is done as soon as possible in order to meet the time to market requirements of a chip. With the technology growing fast, there is tremendous pressure on teams to deliver the chip to customers on time. Any delay in development cycle of chip usually results in a huge loss to the company because the performance grows very quickly with time as predicted by Moore's Law in 1965. It is also necessary to make sure that the system functions correctly under specified specifications for performance and power. This is where system level design, popularly known as Electronic System Level(ESL), comes into picture.

ESL design provides the highest level of abstraction present in the IC design cycle. It provides a platform for hardware software co-design at different abstraction levels. In order to meet the time to market for a chip, it is essential that we increase the level

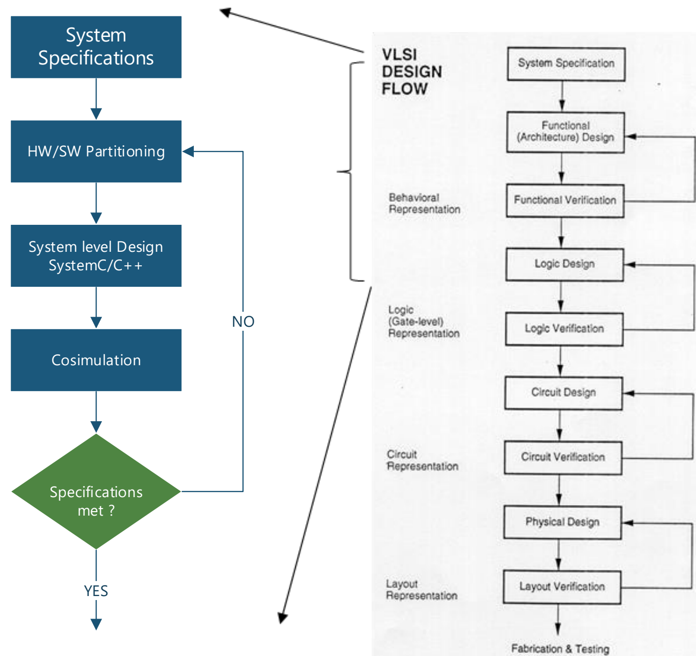


Figure 1.1: System level design in the flow for IC Design

of abstraction to handle complex designs. ESL design deals with functional models of different IPs removing the complex internal details at transistor, gate and RTL level. It is also important to be able to reuse the models of same IP in different ICs. Some of the common components such as microprocessor, memory and system bus are used repeatedly in many SOC designs. Creating common models for these components to be reused would significantly increase the productivity. Most importantly, it is essential that we perform a design exploration at an early stage in IC design cycle to meet performance and power requirements. Also, the power and performance targets should be met when all the components of SOC are simulated together. The platform should be such that it is able to model hardware and software simultaneously and simulate the concurrency present in a real hardware. Virtual Prototyping is one aspect of ESL design which allows users to model hardware in a system level design

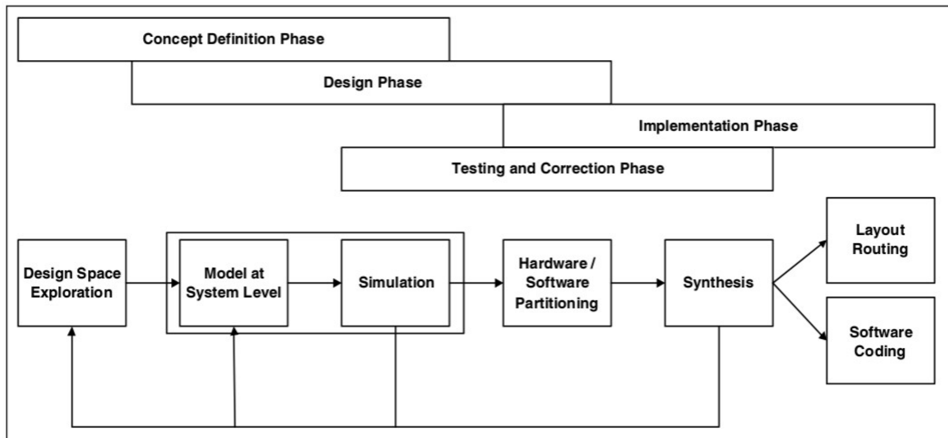


Figure 1.2: ESL design cycle

language, and run it with real applications to evaluate performance and power of final application.

After the functional specifications of a system are finalized, the next step in the process is to design the architecture. The algorithm which is to be implemented is partitioned into hardware and software. With the help of ESL tools, we can perform an analysis of system with both hardware and software components running simultaneously and capture the power and performance of the system. If the requirements are not met, then the process is iterated with different mapping of hardware and software components. Another solution would be to tweak the parameters of different individual components, for example, using a faster memory model or increasing the speed of microprocessor. This step has to be carefully performed by complete analysis as it will define the architecture of the system which cannot be changed afterwards. Also every improvement comes at a cost of something, for example increasing processor speed might also increase the power of design. The parameters of components and hardware software partitioning should be such that it meets the targets which are set by the customer.

There are many tools which are developed by different vendors for ESL design and are available commercially. Although each of these have their own benefits, the scope of this research work is limited to available open source languages for system level design. Specifically, this work aims at developing a faster simulation model for SystemC and TLM2.0 models with the help of GPU. SystemC is a library of C++ classes which can be extended to simulate hardware. One of the main characteristics of SystemC is that it is able to realise the concurrency present in the hardware simulation. Unlike software, hardware components are run in parallel in reality but we simulate it on a software platform which is usually single core and runs only one process at a time. In order to implement a hardware functionality, SystemC therefore imposes a rule on designer that the function implemented should not depend upon the order in which processes execute.

1.2 Overview

Virtual Prototyping has become an essential flow in the design and development of complex embedded applications today. Typically done using SystemC, it models hardware by discrete event simulation in software. Among the numerous advantages that it offers, it allows designers to validate the design, analyze the performance and power of the system under various configurations, facilitates development of software and also plays a key role in partitioning of a system on hardware and software. Architectural decisions are taken at an early phase in design cycle and it is often too costly to make changes at a later stage. Virtual Prototyping using SystemC enables early decision making with a reasonable accuracy and can reduce the time to market for a chip.

This research work tries to analyze the problems associated with the system level design for the ICs being manufactured today. SystemC based approach for

virtual prototyping is a decade old and used extensively by most of the chip designers to analyze power and performance at an early stage. With the new applications constantly demanding high performance and low power, ICs are getting bigger and more complex. As a result, ESL design requires modeling all components of a complex system and simulating them at an early stage. While, the SystemC based approach was a viable solution a decade ago, it needs to be revised to match up with the size and complexity of SOC today. With the saturation of technology scaling and transistor sizing, it is not possible to increase the clock frequency beyond a certain limit. People are therefore moving towards multi-core applications. Embedded systems today have have different processors dedicated for separate tasks to share the workload and still match up the performance. Smartphones, laptops and tablets all have a SOC comprising of multiple cores and this trend is likely to increase in future with people proposing a network of hundreds of cores interconnected through multiple routers. Simulating such a huge system and predicting the performance and power is likely to be a big challenge in future. It is therefore necessary that we look for alternate means for ESL simulation. This work explores some of drawbacks associated with the current platform of system level MPSOC design in SystemC and provides a scalable solution which is more viable to be used in future.

With the growing complexity of today's SOC design, simulating a complete system on software is becoming a challenging task. In order to increase the productivity, people use available reference models of common components like processors, memory, cache and bus. Any new hardware specific for a task can be modeled and plugged into the simulator comprising of all the basic components. A good simulator should keep track of performance counters such as hits, misses, latency and other metrics which can be useful to a designer. This also incurs a simulation latency overhead, slowing it down and so only important metrics can be turned on to match

up simulator's performance.

In order to keep developing SOC with higher productivity at lower costs, it is important to increase the abstraction level even above the system level design with SystemC. Every embedded system design will have multiple modules communicating through an interface. Transaction level modeling increases the abstraction level by providing a better predefined way of communication. It reduces time to model a system and also reduces the probability of error within these interfaces. TLM2.0 is an extension to SystemC which provides a seamless method of communicating between modules. SystemC modeling consists of a number of processes which define the behaviour of the system. TLM2.0 provides an interface to communicate between these modules in form of transactions.

Since most of the system level design work circled around SystemC and TLM, it is therefore important that we keep developing its execution model as we evolve. About a decade old, there has not been a significant change in the execution model of SystemC. The reference SystemC kernel[2] runs as a single threaded application serializing the execution of SystemC model. In order to realise the hardware and its concurrency, each simulation time is considered like a delta cycle in which all the functional components execute. The execution of SystemC is like discrete event simulation model. At any instant of time, all the active processes are evaluated and ran in a random order. If they give rise to new events, then all the processes active to those events are run until they are exhausted. This marks the end of simulation time. It is divided into phases namely evaluate, update and notify. Each phase has a lot of potential for parallel execution, which is not utilized by the reference SystemC kernel. The obvious solution would be to use the host with multiple cores and execute each phase in a parallel fashion[32][33][41][40][38]. Special care has to be taken while distributing the tasks to multiple processors as it should not affect

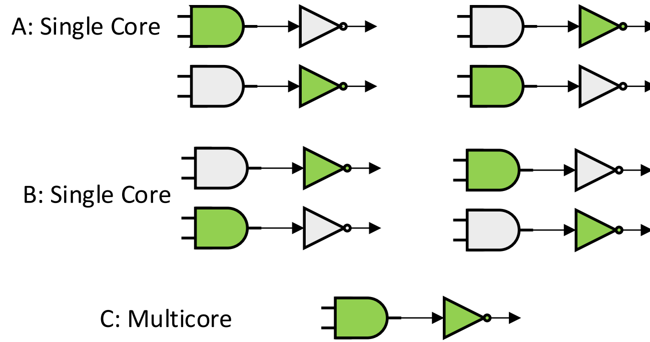


Figure 1.3: Order of execution in single core and multicore hosts

the order in which the events are executed. Since the result of simulation does not depend upon the order in which the processes are executed, it is logical to try to run them in parallel. For example, as shown in figure 1.3, all execution order of processes give the same results in single core as well as multicore hosts. People have tried this previously and shown good results, but it is still not integrated in the open source SystemC source code because of the complications it can produce.

In this research work, we have identified the challenges associated with the reference execution model of SystemC and tried to propose other solutions in multi-core domain to increase the scalability for future generations. In particular, we have focused upon the GPU based approach to simulate any SystemC design with timed event communication models and transaction level models by taking the advantage of data level parallelism.

1.3 Motivation

Even though use of multi-core hosts for Parallel Discrete Event Simulation can improve the simulation speed, they are not effective when process run time in a single

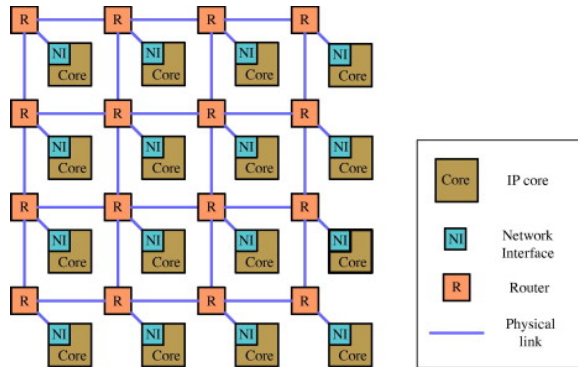


Figure 1.4: Typical NOC based MPSOC schematic

delta cycle is low, because the synchronization overhead becomes comparable to process runtime. Moreover, there are only a limited number of cores available to the end user. So, if the design is large, the number of available cores can be a bottleneck for parallel simulation. If we analyze some of the typically used MPSOC designs in industry, it can be noted that there is a repetition of same modules multiple times in the form of multiple instances(for example multiple instances of homogeneous cores or routers in NOC topology as shown in figure 1.4). In such case, a multi-core host will not be able to take advantage of this property and will have the same synchronization overhead as that of a model where design of each module is different, which is high. In order to tackle this problem, a GPU based approach is explored for system level simulation of MPSOC designs because they execute as single instruction multiple data(SIMD). This solution is ideal because we need to perform multiple simulations of same design but with different parameters.

Exploring the computational power of GPU[39][36], it reduces the synchronization overhead to some extent and has a large number of cores which can be used for simulation of multiple processes in parallel. GPU provides a high degree of data level parallelism which cannot be realized efficiently on multi-core hosts. This approach

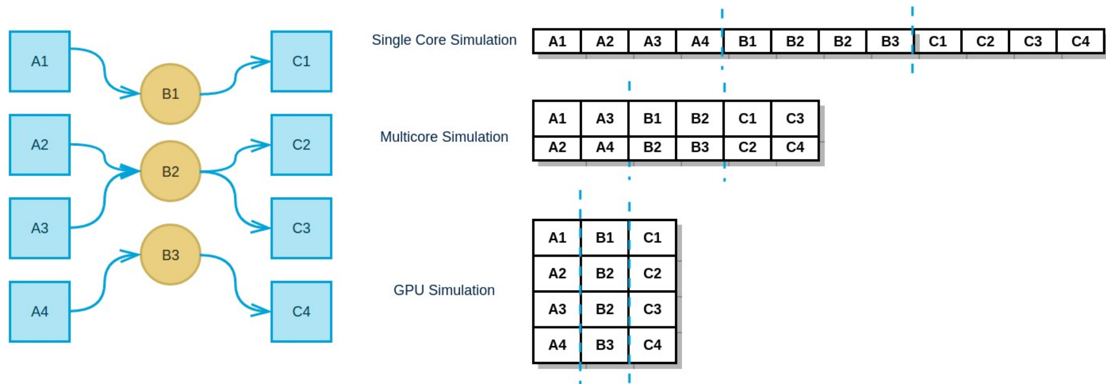


Figure 1.5: Simulation of a design using single core, multicore and GPU host

would be ideal for those case where we have less scope for thread parallelism and more scope for data parallelism, such as a simulation of a network on chip architecture which contains hundreds of routers and cores each having same code to run but on a different data set. In this research work, a novel technique is proposed to parallelize non-synthesizable and transaction level models completely on GPUs achieving maximum parallelism in all the phases of simulation with low synchronization overhead. This technique explores the efficient usage of the available resources in a GPU machine for any SystemC model such as distribution of processes to different thread blocks(multiprocessors), efficient usage of register, shared and global memory and reduction techniques to reduce the runtime in update and notify phases.

2. PREVIOUS WORK

System level design using SystemC and TLM2.0 is a crucial phase in overall time to market of a chip and due to its importance, there has been ongoing research to find different ways to accelerate the simulation. These efforts can be divided into two parts, one where emphasis is paid on the coding style to improve the performance which is referred as front end techniques. Other would be to try to improve the simulation kernel itself which is used as a base for all the designs, also known as back-end techniques. Front end techniques explore the modeling of a general MPSOC environment consisting of multiple processors, memory and other blocks interconnected through bus or network structure. Back-end techniques focus on improving the kernel which is used for modeling hardware in a software environment.

Front end techniques deal with the design of a simulator using a specific modeling language like SystemC. There are many simulators, which used SystemC environment for modeling MPSOC comprising of homogeneous and heterogeneous multiprocessors, AMBA buses for modeling communication structure, memory and other peripheral devices. MPARM[4] is one of these simulators which has been shown to be effective for early architectural exploration. Many other efforts are focussed upon building a MPSOC environment for system level simulations[18][11][37][27]. Some of these[6] are capable of running an entire operating system as well but are really slow in their execution due to implementation complexity. Some of the works focus on the application based system level design such as modeling a system level environment specifically for JPEG, MPEG and H264 decoders[15], others focus specifically on network on chip topology[35][22][16][14][3] as it is gaining a lot of popularity these days. People have also explored design of processors with different ISA using SystemC[29].

2.1 Multiprocessor Work

For back-end techniques, people have mainly worked on utilizing the high computing power of Multiprocessor CPUs and GPUs as the nature of system level design using SystemC can exploit a lot of parallelism. Early research efforts [32] tried to map the evaluation phase of OSCI kernel to multiprocessors at a cost of synchronization overhead. In order to reduce this costly synchronization, others [33] have tried to divide the simulation into different parts, and each part is mapped to certain processes reducing the need for synchronization. Even though both the above efforts produced good results in some cases, they do not utilize the multi-core exploration on other phases of OSCI kernel such as evaluate and update. The discrete event simulation in SystemC is split-up into several delta cycles and each delta cycles have evaluate update and notify phase, and there can be multiple delta cycles for a given time. This makes the improvement in the evaluation phase due to parallelism only a small part of the overall time consumed and moreover it incurs costly synchronization overhead every delta cycle. In order to tackle this problem, people [41] have tried to partition the simulation and map them onto different threads even before the start of simulation and run these threads independently on individual processors. A lookahead time is calculated statically which is the minimum time required for communication for processes from one thread to another thread. The authors call this method parallel time decoupled simulation allowing processes to run independently without synchronization on different processors until a minimum lookahead time. Improving upon the same concept, in [40], authors have proposed a flexible time decoupling based method to even reduce the synchronization overhead but at a cost of lower timing accuracy and determinism, which can be useful if the aim of high level design is only to get a rough estimate of performance. In [38], authors

have designed a highly special purpose parallel simulator which can run the SystemC design on multi-core cost and uses a kernel accelerator which reduces synchronization by reducing the control and increasing the scalability of simulation.

2.2 GPU Work

With growing number of cores on a chip, there is a demand for scalable solution to simulation of large SystemC models. Typically some of the MPSOC models contains hundreds of cores with same code structure executing on different dataset. This points us in the direction of using a GPU based approach which provides a scalable solution to problems with high data level parallelism. Previous works [24][39][36] have already attempted to use GPU for accelerating SystemC models. The first work [24] translates the original SystemC code to a CUDA using a synthesizer by mapping each process to different core. Although it was a good start, it had issues when the process mapped on same multiprocessor shared different code structure, causing lots of branch divergence. This was controlled in the next work [39] by replicating the common process to different multiprocessors. These works [24][39] are restricted only to synthesizable SystemC subsets and do not support timed event-based models which are crucial part of higher level modeling in SystemC. They also do not support transaction level models which are widely used in industry for modeling loosely timed and approximately timed models. In [36], authors have addressed this problem by mapping selected processes to GPU in the evaluation phase of reference SystemC kernel, but their approach require synchronization between CPU-GPU every delta cycle, which is costly. Moreover, they exploit parallelism only in the evaluation phase thereby serializing the update and notify phase.

3. BACKGROUND

This chapter covers some relevant background which is needed in order to understand the optimizations used in this research work. The main focus of this work is to create a simulation model for system level design which is good enough for the future MPSOCs. Since SystemC is widely used in industry as an open source language for system level design, it is important to understand its basic concepts before going into further optimizations. First part of this chapter gives an in depth understanding of SystemC and its execution model. Next, we try to give some insight into the concepts related to transaction level modeling and its use in system level design. Its programming model is briefly described for user to get an intuitive idea of TLM.

The platform provided by above languages is based on C++ libraries and its execution model based on single core is a huge bottleneck for scalable designs. Therefore, some of the techniques are discussed to speed up the simulation by using multi-core hosts and GPU. Conversion of a sequential code into a parallel code is a challenging task because it involves finding scope to run multiple threads simultaneously and still achieve same functionality. Since we would be using GPU extensively, its architecture is described in depth here. Finally, a brief introduction to the approach followed in this work is given which is described in depth in subsequent chapters.

