CHOI, WON HA. System-level Power Prediction Methodology for Mobile 3-D Graphic Engines. (Under the direction of Prof. Paul D. Franzon.)

In this dissertation, we introduce an electronic system-level (ESL) methodology that predicts power consumption of 3-D integrated circuits (ICs) with through-silicon vias (TSVs). This methodology is implemented using SystemC transaction-level model (TLM). For the targeting application that shows a high potential in TSV integration, we choose mobile graphic engines that are widely used in smartphones and tablet computers. For both 2-D and 3-D GPU architectures, this methodology is executed in two stages. In the first stage, we achieve an accurate power prediction by incorporating register transfer level (RTL) physical design flow into the TLM core program. In the second stage, we execute the TLM core where power consumption is computed for each instruction organized into multiple command blocks that represent logic, memory and interconnects. In addition, we introduce floorplanning of the command blocks to make an accurate prediction of interconnect power. Using this methodology, we present a case study on an open-source GPU. Based on this case study, our TLM model achieves an average of 92.2 percent accuracy in power consumption values compared to its RTL counterpart, while reducing the total CPU time of synthesis and routing to a factor of 2.52 and power computation to a factor of 14. Furthermore, the reconfiguration cost of the TLM model in terms of the number of lines in SystemC codes is reduced by an average of 21 percent using thread-focused programming. For the power consumption values, an average of 27.4 percent savings in power consumption with the integration of TSV wide I/Os are reported.
© Copyright 2012 by Won Ha Choi

All Rights Reserved
System-level Power Prediction Methodology for Mobile 3-D Graphic Engines

by

Won Ha Choi

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Prof. Gregory T. Byrd

Prof. W. Rhett Davis

Prof. James M. Tuck

Prof. Paul D. Franzon
Chair of Advisory Committee
DEDICATION

I would like to dedicate this work to my mother Hang Ja Yang. She always stood for me through all the hard and tumultuous times, especially for the last 15 years. She survived so many hardships and showed me how to become strong both mentally and spiritually. She is the only reason why my faith still lives on. Without her, I truly do not know what I would have been.
BIOGRAPHY

Won Ha Choi was born on July, 8th 1984 and is originally from South Korea. He received the Bachelor of Science degree in Computer Sciences from the University of Texas at Austin in 2005. Won Ha received the Master of Science degree in Computer Engineering from University of Southern California in 2008. In that same year, Won Ha began his Ph.D. in Computer Engineering program at North Carolina State University (NCSU). His primary research focus is on 3-D integrated digital circuits and VLSI computer-aided design (CAD) that facilitates the tool flow to realize the integration. His secondary research focus is on parallel programming of VLSI CAD algorithms using General-Purpose Graphic Processing Units (GPGPUs).

Won Ha is currently a student member of IEEE. Upon completion of his Ph.D. work at NCSU, Won Ha plans to relocate to Icheon, South Korea to take a system-level IC design engineer position with SK Hynix Semiconductors.
ACKNOWLEDGEMENTS

First, I would like to deeply appreciate Dr. Franzon for advising me and helping me getting through this dissertation work. When difficulties struck during the middle of my doctoral studies, he helped me to pick a good research topic and encouraged me to continue my studies. I would like to also thank Dr. Davis for providing many suggestions that were very helpful in setting my work straight. Furthermore, I would like to thank Dr. Bourham, Dr. Byrd, and Dr. Tuck for being in my committee and suggesting many ideas that helped exploring this topic.
# TABLE OF CONTENTS

List of Tables .................................................. vii

List of Figures ................................................. viii

Chapter 1 Introduction .......................................... 1
  1.1 Problems Addressed in This Study .......................... 2
  1.2 Main Contributions ....................................... 3
  1.3 Overview of the Future Chapters ......................... 4

Chapter 2 Preliminaries .......................................... 6
  2.1 Investigating and Estimating Benefits of TSV Integration .......... 6
  2.2 Related Works ........................................... 7
    2.2.1 BCA Methodologies ................................... 8
    2.2.2 TLM methodologies ................................. 9

Chapter 3 Methodology Overview .............................. 12
  3.1 Setup Phase ............................................. 13
    3.1.1 Physical Design Flow ............................... 14
    3.1.2 System-level Configurations ....................... 16
    3.1.3 Command Clustering ............................... 17
  3.2 TLM Simulation .......................................... 19

Chapter 4 Power Prediction Setup ............................ 25
  4.1 Power Prediction of Blocks at Logic Tier ................ 25
    4.1.1 Creation of Sub-blocks ............................ 26
    4.1.2 Dynamic Power Prediction ......................... 27
    4.1.3 Tool-based Power Measurement .................... 28
    4.1.4 Standard Cell Library Use ......................... 28
    4.1.5 Caches and Register Files ....................... 28
  4.2 Power Prediction of Blocks at Memory Tier ................ 30
  4.3 Analysis of TSV Wide I/Os ................................ 31
  4.4 Power Prediction of Interconnects ....................... 32
    4.4.1 Assignment of Routing Congestion Degree .......... 35
  4.5 Using Command Clusters .................................. 35

Chapter 5 TLM Simulation ....................................... 40
  5.1 Overview ................................................ 40
  5.2 Thread-focused Programming ............................. 41
  5.3 MDI ....................................................... 44
  5.4 Programming Model of the TLM Core ..................... 46
  5.5 Floorplanning Setup .................................... 48
    5.5.1 Floorplanning Example with TSVs ............... 50
5.6 Update of Simulation History ........................................... 52

Chapter 6 Case Study with Open Source GPU .......................... 53
  6.1 GPU Features .......................................................... 53
  6.2 RTL EDA Flow ......................................................... 56
    6.2.1 Logic Sub-block Generation ....................................... 57
    6.2.2 Memory Sub-block Generation ..................................... 59
    6.2.3 Interconnect Sub-block Generation ............................... 61
  6.3 Command Cluster Generation .......................................... 64
  6.4 TLM Core Execution .................................................. 66
  6.5 Validation of the TLM Model .......................................... 67
    6.5.1 RTL Model ....................................................... 67
    6.5.2 Experimental Results of Three Metrics .......................... 70

Chapter 7 Conclusions and Future Work ............................... 78
  7.1 Summary of Contributions ........................................... 78
  7.2 Future Work ........................................................ 78

References ........................................................................... 80

Appendix .............................................................................. 85
  Appendix A Applying GPGPU in the Setup Phase ....................... 86
    A.1 CUDA Background ..................................................... 86
    A.2 Previous Work on VLSI CAD Algorithms .......................... 88
    A.3 Applying GPGPU ....................................................... 90
LIST OF TABLES

Table 2.1 Power Savings and Computing Resource Gains Achieved by TSV Wide I/Os 7
Table 3.1 Blocks, Sub-blocks, and Command blocks . . . . . . . . . . . . . . . . . . . 19
Table 4.1 Logic sub-blocks used in Power Prediction . . . . . . . . . . . . . . . . . . . 27
Table 6.1 Power, Timing, and Area Information of Sub-Blocks (8ns) . . . . . . . . . 57
Table 6.2 Summary of CACTI Parameters for 64KB texture cache . . . . . . . . . . . 60
Table 6.3 Read and Write Frequencies for Mobile Memory (%) . . . . . . . . . . . . . 61
Table 6.4 TSV Parameters and Minimum Wire Spacings . . . . . . . . . . . . . . . . . 64
Table 6.5 Timing and Power Statistics for First 10 Transactions . . . . . . . . . . . . 67
Table 6.6 Power Consumption reported from CACTI for 8ns clock period . . . . . . . 68
Table 6.7 Average Modeling Accuracy . . . . . . . . . . . . . . . . . . . . . . . . . . 70
Table 6.8 Total Synthesis and Route time . . . . . . . . . . . . . . . . . . . . . . . . . 73
Table 6.9 Speedup of Power Computation . . . . . . . . . . . . . . . . . . . . . . . . . 73
Table 6.10 Total Power Consumption Comparison: 2-D vs. 3-D . . . . . . . . . . . . 75
LIST OF FIGURES

Figure 2.1  Overall flow of processor power modeling in *PowerDepot* [17] . . . . . . . 9
Figure 2.2  a) HTLP example. b) SystemC code for selective synchronization . . . . . 10

Figure 3.1  Top-level Methodology Flow . . . . . . . . . . . . . . . . . . . . . . . . . . 16
Figure 3.2  Inside TLM Core . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
Figure 3.3  Tag Generation inside TLM Generic Payload . . . . . . . . . . . . . . . 22
Figure 3.4  Programming Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

Figure 4.1  Synopsys Library file format . . . . . . . . . . . . . . . . . . . . . . . . . . 29
Figure 4.2  Using Interconnect Sub-blocks for Different Memory Divisions . . . . . . . 34
Figure 4.3  Clustering Command Snapshots . . . . . . . . . . . . . . . . . . . . . . . . 37

Figure 5.1  Module-focused Programming . . . . . . . . . . . . . . . . . . . . . . . . . . 41
Figure 5.2  Thread-focused Programming . . . . . . . . . . . . . . . . . . . . . . . . . . 43
Figure 5.3  Assignment of Interconnect Sub-blocks into MDI Module . . . . . . . . . 45
Figure 5.4  Code Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
Figure 5.5  Virtual Coordinate Assignment . . . . . . . . . . . . . . . . . . . . . . . . 49
Figure 5.6  Floorplanning for Logic and Memory with TSVs . . . . . . . . . . . . . . 51

Figure 6.1  THEIA GPU Core Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 55
Figure 6.2  Schematic of Adder Sub-block with 6ns clock period . . . . . . . . . . . . 58
Figure 6.3  Schematic of Adder Sub-block with 10ns clock period . . . . . . . . . . . . 58
Figure 6.4  Schematic of Multiplier Sub-block with 2 pipeline stages . . . . . . . . . 58
Figure 6.5  Schematic of Multiplier Sub-block with 4 pipeline-stages . . . . . . . . . . 59
Figure 6.6  ALU and Memory Access Sub-blocks . . . . . . . . . . . . . . . . . . . . . 62
Figure 6.7  Layout used for Analysis of Routing Congestion . . . . . . . . . . . . . . 63
Figure 6.8  Command Cluster for the Cross Product Instruction . . . . . . . . . . . . . 65
Figure 6.9  Layout of a GPU Core . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
Figure 6.10 Fraction of Reconfiguration Cost for a) module-focused model, b) threadfocused model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

Figure A.1  Using Index List to Divide Workload between CPU and GPU . . . . . . . 89
Chapter 1

Introduction

An ongoing discussion on 3-D integration of ICs has been active for the past few years as we move towards smaller technology nodes. Specifically, the discussion on the effectiveness of TSV has been active. The TSV technology is currently considered as one of the emerging solutions that can improve electronic system performance in terms of data bandwidth, timing and power consumption. With TSVs, several issues make case for 3-D integration. First, interconnect lengths are reduced with vertical stacking of wafers or dies, reducing interconnect delay. Second, off-chip memory bandwidth significantly increases as vertical vias between the logic and the memory allow construction of wide I/Os. Third, power consumption of memory, on-chip and off-chip interconnects are reduced. Fourth, area reduction is achieved with TSVs for memories since stacking multiple memories with identical sizes can be achieved. Such stacking cannot be achieved with current wire-bonding techniques [13, 14]. Case studies in [13, 46] show evidences of 3-D integration benefits by reducing interconnect lengths and modifying data paths of the 3-D architecture.

Benefits of TSVs, however, are faced by two major challenges. First challenge is the high fabrication and design cost of TSVs. High TSV defects caused by TSV misalignment and vertical height variations contribute to challenges in yield results [24], potentially increasing the fabrication cost significantly. The design cost also increases due to the increased need for
managing thermal issues and increased noise coupling associated with TSVs [50]. The second challenge is the increase in design complexities due to expansion of interconnect dimensions, going from 2-D to 3-D. Design complexity increases as extensive computer-aided design (CAD) support is required to design interconnects and logic/memory modules in both 2-D and 3-D environments. Such support must consider thermal issues and define additional standards in interconnect settings in order to enable TSV integration.

1.1 Problems Addressed in This Study

In this dissertation, we propose solutions to address these two challenges with TSV integration. In order to solve the first challenge, we choose two major driving issues: power consumption and memory bandwidth. For these two issues, we select a specific processor unit where the TSV integration with wide I/Os can significantly improve both power consumption and memory bandwidth of the processor, enriching the motivation appeal for the TSV integration. Therefore, we choose mobile graphic processor unit (GPU) used in smartphones and tablet PCs that are considered commercially popular worldwide. The market for smartphones and tablet PCs are expected to grow in a very fast rate, as they are considered as the next-generation computing devices. Their sales have increased from 170 million to 320 million over the last three years. For the future, tablet market is particularly promising as 62 percent of growth is expected for the next three years [11].

Mobile GPUs used in smartphones and tablet PCs require a very high power efficiency since the power budget of mobile devices is a lot more restrained than that of desktop or laptop devices. Smartphone GPUs also require a high data bandwidth for two reasons. First, high memory usage is necessary in order to process high resolutions in user interfaces and games. Traditionally GPUs are used only for accelerating 3-D graphics applications such as games, but recently GPUs are also responsible for rendering and compositing user interface in operating system (OS). Such operations can allow smoother interface processing than using resource-constrained central processing unit (CPU) [23]. Second, the size of on-chip caches in mobile
devices are limited. This means that it is inevitable to move the majority of memory contents used in graphic processing to off-chip memories, increasing the amount of traffic between off-chip memory and GPUs. Since off-chip memories consume higher power consumption than on-chip memories, a technology suitable for mobile devices needs to reduce power consumption of memory and I/O and substantially improve data bandwidth at the same time.

In order to solve the second challenge, we target simplification modeling as used for analysis of improvements in power consumption achieved by the TSV integration. Before implementing in circuit-level or register transfer level (RTL), developing a design methodology at system-level reduces design complexity by hiding design details at low abstractions. Therefore, we introduce a SystemC TLM based methodology for the power prediction of TSV integrated designs, where the designs contain GPUs and low power mobile memory. In this methodology, we also target flexibility in our model such that additional architectural information or standards in interconnects associated with TSVs can be easily migrated into the model. Moreover, we add features such that power analysis can be thermal-aware with the inclusion of area and power density components.