3.1 SystemC

Until now we have given a brief introduction on the roadmap of using SystemC for ESL design. In this section, we will discuss some concepts related to the execution model of SystemC and usage of its constructs. SystemC as mentioned before is an extension of C++ libraries which are used to realise hardware on software. Its has its own execution kernel which executes these models as if simulating a real

User libraries | SystemC Verification library | Other IP
Predefined Primitive Channels: Mutexs, FIFOs, & Signals
SystemC
Simulation Kernel
Threads & Methods
Channels & Interfaces
Events, Sensitivity & Notifications
Modules & Hierarchy
Data types: Logic, Integers, Fixed point
C++ | STL

Figure 3.1: Architecture of SystemC language

hardware by keeping track of time. Most of the system level design work touches both hardware and software due to the close correlation between two at an early stage. So, designing a hardware in SystemC can provide a good platform for early development of software.

3.1.1 Hardware Constructs

The main advantage of SystemC is that while still developing in an environment compatible with full fledged software, we are able to realise hardware and simulate it efficiently with precise timing notations. A software basically executes a set of instructions using a microprocessor and so runs sequentially. An actual hardware is composed of many transistors and gates running in parallel to achieve a functionality. In order to model that in software, SystemC runs on a kernel which takes care of the inherent concurrency present in hardware simulation.

3.1.1.1 Time

Time is the most important concept which is needed to model hardware using SystemC. There are multiple definitions associated with time. The time which is required by processor to run the code is called processor time, which determines the

efficiency of simulation. If this is the only thing running on machine, then it can also be referred as wall clock time. Simulation time is the overall time for which the hardware is simulated. It implies that when the hardware is fabricated to silicon, it will take that much amount of time to run the desired application.

With the notion of modeling, we are mainly concerned with simulation time. At any point in simulation, the kernel keeps track of the current simulation time and executes all the runnable processes before advancing the time. The class which implements time is known as **sc_time** with resolutions ranging from second to femtosecond. Any hardware component has a certain delay which is referred to its latency. If there is zero delay associated with any model, then it the model is not realistic. A simple construct to realise delay would be to use *wait*. Some of the design have delay in the form of clock cycles, so **sc_clock** class is implemented to achieve that functionality.

3.1.1.2 Hierarchy and Modules

A design of SOC can be quite complex with lots of functionalities. In order to understand and organize them properly, a hierarchical structure is followed throughout the design. A group of processes and ports can be wrapped inside a module, with the instantiation of *SC_MODULE*. This is identical to the structure used in verilog and vhdl for modeling hardware. Since the basic language is still C++, there are predefined macros for defining modules and registering them with processes. A typical module should have input-output ports and methods which define the relation between input and outputs. These methods are linked with corresponding module by defining them in the class definition of module with macro *SC_METHOD* or *SC_THREAD*. Processes should be defined in the constructor function, which is defined using the macro *SC_CTOR* inside a *SC_MODULE*.

3.1.1.3 Data-types

SystemC provides a wide range of data-types for modelling logic signals[7]. Apart from the native C++ counterparts which can be used any-time, SystemC provides classes of data-types with vectors of variable width which could also handle simulation of 'X' and 'Z' signals. It is important that we select the data type which is closest to native C++ implementation in order to get good simulation performance.

3.1.1.4 Communication

Communication is handled in SystemC in two different ways. Inside a module, communication between processes is done through events, their sensitivity and notification. Communication between two different modules is handled by use of a channels which connects two ports.

Firstly, discussing the communication between two modules, it is necessary that we have a channel which derives its member functions from an interface. An interface is a component which is an abstract class and uses the property of polymorphism. It contains virtual methods, i.e. only definition of functions are specified in an interface. A channel and a port should extend from an interface and define the internal functioning of the particular method. The abstract nature of interface allows us to use the same definition for multiple channels implementing different communication details.

Another way of communication is the use of events and their sensitivity. Each process is sensitive to a certain set of events which is declared statically in the registration of process in constructor. Whenever each of these events occur, the process is made runnable and is executed in the current delta cycle. Each process can notify some other events which can invoke the processes sensitive to them. This way, we can switch from one method to another with the help of events.

3.1.1.5 Concurrency

Concurrency is a phenomenon which is very evident in hardware because these units execute in parallel. In order to realise the same behaviour in software, we need to stop the simulation time and execute each runnable process one by one. Consistency throughout the timeline can be maintained by executing all runnable processes at any given instant before advancing the time. The mechanics behind the working of simulation is handled by a kernel which keeps track of all the events occurring in the design and acts accordingly.

The simulation is divided into two phases namely *elaboration* and *execution*. In *elaboration* stage, the simulator reads all the module definitions and creates a hierarchical structure of the hardware. It binds all the processes, ports and signals to their corresponding modules and registers the communication between modules through channels and events. After completing *elaboration*, kernel starts *execution* stage where the actual processing of hardware takes place. The way SystemC simulation runs is popularly known as cooperative multitasking where the kernel gives control to process and waits to it to yield. In order to realise concurrency, there is frequent switching between kernel and processes because the process yield control to kernel when it has finished the work to be done at a particular instant of time. In a real hardware there may be delay associated with each hardware component, whereas in a simulation we need to model delay for a particular task. In general we can use *wait()* and *notify()* statements to generate a delay in simulation time. Both the commands use argument of time to predict when the process will be made runnable next time.

SystemC reference model as shown in figure 3.3, widely used in industry for system level modeling, is based on the concept of discrete event simulation. The

execution phase of simulation is divided into three phases namely - *evaluate*, *update* and *notify*. The execution starts when there is call to **sc_start()**. The time line is divided into a series of discrete set of times and each time instant is broken down into multiple delta cycles. Each delta cycle proceeds through all the phases until there are no events left to process after which simulation advances to next closest time of an active event. In *evaluate* phase, all the runnable methods are queued sequentially and executed in a random yet deterministic order. After their execution, all the signals and channels are updated to their latest values and generated events are updated with their trigger time. In *notify* phase, runnable processes are identified and pushed to runnable queue to execute in next delta cycle. This processes is repeated until there is a call to **sc_stop**. Figure 3.4 shows an example of SystemC design giving control to kernel which identifies the phase and executes accordingly.

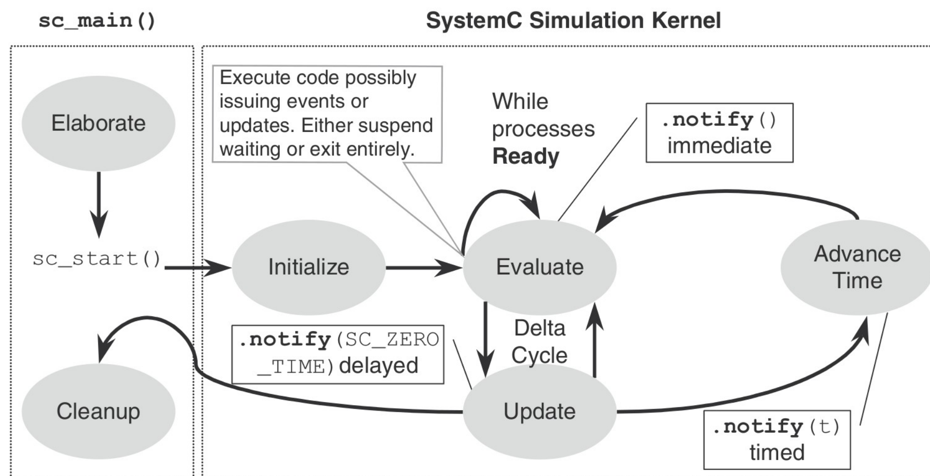


Figure 3.2: SystemC execution kernel

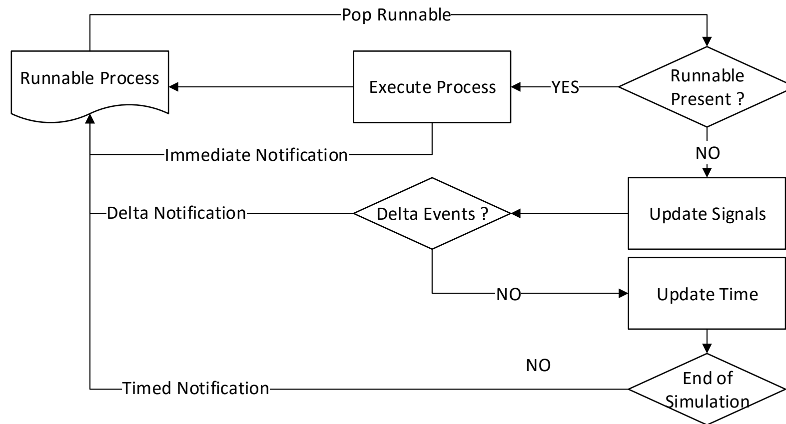


Figure 3.3: SystemC execution flow

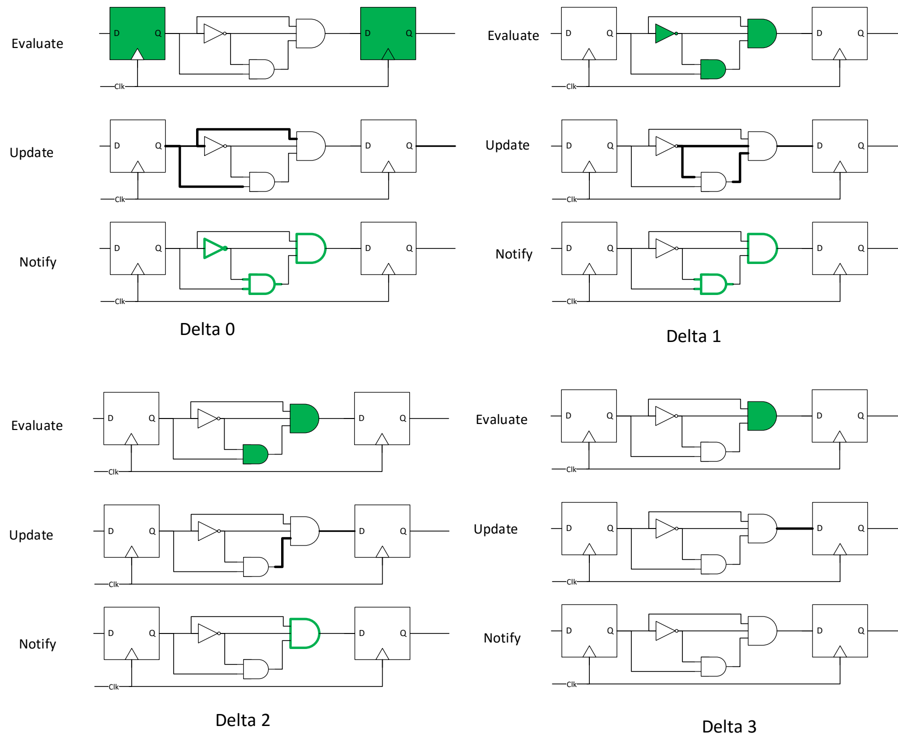


Figure 3.4: SystemC execution example

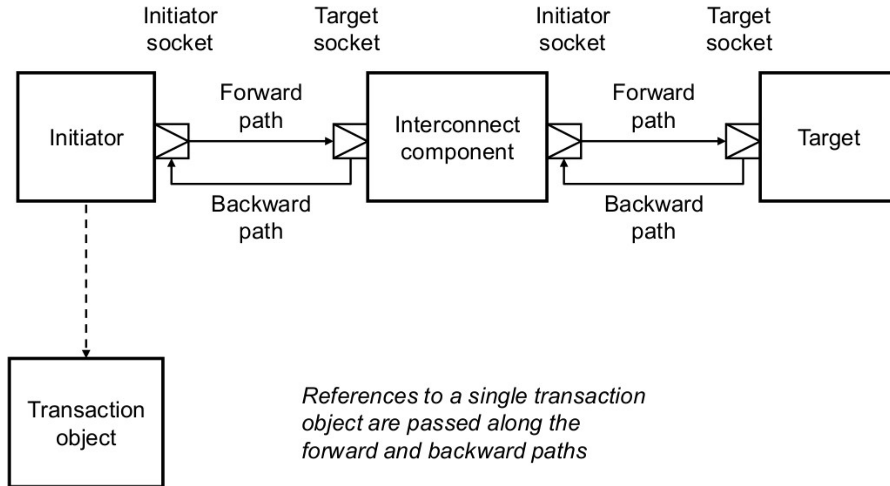


Figure 3.5: Initiator, interconnect and target communicating through TLM sockets

3.2 TLM2.0

In previous chapters, we have discussed the essence of transaction level modeling for SOC designs. In this section, we will mainly discuss its usage which is provided by Open SystemC Initiative(OSCI) as a TLM2.0 standard. This standard helps generalize a way of communicating between modules by the use of sockets and interfaces.

3.2.1 Components

TLM2.0 provides a set of C++ classes which can be used along with SystemC to provide a better framework for communication between modules. If this were to be done with traditional SystemC constructs, it would involve creating ports, defining a channel and extending ports and channel with an interface which can be complicated as well as time consuming. TLM2.0 provides a standardized way of doing memory reads and writes across modules. The key components are described

in the subsections.

3.2.1.1 Initiators and Targets

TLM2.0 consists of a set of transactions which are requested from Initiator and are directed to Target. Initiator is a module which initiates the transaction. The connectivity is defined through sockets in SystemC. Target is a module which receives the transaction, processes it and returns back to the initiator.

3.2.1.2 Socket

Socket is a basic entity through which the communication is made possible. It is very similar to a port in SystemC. Both initiators and targets have to define a socket associated with a type of interface. These are connected together in the connectivity file through usual SystemC constructs. A module may implement both initiator and target sockets if it has to act in both ways.

3.2.1.3 Payload

Each transaction is associated with a bunch of parameters like command type, command address, data pointer, burst length and many other options. All these parameters are compressed into a data-type called payload. Initiator is responsible to initializing these parameters, socket makes sure that it is received correctly at target, and target is responsible for processing the payload and notifying back to the initiator.

3.2.2 Interfaces

Interfaces are similar to intermediate components which are used to transport payloads from initiator to target. They define the behaviour of the transaction as how it should proceed. The sockets are usually defined with the type of interface they are linked. There are four types of interfaces which are used for different purposes

as defined below.

3.2.2.1 Blocking

Blocking transport interface is used when initiator thread is supposed to be blocked until the target thread evaluates the transaction, processes it and returns the result to initiator. The method which is used in TLM2.0 is called **b_transport** which takes pointer to payload and time as arguments. Time argument can be used to annotate the actual physical delay associated with the transaction and can be referenced as a *wait()* call in target. This type of interface is generally used in loosely timed models.

3.2.2.2 Non-Blocking

Non Blocking transport interface is used to initiate a transaction to target and at the same time keeping the current thread alive. This is done using a *nb_transport* call to target. The return from target is made by another *nb_transport* call to initiator from target. The direction of this is opposite to that of the initial call. This is generally used to give the status of processed transaction to initiator if it has been executed correctly. Based on the feedback, the initiator may choose to re-initiate same transaction or start another transaction.

3.2.2.3 DMI

DMI or Direct Memory Interface is used to improve the simulation efficiency. When there are multiple requests to same address in the target memory, it is inefficient to make blocking or non blocking requests repeatedly as each call will have to go through the interface causing delay. As a better way, there is a parameter in the payload to enable the dmi interface. The target will analyze the payload and provide a pointer to memory to initiator if dmi interface is enabled. Subsequent transactions

can be processed directly from initiator without having to go through the interface saving the simulation time and complexity.

3.2.2.4 Debug

Debug transport can be used to read or write to a particular memory address in order to debug the contents of memory at any particular time. The reason for separating it out from normal transport interface is to provide a better debug capability and also to make sure that there is no delays associated with this path.

3.2.3 Models

Transaction level modeling can handle different levels of abstraction depending upon the designer's requirement.

3.2.3.1 Loosely Timed

Loosely timed model corresponds to a transaction which communicates between two concurrent processes with the delay of the transaction specified by user. This delay is not the actual delay in which the transaction will complete, but rather a loose figure based on the nature of individual transaction. They are generally implemented with blocking interface. These models can use temporal decoupling where the some parts of simulation may run ahead of actual simulation time in order to increase speed.

3.2.3.2 Approximately Timed

An approximately timed model corresponds to splitting down the transaction in multiple phases and each phase having its own delay. Typically there are four phases which one needs to be concerned about. Beginning and end of request and response constitute four different types of timing specified. These models cannot be used with temporal decoupling because of presence of multiple phases and each subsequent

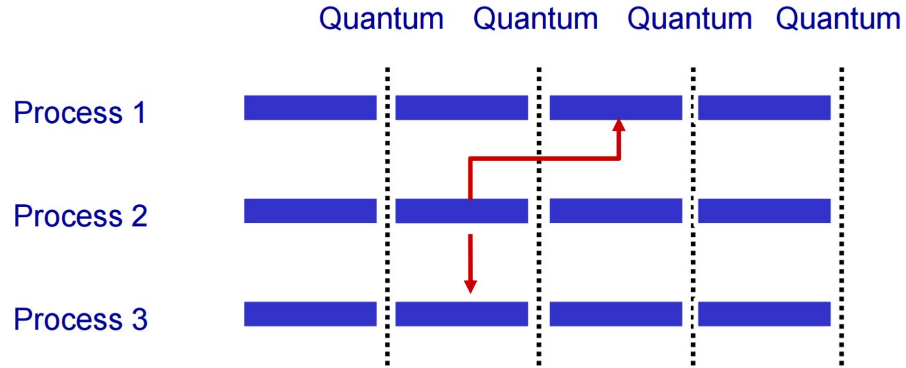


Figure 3.6: TLM loosely timed model

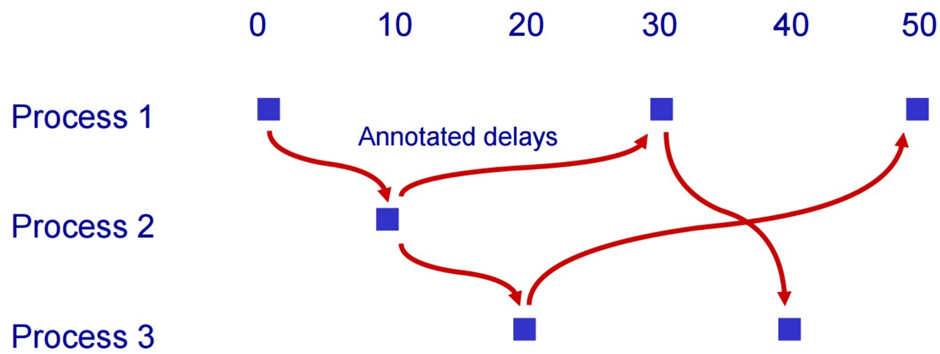


Figure 3.7: TLM approximately timed model

phase is dependent upon previous phase. That is why they need to run in lock step with each other. These models generally use non-blocking interface.

3.2.3.3 Untimed

Untimed models generally have algorithmic description without any annotation of time. So, they are called untimed models. These models usually have single thread implementing the algorithm, so transaction level models may not be beneficial for

this case.