1.2 Main Contributions

We use three performance metrics to evaluate the performance of our TLM model: 1) the total CPU time to complete the entire TLM execution, 2) the accuracy of power consumption against the GPU model’s RTL counterparts, and 3) reconfiguration cost of the TLM model in terms of the amount of SystemC codes. With improving these three metrics in mind, our TLM methodology makes the following contributions:

1. Based on our knowledge, this is the first 3-D power prediction methodology that executes the computation of power consumption entirely in TLM. To achieve TLM execution with a high accuracy power prediction, we create a C++ based power library outside the TLM core. This library stores all RTL samples of logic, memory and interconnects in
the architecture that are used as reference models. By separating RTL-based reference samples from the TLM core, we can achieve the reduction of CPU time while keeping the accuracy high. Specifically for the power computation, we achieve the runtime reduction of more than 93 percent with less than 8 percent degradation in the power prediction accuracy.

2. This methodology also introduces floorplanning inside the TLM core that facilitates an accurate computation of interconnect power consumptions for both TSV-integrated and non TSV-integrated designs. We place the reference samples in the floorplan, with the focus on saving the length of interconnects involved in the communication of logic and memory. After the placement, we measure interconnect power based on distance among all blocks.

3. We introduce thread-focused programming to reduce the reconfiguration cost required in the TLM core when new architectural configurations are introduced. This programming technique assigns each execution thread to keep track of all power information dedicated to the thread, instead of using execution modules defined in SystemC level. Furthermore, we implement a multi-dimensional interconnect (MDI) module to accommodate different interconnect dimensions, allowing less reconfiguration of SystemC codes when implementing both 2-D and 3-D versions of a GPU model. With these techniques, we successfully reduce the reconfiguration cost by more than 20 percent.

1.3 Overview of the Future Chapters

The rest of this dissertation is organized in six more chapters and one appendix. Chapter 2 elaborates further on how TSV integration can benefit the performance of mobile GPUs using a preliminary investigation. In this investigation, the savings of power consumption and computing resources of mobile GPUs are estimated based on industry parameters. In addition, in this chapter we summarize previous contributions of other researches related to system-level
power predictions.

Chapter 3 explains the top-level methodology flow, and the division of work between the physical design flow and the TLM simulation of power prediction. Chapter 4 gives an overview of TSV wide I/Os used in our methodology and detailed steps required to convert power consumption data obtained from RTL physical design flow to TLM input parameters. Detailed steps in executing TLM core and evaluating power savings at transactional level are described in chapter 5.

Chapter 6 presents a case study of our prediction model with a simple GPU model and TSV parameters from industry data. Chapter 7 concludes this paper with a summary of our research work and experimental results. In appendix A, we also provide few general-purpose GPU (GPGPU) programming techniques that can be used in system-level design. Although the contents in the appendix is not directly related to the power prediction itself, such techniques can contribute to further reduction of CPU time in any system-level model of ICs close to ours.
Chapter 2

Preliminaries

In order to understand how the TSV wide I/O technology benefits the power and memory bandwidth performance for mobile GPUs, it is important to have a basic understanding of the capabilities of mobile GPUs in terms of both power consumption and computational capacity. Therefore, we introduce a simple investigation measuring the savings in power per unit bandwidth by TSV integration. Furthermore, we review previous literatures on system-level power predictions, summarizing how other researchers approached the implementation of power prediction methodologies. The applications used in these literatures may not be directly related to mobile GPUs or other mobile devices. However, we describe these system-level design techniques under various levels of abstractions so that we can explain how the roots of our main ideas were derived compared to other studies.

2.1 Investigating and Estimating Benefits of TSV Integration

Current mobile memory devices such as low-power double-data rate (LPDDR2) memory at low voltage of 1.2 Volts produce the I/O power per unit bandwidth of approximately $8mW/Gbps$ [7, 42], where the unit bandwidth is bits per second. With TSV wide I/Os, the I/O power per unit bandwidth is reduced to less than $1mW/Gbps$ as the total bandwidth is increased 8 times greater than that of LPDDR2 with package-on-package (PoP). Specifically,
Table 2.1: Power Savings and Computing Resource Gains Achieved by TSV Wide I/Os

<table>
<thead>
<tr>
<th></th>
<th>Polygon processing rate (MPoly/sec)</th>
<th>Fillrate (Gpixels/sec)</th>
<th>Resource power (mW/FLOP)</th>
<th>Power savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>without TSV</td>
<td>532</td>
<td>16</td>
<td>0.277</td>
<td>-</td>
</tr>
<tr>
<td>with TSV</td>
<td>695.44</td>
<td>20.92</td>
<td>0.212</td>
<td>23.5</td>
</tr>
</tbody>
</table>

the bandwidth increases from 12.8 Gbps to 102.4 Gbps. Moreover, the total I/O power is reduced from 176 mW at 400MHz to 44 mW at 200MHz [39, 42].

Based on these data, we conclude that the I/O power per unit bandwidth can be as low as 0.42 mW/Gbps, provided that all I/O resources in TSV wide I/Os are utilized. For our investigation, we set the average reduction in I/O power per unit bandwidth by TSV integration to 6 mW/Gbps, comparing the best possible I/O power under PoP with the TSV wide I/O power of approximately 1 mW/Gbps. For the mobile GPU used for calculating the power savings estimation, we use PowerVR SGX MP8 GPUs [16] containing eight GPU cores sharing a single texture cache. This GPU operates at the maximum of 400MHz. Based on the PowerVR specifications, the reduction of I/O power alone can allow 31 percent more computing resources for graphic processing than the current resources with the same power budget of 2 Watts. This means that under the same power consumption, more floating point operations per second (FLOPs) can be executed for filling pixels or processing polygons. In addition, 23.5 percent of power consumption per unit bandwidth is reduced. Table 2.1 summarizes these numbers. With additional architectural modifications to this GPU model, further power savings can be achieved with the TSV wide I/O integration.

2.2 Related Works

In this section, we describe works on system-level power prediction methodologies from other researchers. Other works implement methodologies in different levels of abstractions, including bus-cycle accurate (BCA), TLM, and RTL. There are several different applications used for
their models, including System-on-Chip (SoC) peripherals, data buses transferring to dynamic random access memories (DRAMs), SoC processor cores in both single-core and multi-core configurations, and bus matrix communication architectures.

### 2.2.1 BCA Methodologies

Power modeling in [15] evaluates the power and timing performance of 3-D stacked DRAMs for mobile applications, with a BCA SystemC DRAM model. This power model focuses on breaking down the power consumption of memory and I/Os accounting memory state and workload required for the memory processing. Parameterized current usages for memory are used to estimate bus power and internal DRAM power for different bus communication protocols. To record user statistics of the memory, in-house software tools are used to insert direct memory access (DMA) transfers and parallelize the modeling code according to user-specified configurations. Memory statistics are then accurately obtained through a trace file containing all DMA transfers across the Advanced Microcontroller Bus Architecture (AMBA) bus to the DRAM. This model is the most similar study to our research, except that it uses BCA instead of TLM and focuses heavily on capturing memory and I/O power consumptions.

*PowerDepot* [17] generates intellectual property (IP) based power monitors that are embedded into the SystemC design to analyze power consumption in a multi-core SoC featuring Very Long Instruction Word (VLIW) processors. Energy and power models are created from a complete list of switching activities in the SoC, and is organized at instruction-level or pipeline-stage-accurate level. With an energy matrix organizing these power data, *PowerDepot* uses an extension of BCA SystemC modules to execute the calculation of the matrix as the time of SystemC simulation progresses. For the update of the energy matrix, linear regression is used. Figure 2.1 illustrates the overall flow of this modeling.

In an alternative approach, the power consumption of SoCs are estimated with the impact of process variations in mind [6]. Power-state based leakage modeling and Monte Carlo sampling are used to generate SoC power distributions and variability traces. Slightly compromising mod-
eling accuracy to meet a reasonable time limit, this methodology captures inter-dependencies between power profiles and variations in device parameters.

### 2.2.2 TLM methodologies

A TLM methodology introduced by Narayanan et al. [29] accomplishes the estimation of power consumption of peripherals in a SoC. In this study, power characterization is done by generating a hierarchical transaction level power (HTLP) tree structure that captures power information for a particular processor core. HTLP is then used alongside system netlists to derive average power numbers corresponding to each node of the tree. An example of HTLP is illustrated in Figure 2.2 (a). The tree is organized into four different levels, where higher levels represent function based transactions and lower levels represent the transactions closer to actual transactions with all phases available. Simulation speed is considered more important than accuracy in higher levels. The nodes in this structure denote a power model that can be used within the transaction level simulation platform, and the edges denote containment relationships between the nodes.

For comparing tradeoffs between power and timing at the system level, the modeling and simulation techniques shown in the multi-accuracy power and performance TLM [1] is intro-
duced. Using a modified version of SystemC kernel that extends TLM, switching of models at runtime is enabled. This feature allows users to maximize simulation speed during an interval that does not hold a detailed simulation analysis. To implement a generic multi-accuracy module, selective removal of synchronization of the module is used, as shown in Figure 2.2 (b). When clock wait state is omitted, internal synchronization of the model is maintained using dynamic sensitivity lists and synchronization events that are not dictated by clock wait. As a result, the TLM model can run as fast as running C++ codes depending on the frequency of clock wait use.

In addition to TLM, a hybrid abstraction level may be used to implement power macromodels. In Communication Architecture Power-performance Synthesis (CAPPS) [34], the power-timing design space of different bus matrix communication architectures are explored using an energy model based on transaction-level bus cycle-accurate simulation environment (CCATB) [33]. CCATB bases its simulation abstraction at a BCA granularity, but uses only transaction calls instead of signals to obtain simulation speed over pure BCA models. As a starting point, a system testbench consisting of masters and slaves interconnected using the AMBA bus matrix generates traffic patterns of different transaction modes and bursts. Then a full physical design

Figure 2.2: a) HTLP example. b) SystemC code for selective synchronization

```c
void multiLevelWait()
{
    static int speed = DEF_SPEED;
    static int counter = 0;
    if (speed > 0 && counter < speed){
        counter++;
        return;
    } else {
        counter = 0;
        wait();
    }
}```
flow using logic synthesis and RTL simulation is executed alongside multiple regression analysis to extract energy macro models. Finally, the CCATB model is repeated to estimate power consumption of bus communications until the number of solutions required in the output set is satisfied.
Chapter 3

Methodology Overview

Considering the tradeoffs between speed and modeling accuracy, we target to minimize the total CPU time of our power prediction model while conserving a high percentage of modeling accuracy. Specifically, our goal is to reduce the model runtime to few minutes while keeping the accuracy above 90 percent.

In addition to these constraints, we aim to minimize the amount of SystemC codes required to insert when new configurations are introduced in the modeling architecture. As TSV wide I/O integration adds vertical interconnects into conventional 2-D IC design, the prediction model should be flexible to different methods of evaluating the power performance. For some parts of the architecture being modeled, energy and timing parameters accurate to given TSV setting should be fully applied. For some other parts where logic-to-logic transactions dominate logic-to-memory transactions and the use of TSV modeled memory is minimal, the TSV setting should be eliminated without manual modifications to the architecture.

In order to meet all goals specified above, we divide the required work in power prediction to two stages. The first stage is named the setup phase. In this phase, input power consumption data for the TLM core are produced, where the TLM core denotes a set of SystemC TLM modules that execute the power prediction based on TLM function calls and synchronizations. Improving the modeling accuracy against the RTL model is the key objective. The second phase
is named the TLM simulation phase, where its key objective is to reduce the total CPU runtime. Throughout the two phases, we reduce reconfiguration cost of power parameter generation and TLM data structure management.

3.1 Setup Phase

In order to reduce runtime and complexity of our prediction model significantly, architectural-level details defined in our TLM core are entirely in transaction level and are not cycle-accurate. However, the accuracy of the power consumption data may be at risk due to the lack of cycle-accurate information on energy and timing parameters required to compute power consumptions. Especially, the power consumption of non-memory standard cell logics that may be connected to TSV wide memory must be measured with a cycle-accurate calculations so that the effects of the 3-D integration at logic tier can be measured accurately.

Therefore, we choose to incorporate parts of the physical EDA design flow in pre-fabrication RTL. Components of the physical design flow we choose to use are 1) logic synthesis based on commercial synthesis tools such as Synopsys Design Compiler®, 2) place and route tools such as Cadence Encounter®, and 3) Standard cell technology library. These components are necessary in order to obtain base power consumption parameters used in the TLM core, because technology nodes and power calculation methods used in commercial tools determine the RTL power results. However, synthesis and routing may take a long time to complete, where the time required to complete the TLM core may be significantly dominated by the synthesis and routing time. In this case, the size and complexity of logic being synthesized and routed strongly dictates the total time. In order to eliminate this possibility, we decompose the logic into small sub-blocks so that the time required to complete synthesis and routing of one sub-block is very small. The maximum size of a logic sub-block must be small enough that the total runtime used to synthesize all sub-blocks do not overwhelm the overall CPU time taken to complete the TLM core simulation. Therefore, we only incorporate logic that only uses simple arithmetic.

The introduction of physical EDA design flow provides additional advantage in that it may
reduce the learning curve in system-level design for both hardware and software designers. Hardware designers who have little knowledge regarding SystemC programming or other BCA programming techniques can solely focus on creating the sub-blocks in more sophisticated way so that the sub-blocks can be used effectively in the power prediction methodology. Similarly, software designers do not have to implement the model as detailed as BCA. Behavioral functionalities for bus communication at TLM would be sufficient and the coding techniques in high level languages can be applied in a more convenient manner than in BCA environment.

The incorporation of physical design flow into C++ modules is executed in the power library. The power library is a module that stores all histories of power calculations based on standard cell logic library power attributes, synthesis of logic sub-blocks, library parameters extracted from memory and interconnect models, and past TLM simulations. The power library consists of the base sample section that stores newly created gate-level designs for referencing synthesis-based power consumption, and the base parameter section that organizes all power consumption parameters that are directly used in the TLM core. For the top level methodology, we first explain all components related to RTL physical design flow, and then explain all components related to the setup of current TLM simulation and the usage of past TLM simulations.

3.1.1 Physical Design Flow

The top right corner of Figure 3.1 displays all components involved in the physical design flow. All ellipse shaped components are in Verilog or Synopsys library format. For each standard cell defined in the library, two attributes are obtained: leakage power values for computing static power consumptions, and internal power lookup tables for computing dynamic power consumptions. Any libraries from various technology nodes can be used in this methodology as long as they follow the Synopsys library format. The uniformity in the library format is required to allow automatic reading of power parameters within a reasonable runtime. Variations in such format causes several compiler directives to read different kinds of formatting information, increasing runtime unnecessarily. In the power library, the default leakage power values and
all elements in the internal power lookup tables are stored initially. Update of these data are executed as steps defined in the setup phase progress.

In the power library, we also create and store a set of synthesized logic sub-blocks commonly used in any GPU architectures. A common logic sub-block may be in one of the three following types: 1) a simple arithmetic logic unit (ALU) such as an integer adder, comparator, or a multiplier composed of multiple adders, 2) a multiplexer, and 3) a flip-flop that does not contain any arithmetic logic and update data only at the positive clock edge. These sub-blocks also include a single clock signal commonly used throughout all sub-blocks in the GPU logic. We focus on sampling as many sub-blocks related to ALU operations as possible, since the majority of power consumption in GPUs comes from the computation extensive ALU. Further details in generation of these sub-blocks and the simplification of their structures are described in chapter 4.

For predicting power consumption of mobile main memory and its interconnects, we use power consumption data provided by industry specifications. This option is chosen over the option of predicting average power of each memory wire and transceiver capacitance [15], since we omit cycle-accurate power information in order to run our TLM simulation a lot faster than BCA models. We also use power calculation formulas based on industry documents, where we measure the frequency of read and write access of mobile memory by tracking GPU input patterns. For calculating power based on the formula, Several current parameters in the LPDDR2 datasheet such as read and write burst currents, active precharge current, and standby current are used as base variables.

For predicting power consumption of on-chip memory and its interconnects, we use CACTI to extract both static and dynamic power parameters. Since GPU on-chip memories such as general purpose register files and texture caches are implemented as static random-access memories (SRAMs), unit power parameters obtained from memory compilers can be directly used to predict the on-chip memory power consumptions. Power prediction of on-chip non-memory interconnects is executed using routing tools alongside the synthesized logic sub-blocks. Since
interconnect power consumption is dependent on wire length, the parameters related to wire lengths are stored in the power library. These wire lengths are used alongside area information obtained from logic synthesis to predict floorplanning inside the TLM core.

### 3.1.2 System-level Configurations

The top left corner of Figure 3.1 shows components that configure system-level definitions as the inputs of both the TLM core and the power library. All rectangular shaped components are in
SystemC or C++ format. The instruction set architecture (ISA) library generates instructions based on selected ISAs. By default, the ISA library generates 32-bit ISAs based on ARM architecture. Users have an option to add custom 32-bit or 64-bit ISAs compliant to load-store architecture, the architecture commonly used in multi-core architectures and AMBA-compliant architectures [8].

The user configuration module sets up architectural environment such as the number of process cores used in logic design, the number of bits in ISA, memory burst, and memory partitioning mode that determines the number of memory banks per processor core. In addition, the number of adders, multipliers, and comparators used per core, and floorplanning settings are configured for the power library module. For the initial floorplanning, virtual coordinates are introduced in the TLM model in order to predict the interconnect power consumption using routing distance between logic blocks. In a 5 X 5 grid, users can choose which grid should be used for floorplanning of processor cores, caches, and mobile main memories. The actual coordinate values for grid boundaries are determined when the TLM core receives area information from the executions of components related to the physical design flow. When there exists past TLM simulation data, they are also used in the floorplanning configuration of the power library as well as for the update of power consumption values defined in the library. Past TLM simulation data include power consumption data for all logic sub-blocks and memory blocks defined in the library, area information in terms of virtual coordinates placed in the TLM core, and timing information that may update interconnect delays used for the computation of interconnect power at both 2-D and 3-D environments.

3.1.3 Command Clustering

In order to translate power consumption data from the power library into inputs of the TLM core, command clustering is implemented. The main motivation in organizing sub-blocks into commands at instruction level is to provide a way to organize power parameters in terms of TLM transactions. Specifically, command clustering allows power computation in the TLM
core by only accessing GPU instructions and indices of input data arrays. Such organization reduces the programming effort required to create TLM input parameters and complete a single transaction, simplifying the TLM structure and decreasing the reconfiguration cost. In order to allow more flexibility in the user configurations, commands created in this step are compatible to any kind of load-store ISAs.

For each instruction in the ISA, a cluster of commands is created to store all power consumption data created from the RTL design flow. Each command in the cluster executes a specific stage of logic design, data transfer by interconnects between logics and memories, or a specific memory operation. Corresponding power data for each command are stored by accumulating power consumption of all logic and memory blocks used for the execution of the command. For each cluster, a single TLM transaction is executed where each command of the cluster is mapped to each operation executed inside SystemC TLM modules. This means that the volume of computation and computation details of TLM core operations are controlled by those of commands in the cluster.

The outputs of the command cluster module consist of three elements. The first element includes an array of command clusters including power data that will be used in the TLM core, and the array size. The second element includes an array of timings for read and write access executed by memory commands, and for the synchronization of the TLM core. The final element includes an array of virtual coordinates organized in the granularity of logic or memory command blocks.

In order to eliminate confusion among blocks, sub-blocks, and command blocks, Table 3.1 summarizes definitions of these terms. Inside the TLM model, these terms are denoted by different C identifiers. The identifiers for blocks include FETCH, DECODE, CONTROL, ALU, ACCESS (access to both I-O and memory units), ONM (on-chip memory), and OFFM (off-chip memory). The identifiers for sub-blocks are explained in chapter 4. The identifiers for command blocks are identical to those of sub-blocks for two kinds of command blocks: 1) memory command blocks, and 2) ALU command blocks that directly correspond to simple
Table 3.1: Blocks, Sub-blocks, and Command blocks

<table>
<thead>
<tr>
<th>Terms</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>Denotes major GPU modules in the architecture, such as an instruction decoder, controller, ALU, and on-chip cache modules</td>
</tr>
<tr>
<td>Sub-block</td>
<td>Denotes a small logic with a single stage, where a stage takes one or more clock cycles to complete. Smaller in size than a block.</td>
</tr>
<tr>
<td>Command block</td>
<td>Denotes a block that represents a single command defined in the command cluster module, where multiple sub-blocks may be included. Smaller in size than a block, but larger than or equal to a sub-block.</td>
</tr>
</tbody>
</table>

ALU operations such as add and multiply. For other command blocks, the identifiers used in the TLM code are REQ (request), RSP (response), SEL (select), GS (global schedule), LS (local schedule), IOA (I-O access), MEA (memory access), and FSM (Execution FSM).

### 3.2 TLM Simulation

As illustrated in Figure 3.2, the TLM power prediction model is composed of SystemC modules, where each module represents a processing element (PE), a memory block, or a set of interconnects. In the figure, all arrows represent C++ function calls. Arrows between square sockets represent SystemC module specific function calls. A set of multiple SystemC modules are organized as a process core or a memory block, capturing the logic and memory unit at the largest encapsulation. PEs function as transaction initiators that start function calls by passing transaction data structure with TLM 2.0 standard [31]. PEs include instruction scheduler at both global and local scopes, memory access block, and ALUs. For ALUs, we initially assume vector processing units where a single instruction is executed by multiple arithmetic units in parallel. Memories such as register files, caches, and main mobile memories function as transaction targets that respond to initiator function calls by updating data pointers and the response status. Interconnects are responsible for all data transfers between PEs and memory blocks. For the data transfers, approximately timed (AT) model is used to depict synchronous data
transfers. By using AT model, the interconnect models are kept synchronized to a common transactional time rather than allowing each interconnect to run ahead in its own time. Although such synchronization may require extra execution time, it allows the execution threads running in parallel to access and share memories coherently. Even with this feature, AT model takes less CPU time than BCA models.

The main communication protocol used in the model is Advanced eXtensible Interface (AXI) defined in AMBA 3 specification [40]. This protocol is widely used in ARM Cortex-A9 based smart-phones or other mobile devices. In AXI, we use four phases to support AT model: begin
request, end request, begin response, and end response. To provide high parallelism and low-cost DMA in data transfer, four different logical unidirectional channels are featured: address channel, read channel, write channel, and write response channel. While out-of-order transaction issue and completion are featured, this four-phase structure supports the AT model and allows safe depiction of thread synchronization by supporting exclusion rules for request and response. This means that there are no more than one request or response in progress through data socket at any instant. Simultaneous read and write of a channel supported by AXI are depicted by updating the end response times to the same value for both operations.

For the main data structure of the TLM model, a generic payload defined by TLM 2.0 [31] is used for both memory-mapped and non memory-mapped transactions that define GPU instructions. Originally, TLM generic payloads only support memory-mapped transactions such as memory read and memory write, but we extend the usage of the payloads by including tags inside data pointers that indicate the nature of transaction and the power consumption associated with the transaction. For non memory-mapped transactions, we widen the definition of memory write or read by including the register file as one of memory blocks. Since the ISA we use is a load-store architecture, all instructions with the exception of unconditional jump (JMP) or no operation (NOP) must use either a register file, cache or main memory to access data. For JMP or NOP instructions, we assign TLM IGNORE COMMAND defined in TLM 2.0 to indicate that there is neither a read or write operation in this instruction.