3.3 GPU

GPU(Graphics Processing Unit) is a computing unit which provides high level of parallelism in the program. This research work uses the high computing capability of GPUs for speeding up kernel for SystemC execution. Therefore, in this section, a brief overview of GPU architecture and its programming model using CUDA is described. GPU is used to execute programs which are massively parallel in nature. The main quality differentiating GPU from multi-core CPU is that while latter has lesser number of highly powerful cores, the former has large number of small cores capable of handling simpler tasks. They were primarily designed for handling graphics which have large datasets to be processed quickly. Slowly, people started using them to solve other problems as well, which gave rise to General Purpose Graphics Processing Unit(GPGPU). In this work, the main idea of GPU is used to solve a system level simulation problem requiring high level of parallelism. Another important point to note is that the speed-up provided by GPU is primarily due to data level parallelism while the speed-up provided by multi-core CPUs is due to instruction level parallelism.

3.3.1 Overview

Even though GPU has large number of available cores, they are not recommended to be used for sequential code because they are not as powerful as CPU. Any program is divided into sequential code and parallel code. The sequential code is run on a CPU which offloads the task to GPU when the computation required in parallel. A GPU therefore acts like a coprocessor accelerator to speed up the program. It is connected to CPU through a PCIE express bus which provides high bandwidth data rate between DRAM and GPU memory. GPU has large chunk of global memory

which is used for storing the data to be processed. CPU generates the data and transfers it to GPU memory with the help of PCIE bus, GPU then processes and updates the data in GPU memory, CPU lastly transfer the output from GPU memory to its DRAM memory. The time required to complete the data transfer from CPU to GPU memory and vice versa depends upon the speed of PCIE bus, which is usually the bottleneck for parallel computations. It is essential that the speed-up obtained by intensive parallel computation by GPU dominates the overhead due to memory operations for this approach to be useful.

3.3.2 Architecture

The main component of a GPU device is called a streaming multiprocessor(SMP or SM). There are several of these present on a GPU device sharing the global memory of GPU. Each SMP has a capability to execute hundreds of threads at a time. There are several hundred CUDA cores present on a SMP which are responsible for high data level parallelism in the code. These CUDA cores work in SIMD or SIMT fashion where only one instruction is fetched for multiple data, thus making use of the large number of ALU units present in SMP. As evident from the execution strategy, this method involves lots of data reads and writes, so bandwidth of the register file is high enough to make memory operations fast. Apart from the CUDA cores, each SMP consist of a shared memory(L1 cache), register memory, load/store units, special function units and warp scheduler. GPU divides the total number of threads on a SMP into groups of 32 threads called warps. Each of this warp executes one instruction at a time on multiple data.

3.3.3 Programming Model

Programming models of a GPU present a set of rules and guidelines for using the available hardware resources in an optimized manner. A software programmer is

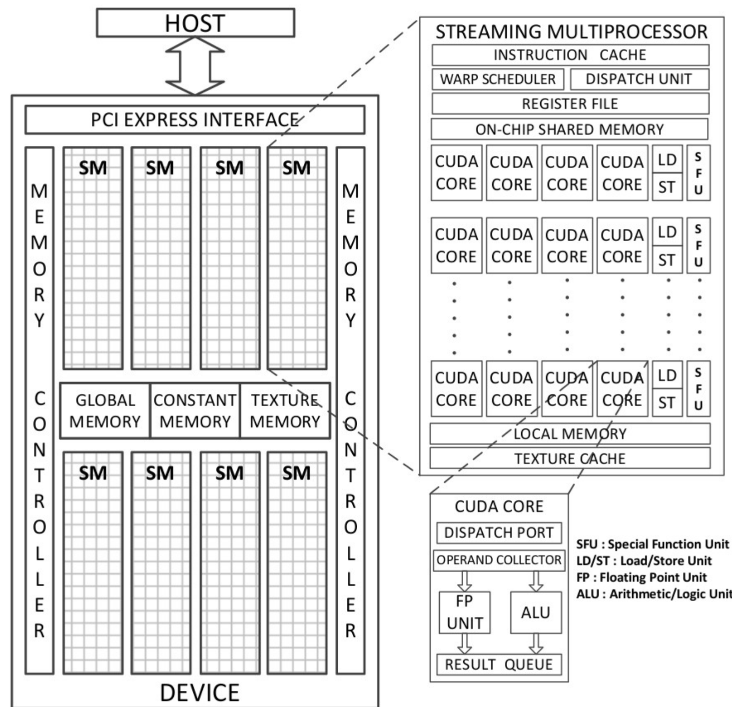


Figure 3.8: Architecture of GPU

generally aware of the programming model which is necessary to code. This model acts as a bridge between software and hardware and tries to efficiently manage the hardware resources based on the requirement as specified by the programmer. This model consists of both CPU and GPU code working together as heterogeneous computing. The part of code present of CPU is called *host* and GPU is called *device*. The call from *host* to *device* is made through call to kernel which invokes GPU processors. Since host and device both work on their independent memory, before and after call to the GPU kernel, a data transfer has to be initiated from host to device memory and vice versa respectively through PCIE express bus. There are dedicated function calls in programming model to make is happen.

Execution model of GPU is characterized into three different types of abstraction

levels forming a hierarchy. These are called *threads*, *blocks* and *grids*. A *thread* is the most basic form of execution unit in a GPU. They can be combined together to form *blocks* or *thread blocks*. Multiple *thread blocks* can combine to form a *grid*. While launching a kernel, we specify the size of thread block as well as grid which helps GPU map the individual threads to actual hardware efficiently. The mapping of grid, block and threads to CUDA cores in SMP is done internally to ensure maximum parallel computation.

A device may have only a limited number of SMPs. Each thread block is mapped to a particular SMP in a GPU which cannot be changed dynamically. All the threads in a thread block are run by the mapped SMP in GPU. Threads are combined in a group of 32 and executed as a warp in a SMP. There are multiple warp schedulers in a SMP making the execution of threads on a SMP concurrent. A SMP may have multiple thread blocks assigned to it and it can choose to switch between warps of different thread blocks depending upon the resources available.

3.3.4 Memory Hierarchy

A GPU device has different level of abstraction in memory as well. Since it provides high degree of data level parallelism, there are large number of memory accesses required for the computation. It is therefore necessary that memory utilization is optimized so that we may not incur significant overhead. Memory used in CPU is generally non programmable, in which programmer cannot define where it should be stored (for example L1 and L2 cache in CPU). In GPU, we have several options for storing memory at a different level of hierarchy as shown in figure 3.10.

Register Memory is the fastest memory available on the chip. Data which is local to individual thread is generally stored in register memory by simply declaring a variable in kernel. This automatically assigns one variable for all the threads active

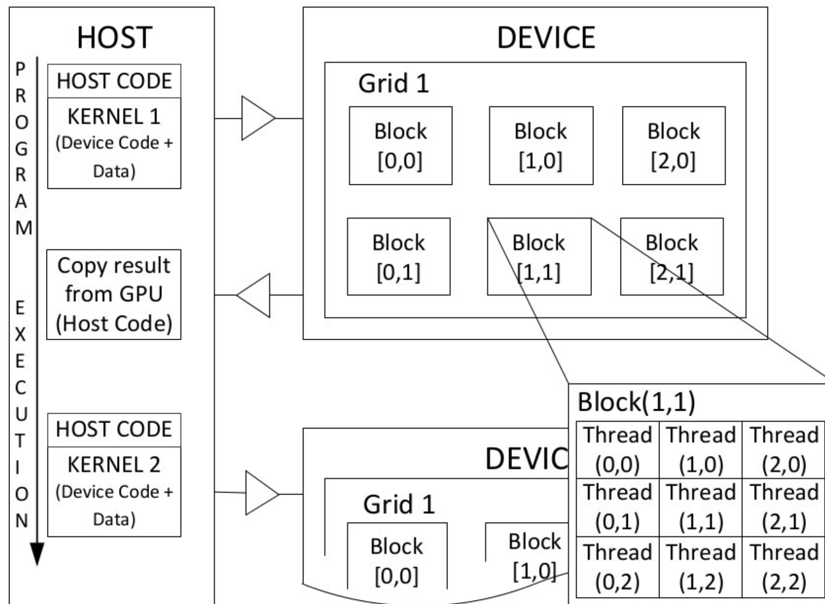


Figure 3.9: Programming model of CUDA

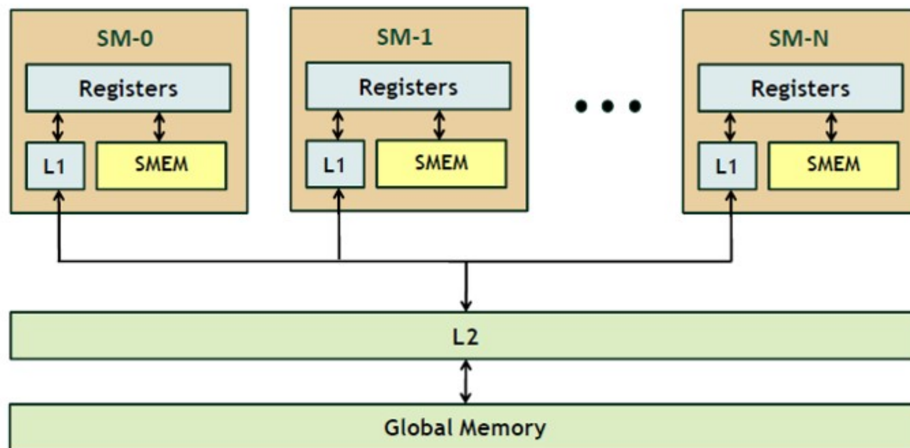


Figure 3.10: Memory hierarchy in GPU

in that kernel. Since this is the fastest form of memory available, it is recommended to use register memory when variables are used frequently. There are only a limited amount of register assigned for each SMP which are shared among all the active warps.

Local Memory is used by GPU when there is an overflow of the available *register* memory space. Similar to register memory, this is reference when a variable is defined in a kernel with its scope limited to thread, but the register memory is full. In this case, physically global memory is used to store all the spilled registers, so this is much slower than the register memory.

Shared Memory is a programmable on chip memory which has very high bandwidth and low latency. This is slightly slower than register memory but still very fast. Each SMP has a fixed amount of shared memory which is partitioned among thread blocks using the particular SMP. This are highly efficient for storing shared variables which are utilized by multiple threads because of the low latency. There is no alternate memory to store this if there is an overflow, and returns an error if more than available memory is required by the program. Shared memory variables are declared in a kernel by appending **__shared__** keyword to it.

Global Memory is the slowest form of memory available of a GPU. It also has the largest amount of size and is shared among all the thread blocks which are even assigned to different SMPs. It is not preferred to use global memory, but since we are bound by the available size of the shared and register memory, we use this to store large amount of data use by GPU in the lifespan of the program. Global memory is not lost in between multiple kernel launches so it is ideal for saving all the data to be used by the GPU program. A part of global memory can be copied to shared memory for faster access.

3.4 Project Description

The first half of this research work aims at developing a framework for system level design which can be used for performance analysis of any general external application. A SystemC based MPSOC framework is designed which consists of homogeneous multiprocessors(based on MIPS), cache memories, bus modeled using TLM2.0 constructs, memory and other external application. A jpeg based encoder is used to test the working of simulator under different configurations of MPSOC and predicts the configuration for best performance. Observations made from this framework are the motivation for next part of project. Upon increasing the number of processors in design, the number of processes running on SystemC single core kernel increases greatly slowing down the simulation. There is a lot of redundancy in the way SystemC code is executed on its kernel which is designed for single threaded application.

Next part of project aims at using the multiprocessors for increasing the simulation speed of system level design using SystemC. With the saturation of clock speed, the trend in today's world is growing towards multiprocessors. Analysing the performance of a design which consists of hundreds of cores can be difficult task. Virtual Prototyping with SystemC can be difficult as single threaded application involves lots of context switches between processes and runs serially. Another important practical aspect is that large system level designs use TLM2.0 framework for bus communication between modules. So, we need a solution which is extensive enough to cover any event based communication model and also able to handle transaction level models using TLM2.0. Visualizing the scalable nature of problem, this research work focuses mainly upon GPU based transformation of original SystemC code such that it can take advantage of performance gain offered by SIMD instruction types.

The key focus of this research work are mentioned below -

- *Mapping Non-Synthesizable SystemC models to GPU:* Previous works [24][39], map original SystemC model to GPU supporting only synthesizable subsets. These models are relatively simple with respect to their communication framework between modules. They typically use value changed events for combinational logic and clock based events for sequential logic. In order to achieve higher level of abstraction, timed event communication is often used in designs, which come under non-synthesizable subsets. We propose a framework to model all types of communication models on GPU.

- *Parallel TLM2.0 simulation on GPU:* As TLM2.0 is increasingly used to reduce the complexity of simulation in cycle accurate models, there is need to parallelize them efficiently. Transaction level models enable us to model memory mapped bus communication model at a higher abstraction level using predefined libraries. We propose a methodology to transform and simulate TLM2.0 loosely and approximately timed models on GPU.

- *Time Decoupled GPU simulation:* As the size of design increases, there is an increase in number of processes to be run in parallel. Processes are hence mapped to different thread blocks such that maximum processes can be run in parallel. Different thread blocks are also used when the processes have different code structure and require multiple fetch units (only one present in a multiprocessor). The simulation needs to be synchronized every delta cycle which is a non-trivial task for multiprocessors in a GPU and has a costly overhead. We propose a time-decoupled method to reduce costly synchronization overhead among threads mapped to different multiprocessors on a GPU.

3.4.1 Approach/Methods

In order to tackle the problem proposed in this work, first approach that is followed here is to design a MPSOC model in a higher level modeling language such as SystemC and identify the shortcomings associated with it. A multi-core simulator is designed in SystemC which is capable of attaching any external application to it and running as a virtual prototype. The process of design of a MPSOC simulator is a complicated task as it involves designing and assembling multiple components which in itself are complicated. As the architectural changes involves tweaking the design and analyzing which configuration gives the best performance, open source simulators are not flexible enough to fulfil this purpose. Our method involves designing a simulator from scratch. The processor, cache, bus and memory models were individually designed and tested. Any external application can be attached to the simulator and it will identify the performance of the system under a particular configuration. Initially this was tested with smaller applications, but later on this was successfully running a jpeg encoder application to compress images. Apart from a few coding styles which improved the speed of simulation, one of the major noticeable improvement was the use of TLM2.0 for communication using buses. This was found to be highly effective in terms of speed for loosely timed models in the initial phase of design. In order to improve the performance of encoder, when the number of processors were increased by an order, it was found that there was a significant slowdown in the simulation speed. This was the motivation for using multi-core techniques for simulation and specifically GPU since the nature of problem involves simulating a huge design consisting of many identical cores dividing the task.

Initially, some effort was spent on using the multi-core hosts by parallelizing the concurrent part of the simulation as by definition of SystemC, the function of the de-

sign should not depend on the order in which kernel will execute the processes. There are previous works proving this technique is indeed very useful for small designs, it is however not scalable. One of the major drawbacks of this approach was when the processes in SystemC do not have large runtime, then the synchronization overhead becomes comparable to the runtime of the process which is certainly not desirable. Another issue is that the number of multi-core threads are usually limited so the solution provided by this approach is not scalable to large designs. A GPU based approached works out best to nullify both the drawbacks as it has significantly low synchronization overhead and also the number of cores in a GPU are large providing a scalable solution.

The hardware architecture of GPU which is desired to be used in MPSOC simulations is slightly involved and works in a unique manner. Each GPU machine has a number of multiprocessors which have their own fetch unit and can run different instructions in parallel. Each Multiprocessor has many small cores which execute in SIMD fashion. For initial part of the project, this work statically identifies the number of processes which are to be executed by the SystemC simulation engine concurrently and maps each process to different cores of a multiprocessor in GPU. There are a few conversion rules associated with different types of constructs during the transformation. The modified code which is GPU compatible is then run on a GPU multiprocessor.

After designing a basic framework for simple SystemC models, next part of the project was to strengthen the communication models associated with SystemC. Synthesizable designs in SystemC typically use clock based communication for sequential designs and signal change based communication for combinational designs. People have not shown interest in a more robust event based communication framework which are used for non-synthesizable designs, so this work involves creating a sim-

ulation model for GPU which can simulate any general communication framework using events asynchronously. In order to create a general simulation model which works in all the above scenarios, the variables used for communication are classified into *sensitive* and *trigger* variables. Based upon the type of communication model, variables used in SystemC are mapped to these variables statically. The simulation is also divided into evaluate, update and notify phase as in the reference SystemC model and each phase is executed in parallel by many cores in a GPU. The mapping of *sensitive* and *trigger* variables to GPU memory is done in such a manner that running them in parallel does not affect the original intent of the design. This work also optimizes the memory assignment of these variables to different types of GPU memory (register, shared and global) to reduce the runtime.

Transaction level modeling is one aspect of system level design which is being extensively for loosely and approximately timed models for predicting the performance and power variations with low simulation runtime, good productivity and efficient work-flow. TLM2.0 is an extension of SystemC which is typically used for memory based bus modeling. This work identifies a TLM2.0 model and transforms it into an equivalent event based communication model as an intermediate step which can then be transformed using the previously mentioned techniques.

As the size of a design increases, it will eventually contain more number of SystemC processes which are to be mapped to GPU. Statically, only a fixed number(say 1024) of processes can be mapped to a thread block of GPU. Some processes eventually will have to be mapped to different thread blocks which may be mapped to different multiprocessors and synchronization between them is a non trivial task with large overhead. In order to make this feasible, the work proposes a time decoupled simulation technique for GPU by statically identifying the maximum time for which no communication is required between thread blocks and synchronize the thread

blocks only at regular intervals.

4. SYSTEMC/TLM2.0 BASED SIMULATOR

Before taking a leap into the GPU based kernel level optimizations in SystemC, it is important to understand the usage of the language and analyze some of the coding styles that can be used to accelerate the simulation. In this chapter, we explore the usage of SystemC to model a fully functional simulator which is able to attach any external application to it and analyze its performance. SystemC is used here to model hardware in a C++ based environment which supports a higher level modeling rather than going into details of gates and transistors. We would be using C language for modelling software to be run on microprocessor, whose code is rather easier to translate into hexadecimal binary which is read by the simulator.

There are many components which are modeled in this simulator and each of these components has its own purpose. A microprocessor model is required to run the software binary which is generally available as an open source Instruction Set Simulator(ISS) with different instruction set architecture(ISA). The problem however with the available ISS is that most of them are available as an executable binary so modifying the internal components according to user's preference is restricted. In this work, a microprocessor model is build from scratch which supports MIPS instruction set and provides a good simulation speed of around 65K instruction per second. This model is completely designed in SystemC and supports some of the advanced techniques such as forwarding, branch prediction and hazard detection and handling. Since we are mostly concerned about the performance of MPSOC environment, we can instantiate multiple instances of each core and redesign the software such that each processor handles certain part of the overall workload.

While a microprocessor play an important part in the whole system, for a fully

Simulator	Speed(instructions per second)
Simplescaler	150KIPS
Gem5	7.5KIPS(8 cores)
SESC	1.5MIPS
MC-Sim	32KIPS(64 cores)
SystemC	65KIPS(8cores)

Table 4.1: Simulation speed comparison of this simulator with other known simulators

functional MPSOC environment, we need a good memory model as well as a bus model which handles all the communication between the processor cores and memory. The memory model described here is relatively simple which handles all the reads and writes. The bus model used here is similar to AMBA AHB Lite protocol which supports multiple masters and slaves. The processor is coupled with a fast access cache memory model which takes advantage of spatial and temporal locality in the code to access memory. In order to test the simulator in a working environment, we instantiate required number of components in the top level design module and verify it with a test-bench which drives the simulator with data and checks for correctness. Any external application can be attached to the simulator and run it with its own software being executed by the microprocessors. In this work, we have first tested the simulator for simpler applications such as a simple multiplier, then a jpeg encoder hardware is attached to MPSOC with multiple cores sharing the workload.