Figure 3.3 shows a flow chart describing how the tags are used to process transactions. There are three tags: memory tag, register tag, and control tag. Memory tag indicates whether the transaction is memory-mapped or not. The next two tags are used to assign correct tasks to non memory-mapped generic payloads. Once the instruction reads the register file for retrieving register data values, the register tag is marked. Once the register tag is marked, the control tag is checked to see if the transaction carries control instructions such as conditional branch. If the control tag is turned on, this indicates that only comparators are used for comparisons of two register values and no register file write is required since only the jump address calculation
is executed to update the program counter (PC). If the tag is turned off, this indicates that register file write must be executed for arithmetic instructions. In this case, an additional power consumption for writing to the register file is accumulated, and the delay for the next transaction is also stored in the data pointer in order to correctly depict the timing information in the TLM core.

The TLM simulation outputs timing and power consumption for all active commands executed in the simulation. The power manager shown in the bottom part of Figure 3.1 gathers all power statistics and creates a final report on the power consumption and the modeling accuracy by comparing the RTL results. The power library is then updated by power consumption and area data of the arithmetic units and flip-flops generated from the RTL, accumulating the simulation history table under instruction-based and coordinate-based categories. Thereby, the
power manager is considered as a tool for enriching the power library by building more resources available for the setup of input parameters used in future TLM simulations. It should be noted that the update of the power library, as shown in the rightmost part of Figure 3.1, is dependent upon the user’s choice. After this step, a single run-through of the prediction methodology ends.

As a supplement to Figure 3.1, Figure 3.4 describes the programming flow of this prediction methodology. Data width parameters denote the number of bits for input and output data for logic, on-chip memory, and on-chip interconnect sub-blocks. In addition to these parameters, standard cell library and industry parameters are two sources used for extraction of power data for small buffer or AOIs and off-chip memory, respectively. Automated processes before the TLM execution are in two categories. The first category includes tool-based processes that are executed for power report generations. The second category includes C++ based processes that are used to generate input text files for the next process. After the TLM execution finishes, then the power prediction results from both TLM and RTL are compared for the analysis of the prediction. It should be noted that the steps required to generate final RTL results are omitted because they are not the part of this methodology’s novelties. The RTL process steps consist of synthesis, placement, route, and recording of switching activities for the entire GPU.
Figure 3.4: Programming Flow
Chapter 4

Power Prediction Setup

In this chapter, we present detailed procedures for computing power consumption prediction data for GPU units and mobile memories by incorporating RTL physical design flow in the setup phase. First, we describe the power computation of logic and memory units without consideration of TSV insertions. Next, we analyze how input parameters of the TLM core related to TSV wide I/Os are structured. Then we explain the power computation of interconnects affected by TSV insertions. Furthermore, we describe how command clusters are used to translate the RTL-based power data into input parameters inserted into the TLM model.

4.1 Power Prediction of Blocks at Logic Tier

As briefly introduced in chapter 2, we use both the standard cells defined in the technology library and the synthesized logic sub-blocks to predict power consumption of blocks at logic tier. Although each cell defined in the standard cell technology library contain power attributes, we cannot use them directly for logic transactions because multiple designs of a logic unit may exist based on different clock periods. For example, for a bitwise addition in logic synthesis under a relaxed clock period, a full adder defined from the library is directly used. However, as the clock period is reduced buffers and primitive logic gates such as and-or-invert (AOI) and exclusive-NOR are used to satisfy timing constraints. Such variety cannot be controlled with
the direct use of library attributes since the assignment of additional logic gates are dependent on the power computation algorithms of the synthesis tools. The library cannot accommodate these dependences without knowing complete details of the copyrighted algorithms. To address this challenge for predicting power consumption of logic cells, we execute four steps. These steps are described from subsection 4.1.1 to 4.1.4. Then we describe how the power consumption of memory blocks at logic tiers are predicted.

4.1.1 Creation of Sub-blocks

Sub-blocks used in our model are synthesized under different clock periods, data widths and pipeline depths. Then these sub-blocks are routed using Cadence Encounter®. ALU-type sub-blocks consist of simple arithmetic logics with a small number of pipeline stages, ranging from 1 to 4. Moreover, we only use fixed-point logic instead of floating-point logic for two reasons. First, the complexity of the sub-blocks are reduced so that the total time required to complete synthesis and routing does not exceed the TLM simulation time significantly. Second, fixed-point logic is a better fit for our model than the floating-point logic because the use of floating-point alongside vector ALUs is not suitable for mobile GPUs at this point due to high power consumptions. To use floating-point logic in mobile GPUs, a scalar ALU is used with write masking technique to achieve lazy vector processing [16]. We choose to use vector ALUs for exploiting a high data parallelism under wider memory bandwidth than that of scalar ALUs. Non-ALU typed sub-blocks such as synchronous multiplexers and cascaded flip-flops provide more options in choosing logic blocks that are used to characterize sub-blocks in the model. Furthermore, availability of such sub-blocks allow an easy variation of pipeline depths and conditional logics in the datapath of the ALU sub-blocks. The end products of this step are stored in the base sample section of the power library. Table 4.1 summarizes all logic sub-blocks that are independent of architectural configurations and used throughout our prediction procedures. In addition, C identifiers (CIDs) representing these sub-blocks are also listed in this table. The logic synthesis clock period for these sub-blocks range from 5ns to 10ns.
data width of the arithmetic units does not exceed 32 bits to limit the size of sub-blocks.

### 4.1.2 Dynamic Power Prediction

In order to set base factors for predicting dynamic power consumptions, we estimate input signal activities for the synthesized logic sub-blocks. For counters, we can simply calculate the toggle rate using the counting patterns. For making switching activity estimates of other logic sub-blocks that access texture cache, we can use the texture input patterns from GPU benchmarks, or randomize the input signal values within the range of the pixel values used in the texture cache. Similarly, we use the GPU instruction patterns from either GPU benchmarks or a randomized ISA pool to estimate switching activities of logic sub-blocks used for instruction fetch, decode and address calculations. After the input activities are annotated, we update switching activities of the logic sub-blocks according to the back-annotated switching activity interchange format (SAIF) alongside standard parasitic exchange format (SPEF) obtained from the routing of the logic sub-blocks. Since we are extracting SPEF and SAIFs for small sub-blocks separately, the total runtime required to complete this step is significantly less than that of the complete RTL.
4.1.3 Tool-based Power Measurement

For all combinations of different clock periods, data widths and pipeline depths, static and
dynamic power consumptions of the logic sub-blocks are extracted from Synopsys Design
Compiler® and stored in the power library. Since all sub-blocks are routed, all interconnect
inside the sub-blocks are also included in the power computation. Therefore, the TLM core is
not required to model interconnect power within these sub-blocks separately.

4.1.4 Standard Cell Library Use

Static power for each cell defined in the library is recorded in the base parameter section of
the power library. Also the average, minimum, and maximum values of internal cell power
consumption are recorded using internal power lookup tables in the library. These data are
used to create base power parameters so that they can be used to adjust power prediction data
based on the power statistics gathered among logic sub-blocks and past TLM power statistics.
Furthermore, they can be used to provide power consumption of extra glue logics that connect
logic sub-blocks in the TLM model. In the TLM code, two CIDs are used for these logics: BUF
representing buffers and AOI representing and-or-inverts.

Figure 4.1 illustrates an example of the Synopsys Library file format [43]. In addition to the
default value for static leakage power consumption, there exists multiple leakage power values
for each combinational standard cell, where each value represents the leakage power for each
input combination. We only store minimum, maximum, and median leakage values for future
reference when we compare them to the leakage power values of sub-blocks. For the internal
lookup table in sequential cells, we iterate every element of rise and fall energy values to obtain
minimum, maximum, and mean values.

4.1.5 Caches and Register Files

Some 3-D architectures with TSVs place caches on different tiers than logic tiers [48, 53] by
placing caches per each level in separate memory tiers. Our model simplifies such setting.
Processor caches at all levels and register files are located at a single logic tier, where only mobile main memories are placed in the memory tier. In addition, we set texture cache as the only cache that communicates with the mobile memory. In the TLM code, two types of CIDs are used to represent these on-chip memory sub-blocks: RF denoting register file, and TEX denoting texture cache.

CACTI 6.5, an extension of CACTI 6.0 memory compiler [27], is used to build a custom cell library for each cache or register file unit used in the model. The use of CACTIs allows a fast computation of memory power without considering synthesis of memory blocks using standard cells. We modify memory model generator script created in NCSU Electronic Design Automation (EDA) group to be compatible with CACTI 6.5. Then, we extract power consumption data directly from the newly generated library files. To predict dynamic power consumption...
of caches and register files, average read and write energy values in the internal power lookup tables are divided by clock periods, providing read and write power for different clock periods. Furthermore, cache input signal activities are estimated in the same fashion shown in subsection 4.1.2 to extract switching power data. For static power prediction, CACTI outputs are directly used. For small register files under the size of 2KB, we found some inaccuracies in internal power measurements from CACTI C++ program. Based on our observation, such inaccuracies are caused by formatting bugs in printing very small power values. For a value under $1\mu W$, the power values are printed in $nW$ where it should be in $\mu W$.

### 4.2 Power Prediction of Blocks at Memory Tier

For the mobile memory, read and write frequencies of the memory are initially computed because the frequencies are the input parameters used for the mobile memory power calculation methods in [44]. The frequencies are computed out of the total number of transaction cycles used in the TLM simulation, where a transaction cycle denotes a delay between two TLM modules that may take one or more clock cycles to complete. Read and write frequencies are dependent on the texture cache size used for graphic processing, and the memory bandwidth. Therefore, the architectural configurations become a major factor in deciding the mobile memory power consumption. We provide an option for users to insert current parameters from electrical specification as inputs to the power calculation. Then the power consumption prediction value for the mobile memory will be computed in the TLM core based on these parameters.

For the power computation of the PoP LPDDR2, we set the memory bandwidth to 32 bits. For TSV wide I/O memory, we allow its memory bandwidth to increase up to 512 bits where other bandwidth options include 64, 128 and 256 bits. The 512-bit configuration in TSV wide I/O memory [20] can support the same power consumption as that of the 32-bit PoP LPDDR2. For the 256-bit configuration, therefore, we set the memory power consumption affected by active and standby mode to half of those from the LPDDR2. Moreover, we set the refresh power to be reduced 33 percent from that of LPDDR2 with the help of dual period refresh
scheme [20]. These power parameters can be modified by users if different current or power parameters of TSV wide I/O memory are to be used in the model.

4.3 Analysis of TSV Wide I/Os

As we try to integrate TSV wide I/Os, additional factors must be addressed for predicting power consumption of the 3-D integrated design. Especially, the complexity in computing interconnect power with the vertical stacking of TSVs increases compared to that of the existing power prediction methods for 2-D designs. We evaluate both positive and negative aspects of such additional factors, and decide how they should be incorporated in our design in a way that the accuracy is improved without slowing down the system level simulation time significantly. On the positive aspects of the TSV integration, we must consider quantifying the interconnect length reduction due to the elimination of long cross-chip interconnects between logic and memory [18]. In addition, we must consider measuring how the interconnect length (wire length) reduction and architectural modifications contribute to the 3-D power reduction as suggested in [45]. On the negative aspects, we must measure the parasitic capacitance of TSVs since they cannot be considered negligible. For instance, the capacitance of a 50$\mu$m-tall TSV with area of $5\mu$m $\times$ $5\mu$m is equivalent to that of a 400$\mu$m-long interconnect in 45nm technology [19]. Moreover, we find that additional interconnect length needed to route around newly inserted TSVs must be taken into account to measure the 3-D effect of wire lengths more accurately. It should also be noted that thermal issues cannot be ignored in the analysis of TSV integration, so relating thermal issues with the analysis of 3-D power consumption is essential even though a direct thermal analysis is not executed.

In order to quantify interconnect length reduction and power reduction for TSV-integrated ICs, we organize a set of interconnects as a sub-block and extract power consumption of these interconnect sets using procedures that will be explained in the next section. These sub-blocks are available for both 2-D and 3-D architectures. For the interconnect sub-blocks that are placed differently for the 3-D architecture, we isolate these sub-blocks so that they are composed of
interconnects that solely communicate between cache or logic blocks and the mobile main memory. Then we update power consumption values of these sub-blocks by replacing the total interconnect parasitic capacitance with TSV parasitic capacitance. These sub-blocks are also used as facilitators to architectural modifications. Multiple schemes on memory partition [45, 47] in the existing architecture can be used to reduce power consumptions by decreasing capacitance per bitline in memory access and increasing memory bandwidth via parallel memory access. Currently, the total number of memory divisions is limited to 8 in order to accommodate limitations in the mobile power budget.

For the measurement of the parasitic capacitance of TSVs, we allow users an option to provide the capacitance value as an input parameter in the user configuration module. For the measurement of additional interconnect length required to route around TSVs, we insert the notion of coordinates in order to provide an accurately total area for the whole architecture. For this purpose, the following parameters are taken as inputs in the user configuration mode: 1) TSV width, 2) minimum spacing between TSV and interconnect, 3) minimum spacing between two interconnects, 4) minimum width of a metal interconnect (wire), and 5) a routing congestion degree. These five parameters are used to compute the additional total wire length required. The routing congestion degree can vary from 0 to 1. We assume that TSV area is TSV width squared, following the references of TSVs used in our model.

### 4.4 Power Prediction of Interconnects

We categorize interconnects in two: horizontal and vertical. As defined in [19], horizontal interconnects refer to wires at logic tier that their delays and power consumptions may be affected by TSV insertions. They require additional on-chip interconnects to get around TSVs. Vertical interconnects refer to wires that are not affected by TSV since they are used exclusively to communicate between logic and memory tiers. In order to predict interconnect power without having complete information on all interconnects in the architecture, we newly gather a small number of routing samples by synthesizing and routing sub-blocks in the basic sample section.
of the power library. There are two types in these routing samples: individual sample that connects two sub-blocks, and shared sample that connects more than two sub-blocks. A set of interconnects in each sample are characterized as a sub-block that are used to connect two or three logic or memory sub-blocks. Input signal activities measured in power prediction of logic sub-blocks are re-used to extract pre-route and post-route switching power consumptions, where the difference between the two denote the total interconnect power for all wires used to route multiple sub-blocks.

By using interconnect sub-blocks, we can realize different types of memory division by adding the sub-blocks when new wirings are required, as illustrated in Figure 4.2. For each PE block, a 16-1 synchronous multiplexer and a flip-flop with the drive strength of 2 are used to select the correct address to be read from the texture cache. Each rectangle labelled with an "I" denotes an individual sample, and a rectangle with an "S" denotes a shared sample. These labels are also used as the CIDs of these sub-blocks in the TLM code. An individual sample shown in the figure shown has the 32-bit data width, while a shared sample covers 64 to 256 bits depending on the memory division configuration. As shown in the figure, we can differentiate the design by simply changing the selection of interconnects used to connect the texture cache. When the shared interconnect is used to connect the cache, then we insert additional logic required to control the cache sharing among all processor cores.

In addition to computing interconnect power, we store the total interconnect length for each routing sample in the power library. By using the formula for interconnect power consumption [51], the total interconnect length $L_{tot}$ in terms of the interconnect power $P_{int}$ is computed as:

$$L_{tot} = \frac{20P_{int}}{C_iVdd^2f_c}$$

(4.1)

where $C_i$ is the distributed wire capacitance per unit length and $f_c$ is the clock frequency. $C_i$ is dependent on the process technology, varying from $248fF/mm$ at $90nm$ to $300fF/mm$ at $32nm$ [2, 38]. All components in the formula can be varied by user configurations.

With the computations for interconnect power at 2-D architecture available, we include
TSVs and additional wiring capacitance caused by TSV insertions. We set $37 fF$ [50] as the TSV capacitance with area of $5\mu\text{m} \times 5\mu\text{m}$. We set the additional interconnect length per net to $2L_s + W_{tsv}$, where $W_{tsv}$ denotes the TSV width and $L_s$ denotes the minimum required spacing between TSV and interconnect. This value is the maximal length required to route around the TSV when the TSV is located at the center of the routing path. For each interconnect sample obtained, we use the following formula to compute the interconnect power with TSVs:

$$L_{extra} = b(2L_s + W_{tsv})\left[\frac{(W_{tsv} - W_m)}{(W_m + W_{msp})}\right]$$  \hspace{1cm} (4.2)

$$P_{int3d} = \frac{C_i}{20}(L_{tot} + \frac{C_{tsv}}{C_i} + L_{extra})Vdd^2 f_c$$  \hspace{1cm} (4.3)

$b$ denotes the factor between $0$ and $1$ that determines the degree of severity in routing congestion. $W_m$ denotes the minimum width of a metal wire, and $W_{msp}$ denotes the minimum
spacing between two wires. $C_{tsv}$ denotes the TSV capacitance. $L_s$, $W_m$ and $W_{msp}$ may vary depending on different TSV bonding techniques and design rules. Default values for these three parameters are referenced from design rules defined in FreePDK45-3D, the 3-D version of FreePDK45 [41].

4.4.1 Assignment of Routing Congestion Degree

In order to estimate the routing congestion degree, we obtain some additional samples that show a logic layout. Creation of a layout is necessary since we need to see the actual physical routing result as a reference to determine the congestion degree. Therefore, we define a simple logic where a sub-block from ALU and memory access logic sub-blocks are included. The number of sub-blocks used for the memory access logic is less than 10 for the purpose of simplification and small estimation runtime. The sampling logic is a good candidate for the logic that may affected by TSV insertions, since the memory access logic is connected to ALU, texture cache, and the main memory. After the synthesis and routing, a layout is obtained using Cadence Encounter.

4.5 Using Command Clusters

After generating all logic, memory and interconnect sub-blocks used for the computation of power consumption, these sub-blocks are converted as command clusters so that they can be passed as input parameters to the TLM model. In the TLM model, we define a command as a part of procedures required to complete operations carried out by a single GPU thread. In addition, we define a module as a SystemC class instance that represents a block, where one command is characterized by one or more sub-blocks.

While it is acceptable to pass sub-blocks as input parameters to the TLM core, instruction-level command clustering further simplifies the execution process in the TLM core. Command clustering significantly reduces the number of array elements necessary to insert as input TLM parameters since not every detail from each sub-block does not need to be passed as parameters. For example, every instruction goes through instruction fetch and decode and it is repetitive.
to track the power consumption of all sub-blocks associated with these stages. In this case, we simply calculate the total power consumption required to go through these two stages and pass these data as only a single command. Such setting also reduces the reconfiguration cost during architectural modifications since there are less array elements to modify than having sub-blocks as elements. In addition, command clustering allows an easy translation between power analysis per sub-block and power analysis per instruction.

A Command cluster is generated by selecting multiple commands that are required to complete each instruction in the instruction list. There are four types of commands:

1. **Intra-module Logic Command**: This command executes a single stage processing logic inside modules, where one stage represents a part of a logic transaction that will be executed by logic sub-blocks. Depending on the pipeline depth of the logic sub-block, a single stage may take multiple clock cycles to complete, as shown in a 4-stage multiply command (MULT4) that takes 4 clock cycles. As shown in the top part of Figure 4.3, a command block takes one or more logic sub-blocks, and an interconnect sample that connects the command block to another.

2. **Intra-module Memory Command**: This command executes a single stage memory operation. The structure is similar to that of a intra-module logic command, but only read and write commands are supported. Read and write commands have two stages: address access and data access. Only caches, register files and mobile memories use this command.

3. **Intra-module Interconnect Command**: This command executes data transfers between sub-blocks within a PE. We label these commands uniformly with CONNECT. These commands are composed of interconnect sub-blocks created from logic synthesis and routing.

4. **Inter-module Command**: This command executes data transfers between PE and memories. We label these memory-mapped commands as either REQUEST or RESPOND. We use either CACTI or main memory interconnect information to feed power and timing val-
ues to these commands. Depending on the timing constraints, buffers from the technology library are included as a small logic sub-block.

Each command includes timing, power, and area parameters that will be used in the TLM simulation. The timing parameters are obtained from clock period, longest path delays reported during the synthesis of logic sub-blocks, and delays specified in the cache and mobile memory blocks. Power parameters are obtained from all power information of logic sub-blocks, memory blocks and interconnect samples obtained from previous procedures. Area parameters are

Figure 4.3: Clustering Command Snapshots

Cluster #1: \{MULT4, ADD1\_1, SELECT, REQUEST, ADDR READ, \ldots\}

Cluster #2: \{DIV3, ADD1\_3, CMP\}

Figure 4.3: Clustering Command Snapshots
included by adding all area of corresponding logic or memory sub-blocks and interconnect samples, where the area are obtained from logic synthesis or memory specifications. Such inclusion of input parameters is defined as a command snapshot, an attempt to capture the actual logic gates defined by the RTL.

Examples of command snapshots are shown in the bottom part of Figure 4.3. When all commands are defined, a cluster of required commands for each instruction is created and stored in a command cluster instance. This instance is accessed in the TLM core when an instruction is fetched and its execution starts. When assigning commands in a command cluster, we consider two tie-breaker cases. First, when there are multiple identical logic sub-blocks that can be included in the cluster such as ADD_1 and ADD_2, we randomly select one command and mark it as occupied so that the other clusters can take the unoccupied command. Second, when a command fans out to two different commands, we let all associated logic commands to be tied together to execute simultaneously, as shown in SELECT and CMP in the bottom part of Figure 4.3. These pair are executed when any of three add commands is executed in the previous transaction cycle. However, the next command to execute depends on the previous command of the cluster. Therefore, the first cluster in Figure 4.3 executes memory access commands when SELECT and CMP execute, but the second cluster does not because its previous command is not related with memory transactions.

When TSV insertions are considered, we only modify the computation methods in inter-module commands such as REQUEST and RESPOND. Additional intra-module interconnect commands needed for routing around TSVs are included in the inter-module command, since we classify that such addition is impacted by the interconnect between two modules.

The accuracy of power consumption from the use of command clusters depends on the richness of detail in the available sub-blocks. Provided that the actual power consumption of a logic sub-block measured by the RTL has the total of $N$ logic gates, the number of command snapshots $M$ can be between 0 and $N$. The reconfiguration cost for modifying architecture is reduced by organizing power data at command level because the management of these data
becomes independent of architectural configurations. Furthermore, commands consist of arithmetic and memory commands that are universal to any ISAs, eliminating necessity to modify any code based on ISAs.

For all command clusters used in the TLM core, we store the cluster contents in an 2-D array where a cluster representing a single GPU instruction is stored for each row. The number of commands is equal to the column size minus 1, where the column size is always stored at the column index 0 of each row. The capacity of the array is the maximum number of commands among all clusters.
Chapter 5

TLM Simulation

In this chapter, we first explain several programming techniques that enhance the simplicity and flexibility of the TLM model. Specifically, these techniques focus on modifying the transaction entity such that a single SystemC module can be used to implement all data required for both 2-D and 3-D architectures. Second, we briefly describe the basic structure of the TLM core with few code examples. Third, we introduce floorplanning setup schemes that allow assignments of virtual coordinates that are used to compute interconnect power consumption accurately. Finally, we explain how the power statistics gathered from TLM simulations can be used to improve the accuracy of future simulations using history tables in the power library.

5.1 Overview

Each transaction in the TLM model represents an active thread in processor cores. Using SystemC ports and synchronization methods, we model thread-level parallelism for single-instruction multiple-data (SIMD) architecture used in mobile GPUs. Since the main target of this model is to predict power consumption rather than depict an accurate execution of GPU instructions, we do not tie in parallel programming interfaces such as Portable Operating System Interface (POSIX) threads shown in SystemC-SMP [36]. Instead, we keep track of execution time for each thread and allow the tracking of such timing to be executed independently, and in
sync. For modules in the TLM model, a SystemC extended class called *PowerThread* enhances modeling flexibility by providing independence in management of execution threads. In order to develop 3-D environment in TSV defined interconnect, a new SystemC class named multi-dimensional interconnect (MDI) is developed to provide flexibility in interconnect modeling without reconfiguring the actual system descriptions of interconnect.

### 5.2 Thread-focused Programming

For each execution thread used in our TLM model, a generic payload is used to store transaction information such as instruction, data to be transferred, data width, and memory burst size. In addition to transaction details, power consumption statistics must be managed internally to report the intermediate and final power consumption results. One approach in storing and managing the computations of the power consumptions is to let each module handle the

![Figure 5.1: Module-focused Programming](image-url)
computations with its dedicated data structures under the power tracking class type POW, as shown in Figure 5.1. In addition to existing TLM functions to manage transaction payloads, each module stores a POW class instance that keeps track of all power statistics due to thread executions, and these results are reported during the middle or at the end of the simulation. The method to store these POW class instances can vary. A large array of POW can be created at the top-level SystemC power manager module which can access the POW instances using appropriate indexing schemes, or each POW instance can be accessed via pointers where a single linked list contains a list of pointers.

However, this module-based approach may increase the reconfiguration cost significantly. When power statistics are reported at the end of the simulation, the power manager must monitor all modules that have POW class instance as their members. Therefore, whenever new components are added to the model, a monitor that stores power statistics for the new component and reports to the power manager must be reconfigured. When power statistics are reported during the middle of the simulation, further problems arise as more time is required for intermediate monitoring. Such monitoring requires synchronization and intermediate function calls every time the statistics are reported, adding redundancy in simulation. To avoid such redundancy, additional arrays could be used to store timing required to synchronize with other threads with the same instruction group. However, the reconfiguration cost increases because additional mapping of waiting threads into corresponding transactions must be implemented for the timing array to work with POW class contents.

As an alternative solution, a thread-focused approach is taken where each thread is responsible for keeping track of its power consumption statistics, and the model representing modules are independent from the management of power consumption statistics. Each thread in a PowerThread instance remembers its execution path, storing power information of modules that the thread traverses. This thread instance is stored as a data carried in a transaction payload, as shown in Figure 5.2. The actual data from algorithmic computations are stored inside the data width spot of the payload. Data width and memory burst of each transaction
Figure 5.2: Thread-focused Programming

is stored as local non-static variables of a \textit{PowerThread} instance to allow update of these attributes during simulation. For each instruction, each thread associated with the instruction reports the power statistics to the power manager only when the thread terminates. Since all power statistics during the entire simulation run are stored in the path list of \textit{PowerThread}, synchronous monitoring or intermediate function calls among modules are not required.

The path list of \textit{PowerThread} stores the ID of modules that the thread uses for its execution, the power consumption statistics during the traversal, and a list that describes the method used for the computation of the power consumption. For consistency of AT synchronization, power consumption statistics also include the time stamp indicating the entrance and exit of the thread. Each element added to the method list indicates which unit power calculation was used whenever the power is incremented. In order to indicate that the current transaction is a thread waiting on other threads in the same instruction group, we set the size of the path list fixed and allow peaking of other threads’ execution status by calculating the offset of the heap location.
where other instances of *PowerThread* are stored. The fixed size is set to 100, assuming that the total number of active threads inside the mobile GPU does not exceed that number due to the fact that the number of cores in mobile GPUs are smaller than that of desktop or laptop GPUs.

### 5.3 MDI

The MDI module models all communications among all modules within the TLM model. The MDI first predicts the power consumption of data transfers between PEs and memory blocks, using input power parameters for potential vertical interconnects that can be substituted by TSV vertical wires. The MDI then predicts the power consumption of interconnects within a PE by taking an input array of interconnect power data from each PE block. These interconnects qualify as potential horizontal interconnects that can be affected by additional on-chip routing due to TSV insertions.

The MDI module receives the dimension mode and the logic-to-memory partitioning mode upon its instantiation. The dimension mode determines whether the MDI will be used for 2-D or 3-D settings. The partitioning mode determines how the interconnect will be mapped into the memory targets from initiators. Depending on the partitioning mode, the MDI can set each interconnect element as a single shared bus that connects to multiple banks of memory or multiple processor cores, or as multiple buses where each interconnect is dedicated to each memory. The number of interconnects partitioned for the MDI can be any positive integer, and all information dedicated to each interconnect is stored in the module.