Finally, we implement some of the multi-core techniques in kernel to parallelize the execution of concurrent processes and analyze its effectiveness over the traditional SystemC kernel.

4.1 Microprocessor Design

A microprocessor is a design element which takes an instruction, decodes its functionality and executes it using the ALU unit. The results are stored back to registers

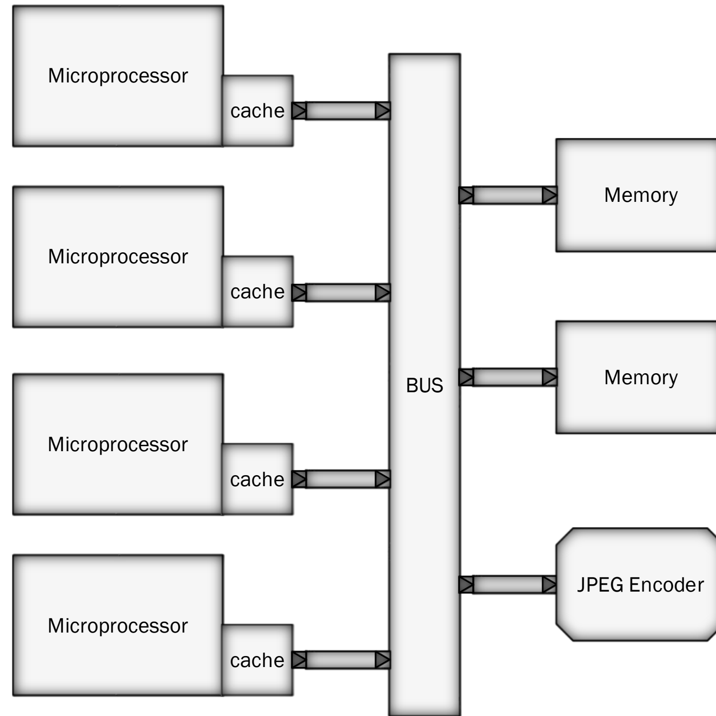


Figure 4.1: Schematic of simulation connected to a jpeg encoder

or main memory depending upon the instruction. Some of the basic components used in its design are defined as follows.

4.1.1 Register File

Every microprocessor needs some amount of fast coupled memory for doing temporary calculations such as program counter, stack pointer, frame pointer and other temporary variables. A register file is a fast but has limited memory. There are 32 integer registers and 32 floating point registers present in the register file. The width of the data provided is 32 bits.

4.1.2 ALU Design

Just like other components, a higher level design for ALU is created in SystemC which is then integrated with other components to achieve the full functionality of a microprocessor. ALU is currently configured to operate on all the arithmetic and boolean functions. It has its own ALU Control Unit which is responsible for translating the control signals into ALU function.

4.1.3 Single Cycle Design

There are 5 main stages in the execution of a microprocessor namely, fetch, decode, execute, memory and write-back. Advanced microprocessors coming lately have much more complicated stages, but in order to explore the usability of SystemC, we use a simple MIPS based design which is used in academia. All the above components are connected together with respect to each phase and the simulation is based on events. In this kind of design, all the five stages are executed in one cycle and so it is called single stage design. Apart from the components designed above, we need a logic for Control unit which decodes the instruction and sets the value of control signals according to it. In order to maintain the simulation speed, emphasis is made to ensure that native C++ constructs are used where ever possible.

4.1.4 Pipelined Design

In order to introduce pipeline in the SystemC design of microprocessor, double buffering of **sc_signal** makes it easier to model registers of hardware. All the contents of control signals and other registers are stored in a register file to update the contents of it. On the other hand, older contents of the same register file are used for processing information in next stage. In the update stage, old contents of registers are updated with the new values which is identical to what happens in a real hardware.

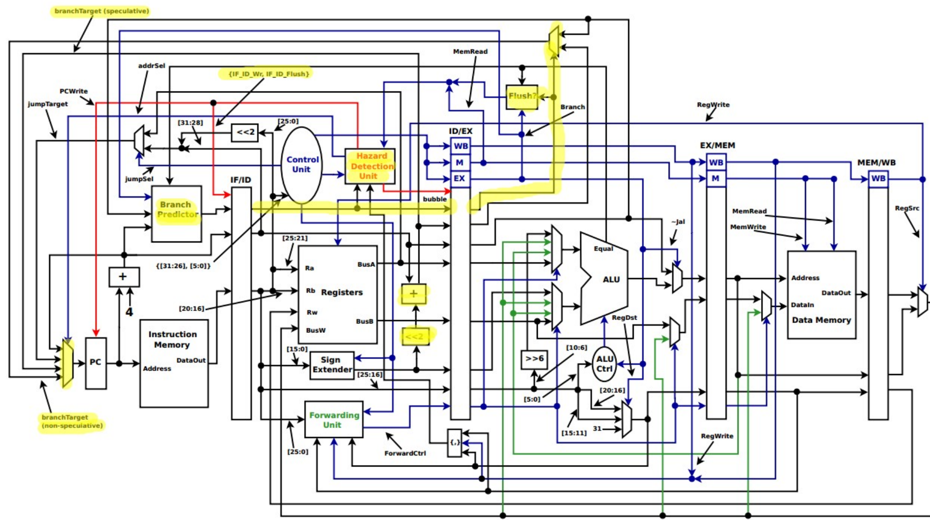


Figure 4.2: Schematic of MIPS based microprocessor designed in SystemC

In the process of design, there are many signal connections from one stage to another to ensure the proper working under all hazard situations. There are two types of methods in which simulation can proceed. One is where we design all the components like hardware with processes sensitive to events, and other is to follow a path where least number of processes are executed every cycle. Former, which is more similar to a hardware like simulation, is not that effective because it causes one process to run multiple times in a delta cycle until all its inputs are finalized. Second method uses the flexibility provided by software to model some components like hardware and execute them on software. A process is not scheduled to be executed until all its input are finalized, this makes this model more effective in terms of simulation. In order to avoid data collision, the simulation proceeds in opposite order of their actual execution in a particular cycle. This means we would be simulating in the order of writeback, memory, execute, fetch, decode. The main idea behind this is to prioritize the instructions which are ahead in time so that their

data is not collided with that of next instructions.

4.1.5 Forwarding Unit

This is a block placed in decode stage to prevent read after write hazard. In many cases, we have a immediate read proceeding a write to same register, which would normally give wrong results as this is a 5 stage cycle and the input is required in the third stage itself. In this case, their are direct paths of the temporary result being fed to previous stages to remedy the hazard.

4.1.6 Hazard Detection

The method of forwarding does not work in all the cases, and for some cases we have a real hazard which cannot be solved by moving the result. For those cases, we have a detection unit in decode stage which identifies the hazard and inserts bubbles in the data-path. A bubble is a nop instruction which does nothing and is only added to ensure that the hazard is resolved.

4.2 Bus Modeling

System bus is a component in SOC which is a hub for communication of multiple components. It could handle multiple channels of communication at a time or only support one channel depending upon the design. It usually consists of an initiator which places a request to a bus, which decodes the target intended by the initiator and forwards that request to a particular recipient. The target processes the request and returns back the result to bus which in turn finishes the transaction by providing initiator the results. In this design, we consider a AHB type bus protocol which can handle multiple masters and multiple slaves. There is a arbiter unit present inside the bus which decides the priority of transaction coming to bus and queues up for being processed.

4.2.1 Usage of Transaction Level Modeling

Bus is an important component which is used in all the SOC designs for communication between modules. A typical bus model designed in SystemC requires lots of effort on the user end. Also the simulation model of SystemC based bus is cycle accurate with the simulation executing in multiple delta cycles caused by the notification of events. This model can nevertheless be designed with impeccable accuracy upto cycle level but during the system level design, we are not really concerned with accuracy that high. The aim at that stage is to create a design that has fair amount of accuracy and is still able to predict the performance and power variations. Transaction level design is hence used for most of the bus based memory accesses. As an extension of SystemC, TLM2.0 is quite stable with predefined interfaces for communication whose parameters are defined through a payload. In our MPSOC design, the SystemC bus model is replaced with TLM2.0 model which is approximately timed. with some relaxation in accuracy, the simulation model works around 10 times faster than the original model.

4.3 Memory

4.3.1 Data Memory

All the operations which a microprocessor performs for a program requires a permanent storage in memory such DRAM which is a high latency memory connected to the bus. Access to DRAM are usually in the form of burst reads as we want to take as much data as possible to explore spatial locality. The model used in this simulator is relatively simple with each memory module providing ports and interfaces which can be connected to bus and accesses by any master.

4.3.2 Instruction Memory

A microprocessor fetches an instruction every cycle in the form of 32 bit numbers which are processed later in the cycle. Assembler gives an output of the list of hexadecimal numbers which are supposed to be processed by the microprocessor. Instruction memory is a class which is designed in such a way that any request coming to it goes to successive line in the executable file and returns its contents to microprocessor. Main storage of instruction memory is also in DRAM which is cached by the microprocessor. In order to get the ideal performance, we would have an instruction cache memory in between microprocessor and instruction memory, but is left out in this simulator for simplification.

4.3.3 Cache Memory

We have a configurable cache memory which is connected to microprocessor to enable fast access to DRAM contents by exploring temporal and spatial locality. Although, it can be increased, in the current model we just have one layer of cache memory which can be configured at the start of simulator. The design of cache memory model is done in a hierarchical manner with class definitions of a memory location(node) extending cache line(node set), extending cache memory. Each class has internal functions which facilitate the working of cache in a smooth manner. Although there is cache for both instruction and data memory, we are currently using only data cache for simulations here. However, instruction cache is a simple extension of the above.

4.4 Running the Simulator

Simulator designed in this project is a full SOC system level design where both hardware and software needs to be designed. The simulator has pre-designed com-

ponents used in a SOC such as microprocessor, bus, memory and other hardware resources. The user needs to instantiate these components with their required configuration such as cache size, number of processors, memory size etc. and connect their ports appropriately. The user should also create a system level hardware design of the prototype and attach it to the bus. Important thing to take into account while doing this is to declare the address space of the hardware such that it is accessible to other components in the simulator.

The next part is to write the software which will actually run on the microprocessor designed in SystemC and use the hardware components connected to the bus. In case of multi-core, ideally we would be writing a parallel code in *openmp* or any other parallel language, but this would requires an operating system for managing the workload. In order to simplify things here and achieve the same results, we would be dividing the workload to individual processors manually by writing a C code for each of them. After compiling and assembling the code, it is converted into executable format which is a list of 32 bit instructions ready to be executed by the processor. This forms the instruction memory of the microprocessor. All the parallel cores execute the code, access the hardware or memory when required.

In this work, first hardware that was tested to check the working of simulator was a simple multiplier whose inputs were given from the software and processed on hardware and result used by software again for further tasks. Moving towards more complicated hardware models, we have also built a JPEG encoder model which takes an input .png image and converts it into .jpeg format.

4.5 Multi-core Execution Kernel

Going forward, we explore some multi-core techniques to accelerate the SystemC kernel because it is too slow for event driven MPSOC simulator. The obvious solution

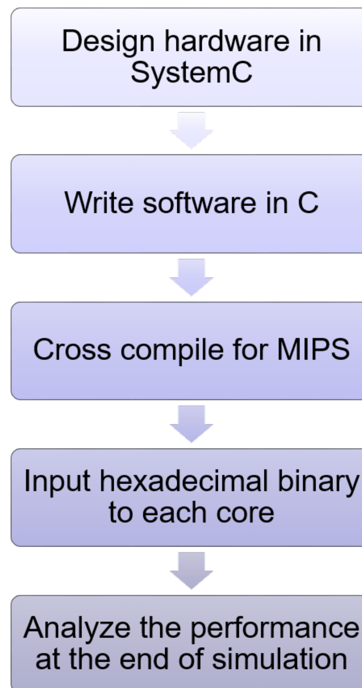


Figure 4.3: Working of simulator with an external application

is to identify the concurrent processes in *evaluate* stage and assign them to different processors in a multi-core environment. We tried different scheduling techniques and analyzed their performance on different designs namely, multiplier and JPEG encoder.

Static scheduling is the simplest form of multi-core where threads are assigned to multi-processors based on their index serially. This way, all the processors know which processes they are going to execute in the beginning. There is negligible overhead due to synchronization in this case, but the processes can be highly skewed in runtime. So, generally, the performance of this form of scheduling is the worst.

Dynamic scheduling is an improvised method where processors pick up threads to execute when they have finished execution of previous threads. In this case, the operating system maintains a queue of threads to be executed and processor

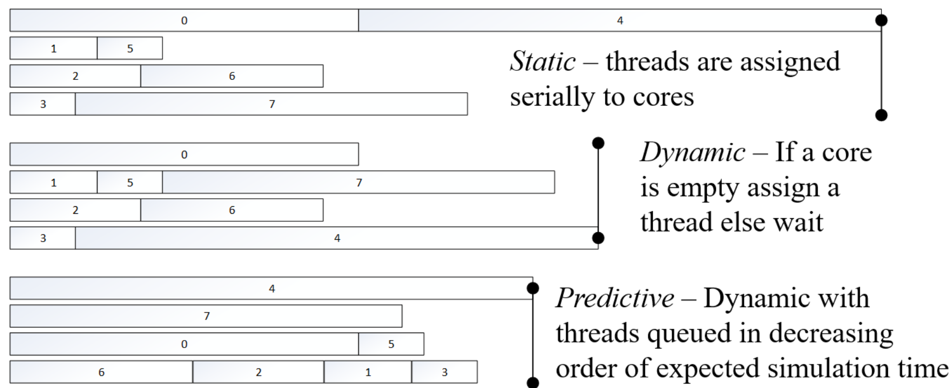


Figure 4.4: Different types of scheduling in multiprocessor host

picks one thread from top of the queue after finishing its current thread. This form of scheduling is quite balanced but has significant synchronization overhead for managing the queues and assigning them to different multiprocessors. Overall, this method is still better than static scheduling in general.

Predictive scheduling is another improvement on the dynamic scheduling when the processes are first sorted in order of their decreasing expected runtime and then assigned dynamically to multiprocessors. The expected runtime is the time which the process used in the previous delta cycle to run. Based on the prediction, multiprocessors are assigned threads such that most time taking threads are separated out among different multiprocessors. This provides an even more balanced distribution of threads and is usually slightly better than dynamic scheduling. The performance improvement is not a drastic, and so we look forward to other means of parallel simulation to achieve better results. From next chapter, we start looking at GPU based techniques to accelerate SystemC kernel.

5. GPU BASED APPROACH FOR SYSTEM LEVEL SIMULATION

5.1 Overview

As discussed in previous chapters, a GPU is a device with huge number of cores which can run in parallel exploiting data parallelism. In order to run a SystemC code on GPU, the kernel which is responsible for executing the processes should be compatible. However, this is not a trivial task as GPU programming is a bit different from standard multi-core programming. It has multiple levels of abstraction in cores as well as memory. An ideal way to handle this would be to transform the SystemC kernel which is designed for single processor into a GPU compatible kernel. This could however be quite complex to handle as there are lot of internal details in the kernel which cannot be directly translated into a GPU based code.

The flow which has been used in this work involves a parsing the SystemC code and translating it directly to a CUDA C code which can be run on a GPU. This method analyses the SystemC code and maps it to GPU in a way that it would still exhibit concurrent execution. In simple words, single core kernel is flattened with all the processes assigned a different threads on GPU. The methods which are supposed to be running concurrently on hardware will actually be executed by different threads of a GPU resulting in large speed-up.

5.2 GPU Execution Flow

The flow of execution in GPU is very similar to that of reference SystemC kernel. The main concept lies behind identifying the places in code which can be executed in parallel without changing the final outcome. The reference SystemC kernel executes a delta cycle in three stages. Delta cycles are executed until all the events have exhausted. This is how a discrete event simulation progresses. While doing the same

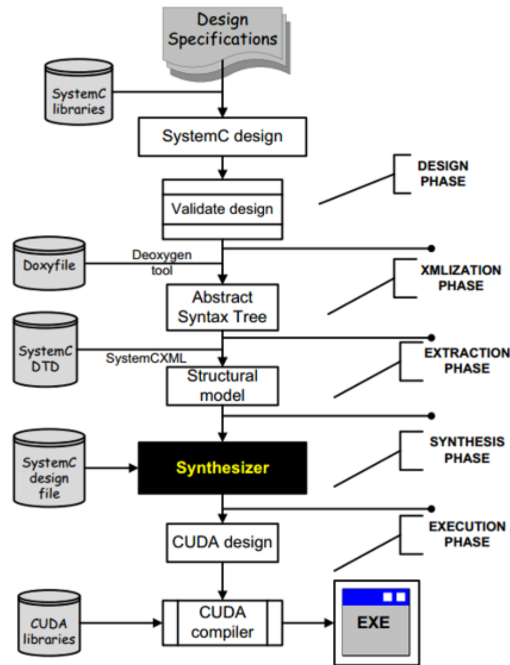


Figure 5.1: SystemC to CUDA conversion

thing in GPU, each phase takes the maximum benefit of parallel computation which GPU has to offer.

The three phases of SystemC reference kernel execution are *evaluate*, *update* and *notify*. In evaluation phase, all the runnable processes execute on a processor to produce some outputs. SystemC has a rule which is important for the concurrent execution of processes on a single processor. It states that the final outcome of the function modeled through SystemC should not be dependent upon the order in which concurrent processes execute on a single core kernel. This rule applies to multi-core simulation as well because the parallel execution of processes will cause race conditions in code if the final outcome is dependent upon the order of execution. Applying the same rule, all the processes are statically assigned to a thread in the beginning when the code is translated. One main advantage here is that since the

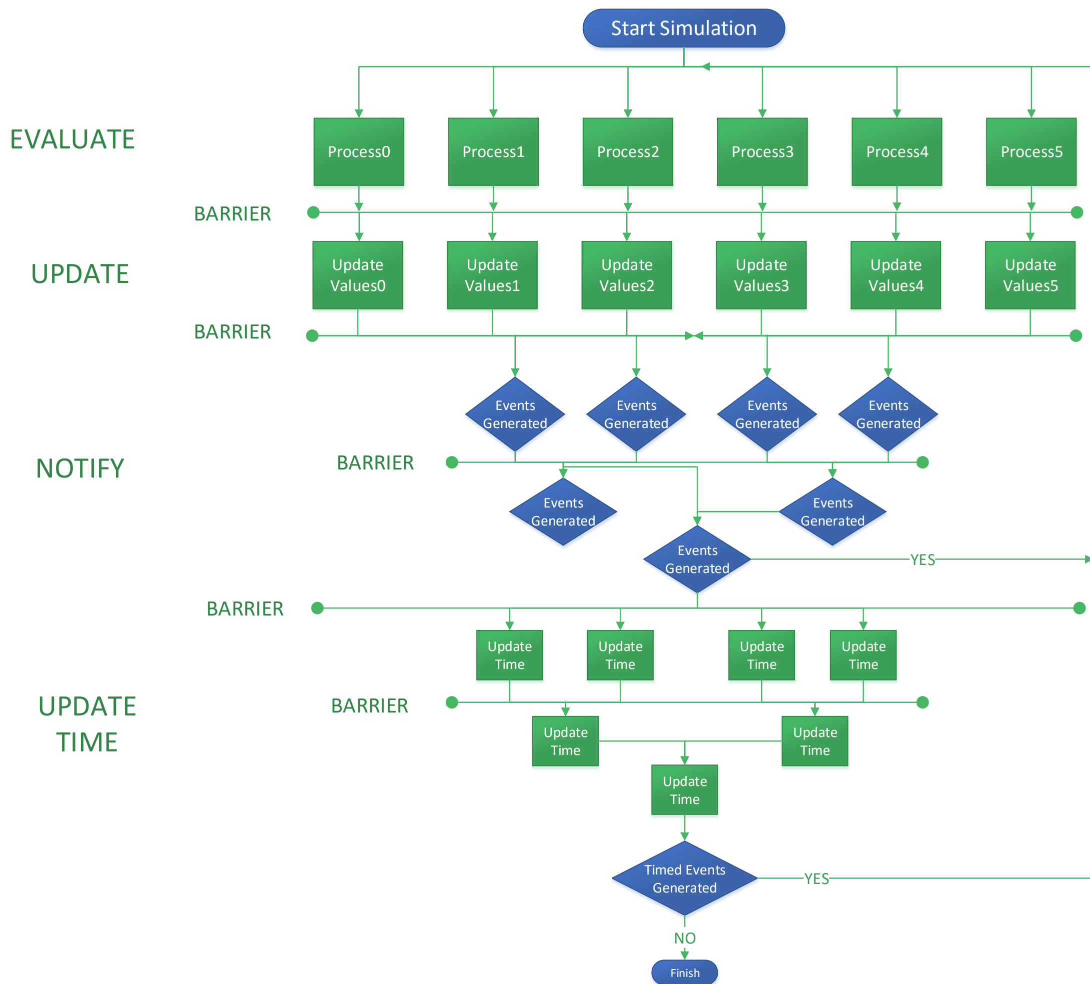


Figure 5.2: Execution of SystemC kernel in GPU

number of parallel cores are high, a large number of parallel processes can be handled with less synchronization overhead. Also care must be taken to make sure that multiple cores are not writing concurrently to a shared resource, which will then lead to race conditions in the code. This kind of scenario is generally not preferred in a SystemC design, but if it has to happen then we shall use the constructs which allow only one thread to access that shared variable. All the possible processes in the code are assigned a thread at the beginning. It is possible that in some cycles

only a few processes needs to be executed. In order to handle this, every thread has a conditional statement at the beginning of the code which checks if the sensitivity of the process is met and only then proceed towards execution. This stage is followed by a barrier of synchronization to make sure that all the processes have executed on GPU before moving to next stage.

After the evaluation of all the processes is done, the next phase is to update the variables with new values which are evaluated in previous phase. As we will see in later sections, different types of events have different methods of updating the variables. Describing briefly, simplest concept is to have a double buffering of variables such that it would not affect the results of evaluate phase. In a single core SystemC kernel, all the updates to events or variables are made one by one in a serial fashion. This process is accelerated in GPU by assigning each variable to a core which updates it every cycle. The order of update does not matter in the outcome, so making this phase in parallel does not need special care to check race conditions of shared variables. Similar to evaluate phase, here also we have a conditional check to update only the required variables.

Third and final phase in a delta cycle of execution is notify phase. Its purpose is to evaluate whether there are pending events for the current delta cycle. If active events are indeed found, then we go through another phase of delta cycle right from the evaluation phase. Only those processes are made runnable which are sensitive to the events found in notify phase. The process of finding if there is atleast one new event generated translates to finding an OR reduction of all the possible events of the design. Translating this phase is not as trivial as the previous two because it involves a reduction operator. Although this phase cannot be completely made parallel, it is possible to accelerate it into logarithmic runtime. This involves use of some common techniques which are used in GPU for reduction operators.

5.3 SystemC to CUDA Translation

SystemC is set of libraries in C++ which provide discrete event simulation framework for modeling hardware on software. In order to map this model to GPU which recognizes CUDA, we perform a transformation with the help of a synthesizer similar to the previous work[24]. This synthesizer parses the SystemC code and writes out a CUDA kernel code which can directly run on a GPU. Before getting into the details of the translation, there are few terminologies which a user needs to be familiar with.

5.3.1 Types of Events

Since, this work mainly deals with creating a framework for advanced communication models, we need to first classify the types of events which are responsible for communication. In general, we can classify the events into three categories -

1. Timed Synchronous Events: These are the events which occur regularly at a fixed interval of time, for example, positive or negative edge of clock.

2. Value Changed Events: These are events which get triggered when any signal changes its value.

3. Timed Asynchronous Events: This type of events are triggered asynchronously at irregular intervals, for example, communication of the type wait(delay), wait(event) and notify(event, delay).

Each of these events have a specific way of communication which needs to be handled in the translation. The distinguishing factor from the other approaches that try to use GPU for SystemC simulation is that while other works are generally focused upon synthesizable models which have rather simpler means of communication. In this work we develop a general model for SystemC simulation which can handle any synchronous as well as asynchronous means of communication.

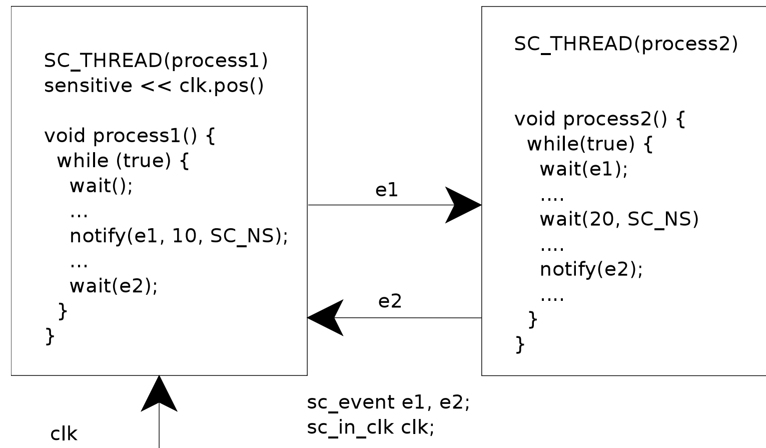


Figure 5.3: Timed event based communication in SystemC

5.3.2 Types of Variables

We typically need two variables for all events to deal with all of the above methods of communication. This method can be related to double buffering mechanism used in discrete event simulations in order to avoid race conditions. These are briefly described below -

1. Sensitive Variables: These variables contains information about the previous delta cycle of a particular event and does not take into account any updates due in the current evaluate phase. The advantage of this variable is that it separates out sensitivity and update of same variable resulting in race conditions. It typically has a boolean value which is read by each process in the beginning to determine if it needs execution. These variables are read in evaluate phase and written in update and notify phase.

2. Trigger Variables: These variables are updated by processes in evaluate phase depending upon the function modeled. They contain the information about when the signal or event is supposed to be triggered next. The variables are

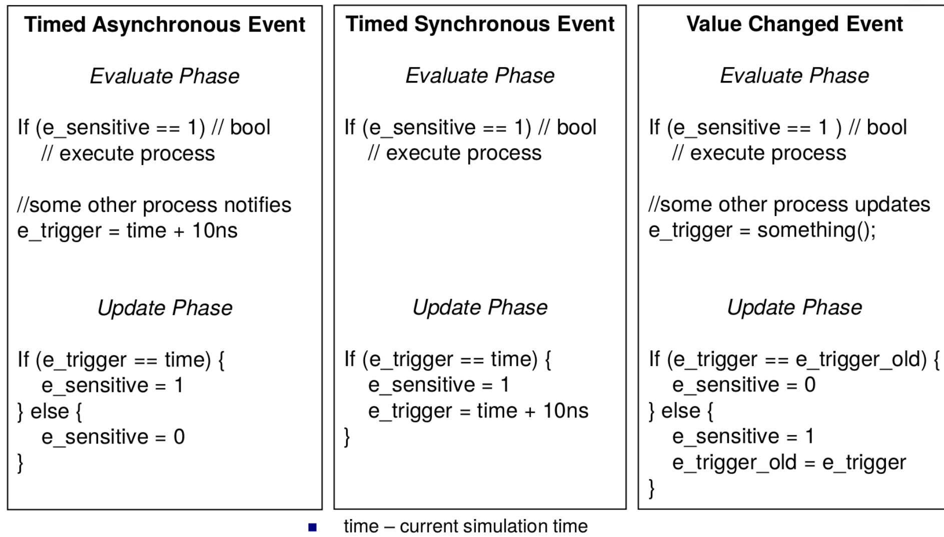


Figure 5.4: Event mapping to GPU

altered in evaluate phase and depending upon their value, they update the sensitive variables in update phase. In case of timed events, these variable refer the next time when the event is active. In case of value changed events, there is double buffering in trigger variables with allocated space for new and old values. These variables are written in evaluate phase and read in update and notify phases.

5.3.3 Algorithm

This section describes the details of algorithm which is used in translating SystemC code to CUDA. The first part of translation is to extract the hierarchy of the hardware and list of all the ports and channels in the design. In order to gather that information, we run the SystemC code to a parser such as Doxygen tool which creates an abstract syntax tree of all the class in the code and lists out their variables which in our case are the boundary ports of modules in XML format. These variables then can be used later on by a script to map processes to cores and also to identify

the variables and their scope.

In the process of translation, we proceed in the same manner as the reference SystemC OSCI kernel progresses. Firstly, the algorithm identifies all the events and their types based on the previously discussed categorization and also allocates the memory required for declaring the variables. All these events are declared with their corresponding data types. The memory allocated to events can be shared or local depending upon the usage of variables which is discussed later. The variables which are used to declare events are combined and renamed as arrays so that they can be coded in the form in SIMD which works much faster on GPU. More details on this will be discussed later in the chapter. Now, from the data generated from the XML, we declare all the parameters in the beginning of kernel. Before executing any code, there is an initialization phase, where all the methods and threads are initialized unless stated otherwise. This attribute is captured while parsing and is used in the synthesizer's translation script. Only the part of process which is run until it yields control is translated into initialization phase.

Next phase is translate process and map them to different cores on GPU. It is easy to identify in the beginning where each process has yielded control to kernel. Processes in SystemC are of two types, SC_METHOD and SC_THREAD. SC_METHOD can be converted into SC_THREAD by simple modifications in code and now we describe a methodology to translate SC_THREAD to a CUDA core. Each SC_THREAD has some points in code where it yields control to kernel. The role of parser is to identify these points and translate the process into switch case statements as shown in figure. It also adds a variable to identify which case block is supposed to run. The part of code in each case block is protected by a conditional *if* statement, which identifies the sensitivity of thread at that particular point. It adds a conditional statement with all the *sensitive* variables of events as its parameters.

If the event is found, the block is executed until there is a break statement. All the updates to signals are translated to *trigger* variables, to avoid the race conditions in code. For value changed events, the new trigger variable is simply updated whereas for timed events such as wait or notify, the trigger variable(which is time) is updated to the next time when the process is supposed to be made runnable.

The above algorithm to map SystemC threads to CUDA threads is a specific example of a generalized mapping where there are no conditional statements in the code. In a general case, we may have multiple conditional statements changing the order of execution of blocks of a thread. The way to handle it is to create all possible combinations of blocks of code which can run in one delta cycle and divide them into separate switch case blocks. Even before the code is actually run on GPU, we can identify statically which is the next block one process has to execute depending upon the branch's result. Using this idea, it is possible to map any possible SystemC process to a GPU thread which will be run in parallel while execution.

After the evaluate phase is translated, we have a barrier synchronization command which ensures that all the processes have been completed before proceeding to update stage. In this stage, since we have categorized all the events previously, it is easier to write them down in single instruction multiple data fashion. Firstly, for all the timed events, we have a conditional statement to check if the current time(trigger variable) is equal to the time when the corresponding event is supposed to trigger. If that is the case, then we enable its event (sensitive variable). The trigger variable is reset to a negative value so that it does not trigger in the next delta cycle of the same time. If this was not done, then we would be stuck in an infinite loop. For all the value changed events, we compare if the new value is equal to the old value of trigger variable. If there is a change then again we enable the event(sensitive variable). While translating into GPU code, we have already organized all the events as arrays

and so updates to them can be issued by a single command with different thread ids as their thread index which picks up different data for same instruction. This style of coding make the simulation much faster because it makes full utilization of GPU which works on the concept of single instruction multiple data.

Moving on to next stage, we again have a barrier synchronization to avoid race conditions. In notify stage, we basically have to check if there is atleast one event pending to be addressed. All the sensitive variables of the events acts as a means to find if event is triggered or not. An OR reduction of all these variables should be able to predict if atleast one of the events is active. The whole delta cycle is repeated if an event is found. This process is repeated until all the events have been addressed. In the process of translation, the OR reduction is not as trivial and cannot be finished in O(1) as two threads combine in each iteration to give the result. There are several reduction techniques[26] which takes care of it in logarithmic time by making use of all the cores to reduce as many possible threads as possible.

Coming out of the while loop, if no pending events are found, the next step is to advance time to the closest possible time in future when any event is active. This requires finding the minimum positive value of the trigger variable of all events. The positive term is highlighted in this case because if an event is served, its trigger variable is reset to a negative value in update stage. Finding the minimum positive value of an array is again similar to reducing the array by taking two candidates in array each time and storing them in same array. It takes logarithmic time to calculate the next time to be simulated. Time is advanced to that value and the delta cycle starts again.

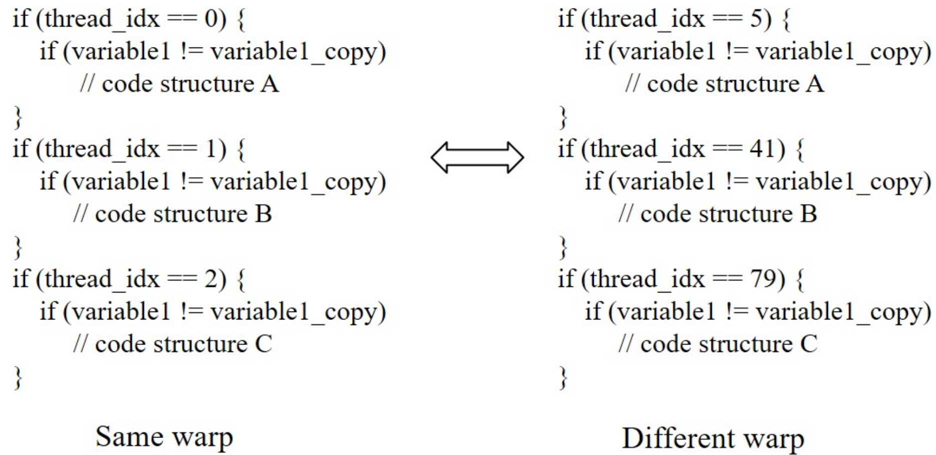


Figure 5.5: Code mapping to multiple warps in GPU

5.4 Different Code Structure Optimization

GPU produces best results when we have single instruction on multiple datasets. Each SMP has a single fetch unit and if threads are running code with different structure, then they are effectively serialized. Execution progresses in GPU is in a group of 32 threads which is called a warp. If a warp has different code structure, then it cannot be executed in parallel in any way. The solution here is to identify the processes with different code structure and map them to different warp. Figure 5.5 shows how to map them to different warps.

5.5 Shared Memory Optimization

Not much work is done to explore the memory assignment of variables as all of them use shared memory to store all the variables in previous approaches. If processes having different code structure are sensitive to same event, then they are mapped to different warps with reads to same variable as shown in figure 5.6. Memory coalescing happens only when multiple threads in a warp access same shared memory location. In the case mentioned here, memory coalescing is not possible since warps are dif-

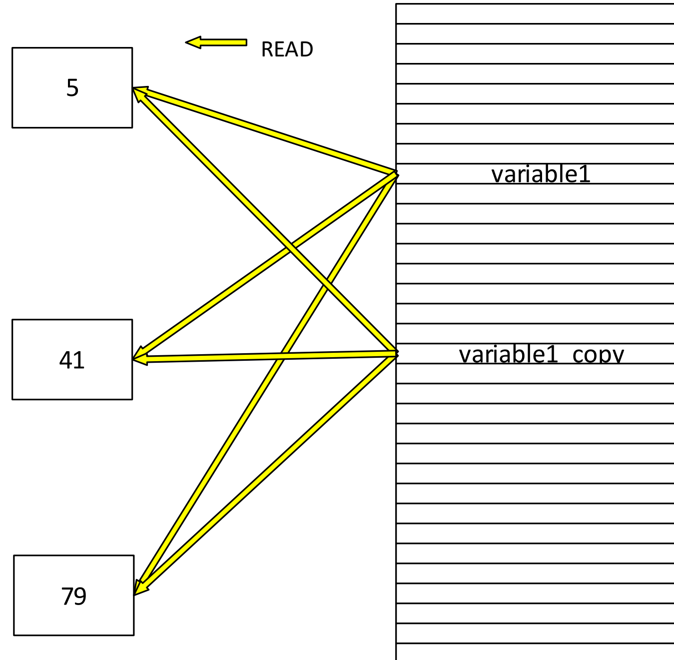


Figure 5.6: Shared memory conflict in sensitive variables across multiple warps

ferent, resulting in memory access serialization. If a *sensitive* variable is common to two processes which does not share the same code structure, then assigning them to shared memory can be costly as it will cause multiple reads to same memory location. Because of this, they are individually stored in every thread's register memory, reducing the serialization of reads. For remaining cases, we assign *sensitive* variables to shared memory, as register memory is a limited resource. *Trigger* variables are stored in shared memory because they do not produce conflicts.

5.6 GPU Execution of Non-Synthesizable Model

In this section, we give an overview of the execution steps followed in the transformed CUDA code by taking a example shown in figure 5.3. It is composed of two types of blocks, namely initiator and target, communicating through timed events.