The dimension, length, and transaction type of all interconnects within the MDI cannot vary, meaning that if one interconnect with a unit length carries a 2-D logic-to-logic transaction, all others must carry the same type of interconnect attributes. Variations in transaction type among interconnects can increase reconfiguration cost of the model significantly, because all power consumption calculation schemes must be managed individually for each interconnect. In addition, when the dimension or length is changed for parts of interconnects within the
MDI, the power calculation scheme must be modified for each interconnect. The only variation in transactions allowed for interconnects in the MDI module is memory type, provided that dimensions are uniform. One interconnect can be set for 3-D memory transactions from logic to cache, and another can be set for 3-D memory transactions from logic to main memory.

The number of interconnects inside each interconnect element defined in the MDI matches the number of interconnects included in an interconnect sub-block that represents the element, as shown in Figure 5.3. This figure shows two arrays, one dedicated to PE to memory interconnects, and the other dedicated to all interconnects within all PE blocks composed of logic sub-blocks. Two arrays display interconnect elements of a single PE block shown in figure 4.2. The label in each array element denotes the type of interconnect sample and the data width. The mapping shown in the figure allows a direct use of all interconnect power consumption calculations mentioned in chapter 4. When buffers are included in the interconnect, buffer power is also considered for power calculation using power values from the standard cell library.

Each interconnect stores its unique ID, initiator ID, and target ID. The IDs of initiator and target are later used for the analysis of the total power consumption of the entire block.
that interconnect connects. After all threads finish their executions and all timing and power information are reported to the evaluation stage of the model, the power information for all interconnects will be sorted in the ascending order of the power consumption based on the IDs of their neighboring components at each time interval. Based on this information, the timing and the location of a high power consumption area can be detected and can be applied into the new partitioning and floor-planning solution for the next simulation of the architecture. Thread-focused programming applies to the MDI as well. The MDI does not store power consumption information and leaves the responsibility to individual threads that pass through the MDI.

5.4 Programming Model of the TLM Core

As we described programming techniques in the previous sections, we now provide few code examples in the TLM core. Figure 5.4 shows 1) the top-level module that instantiates the processor core, MDI, and the memory unit modules, 2) the processor core module that includes scheduler and execution unit blocks, and 3) the execution unit block that includes instruction decoder and ALU blocks. Multiple C++ classes that represent logic or memory blocks are included in each processor core unit or memory unit. All power and timing parameters of sub-blocks that correspond to commands executed in each block are taken as inputs to the module that contains the block. For example, power consumption data of adders and multiplier sub-blocks are inserted into ALU class, and are used when power calculation of the data values in generic payloads is executed. Similarly, power consumption data of texture cache are used when generic payloads arrive the memory module that contains cache instances.

For each module, we use SC::THREAD process that contains its own thread of execution. Compared to SC::METHOD that does not keep a state of execution, SC::THREAD uses wait() statements that are used to suspend current function and update any class members [3]. Therefore, we find this process as a convenient option to model GPU threads executing in parallel. However, due to the limitations in SC::THREAD the number of process core and memory blocks must be declared explicitly, and no arrays can be used to dynamically define this number. Array
initializations of any instance always use default constructors of the instance. To insert appropriate information using default constructors, several pointers containing the addresses of power or delay parameters must be included using a copy function. But a copy function, which is not a constructor, cannot be utilized in SC_THREAD because initialization of all input parameters and binding of SystemC ports may not recognize the changes made by this function.
5.5 Floorplanning Setup

In the setup phase, area information for all commands are obtained. However, floorplanning of these commands are necessary to accurately predict the actual interconnect power consumption, which is dependent on distance among multiple sub-blocks in terms of lengths and widths. Therefore, we assign virtual coordinates to all commands executed in the TLM model. For each instruction to be executed, a command cluster corresponding to the instruction is accessed. For each command in the cluster, we define a command block \( c \) as the set of all logic or memory sub-blocks and interconnect samples associated with the command. Area of the command block are denoted by \( A_c \). We set width of the command block \( w_c \) as \((A_c)^r\) and length of the command block \( l_c \) as \((A_c)^{(1-r)}\), where \( r \) ranges from 0 to 1. \( r \) is initially set to 0.5 to place the block as square-shaped, but may change depending on the positioning of the neighbouring command blocks.

An example of a coordinate assignment is illustrated in Figure 5.5, where the number inside each instance denotes the order of assignment and the coordinates are in the units of 10\(\mu m\). Coordinates of the command blocks are assigned and updated in the order of the commands executed in an instruction. In order to assign the direction where each command block should be placed, we first access the 5 X 5 grid that selects the positioning of logic and memory blocks. \( G \) in the grid denotes the area designated for logic commands, and \( M \) denotes the area for memory commands.

We start executing the floorplanning from an arbitrary grid assigned to logic command blocks. Suppose the first command cluster shown in figure 4.3 is executed first, and the second command cluster is executed next. We designate the direction of floorplanning for the first cluster based on the grid positioning for memory-mapped commands. As shown in Figure 5.5, we limit such that all logic command blocks that execute memory commands in the future are be placed in the north and east directions since the memory blocks are designated on these directions. The other two directions are selected for assigning the coordinates of any commands that are not connected with memory-mapped commands. While ADD1,3 is placed south of
DIV3, CMP is placed east of these two commands because CMP is connected with SELECT. When TSVs are introduced in the floorplan, all logic command blocks are considered for re-adjustment of coordinates. When the space for TSV is not available, the command blocks with the longest routing distance to their neighbouring blocks are chosen for re-positioning.

When two command blocks such as ADD1_1 and SELECT must be connected with an interconnect sample, the sample is selected as either individual or shared depending on the order of the command block assignments. In the case of these two blocks, an individual sample is first chosen since ADD1_2 is not iterated at this time. When ADD1_2 is iterated, an individual sample is also chosen. Shared interconnect sample are chosen only when the iterating command has two destination commands already assigned in the floorplan. While such selection may not
place interconnects optimally, the complexity of the assignment is simplified. At the end of
the TLM simulation phase, the selections of interconnect samples can be modified when the
update of the most optimal virtual coordinates for all command block is executed in the power
manager.

Prior to the power manager execution, the updates of command block coordinates inside
the TLM core are tracked by all threads that execute the command block. Multiple threads
that execute the same command block may have different coordinates for the block, due to
the difference in the order of TLM executions among threads. Since threads are terminated in
order, however, the final coordinate of a command block can be assigned correctly by assigning
the coordinate tracked by the thread that terminates the last. For the future TLM simulations,
the final coordinates for all command blocks are assigned in all threads that are initialized for
the subsequent simulation. With this setting, there is no need to keep track of the coordinates
within TLM modules, reducing the programming effort required inside the TLM core.

Based on the floorplanning, the lengths of all interconnects are determined. We compute
the total interconnect power consumption based on these length values as well as the dimension
settings. In the case of TSV wide I/Os, routing congestion factor $b$ and design rules are also
considered. For the power consumption of memory or logic blocks, we assign the input switching
activities obtained from the setup phase for all units that were used in any active commands. For
logic, memory blocks and interconnect samples that were inactive, we set the switching power
to 0 and only evaluate the internal power consumption from the input power parameters.

5.5.1 Floorplanning Example with TSVs

Now we look at another example for virtual coordinate assignment of blocks with TSV insertions
involved. Since the width of TSVs are already defined in the construction of the TLM core,
there is no need to explicitly define TSV as a separate command block to find the total area.
We initially add TSV area to all interconnect command blocks that are either connected to
TSV wide I/O memory or routed around TSVs. In order to accurately determine the total
TSV area, the memory burst information must be stored in interconnect command blocks. The memory grid is categorized differently than that of existing 2-D architecture, since caches are placed on the logic tier and mobile main memories are placed on the memory tier. As shown in Figure 5.6, memory grids are divided into logic-tier memory and memory-tier memory, where logic-tier memory is denoted as \( LM \) and memory-tier memory is denoted as \( MM \). There is a new notation named \( GM \), which represents the grid where TSV connects \( G \) and \( M \). Vertical arrows pointing down denotes the REQUEST command, and vertical arrows pointing up denotes the RESPONSE command.

The rules for coordinate assignment are similar to those in 2-D environment, except that the update of coordinates for logic grids and memory-tier memory grids are adjusted so that these two grids share the same horizontal coordinates. Logic or \( LM \) command blocks that communicate with memory command blocks must have coordinates within the range of the \( MM \) command block, in order to fit the requirements to be in the \( GM \) region. As shown in Figure 5.6, with the \( MM \) command block spanning from 3 to 22 in the \( x \) direction and 0 to 5 in the \( y \) direction, the address calculation logic command, the adder command, and the texture cache \( LM \) command blocks must be placed within these ranges.

Figure 5.6: Floorplanning for Logic and Memory with TSVs
5.6 Update of Simulation History

After the simulation of the TLM core is complete, we compare the power statistics obtained from the core with the actual RTL power statistics. Users have an option of choosing which logic sub-blocks are to be updated in their power statistics for future simulations. When such update occurs, the history table stored in the power library is also updated.

Although it is optimal to store all history data in order to improve the modeling accuracy under various architecture and benchmark settings, we must keep the size of the table relatively small to avoid simulation slowdown. Therefore, we choose to only store history of logic sub-blocks, and leave the update of memory an interconnect sub-blocks to the initial user configuration in the setup phase. For the history table, we choose to drop the least common units used for a fixed number of TLM simulations, and keep commonly used units. The selection of commonly used units depend on numerous attributes including flip-flop drive strengths, the type of AOIs used for the logic, and the insertion of buffers in the inputs and outputs of the sub-blocks. In addition to power statistics, the area information for all user-specified sub-blocks are updated to reflect a better initial estimation of floorplanning in the TLM core.
Chapter 6

Case Study with Open Source GPU

To test our power prediction methodology, we choose a simple GPU architecture named THEIA [49]. THEIA is a multi-core architecture that uses ray casting in its rendering of 3-D scenes to 2-D scenes. Ray casting is very similar to ray tracing except that ray casting does not trace secondary rays recursively, whereas ray tracing does [4]. Under a reconfigurable processor architecture such as THEIA, ray casting is advantageous to mobile devices compared to ray tracing due to the low power consumption required to complete the casting algorithm [32]. Moreover, the parallel nature of ray casting requires high memory bandwidth, making a case for the TSV wide I/O integration. For the remainder of this chapter, we revisit all procedures explained in chapter 4 and 5, and describe how these procedures are used to predict the power consumption of a THEIA-based GPU and the power savings due to the TSV wide I/O integration.

6.1 GPU Features

Each GPU core in THEIA architecture has pipelined SIMD ALUs with fixed-point arithmetic logic on 3-D vectors. The number of cores we set is 8 for the mobile GPU we model, but this number can vary from 2 to 16 where the number is a power of 2. Since THEIA is a freeware available for public use, the OpenCores Wishbone bus architecture [52] is originally installed in the overall architecture. THEIA includes a custom 64-bit ISA that includes logic, arithmetic
and control instructions. All specifications described above are all configurable, and even the rendering algorithm for texturing bi-linear filtering is configurable.

Several modifications in GPU settings are made from the original THEIA. First, the size of texture cache is reduced from 64KB per core to 16KB so that it can satisfy the power budget of mobile GPUs, which is approximately 2 Watts. Second, the ISA library made some modifications in the THEIA ISA setting so that 32-bit ISA referenced from ARM load-store architecture can be used. Third, several pipeline stages are added into the fixed-point vector ALU so that the post-synthesis clock frequency can increase from 62.5MHz to 143MHz. Finally, we modify the bus architecture such that it reflects the protocol closer to the one indicated in AMBA 3.0 than the Wishbone protocol. We are not able to fully realize every single functionality specified in AMBA 3.0, but we try to deviate from existing Wishbone specifications as much as possible.

We set the total number of cores in the GPU to 8. We try to allow as many GPU cores as possible in the architecture to find out the architecture that has the maximum memory bandwidth under the mobile power budget. After executing the validation of our TLM power prediction with the full RTL implementation of the architecture, we find that using 16 cores in the architecture creates power consumption that is higher than the standard thermal power limit (TPL) of smartphone GPUs. Although the mobile power budget is 2 to 3 Watts, the power consumption allowed for GPUs are much smaller since CPU, mobile memory, screen display, and global system for mobile (GSM) technology must share this budget. Usually around 100mW of power consumption is allowed for the average workload of smartphone GPUs [5], and the 16-core configuration exceeds the limit significantly. More details on these measurements are discussed in later sections of this chapter that describe the RTL validation of the GPU model.

Figure 6.1 illustrates the architecture of a GPU core that is the modified version of the THEIA architecture. This core is centred by several register files and a texture cache. A 1KB general purpose register file (RMEM) is read or written by the execution unit (EXE) of the GPU core, while the I/O unit cannot see these data. 1KB input register files (IMEM and XMEM) are written by the I/O unit, and read by EXE. There are two 1KB swap register files called
MEMA and MEMB, allowing I/O unit to read processor data while EXE writes processor, and vice versa. All register files have a data width of 96 bits to support 32-bit data from $x$, $y$, and $z$ dimensions. A single 4KB instruction memory feeds the general instruction to all GPU cores. Based on these instructions, parallel instructions specific to the GPU are decoded with the help of a 4KB ROM. For memory operations, the texture cache has the 32-bit data width. Size of the texture cache varies from 16KB to 128KB, depending on the mode of cache division obtained from user configurations. It should be noted that the GPU reads memory data exclusively from the texture cache while it writes the output data directly to mobile main memory since we
find that the frequency of memory write is a lot less than that of memory read. Due to these characteristics, we find that it is wasteful to insert output caches that stores resulting texture values from graphic processing.