Figure 5.7 shows the corresponding execution steps performed on GPU, where solid and dashed arrows show conditional true or false statements respectively while reading the variables from memory. Apart from communication via events, we consider different code structures among initiators and targets individually. In order to avoid shared memory conflicts, our synthesizer maps *sensitive* variables for events to register memory and *trigger* variables to shared memory. In execution phase, all the cores read their *sensitive* variables from register memory in parallel. Since initiators are triggered, they execute until suspended and update the trigger time for events sensitive to targets by doing writes to shared memory as shown in figure 5.7. In update phase, all the cores execute in parallel to check the current time against trigger time, and enable the corresponding event if there is a match. Since *trigger* variables

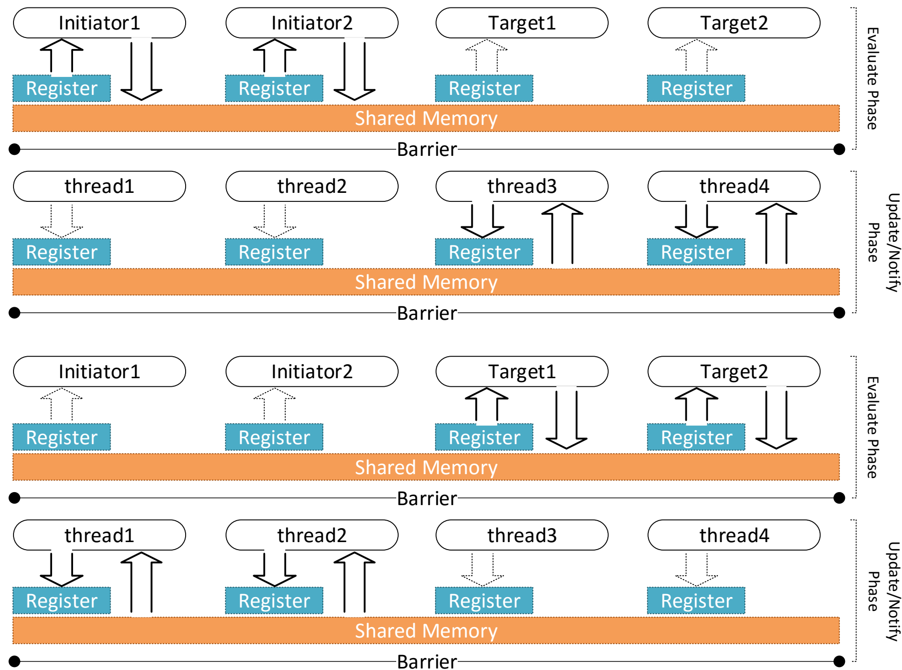


Figure 5.7: Execution mechanism of timed event communication in GPU using shared and register memory

Listing 5.1: *TLM Model Mapped to GPU*

```
// local sensitive variable shared trigger variable(time)
bool ev_sensitive;
int ev_trigger[2];
do {
 do {// Evaluate Phase
  switch (idx) {
   case 0:// initiator thread wait on e1
    if (ev_sensitive == 1) {
        // initialize payload ....
        // trigger e2 after delay
        ev_trigger[1] = time + delay;
    }
    break;
   case 1:// target thread wait on e2
    if (ev_sensitive == 1) {
        // decode payload ...
        // process transaction ...
        // trigger e1 in current delta cycle
        ev_trigger[0] = time;
    }
    break;
  }
  __syncthreads();
  // Notify and Update Phase
  // Check time against event trigger time
    if (time == ev_trigger[idx]) {// set event, reset time
    ev_sensitive = 1; ev_trigger[idx] = -1;
  } else {// unset event
    ev_sensitive = 0;
  }
  __syncthreads();
  ev = or(ev_sensitive); // find any active event
 } while (ev); // do until all deta cycles finish
 time = min_time(ev_time); // next trigger time
} while (time <= MAXTIME); // do until maxtime
```

reside in shared memory, this step causes reads to shared memory and writes to register memory. There is no conflict in reads here as each thread is assigned separate *trigger* variable stored in shared memory. After some time, events sensitive to target processes become active and same steps are followed again.

5.7 Support for Transaction Level Modeling

In order to parallelize transaction level models, we partition the process of conversion from TLM2.0 to GPU into two phases. TLM2.0 models are transformed into CUDA compatible model by using timed event based communication model as an intermediate step. Since the implementation of latter is discussed in depth in previ-

ous sections, we are only concerned about the former. Figure 5.8 shows a possible conversion of TLM2.0 model into event based communication model.

We study 4 basic interfaces which are widely used in most TLM2.0 models namely, blocking, non-blocking, direct memory interface and debug interface. All these interfaces are briefly described in previous chapters. In this section, we describe only transformation of a blocking interface into a timed event communication model. For other interfaces, the transformation is a trivial extension of the methodology described below.

5.7.1 Blocking Interface

In this example, we have single initiator and target connected through a socket as shown in figure 5.8. At the beginning, initiator initializes the payload with parameters such as transaction command, address, data pointer etc. In a blocking interface, b_transport is a method which initiates a request from initiator to target. When initiator makes a call to b_transport, it suspends the execution of initiator thread and triggers the execution of b_transport method implemented in the target. Initiator thread is suspended until the target method completes its execution. This protocol can be transformed to an equivalent model using two events, e1 and e2, and replacing the b_transport call as shown in figure 5.8 with notify(e1) and wait(e2) statements. The b_transport call on the target also undergoes some changes becoming sensitive to e1 and has a delay mentioned in the argument of b_transport call using wait(delay). Finally we add notify(e2) statement in target to return control to initiator thread upon execution. The payload transaction which is passed as a pointer in the b_transport method is stored in shared memory of GPU since it is used by multiple threads. Once the model is transformed into events and waits, it can be mapped to GPU easily using transformation into sensitive and trigger vari-

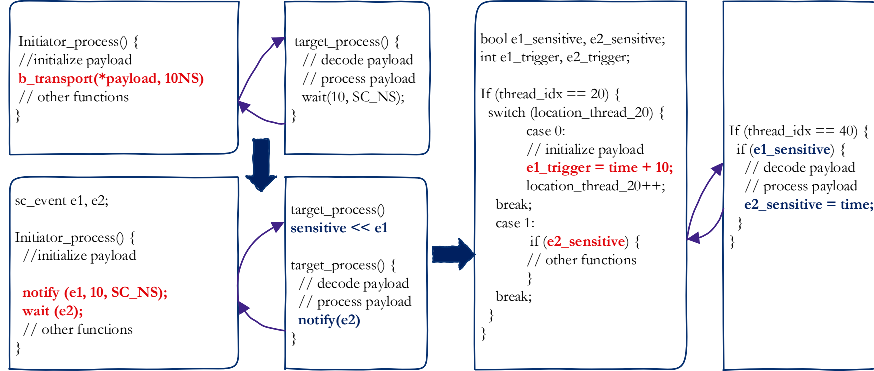


Figure 5.8: TLM transformation of blocking interface

ables and increasing the delay for time. Listing 5.1 shows a brief transformation of TLM2.0 model finally to GPU with detailed comments. Event e1 and e2 are mapped to ev_trigger[0] and ev_trigger[1] respectively.

5.7.2 Non Blocking Interface

A non-blocking interface can also be easily mapped to GPU threads in a similar fashion by transforming the original TLM2.0 model into event based model with wait and notify statements. The only difference here from the blocking interface is that while in blocking interface the initiator thread waits until there is a return from the target thread, so we have a wait(e2) in the end. In a non-blocking interface, the initiator thread continues to run even after it has started the execution of b_transport in the target thread. So, while transforming, we remove the wait(e2) statement from the initiator thread as shown in figure, to achieve same results. But, non-blocking interface have forward as well as backward transport calls, therefore two events are used in this scenario as well. The transformation is completely shown in figure 5.9.

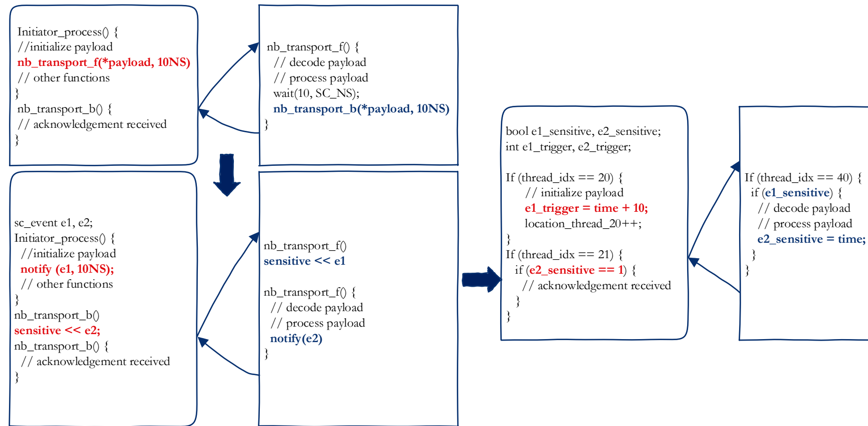


Figure 5.9: TLM transformation of non-blocking interface

5.7.3 Direct Memory Interface

A direct memory interface makes the use of fact that if there are going to be multiple transactions for same target, then we can get a pointer to memory from the target to initiator which can do reads and writes directly at the initiator. In this method, during the first transaction, the target sends a hint to initiator in the payload which tells it that direct memory interface can be established. In initiator we check for the clue and if it is found, we get the pointer to memory through a call to get_direct_mem_pointer() and do the rest of the read writes directly through that pointer.

Transforming this to GPU requires an additional hint variable in the shared payload and some conditional statements to check if the DMI is allowed on the memory. The initiator thread needs some additional checks and functions to retrieve the pointer from target thread to current thread after which it can complete the transaction in single delta cycle. It need not go through all the notify and wait cycles caused through blocking interface.

5.7.4 Debug Interface

Working of debug interface is similar to that of a blocking interface except that it does not have any notion of timing associated with it. It is only used to get a status of all the memory at any point of time. Transformation of this is very similar to that of a blocking interface by removing the wait(time) statement which causes the delay.

6. TIME DECOUPLED GPU SIMULATION

Until now we have only discussed how to accelerate any SystemC model execution using GPU based approach. We have talked about the translation of code from SystemC to GPU where the mapping is done only on a single SMP. A GPU has multiple SMPs and there are multiple reasons because of which we need to make use of these SMPs.

6.1 Problems with Single SMP Framework

1. Different Code Structure: We have mentioned in the previous discussions that the acceleration from GPU is mainly due to its SIMD nature which allows multiple instances of a single design to simulate faster. This however does not guarantee the best results when the design is largely dominated by elements which have different code structure. If there are instances of different designs integrated into a MPSOC, then the code will run more or less serially. This is because there is only one fetch unit in a SMP of a GPU and it consists of multiple stream processors which are responsible for execution of parallel threads. If the code structure is completely different, then it is hard to find the scope for parallel simulation. In order to avoid it, we want to split the code with different code structure to different SMP which have different fetch unit operating completely in parallel. This enables us to use the multiple fetch units to allow some instruction level parallelism in the code.

2. Large Problem: Another drawback of GPU based approach is that if a problem size is very large, we may not be able to map it to a single SMP. When there are large number of functions mapped only to a single SMP, the device will

split them into warps of 32 threads. There will be a lot of context switching in the code as only limited number of functional units are present which can execute in parallel. The other SMP on the other hand are sitting idle. So, we would want to distribute the work load to multiple SMP when the size of problem is large.

6.2 Trade-off in using Multiple SMP

Multiple SMPs provide us with some scope of instruction level parallelism in GPU, but these are usually very limited. The number ranges from 2 from low end GPU to 26 in high performance GPUs. In the hardware that we have used, we are provided with 13 SMPs in a GPU. Dividing the code onto multiple threads may sound good but there is a trade off in performance caused by multiple factors -

1. Shared Memory: When the processes are localized to one thread block which is mapped onto a single SMP, shared memory is readily available for storing an variables common to both the threads. When we divide the processes to different thread blocks, there is no guarantee that they will be mapped to same SMP. So, shared resources go to global memory which is much more costly in terms of latency than shared memory. So, use of different thread blocks should be such that common resources are handled in global memory with minimum cost in latency.

2. Synchronization Overhead: In the execution of a delta cycle in SystemC, there is a synchronization barrier among all the processes between each phase to make sure that some processes may not run ahead of time and cause race conditions. Now, when we split some processes to different thread blocks, we need to make sure that both the thread blocks are synchronized at the end of

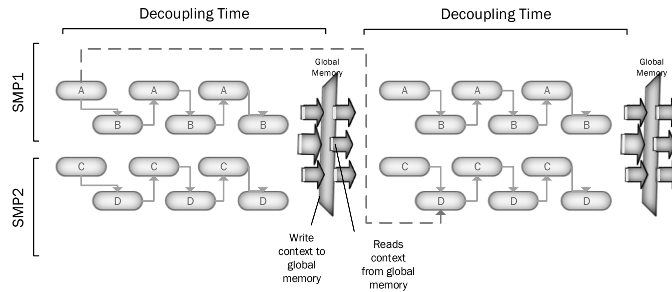


Figure 6.1: Time decoupled simulation

delta cycle. Synchronization between thread blocks is not as trivial as between the threads and is much more latency consuming. Main cause of delay is because the control is transferred to CPU where both the thread synchronizes and returns back the control to GPU. This context switch causes a lot of delay in the execution. Also there would be data transfers in some cases from GPU to CPU memory to determine stop condition which is costly. So, we need to use this technique such that synchronization is as low as possible.

In [42], authors have analyzed some techniques for efficient inter block synchronization. Nevertheless, it still becomes a bottleneck for achieving high performance with multiprocessors. In [39], authors have replicated common process to different multiprocessors to remove block synchronization. This is not feasible in case of transaction level models and timed event communication models. In this work, we have implemented time decoupled simulation to reduce number of synchronizations between multiprocessors of a GPU, effectively improving the performance. This approach was recently studied for its benefits on multi-core hosts[41][40].

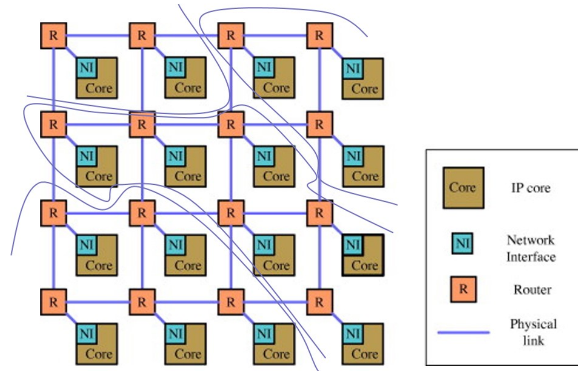


Figure 6.2: Partitioning of a design for time decoupled simulation

6.3 Time Decoupled Parallel GPU Simulation

For large designs which cannot be mapped to a single multiprocessor, we partition the design and map them separately such that minimum communication time between them is maximized. Each of the multiprocessor is run in parallel up-to a decoupling time, reducing lots of costly memory transfer operations. Communication between the two process groups shown in figure 6.1 happens rarely, which motivates us to run them on multiprocessors in parallel without synchronization up-to a decoupling time. Without time decoupling, there would be lots of redundant synchronizations at every step even though the processes mapped to different multiprocessor communicate after decoupling time. The decoupling time between multiprocessors cannot exceed the minimum communication time which would cause race conditions in the code otherwise.

This kind of technique requires some hints from user end to provide some communication delay across interfaces in order to create a partition. If the decoupling time exceeds this limit, then there is a possible chance of losing some transactions. So, this technique should be used carefully. Figure 6.2 shows a possible partitioning of a NOC design to different SMPs of a GPU.

7. EXPERIMENTAL RESULTS

7.1 Experimental Setup

We compare our results with OSCI SystemC reference kernel which runs as a single threaded application on a Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz machine. For GPU simulations, we use NVIDIA GK110GL [Tesla K20m], which can run a maximum of 2496 threads at a time with each core clocked at 706MHz and has 13 different multiprocessors(SMPs).

7.2 MPSOC Simulator

Firstly, we will start off by stating some of the results for simulation latency produced by SystemC MPSOC simulator which forms the basic motivation for moving towards multi-core architectures. We have designed the simulator with processor cores capable for running MIPS ISA, with all the transactions which are either a miss in cache or addressed to the external application(jpeg encoder) directed to their respective slaves after going through the bus. Bus modeling is done in two different ways, one is by designing an event based cycle accurate model and the other is by using transaction level models. As seen from the results in table 7.1, after simulating it with actual external application, transaction level modeling produces much faster results than the cycle accurate SystemC model.

Benchmark	SystemC	TLM	SpeedUp
Multiplier	1.24	0.09	13.8x
jpeg encoder(128X128)	86.40	11.40	7.6x
jpeg encoder(256X256)	376.81	42.79	8.8x

Table 7.1: Variation of latency(in sec) with different types of bus modeling in a host with 4 cores

Number of Simulator cores	Latency
3	11.4
6	24.8
9	41.6
12	56.3

Table 7.2: Variation of latency(in sec) with increasing number of cores for encoding 128X128 image

Benchmark	OSCI	Static	Dynamic	Predictive
Multiplier	1.22	1.09	1.06	1.11
jpeg encoder(128X128)	84.76	56.68	36.23	34.95
jpeg encoder(256X256)	369.12	191.30	128.49	119.15

Table 7.3: Variation of latency(in sec) across different types of scheduling in a host with 4 cores

For the jpeg encoder benchmark, we increase the size of design and share the work to additional cores to increase the parallelism in hardware. As shown in table 7.2, with increasing number of cores in design, the latency increasing almost linearly. More the number of cores, greater are the number of processes with same code structure. This is the key motivation for using GPU instead of multi-core host.

In order to accelerate the simulation, we try some of the multi-core techniques on the same benchmarks. Multiplier has a simple hardware model used to test initial results. For more complicated application, we consider jpeg encoder compressing 128X128 and 256X256 images by splitting some part of work to hardware and other to software. Using a host with maximum of 4 cores, we can infer from table 7.3 that dynamic and predictive scheduling provide good results. There is although not a major difference because the savings in time provided by predictive scheduling is comparatively low. Static scheduling does not work at all in all these cases as it has more chances to produce skewed scheduling.

7.3 Synthetic Benchmarks for GPU

In this section, we evaluate the performance of our techniques by simulating benchmarks which are most commonly used in industry and closely mimic the problems encountered in large scale parallel SystemC simulation. We start with a sequential design(Shift Register) consisting of a series of flip-flops connected to each other. Only consisting of timed synchronized events, this example is used for analyzing performance improvements from optimized memory allocation. We create a handshake benchmark as shown in figure 5.3 which consists of timed notifications and waits to implement a handshaking protocol between two processes.

7.3.1 Memory Optimization

Firstly, we compare the efficiency of our approach which maps *sensitive* variables to register memory instead of shared memory in certain cases as described in section 5.5. Since we are comparing results against previous approach, we consider 64 instances of both flip-flops in Shift Register and Handshake benchmark. As seen from the results in Table 7.4, for a sequential circuit, we are getting a speed-up of 5x more than the existing approach [24], and 10x speed up relative to the reference OSCI model. Our model is better in this case than the previous work[24] because we prevent the conflict in shared memory by allocating common events to register memory. Extending our approach to Handshake benchmark which implements timed events, we find 2.7x improvement in results when the *sensitive* variables were mapped to register memory rather than shared memory. Both the benchmarks show that distributing event variables to shared and register memory is better than assigning them always to shared memory.

Benchmark	OSCI	GPU_Sh	GPU_Reg	SpeedUp with GPU_Sh	SpeedUp with GPU_Reg
Shift Register	118.1	51.1	10.3	2.3x	11.5x
Handshake	129.0	63.8	24.1	2.0x	5.4x

Table 7.4: GPU_Sh is the scgp[24] implementation of benchmarks which sensitive variables are mapped to shared memory and GPU_Reg is our approach where sensitive variables are mapped to register memory. Simulation time(in ms) comparison across benchmarks

7.4 Typical SystemC Benchmarks

In order to evaluate our study on implementations which are used in industry, we use some of the typical SystemC models[31] which are used to evaluate performance of system level modeling. It covers a variety of applications from different domains

S2CBench : 12+1 designs

Design	Type	Domain	Optimizations Tested
qsort	dd	Auto/Ind	Loops, arrays, functions pointers
sobel	dd	Auto/Ind	Loops, functions, IO array expansion, multi-dimensional arrays expansion, fixed arrays (ROM, logic)
aes cipher	dd	Security	IO array expansion, multi-dimensional arrays expansion , large fixed arrays
kasumi	dd	Security	Multi-processes, delay report accuracy
md5c	dd	Security	#define macros, delay report accuracy
snow3G	dd	Security	Templates, delay report accuracy, function synthesis
adpcm	cd	Telecom	Structure synthesis
FFT	dd	Telecom	Floating point, trigonometric functions
FIR	dd	Consumer	IO array expansion, arrays, loops, functions, sum of products
Decimation	dd	Consumer	Resource sharing across loops, fixed point data types
Interp	dd	Consumer	Polynomial decomposition, fixed point data types, sum or products
IDCT	dd	Consumer	#include statement to initialize arrays, loops, functions,
Disparity	cd/dd	Consumer	Hierarchical design, multi-dimensional array expansion, synthesis running time

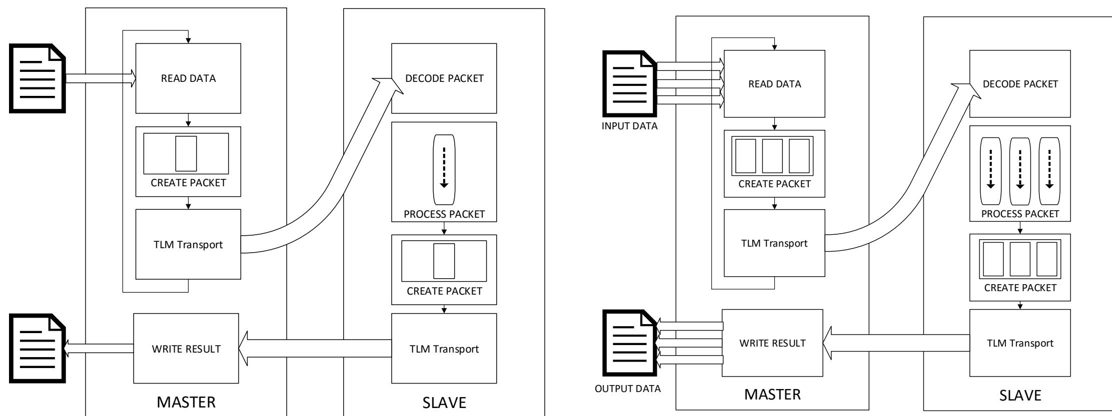
Figure 7.1: List of benchmarks

such as Auto/Ind, Security, Telecom, Consumer etc., described in figure 7.1. In each of these benchmarks, we have an initiator, which collects the data and communicates with the the target block through transaction level modeling. Target block processes the data based on its implementation, for example - fft, uart, interpolation etc., and sends the result to initiator through transaction level modeling again. These benchmarks are slightly modified to handle the communication between modules by TLM2.0 library. With the algorithm described in this work, we map these benchmarks to GPU and run for 5ms, where the delay between communication is taken 1ns. As shown in figure 7.2a, we have a master which extracts a set of data from input file, processes it to form a packet and initiates a transport call through TLM to slave. When the packet is received, slave decodes the contents of packet and processes the input data depending upon the application such as fft, idct etc. The output result is again transported through a TLM call to master which writes the result to a file.

7.4.1 Packet Size Variation

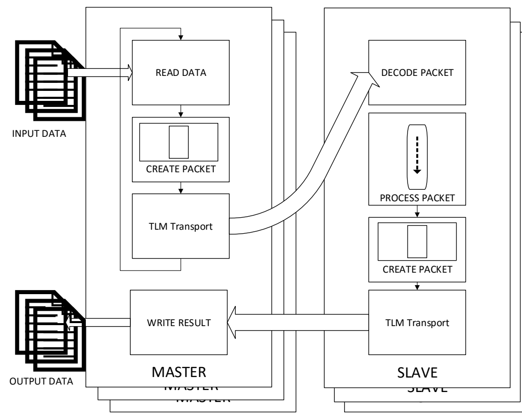
In order to achieve the scope for parallel threading, we firstly increase the size of data being extracted from input file. Figure 7.2b shows the corresponding block diagram of the resultant design. The packet now contains multiple sets of data which are to be transported through the same transport call to slave. At the other end, when the packet is decoded, multiple threads are spawned to process each set of data in parallel. Output results from all processes are gathered into a packet and transported back to master. For this experiment, total input data is kept same across different packet size.

Figure 7.3 shows a comparison of speed-up and latency for 5 different benchmarks by varying the packet size and hence increasing the number of parallel processes.



(a) Normal schematic with TLM

(b) Increase data sets in a packet



(c) Increase total number of instances

Figure 7.2: Schematic of benchmark with different variations

As we can observe from the graph, the latency of GPU decreases rapidly as we include more datasets in a packet as a result speed-up also increases. The change in latency for OSCI kernel is minor since effectively single core has to do same amount of work. One important point to note here is that the benchmarks having higher runtime also have the higher speed-ups increasing linearly. On the other hand, low runtime benchmarks saturate after a point because when the runtime is very low, synchronization overhead starts getting comparable.

Table 7.5: Simulation time(in s) of benchmarks with increasing packet size

Number of Instances	fft		aes		adpcm		idct		interpolation	
	OSCI	GPU	OSCI	GPU	OSCI	GPU	OSCI	GPU	OSCI	GPU
1	15.74	52.43	184.12	240.76	13.23	58.23	12.11	30.58	320.84	563.72
4	15.45	12.43	178.94	76.45	13.08	24.54	11.98	7.54	313.12	172.32
16	15.11	3.43	172.49	22.89	13.02	6.47	11.90	2.11	308.74	45.65
64	14.83	1.30	168.66	8.43	12.90	1.78	11.86	1.08	298.90	13.54
256	14.22	0.85	154.23	3.02	12.86	0.97	11.83	0.78	292.50	3.96
1024	13.76	0.72	142.75	1.56	12.82	0.77	11.81	0.67	284.32	1.21

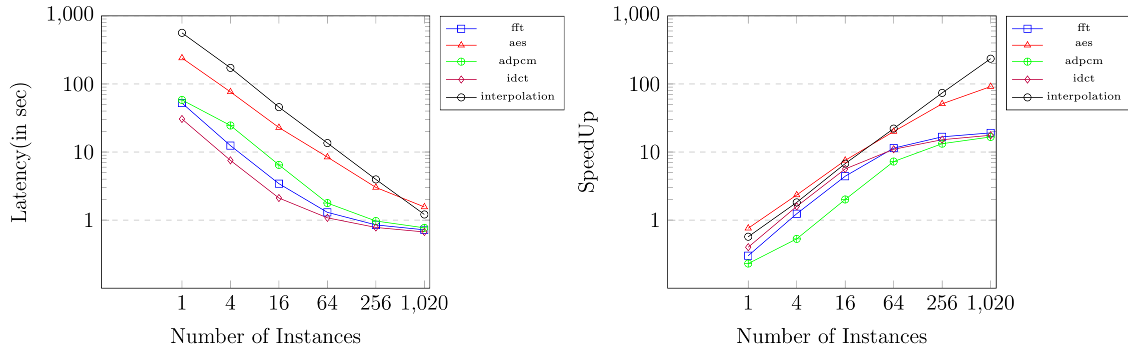


Figure 7.3: GPU latency and speed-up variation with size of packet

7.4.2 Number of Instances Variation

Another form of parallelism can be achieved by simply duplicating the instances in design to perform multiple operation at the same time as shown in figure 7.2c. As we increase the number of instances, the total data size also keeps on increasing. In order to limit the simulation latency to numbers which are not too slow, we reduce the initial runtime to 50000ns.

We can infer from the graph 7.4 that with increasing number of instances of the design, the speed-up increases sharply. Latency of design running on GPU is almost constant compared to that on single core. There is a sudden increase in latency when we simulate 256 instances which is because at that point we have to use multiple

Table 7.6: Simulation time(in s) of benchmarks with increasing number of instances

Number of Instances	fft		aes		adpcm		idct		interpolation	
	OSCI	GPU	OSCI	GPU	OSCI	GPU	OSCI	GPU	OSCI	GPU
1	0.07	0.06	0.53	1.22	0.05	0.03	0.04	0.02	0.60	0.35
4	0.19	0.06	2.28	1.23	0.13	0.03	0.11	0.02	2.35	0.38
16	0.69	0.07	8.63	1.24	0.47	0.04	0.43	0.03	11.95	0.41
64	3.16	0.07	31.54	1.27	1.64	0.05	1.50	0.04	37.28	0.45
256	12.51	0.23	130.42	1.39	6.51	0.18	5.78	0.12	140.24	0.67
1024	47.12	0.26	511.80	1.41	21.54	0.24	22.52	0.14	529.02	0.73

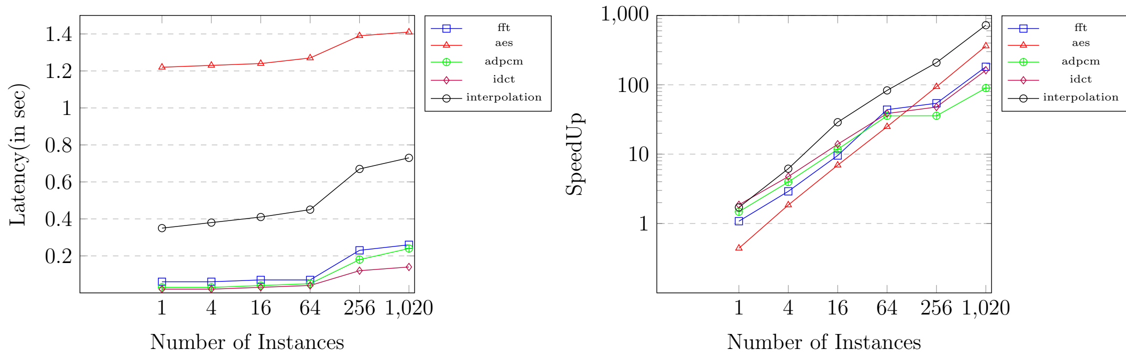


Figure 7.4: GPU latency and speed-up variation with increasing number of instances

SMP of GPU because one SMP cannot accommodate all the processes. This will cause additional latency due to synchronization across blocks and also global memory transfers. This overhead is comparable when the runtime is low for a benchmark and so it affects the speed-up slightly. For benchmarks with high latency, the speed-up is linearly growing and this effect is negligible.

7.4.3 Synchronous vs Asynchronous

Synchronous designs are the ones where processes are trigger at regular interval of time which is defined by clock period. In the above mentioned benchmarks, we can introduce a constant delay after each cycle to make it synchronous. In order to make it asynchronous, we have randomized the delay such that processes would

Table 7.7: Simulation time(in s) of benchmarks with increasing number of instances for asynchronous design

Number of Instances	fft		aes		adpcm		idct		interpolation	
	OSCI	GPU	OSCI	GPU	OSCI	GPU	OSCI	GPU	OSCI	GPU
1	0.08	0.06	0.53	1.22	0.05	0.04	0.05	0.04	0.71	0.67
4	0.24	0.15	2.35	2.45	0.14	0.09	0.12	0.06	2.23	1.04
16	0.81	0.36	10.22	6.87	0.48	0.25	0.48	0.15	12.57	2.79
64	3.73	1.09	34.66	17.12	1.69	0.87	1.45	0.39	39.11	5.70
256	14.23	4.12	145.77	45.68	8.38	3.23	5.98	1.23	156.43	16.33
1024	47.69	6.23	643.01	103.28	20.89	7.45	24.37	3.07	594.12	28.31

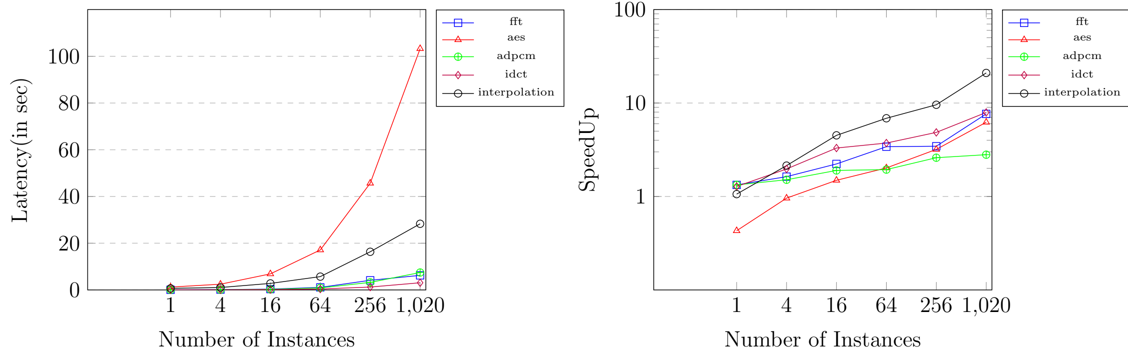


Figure 7.5: GPU latency and speed-up variation with increasing number of instances for asynchronous design

be triggered at irregular intervals of time. As seen from the results in figure 7.5, synchronous benchmarks simulate faster which is expected because it is able to make the most out of data level parallelism. The acceleration provided by GPU in this work is localized to processes simulating at a given time instant. So, if different instances of same design are being simulated at different times, then their execution in GPU will be serialized as well.

7.4.4 Scalability

Figure 7.6 shows the scalability of our solution by increasing the number of instances to 16384. A design of this size cannot be effectively simulated on OSCI

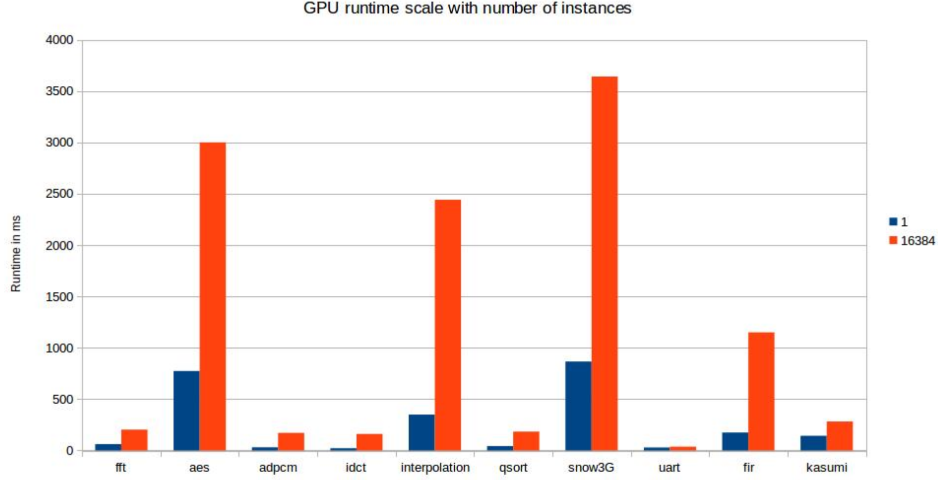


Figure 7.6: Runtime variation with number of instances across benchmarks

reference kernel because of linear scaling whereas GPU can perform the same task with only 4-5x increase in simulation time when comparing 16384 instances to single instance. This is achieved by mapping the instances to different multiprocessors as there is no communication between instances. With addition of some form of cross communication, the speed-ups are relatively lower which will be discussed in later sections.

7.5 Multi-Initiator Target

This example has a high resemblance with any typical MPSOC model where we have multiple processors connected to multiple memory interfaces through a bus which serves the purpose of arbitrating and transferring the packet to its correct destination. As shown in figure 7.8, the connectivity can be understood from the diagram. All the previous benchmarks that we have used until now, had minimum cross communication among duplicate instances. In this example, we will have some form of cross communication as the number of initiators and targets can be different. Number of Initiators are varied in this experiment to find out how much does the

Table 7.8: Simulation time(in ms) of multi-initiator socket with increasing number of initiators across different platforms

Number of Instances	OSCI	multicore(2)	multicore(4)	GPU
1	2.5	7.8	12.1	6.2
4	8.2	10.3	13.6	6.2
16	28.1	21.1	19.1	6.6
64	67.2	49.4	41.7	7.0
256	175.6	122.5	79.4	8.8
1024	402.4	269.3	151.3	9.4

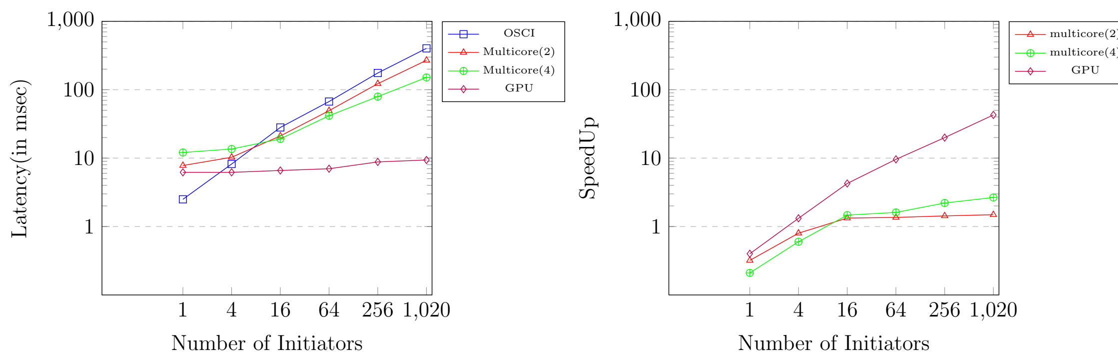


Figure 7.7: GPU latency and speed-up variation with increasing number of initiators for multi-initiator target

latency of GPU varies with respect to that of single core OSCI implementation and multi-core(2 and 4) implementations with dynamic scheduling.

In this experiment, we are comparing the simulation latency of single core with multi-core implementation as well as GPU. The scheduling algorithm used in here is dynamic since it is easier to implement and has a balanced distribution of workload. Analysing the data, we can infer from graph 7.7 that GPU based simulation model gives much better results when the number of instances are high because it is able to make the most out of GPU's data level parallelism whereas multi-core kernels are better than single core but not as good as GPU. Also when we have a single initiator, we would expect the single core kernel to give better results as there is no scope for

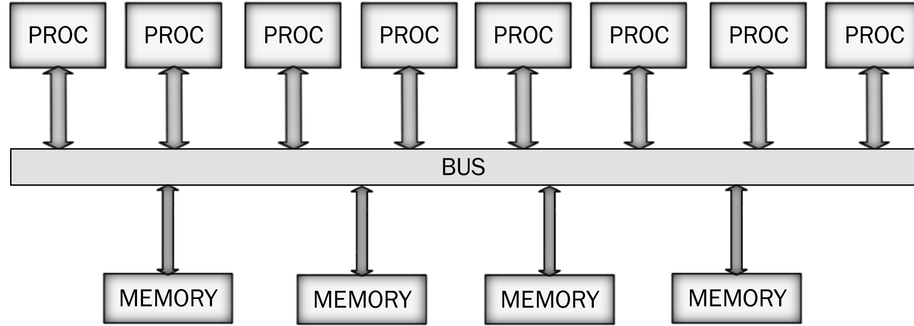


Figure 7.8: Multi-initiator target simulation model for benchmark

Table 7.9: Simulation time(in ms) of NOC with increasing size across different platforms

Number of Instances	OSCI	multicore(2)	multicore(4)	GPU
2X2	40.3	140.3	164.2	89.4
4X4	171.2	342.6	312.1	160.6
8X8	823.5	1109.1	845.1	311.3
16X16	3695.3	3987.8	2335.6	802.4
32X32	17342.7	13195.6	7326.2	1345.1

parallelism in the design, and which is confirmed from the results.