For the memory bandwidth, we support 32-bit or 256-bit memory burst depending on the dimension setting. Although the TSV wide I/O memory we use has a capability of supporting the maximum of 512 bits, the maximum number of cores is 8 and the maximum bandwidth we test is 256 bits. The memory controller used in this GPU is the modified version of the memory controller implemented in [46]. The 32-bit PoP LPDDR2 used in this experiment [12] has the size of 2Gbits with 8 memory banks, and operates at 400MHz clock rate. For sub-blocks on the 3-D memory tier, 50\textit{nm} technology is assumed from [20] and the TSV wide I/O memory has the identical size as LPDDR2 for a fair comparison.

6.2 RTL EDA Flow

The 45\textit{nm} Nangate standard cell library [28] is used for all logic and memory sub-blocks on the logic tier. Compatible with NCSU FreePDK45, this cell library contains more than 100 standard cells with different drive strengths and facilitates the testing of the physical EDA design flow at RTL. For processing logic synthesis of the synchronized GPU design, we use the slowest characterization corner with the nominal temperature of 125 Celsius and the nominal voltage of 0.95\textit{V}. For the wiring delay, the delay model in [26] is used, where 50\textmu m of M3 metal layer contains 12.5585\textit{fF} of wire load capacitance and roughly 20 \textit{ps} of wire delay is assumed. Furthermore, we set that all cells are primarily driven by D flip-flops with the drive capacitance load of 0.879\textit{fF} and fanout of 4. All settings mentioned above are used identically for the creation of logic, memory and interconnect sub-blocks for TLM power prediction specified in section 6.2, and the RTL simulation of THEIA architecture specified in section 6.5.
Table 6.1: Power, Timing, and Area Information of Sub-Blocks (8ns)

<table>
<thead>
<tr>
<th>Sub-blocks</th>
<th>Area (µm²)</th>
<th>Internal (µW)</th>
<th>Switching (µW)</th>
<th>Leakage (µW)</th>
<th>Post-route delay (ns)</th>
<th>Pre-route delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder (2-stage)</td>
<td>804.38</td>
<td>79.76</td>
<td>52.28</td>
<td>4.29</td>
<td>8.50</td>
<td>6.22</td>
</tr>
<tr>
<td>Multiplier (4-stage)</td>
<td>5293.67</td>
<td>232.75</td>
<td>130.09</td>
<td>32.31</td>
<td>10.32</td>
<td>6.27</td>
</tr>
<tr>
<td>Divisor (3-stage)</td>
<td>3048.89</td>
<td>165.93</td>
<td>84.49</td>
<td>19.01</td>
<td>8.87</td>
<td>6.51</td>
</tr>
<tr>
<td>Comparator</td>
<td>597.44</td>
<td>14.38</td>
<td>8.44</td>
<td>5.62</td>
<td>8.53</td>
<td>6.16</td>
</tr>
<tr>
<td>Square Root</td>
<td>558.07</td>
<td>28.17</td>
<td>16.20</td>
<td>4.57</td>
<td>6.41</td>
<td>4.04</td>
</tr>
<tr>
<td>32-1 MUX</td>
<td>1107.89</td>
<td>26.39</td>
<td>24.69</td>
<td>7.18</td>
<td>4.27</td>
<td>2.71</td>
</tr>
<tr>
<td>FFS16</td>
<td>589.72</td>
<td>36.77</td>
<td>32.72</td>
<td>1.78</td>
<td>2.23</td>
<td>1.28</td>
</tr>
<tr>
<td>FFS32</td>
<td>1127.31</td>
<td>120.53</td>
<td>60.06</td>
<td>2.93</td>
<td>2.97</td>
<td>1.56</td>
</tr>
</tbody>
</table>

6.2.1 Logic Sub-block Generation

Using the synthesis scripts with the Nangate cell library, we create logic sub-blocks. The clock periods used for these sub-blocks vary from 5ns to 10ns, with the step size of 1ns. As examples, Figure 6.2 and Figure 6.3 show the post-synthesis schematic of a fixed-point adder at 6ns and 10ns clock periods, respectively. By comparing two figures, we see that a large number of extra AOI gates and buffers are required in the middle of the logic for the 6ns design than for the 10ns design. Figure 6.4 and Figure 6.5 show the 6ns high-radix Booth multiplier with 2 and 4 pipeline stages. For the 2-stage multiplier, a high volume of AOI gates and buffers are required in the beginning stage of the design to reduce the input delay, whereas the 4-stage multiplier requires extra volume of logic in the end stage of the design because of additional flip-flops.

In order to predict switching activities used to compute dynamic power consumption, we implement several test fixtures with list of instructions or texture inputs from THEIA GPU benchmarks we use, and update the dynamic power consumption data for all logic sub-blocks. Table 6.1 organizes the power, timing, and area information of all major logic sub-blocks under the global clock period of 8ns. FFS16 and FFS32 denote flip-flops that contain multiplexer
Figure 6.2: Schematic of Adder Sub-block with 6\(\text{ns}\) clock period

Figure 6.3: Schematic of Adder Sub-block with 10\(\text{ns}\) clock period

Figure 6.4: Schematic of Multiplier Sub-block with 2 pipeline stages
logic which takes 16 32-bit inputs and 32 32-bit inputs, respectively. Pre-route delay refers to the delay measured right after the completion of logic synthesis, and post-route delay refers to the delay measured after routing the logic and taking all parasitic capacitances and resistances into consideration. In addition to the logic sub-blocks mentioned in the table, we include 16 more combinations of multiplexers and flip-flops by varying the data width of inputs and the type of multiplexer logic.

6.2.2 Memory Sub-block Generation

As mentioned in section 4.1.5, CACTI 6.5 is used to extract all power information for caches and register files at logic tier. For the input configuration of CACTI, we set the block size to 32 bytes. Cache associativity is set to 4 for texture cache and 1 for all other register files. 45\text{nm} technology is used to match the Nangate logic process technology. Operating temperature is set to 350 Kelvin. Page size is set to 1KB, and tag size is varied so that CACTI can calculate the minimum number of required tag bits. For the design objective for the cache access, we focus more on reducing leakage power than reducing dynamic power due to the large leakage power found in texture caches with the size of 16KB or larger. For the interconnect analysis, we use the default CACTI mode where both full-swing and low-swing interconnects are considered to find an optimal configuration. Table 6.2 summarizes all power consumption, timing, area, and
Table 6.2: Summary of CACTI Parameters for 64KB texture cache

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area ($mm^2$)</td>
<td>1.50447 (1.18332 x 1.2714)</td>
</tr>
<tr>
<td>Access time (ns)</td>
<td>0.41308</td>
</tr>
<tr>
<td>Cycle time (ns)</td>
<td>0.26593</td>
</tr>
<tr>
<td>Total dynamic read energy/access (nJ)</td>
<td>0.09829</td>
</tr>
<tr>
<td>Total dynamic write energy/access (nJ)</td>
<td>0.10571</td>
</tr>
<tr>
<td>Decoder data energy (pJ)</td>
<td>0.23677</td>
</tr>
<tr>
<td>Wire delay (ns/mm)</td>
<td>0.03796</td>
</tr>
<tr>
<td>Wire power (mW/mm)</td>
<td>0.02344</td>
</tr>
</tbody>
</table>

interconnect data of 64KB texture cache obtained from the CACTI execution.

For mobile main memories, we use the area information from industry parameters [12, 20] to create a single memory block. Then, we divide this block into sub-blocks where each sub-block is responsible for transmission of 32-bit data. This means that for the LPDDR2 we use, only one sub-block is modelled since the memory burst is only 32 bits. For the 256-bit TSV wide I/O memory, however, the block is divided into 8 sub-blocks. It should be noted that such division of memory is not related to the total number of banks in the memory.

Before the initialization of the TLM core, we need to measure an accurate read and write power consumptions based on the mobile memory read frequencies and write frequencies for all combinations of GPU architectures and benchmarks in percent. The frequencies are listed in Table 6.3 for three different Verilog testbench benchmarks in the THEIA package. We label these benchmarks as THEIA1, THEIA2, and THEIA3. The total number of transactions used for the three benchmarks is 64,000, 48,000, and 63,000, respectively. The read and write frequencies are heavily affected by the architectural settings for the texture cache, because the cache size and the data bandwidth directly contributes to the number of clock cycles required to complete read operation. As the number of cores shared by each texture cache decreases from 8 to 1 and the size of texture cache decreases, the read frequencies increase and the write frequencies decrease.
Table 6.3: Read and Write Frequencies for Mobile Memory (%)

<table>
<thead>
<tr>
<th>Number of Cores per Cache</th>
<th>THEIA1 read</th>
<th>THEIA1 write</th>
<th>THEIA2 read</th>
<th>THEIA2 write</th>
<th>THEIA3 read</th>
<th>THEIA3 write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50.44</td>
<td>1.18</td>
<td>43.56</td>
<td>1.11</td>
<td>49.32</td>
<td>1.25</td>
</tr>
<tr>
<td>2</td>
<td>45.42</td>
<td>2.23</td>
<td>39.90</td>
<td>1.98</td>
<td>45.01</td>
<td>2.23</td>
</tr>
<tr>
<td>4</td>
<td>36.73</td>
<td>3.99</td>
<td>30.00</td>
<td>3.11</td>
<td>35.89</td>
<td>3.25</td>
</tr>
<tr>
<td>8</td>
<td>17.76</td>
<td>6.21</td>
<td>15.13</td>
<td>4.02</td>
<td>17.12</td>
<td>5.91</td>
</tr>
</tbody>
</table>

for all three benchmarks. Specifically, the read frequency increases up to 50 percent of all computations. This means that the reduction of read power consumption by the TSV wide I/O integration can be significantly effective in decreasing the overall system power consumption, especially for the setting where a small texture cache is dedicated to each core. This observation supplements the advantage of such setting in the 3-D architecture specified in [45].

6.2.3 Interconnect Sub-block Generation

To create interconnect sub-blocks, we choose multiply-add (MAD) units as reference models. We connect one multiplier and one adder from the list of logic sub-blocks to create a fixed-point MAD logic. After repeating the synthesis, routing, and extraction of switching activities as mentioned in previous sections, we obtain the power data of the MAD logic. Then we subtract the power consumptions of the multiplier and the adder from the MAD power, and we store this result as the power consumption of an individual sample that connects two sub-blocks. In order to obtain the power consumption of a shared sample that connects three or more sub-blocks, we add more adders or comparators into the MAD logic and repeat the process.

For all interconnect samples, the length of the interconnect is estimated by the interconnect equations described in chapter 4. For an interconnect sample that connects a logic sub-block and a off-chip memory sub-block, we use a fixed off-chip wire capacitance specified in [12], varying from $1pF$ to $2pF$. This value substitutes the product of interconnect length in $mm$ and $292fF/mm$. $292fF/mm$ is the on-chip wire capacitance per unit length for 45$nm$ technology [38]. Figure 6.6 illustrates how interconnect samples are placed among ALU and memory access
Figure 6.6: ALU and Memory Access Sub-blocks

sub-blocks for the setting where 2 GPU cores share a texture cache. All outputs of ALU are connected to both the execution finite state machine (FSM) and the I/O block that interfaces memory addresses for register files and texture cache. Each interconnect sub-block, sampled individually or shared, has a 32-bit data width in this figure. We also illustrate how TSVs are inserted between logic and memory. TSVs are added into the shared interconnect sub-block that connect logic, texture cache and TSV wide I/O memory. TSV parameters and minimum wire spacing requirements are listed in Table 6.4. The default routing congestion rate used for predicting additional interconnect power introduced by TSV insertions is set to 0.2. This number is based on the observations made by extracting a routed layout of a logic that consists of a multiplier, two adders, and a 256-bit memory controller. The layout is shown in Figure 6.7. We find that around 20 percent of interconnects related to ALU connections lie within the
Figure 6.7: Layout used for Analysis of Routing Congestion
Table 6.4: TSV Parameters and Minimum Wire Spacings

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSV area ($\mu m^2$)</td>
<td>25 ($5\mu m \times 5\mu m$)</td>
</tr>
<tr>
<td>TSV capacitance ($fF$)</td>
<td>37</td>
</tr>
<tr>
<td>$L_s$ ($\mu m$)</td>
<td>0.88</td>
</tr>
<tr>
<td>$W_m$ ($\mu m$)</td>
<td>0.8</td>
</tr>
<tr>
<td>$W_{msp}$ ($\mu m$)</td>
<td>0.8</td>
</tr>
</tbody>
</table>

candidate region of TSV insertion for the memory controller.

6.3 Command Cluster Generation

After creating all sub-blocks that are to be used in the TLM model, we generate command clusters by perusing GPU instructions given in THEIA. First, we create a C++ program that reads all synthesis and route text reports created by sub-block generations. All power, timing, and area statistics are stored in array structures where each array element corresponds to a single command.

After the initial loading of data is complete, we translate THEIA specific instructions into general load-store ISA instructions that are used in the TLM core. First, we create a set of command clusters where each cluster corresponds to generic load-store arithmetic and control instructions. With these clusters available, a direct translation can be made for many arithmetic and control instructions in THEIA such as ADD, MUL, JMP, and conditional branches. The only work required for such translation is the recognition of the number of clock cycles required to complete these THEIA instructions. For example, the THEIA specification [49] specifies that the addition of two fixed-point 32-bit data takes 2 clock cycles to complete, so we translate this instruction into ADD2 command specified in the power library. For some instructions such as division (DIV), such latency is variable depending on the function codes specified in the ISA. We vary the assignment of commands depending on the function code defined in the ISA specification. When the function codes are not defined explicitly in the ISA specification, which is the case
in THEIA, we take the median value of the number of clock cycles between the minimum and the maximum.

There is another type of instructions where the direct translation may not apply. GPU-specific memory or arithmetic instructions such as vector magnitude (MAG), cross product (CROSS), dot product (DOT), rearrange dimensional components (SWIZZLE3D), and texture read (TMREAD) fit into this type. For arithmetic instructions of this type, the algorithmic description specified in the THEIA specification is used to select appropriate commands and their order of execution. Figure 6.8 illustrates the structure of the command cluster for CROSS as an example. In this example, we also illustrate all sub-blocks inside CONNECT commands to show which sub-blocks are used for each inter-module interconnect command. For memory

Figure 6.8: Command Cluster for the Cross Product Instruction
instructions of this type, we assume a three-stage computation that consists of 1) identification and selection of input operand values of addressing caches or main memory, 2) address calculation using an integer adder, and 3) memory access where the data from memory is processed and propagated as input operands to other logics. During the translation of memory instructions, the main source of under-estimation in the prediction of power consumption is the possible absence of memory access peripherals that allow a simultaneous execution of arithmetic blocks with memory blocks. The effects from such absence can be reduced by increasing the availabilities of such peripherals as sub-blocks in the power library.

6.4 TLM Core Execution

In order to test the accuracy of our prediction model in the TLM core, we prepare four different GPU architectures: 1) all 8 GPU cores share a single texture cache of 128KB, 2) 4 cores share a 64KB cache, 3) 2 cores share a 32KB cache, and 4) a 16KB cache is dedicated to each core.

Table 6.5 shows instruction type, thread ID, instruction group ID, start time, end time, and the average power consumption for the first 10 transactions of the THEIA1 benchmark initiated in the TLM core. As shown in the table, the order of transaction completion can be out-of-order among all threads that share the same group ID. The ordering of thread execution inside the TLM core is affected by the priority of the floorplanning specified in section 5.5, where the command blocks related to memory transactions are the main focus of area optimization with TSV integration in consideration.

However, we make sure that the creation, dispatch, and completion of each thread is in the order of group ID so that the synchronization of threads is preserved and any potential data dependencies are addressed correctly. The delays specified in the TLM wait statements are modified, when the synchronization requires to increase the waiting time because of other threads within the same group finishing the execution of their instructions.
Table 6.5: Timing and Power Statistics for First 10 Transactions

<table>
<thead>
<tr>
<th>thread ID</th>
<th>instruction type</th>
<th>group ID</th>
<th>start time</th>
<th>end time</th>
<th>average power(mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADD</td>
<td>1</td>
<td>10</td>
<td>30</td>
<td>2.687</td>
</tr>
<tr>
<td>2</td>
<td>ADD</td>
<td>1</td>
<td>10</td>
<td>40</td>
<td>2.882</td>
</tr>
<tr>
<td>3</td>
<td>ADD</td>
<td>1</td>
<td>10</td>
<td>30</td>
<td>2.687</td>
</tr>
<tr>
<td>4</td>
<td>ADD</td>
<td>1</td>
<td>10</td>
<td>30</td>
<td>2.687</td>
</tr>
<tr>
<td>5</td>
<td>MUL</td>
<td>2</td>
<td>30</td>
<td>70</td>
<td>3.463</td>
</tr>
<tr>
<td>6</td>
<td>MUL</td>
<td>2</td>
<td>30</td>
<td>70</td>
<td>3.463</td>
</tr>
<tr>
<td>7</td>
<td>MUL</td>
<td>2</td>
<td>30</td>
<td>70</td>
<td>3.463</td>
</tr>
<tr>
<td>8</td>
<td>MUL</td>
<td>2</td>
<td>40</td>
<td>80</td>
<td>3.563</td>
</tr>
<tr>
<td>9</td>
<td>ZERO</td>
<td>3</td>
<td>80</td>
<td>90</td>
<td>2.219</td>
</tr>
<tr>
<td>10</td>
<td>CROSS</td>
<td>4</td>
<td>70</td>
<td>150</td>
<td>4.277</td>
</tr>
</tbody>
</table>

6.5 Validation of the TLM Model

After completing all steps specified in the TLM power prediction methodology where the complete RTL implementation of the THEIA GPU is not available, we fully implement a GPU at RTL to evaluate the modeling accuracy of our prediction model. After the validation of our model at 2-D environment, we safely move on to modifying the architecture settings specific to TSV wide I/Os. Specifically, all configurations related to memory blocks and interconnects are included in these modifications.

6.5.1 RTL Model