7.6 Network on Chip

As the number of cores on a SOC increases, the congestion in the bus grows quickly because all the cores have to go to memory through it. As a result, network on chip architecture is growing popularity these days in order to create a scalable network which can communicate through a chain of routers interconnected to each other. Another advantage that this kind of design offers for our evaluation is that it has a lot of branches in the code and also cross communication across multiple instances. Figure 7.9 shows the connectivity of a single router in a network of chip architecture.

Figure 7.10 shows the latency and speed-up variation of NOC implementation

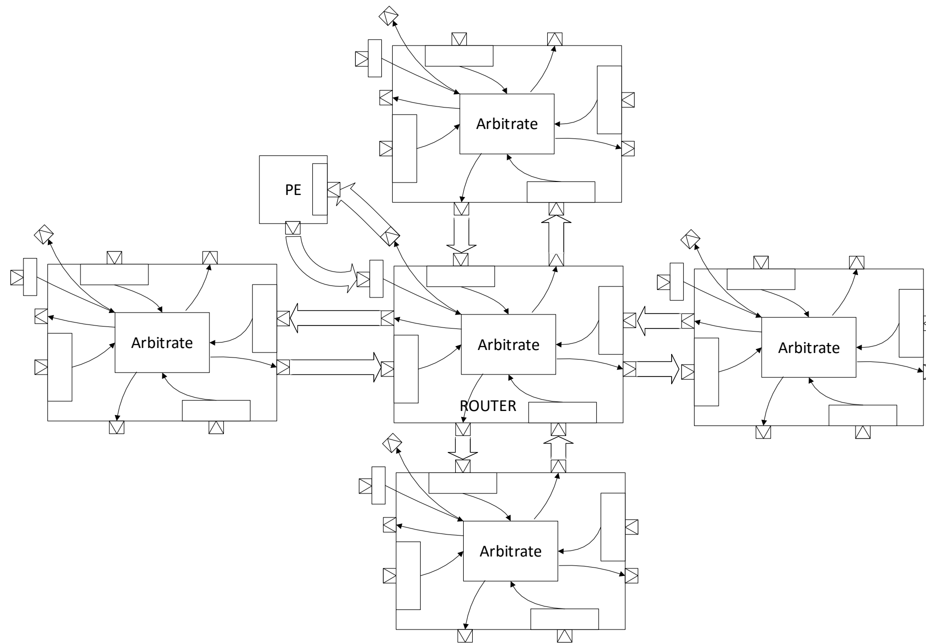


Figure 7.9: Schematic of NOC Router

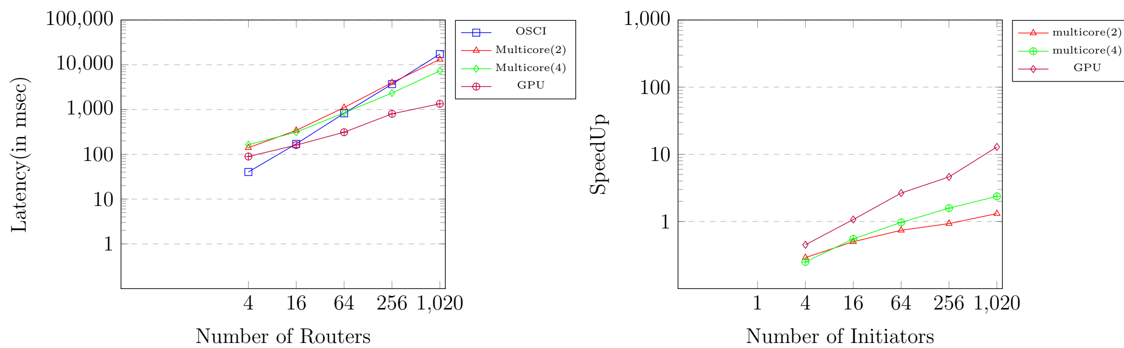


Figure 7.10: GPU latency and speed-up variation with increasing size of NOC

across different simulation platforms such as single core, multi-core(2 and 4) and GPU. The trend in the results is very much similar to that of figure 7.7 but the speed-up provided by GPU for this case is not as high as previous cases. The main reason being that with the complicated nature of design, there are lots of conditional

statements in the code which causes some amount of branch divergence in the code.

7.7 Time Decoupled Simulation

Figure 7.12 shows the variation of simulation time with increasing decoupling time for the Handshake benchmark where processes are mapped to different multiprocessors. As shown in figure 7.11 all the processes in a block communicate every 1ns and inter-block communication happens every 20ns. Both the blocks are mapped to different SMP and synchronization is restricted to happen every 20ns instead of every 1ns. We can observe that simulation time decreases exponentially upon increasing the decoupling time and stabilizes to a particular value eventually. In order to achieve maximum benefit of time decoupling, we should partition our design such that we lie as close to the stable region as possible. As we increases decoupling time, it cannot go beyond the communication time between two processes mapped to different multiprocessors, otherwise it will cause race conditions in design. So, minimum time for communication between events should be as high as possible.

In order to evaluate this optimization more clearly, we would be applying time decoupled simulation approach to NOC benchmark by restricting the communication across columns to 20ns. Communication across rows can still happen every 1ns. In this way, as shown in figure 7.13, we divide the simulation into multiple zones with each zone occupying one of the row. After simulating it for two configurations and varying the simulation latency with decoupling time, we can observe that there is a sharp decrease in latency in both the cases with increasing decoupling time and it tends to saturate close to 20ns when synchronization overhead and global memory access overhead are minimum. It is also worth noting that latency for the bigger design decreases more rapidly because it uses more SMPs and more global memory.

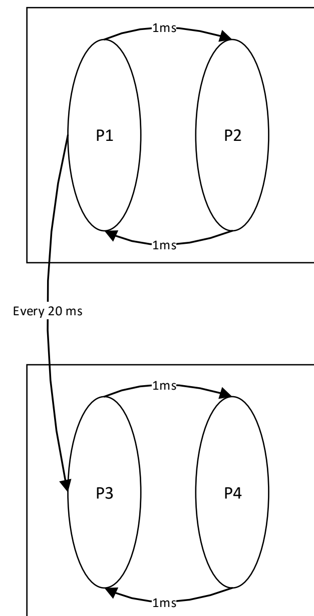


Figure 7.11: Handshake benchmark for time decoupled simulation

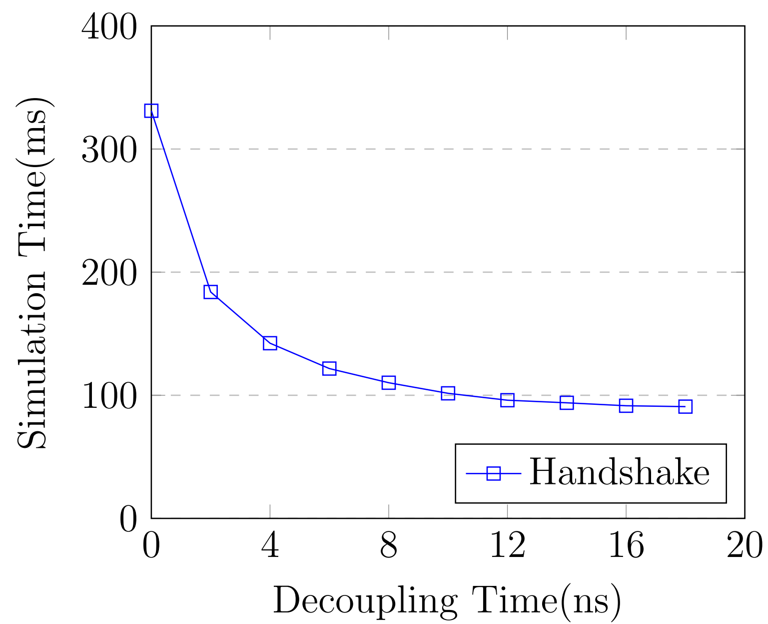


Figure 7.12: Time decoupled parallel GPU simulation for handshake benchmark

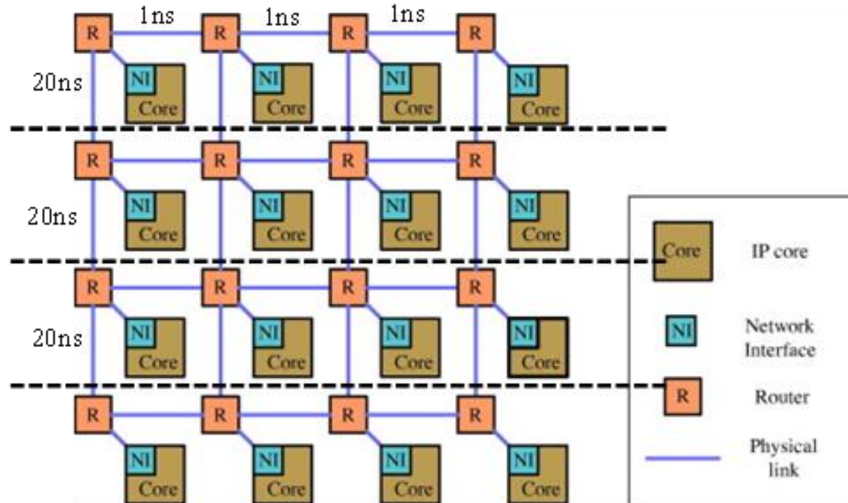


Figure 7.13: Time decoupling partition for NOC

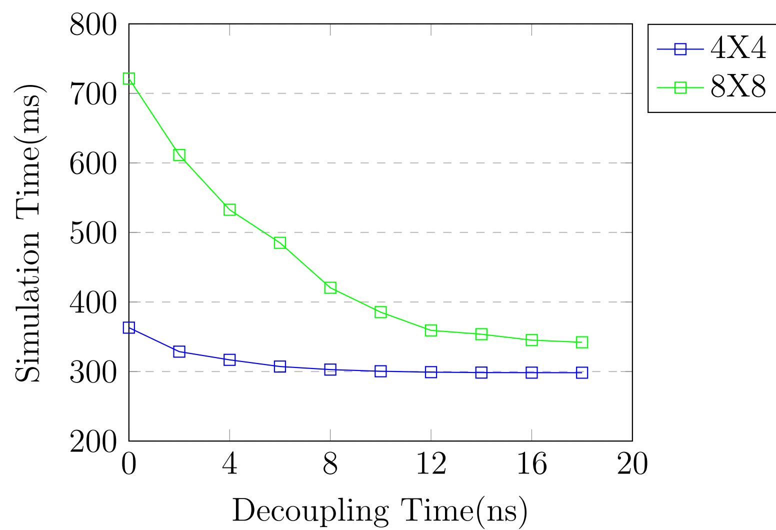


Figure 7.14: Time decoupled parallel GPU simulation for NOC

8. CONCLUSIONS AND FUTURE WORK

The future of computing devices is growing towards multicore architectures and very soon it is likely to come into mainstream products. Even current generation devices have dual to octa-core architectures. The demand is likely to grow higher in future with semiconductor companies trying to deliver products with a shorter time to market. The first step of IC design involving system level simulation is critical because it defines the architecture which rest of the teams are going to work on. Single core platform for simulation are no longer acceptable because of the increasing design size. This work provides a platform for simulation of SystemC models on GPU capable of handling complex designs involving timed event based communication which is not previously done before. It provides an infrastructure which can handle any general non-synthesizable SystemC design on a GPU device.

This work can be further improved in future by adding some techniques to resolve branch divergence in GPU which can be a serious problem if the number of conditional statements are more. This work involves a process to transformation of original SystemC code into CUDA compatible code which can be a lengthy process. Ideally, we would like to have an environment where we could just write the code for our design and it can simulate without the tedious process of transformation like on a OSCI single core kernel. Although GPU can provide a scalable solution, it may not be good in all the cases(for example when there is branch divergence). So, it should be able to identify beforehand if the design is best suited for CPU or GPU and run it accordingly. Also, not each and every construct has been shown to translate in this work. Future work could involve a simulator which defines translation of all the SystemC constructs.

REFERENCES

[1] NVIDIA, CUDA C Programming Guide v7.5. https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[2] Open SystemC Initiative, SystemC. www.systemc.org.

[3] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 33–42. IEEE, 2009.

[4] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *Journal of VLSI signal processing systems for signal, image and video technology*, 41(2):169–182, 2005.

[5] Valeria Bertacco, Debapriya Chatterjee, Nicola Bombieri, Franco Fummi, Sara Vinco, Anirudh M Kaushik, and Hiren D Patel. On the use of gp-gpus for accelerating compute-intensive eda applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1357–1366. EDA Consortium, 2013.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[7] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up, Second Edition.* Springer Publishing Company, Incorporated,

2nd edition, 2009.

[8] Nicola Bombieri, Sara Vinco, Valeria Bertacco, and Debapriya Chatterjee. Systemc simulation on gp-gpus: Cuda vs. opencl. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 343–352. ACM, 2012.

[9] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107. IEEE, 2008.

[10] Philippe Combes, Eddy Caron, Frédéric Desprez, Bastien Chopard, and Julien Zory. Relaxing synchronization in a parallel systemc kernel. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*, pages 180–187. IEEE, 2008.

[11] Jason Cong, Karthik Gururaj, Guoling Han, Adam Kaplan, Mishali Naik, and Glenn Reinman. Mc-sim: An efficient simulation tool for mpsoc designs. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 364–371. IEEE Press, 2008.

[12] David Richard Cox. Ritsim: distributed systemc simulation. 2005.

[13] Rainer Dömer, Weiwei Chen, Xu Han, and Andreas Gerstlauer. Multi-core parallel simulation of system-level description languages. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 311–316. IEEE, 2011.

[14] Fernando Escobar, Mauricio Guerrero Hurtado, Lorena García Posada, Antonio García Rozo, et al. Performance evaluation of a network on a chip router

using systemc and tlm 2.0. In *Circuits and Systems (LASCAS), 2011 IEEE Second Latin American Symposium on*, pages 1–4. IEEE, 2011.

[15] FA Escobar-Juzga and FE Segura-Quijano. Performance analysis of a jpeg encoder mapped to a virtual mpsoc-noc architecture using tlm 2.0.

[16] M Teresa Medina León. *Fast modelling and analysis of NoC-based MPSoCs*. PhD thesis, Eindhoven University of Technology, 2006.

[17] Ye Lu, Sakir Sezer, and John McCanny. Tlm2. 0 based timing accurate modeling method for complex noc systems. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 2900–2903. IEEE, 2010.

[18] S. Mahadevan, M. Storgaard, J. Madsen, and K. M. Virk. Arts: A system-level framework for modeling mpsoc components and analysis of their causality. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, sep 2005.

[19] A. Mello, I. Maia, A. Greiner, and F. Pecheux. Parallel simulation of systemc tlm 2.0 compliant mpsoc on smp workstations. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 606–609, March 2010.

[20] Jones Y Mori and Michael Huebner. A high-level analysis of a multi-core vision processor using systemc and tlm2. 0. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6. IEEE, 2014.

[21] Youssef N Naguib and Rafik S Guindi. Speeding up systemc simulation through process splitting. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.

[22] Kensuke Nakajima, Tomohiro Hieda, Ittetsu Taniguchi, Hiroyuki Tomiyama, and Hiroaki Takada. A fast network-on-chip simulator with qemu and systemc. In *Networking and Computing (ICNC), 2012 Third International Conference on*, pages 298–301. IEEE, 2012.

[23] Mahesh Nanjundappa, Akhil Kaushik, Hiren D Patel, and Sandeep K Shukla. Accelerating systemc simulations using gpus. In *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, pages 132–139. IEEE, 2012.

[24] Mahesh Nanjundappa, Hiren D Patel, Bijoy Jose, Sandeep K Shukla, et al. Scgpsim: a fast systemc simulator on gpus. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 149–154. IEEE, 2010.

[25] Jyothi Swaroop Arlagadda Narasimharaju. *SystemC TLM2. 0 Modeling of Network-on-Chip Architecture*. PhD thesis, Arizona State University, 2012.

[26] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[27] Pablo Montesinos Ortego and Paul Sack. Sesc: Superescalar simulator. In *17 th Euro micro conference on real time systems (ECRTS05)*, pages 1–4, 2004.

[28] Kalyan S Perumalla. Discrete-event execution alternatives on general purpose graphical processing units (gpgpus). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 74–81. IEEE Computer Society, 2006.

[29] Mohammad Abdul Qayum. *Design of a Mips Instruction Set Simulator for Multicore Processor Research in Systemc*. PhD thesis, Oklahoma State University, 2010.

[30] Hao Qian and Yangdong Deng. Accelerating rtl simulation with gpus. In *Proceedings of the International Conference on Computer-Aided Design*, pages 687–693. IEEE Press, 2011.

[31] Benjamin Carrion Schafer and Anushree Mahapatra. S2cbench: Synthesizable systemc benchmark suite for high-level synthesis. *Embedded Systems Letters, IEEE*, 6(3):53–56, 2014.

[32] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parsc: synchronous parallel systemc simulation on multi-core host architectures. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 241–246. ACM, 2010.

[33] Christoph Schumacher, Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Laura Tosoratto, Alessandro Lonardo, Dietmar Petras, and Thorsten Grotker. legasci: Legacy systemc model integration into parallel systemc simulators. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 2188–2193. IEEE, 2013.

[34] Shye-Tzeng Shen, Shin-Ying Lee, and Chung-Ho Chen. Full system simulation with qemu: An approach to multi-view 3d gpu design. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3877–3880. IEEE, 2010.

[35] Lin Xiang Shi and Zhe Zhang. A hybrid noc-based mpsoc simulator. In *Advanced Engineering Forum*, volume 6, pages 238–242. Trans Tech Publ, 2012.

[36] Rohit Sinha, Aayush Prakash, and Hiren D Patel. Parallel simulation of mixed-abstraction systemc models on gpus and multicore cpus. In *Design Automa-*

tion Conference (ASP-DAC), 2012 17th Asia and South Pacific, pages 455–460. IEEE, 2012.

[37] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David. Sesam: An mpsoc simulation environment for dynamic application processing. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1880–1886, June 2010.

[38] N Ventroux, J Peeters, T Sassolas, and James C Hoe. Highly-parallel special-purpose multicore architecture for systemc/tlm simulations. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 250–257. IEEE, 2014.

[39] Sara Vinco, Debapriya Chatterjee, Valeria Bertacco, and Franco Fummi. Saga: Systemc acceleration on gpu architectures. In *Proceedings of the 49th Annual Design Automation Conference*, pages 115–120. ACM, 2012.

[40] Jan Henrik Weinstock, Rainer Leupers, and Gerd Ascheid. Parallel systemc simulation for esl design using flexible time decoupling.

[41] Jan Henrik Weinstock, Christoph Schumacher, Rainer Leupers, Gerd Ascheid, and Laura Tosoratto. Time-decoupled parallel systemc simulation. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 191:1–191:4, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[42] Shucai Xiao and Wu chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[43] Dexue Zhang, Xiaoyang Zeng, Zongyan Wang, Weike Wang, and Xinhua Chen. Mcvp-noc: Many-core virtual platform with networks-on-chip support. In *ASIC (ASICON), 2013 IEEE 10th International Conference on*, pages 1–4. IEEE, 2013.

[44] Yuhao Zhu, Bo Wang, and Yangdong Deng. Massively parallel logic simulation with gpus. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(3):29, 2011.

[45] Hao Ziyu, Qian Lei, Li Hongliang, Xie Xianghui, and Zhang Kun. A parallel systemc environment: Archsc. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 617–623. IEEE, 2009.