The original RTL model of the THEIA architecture is not synthesizable, so we modify the entire THEIA RTL package by removing all unsynthesizable constructs and unrolling while loops. Since memory blocks are not synthesizable, the hierarchy of the GPU is also restructured so that the GPU logic and the memory blocks such as register files, texture cache and ROM are completely separated. Furthermore, we add pipeline stages to vector ALU modules defined in THEIA so that the clock period can be reduced and any latches in the design are removed. We also add pipeline stages to the memory access module to general-purpose input register files since we find that the critical path delay between ALU access and input register file access
Table 6.6: Power Consumption reported from CACTI for 8ns clock period

<table>
<thead>
<tr>
<th>Memory units</th>
<th>Size(KB)</th>
<th>Internal power(mW)</th>
<th>Static power(mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Files (IMEM, MEWA, MEMB, RMEM, XMEM, ROM)</td>
<td>1</td>
<td>0.3745</td>
<td>1.1710</td>
</tr>
<tr>
<td>ROM</td>
<td>4</td>
<td>0.9037</td>
<td>3.5918</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>4</td>
<td>0.3535</td>
<td>3.5231</td>
</tr>
<tr>
<td>Texture Cache</td>
<td>128</td>
<td>0.3261</td>
<td>20.4378</td>
</tr>
</tbody>
</table>

exceeds all other delays detected in the original architecture. The number of bits required to access memory blocks are reduced from those in the original implementation so that no address bits are unused, creating a better synthesized design.

The layout of a core generated by Cadence Encounter is shown in Figure 6.9. The target core utilization rate used for the Encounter routing is 0.7. During the creation of this layout, a rise phase delay between 375.5ps and 415.9ps is reported, and a fall phase delay between 369.3ps and 429.8ps is reported. 40.4ps of rise skew and 60.5ps of fall skew are reported. The buffer rise delay ranges from 46.3ps to 96.7ps. The buffer fall delay ranges from 35ps to 49.8ps.

8 of such cores are attached with all memory blocks at the logic tier and a memory controller. We report that a 32-bit memory controller we use for the 2-D setting consumes the total power of 0.648mW. In addition, a 256-bit memory controller we use for the 3-D setting consumes 4.653mW. Table 6.6 describes the internal and leakage power consumptions reported from the CACTI memory compilation under the clock period of 8ns. As shown in the table, the total leakage power consumed by texture caches is around 25mW and slightly increases as the number of cores per cache decreases. This means that in order to increase the memory bandwidth further than 256 bits, design techniques to reduce leakage power consumption of caches may need to be accompanied. Doubling the texture cache capacity is likely to raise the total static power consumption of caches over 50mW, and leaves little room for power consumption of other memory blocks or GPU core blocks.
Figure 6.9: Layout of a GPU Core
6.5.2 Experimental Results of Three Metrics

Modeling Accuracy

First, we measure the average modeling accuracy of our prediction model for each benchmark. The average is taken from the four different texture cache settings. The average modeling accuracy is measured only for GPU cores and mobile main memory. We exclude register files and texture caches from this measurement because identical power statistics are used for those blocks in architectures of both dimensions. For the RTL measurement, we combine the average power consumption of all GPU logic units and the mobile memory that were active during the entire simulation. For the TLM measurement, we combine the power measurement of all sub-blocks that were used as commands in the TLM core simulation. Overall, 92.2 percent is reported as the average modeling accuracy. Table 6.7 shows the accuracy, and the total system power consumption that includes the total GPU power and the mobile memory power. The TLM model consistently overestimates the GPU power, especially in the dynamic power related to logic and interconnect sub-blocks.

The factors that contribute to the inaccuracy are the differences in the measurements of switching activities, read and write frequencies to mobile memory, and wire length estimated for additional routing due to TSV insertions. However, we claim that such inaccuracies can be isolated and maintained in control for two reasons:

1. Since the switching activities are recorded using sample sub-blocks obtained from the setup phase, the logical behavior of these sub-blocks dictate the overall prediction behavior of our model. This behavior may not be so problematic, however. The sampling of sub-

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Accuracy (%)</th>
<th>Total System Power</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RTL(mW)</td>
</tr>
<tr>
<td>THEIA1</td>
<td>94.1</td>
<td>312.76</td>
</tr>
<tr>
<td>THEIA2</td>
<td>89.4</td>
<td>291.42</td>
</tr>
<tr>
<td>THEIA3</td>
<td>93.2</td>
<td>309.55</td>
</tr>
</tbody>
</table>
blocks on ALU-related logics and a majority of GPU operations are ALU-related, a high modeling accuracy can be achieved as targeted.

2. In order to minimize accuracy error due to read and write memory frequencies, we collect the same volume of texture input data sets for the setup phase as the one used in RTL. Since we use SAIF files that collect the average switching activities and our architecture only allows LPDDR2 or TSV wide I/O read based on texture cache access, keeping the same volume of sample data can achieve a high accuracy in predicting read and write frequencies.

For inter-module interconnect commands, the accuracy of power prediction strongly depends on the accuracy of interconnect length. Therefore, the difference in the quality of floorplanning optimization techniques between the RTL model and the TLM model is likely to be the major factor in compromising the accuracy of interconnect length. In order to reduce such inaccuracy, numerous existing floorplanning algorithms need to be modified in terms of code structure so that they can fit into our TLM model. This is left for future work.

The average accuracies reported in Table 6.7 is measured where three benchmarks are executed independently, meaning that none of the results from a benchmark simulation affects other benchmark results as suggested in section 5.5. We re-measure the average accuracy again by executing the three benchmarks serially, where the power statistics obtained from the RTL simulation results in THEIA1 update the history table in the power library, applying to the TLM simulations of THEIA2 and THEIA3 benchmarks. Specifically, 12 logic sub-blocks created in our prediction model are updated, where the total number of logic sub-blocks created is 18. Mainly these logic sub-blocks are related to multiplication and addition instructions. Moreover, a total of 2 buffer gates with the drive strength of 4 are added between a 4-stage multiplier and a 2-stage adder. For the memory sub-blocks, only the input register file power values are updated. For the interconnect sub-blocks, the power numbers of all sub-blocks that communicate with logic sub-blocks are updated. The input characteristics of THEIA2 do not correlate strongly to those of THEIA1, so the accuracy improvement for THEIA2 is measured only 0.7 percent. However,
the accuracy of THEIA3 improves from 93.2 percent to 97.4 percent since the characteristics of
THEIA1 and THEIA3 are somewhat similar in terms of texture input patterns and instructions
that carry out these patterns.

**Total CPU time**

Under three benchmarks, we also measure the average speedup of our model against the RTL
model. In order to remove any factors external to the CPU execution, we choose the total CPU
time as the time metric. The total CPU time required to complete all tasks in the RTL model
includes four components: a) CPU time taken to complete logic synthesis of the entire GPU
core using Synopsys Design Compiler, b) CPU time taken to route the final netlist and create a
SPEF file using Cadence Encounter, c) CPU time taken to complete the extraction of switching
activities using the SPEF file, and d) CPU time required to analyze the switching activities
and report the power consumption, area, and timing of the GPU core. The total CPU time
required to complete all tasks in the TLM model also includes four components: 1) CPU time
taken to synthesize all logic sub-blocks and interconnect sub-blocks at the logic tier, 2) CPU
time taken to route these sub-blocks, 3) CPU time required to extract switching activities of
all sub-blocks and organize the power, timing, and area numbers into command clusters, and
4) CPU time taken to complete the TLM core execution.

For the RTL model, synthesis and route time are updated only once for the given clock
period. Switching activity extraction and power report time are updated for each benchmark.
For similar reasons, synthesis and route time in the TLM model are updated only once. Also,
switching activity extraction, command generation, and TLM core execution time are updated
for each benchmark. It should be noted that the TLM synthesis and route time consists of
the time required to generate all sub-blocks with variations in the minimum clock period,
especially in flip-flops and multiplexers. For the SystemC TLM execution, the ANSI C time
function is used for the time measurement. For other components in both TLM and RTL, the
time command from Linux version 2.6 is used. From the time command, we use the total CPU
Table 6.8: Total Synthesis and Route time

<table>
<thead>
<tr>
<th>Component</th>
<th>RTL(min)</th>
<th>TLM(min)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesis</td>
<td>43.803</td>
<td>12.881</td>
<td>3.40</td>
</tr>
<tr>
<td>Routing</td>
<td>33.043</td>
<td>17.626</td>
<td>1.87</td>
</tr>
<tr>
<td>Total</td>
<td>76.846</td>
<td>30.507</td>
<td>2.52</td>
</tr>
</tbody>
</table>

Table 6.9: Speedup of Power Computation

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Simulation Time (min)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RTL(c)</td>
<td>TLM(3)</td>
</tr>
<tr>
<td>THEIA1</td>
<td>171.40</td>
<td>9.66</td>
</tr>
<tr>
<td>THEIA2</td>
<td>146.44</td>
<td>8.51</td>
</tr>
<tr>
<td>THEIA3</td>
<td>154.05</td>
<td>8.63</td>
</tr>
</tbody>
</table>

In Table 6.8, we present the speedup of the synthesis and routing. Overall, an average speedup of 2.52 is achieved when multiple small sub-blocks are synthesized and routed instead of a large GPU block. The majority of the overall speedup is, however, achieved from the actual power computation where switching activities are extracted from a simulation and power consumption values are computed. Table 6.9 summarizes the power computation time for both models. RTL(c) and TLM(3) denote the third time components of their respective models described in the above paragraph. RTL(d) and TLM(4) denote the fourth time components of their respective models. An average speedup of 14 is reported for these time components under the three benchmarks.

3-D Power Savings

Table 6.10 summarizes the power savings reported for all combinations of GPU architectures and benchmarks. For the 3-D settings, the configuration for 256-bit memory burst setting mentioned in section 4.2 is used. Out of all 12 cases, 27.4 percent of average total power savings for the GPU with the TSV integration are reported. The maximum saving is reported 29.5 percent, and the minimum is reported 23.6 percent. The maximum power consumption reported in the
3-D environment is measured less than the minimum power consumption reported in the 2-D environment. This result implies that although the leakage power may increase slightly, multiple small texture caches can be used in the mobile GPU with less delay and less power than those of a large texture cache currently used in mobile processors.

Breaking down the 3-D power reduction in detail, we find that as the number of cores per cache increases, the importance in the I/O power reduction increases. Conversely, the importance in the memory power reduction increases as the number of cores per cache decreases. When multiple cores share a single cache that bursts 32-bit for each clock cycle, the read frequency of the TSV wide I/O memory is reduced and the I/O power per given clock period decreases. We report that the proportions of I/O power reduction over the total power reduction are approximately 52 percent and 34 percent for the 8 cores per cache and the 4 cores per cache settings, respectively. When each core has its dedicated 16KB texture cache, 256 bits of data are transmitted to the 8 32-bit texture caches. In this case, the read frequencies are maximized and the I/O power increases, meaning that the memory power reduction due to the TSV integration becomes more important than the memory read burst settings. We report that the proportions of I/O power reduction over the total power reduction are approximately 24 percent and 10 percent for the 2 cores per cache and the 1 core per cache settings, respectively.

Based on all 12 cases, an average of 30 percent in the total power reduction is due to the I/O power reduction, and 70 percent is due to the memory power reduction.

Based on the power consumption per block measured from the TLM core, we also report that arithmetic logic blocks in the vector ALU consume more than 65 percent of the overall GPU core logic power excluding register files and a texture cache, proving that mobile GPUs must be aware of high ALU power consumption since GPUs are very compute intensive with the majority of instructions devoted to ALU instructions. In particular, 6 sets of high-radix fixed-point multiplier consumes approximately 40 percent of the overall core logic power. This shows that replacing fixed-point arithmetic logics with high-power floating-point logics require a very careful design modification in order for those logics to fit into the current vector ALU.
<table>
<thead>
<tr>
<th>Number of Cores per Texture Cache</th>
<th>THEIA1</th>
<th>THEIA2</th>
<th>THEIA3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2-D(mW)</td>
<td>3-D(mW)</td>
<td>%</td>
</tr>
<tr>
<td>1</td>
<td>350.896</td>
<td>247.261</td>
<td>-29.5</td>
</tr>
<tr>
<td>2</td>
<td>335.500</td>
<td>237.901</td>
<td>-28.7</td>
</tr>
<tr>
<td>8</td>
<td>257.356</td>
<td>193.909</td>
<td>-24.6</td>
</tr>
</tbody>
</table>
Reconfiguration Cost

The reconfiguration cost in our TLM model is also measured for both the module-focused model and the thread-focused model. The total number of SystemC code lines in the TLM core is a little over 5,000. Pie graphs shown in Figure 6.10 illustrates the fraction of SystemC codes added or modified due to the change in benchmark setups from THEIA1 to THEIA2, showing the difference in the reconfiguration cost between two models. RC denotes the fraction where the codes are reconfigured when applying a different benchmark from the previous one. US denotes the fraction of codes required for the setup of user setting. CD denotes the fraction of codes required for class definitions. For the two models, the unmodified codes from the first benchmark setup consume 70 and 74 percent of the total amount of codes, respectively. For the module-focused model, 24 percent of the entire code must be modified when all components in the user configurations are changed in the 8-core architecture. This percentage decreases to
19 percent in the thread-focused model. Therefore, we claim that the reconfiguration cost is reduced by 21 percent. Breaking down the code reduction in detail, we find that a simple data pointer access in the thread-focused programming model allows to reduce over 80 percent of the total reductions. The remaining reduction is due to the elimination of class instantiation that keeps track of power statistics for each SystemC module.
Chapter 7

Conclusions and Future Work

We conclude this dissertation by providing a summary of our contributions in power prediction methodology of 3-D graphic engines. Furthermore, we explain the future tasks related to this research.

7.1 Summary of Contributions

This paper has shown the power prediction methodology for mobile graphic engines that integrate TSV wide I/Os. Our experimental results show that using command clusters with a partial incorporation of physical design flow provides an average of 92.2 percent accuracy compared to the RTL of the GPU architecture. Furthermore, it is shown that an impressive average of 27.4 percent power improvement can be achieved from 3-D TSV integration. With the utilization of thread-focused programming and MDI, the reconfiguration cost in terms of the number of lines in SystemC codes is reduced by an average of 21 percent.

7.2 Future Work

The power prediction model in this dissertation is to be featured as one of the components in PathFinder3D [37], the open source design kit and architecture evaluator used to explore
the various aspects of the TSV 3-D integration. Other components include thermal simulator featuring WireX [25], a physical thermal extractor that creates a thermal netlist. In order to integrate with this thermal simulator, our power prediction model needs to be elaborated in the floorplanning section that directly relates to the thermal performance of 3-D ICs.

Another area we can elaborate this work in the future is the incorporation of general-purpose GPU (GPGPU) programming to reduce simulation time further in the setup phase. In other words, GPUs can be also used as a device to improve the performance of our prediction model. GPU specific languages such as CUDA [30] can be used to process all direct translations of power data into input arrays used in the TLM core. The challenge in this approach will be to identify parallelizable sections in the setup phase. In addition, it is important to achieve a massive amount of data processing inside GPUs so that device communication overhead between GPU and CPU does not affect the simulation time. Appendix A briefly describes works that describe the GPGPU programming applied for VLSI CAD algorithms, and relate these works into our research.
REFERENCES


APPENDIX
Appendix A

Applying GPGPU in the Setup Phase

As a brief appendix, we introduce some GPGPU techniques that can be used to reduce runtime of C-based codes significantly. Specifically, we elaborate on how division of workload between the CPU host and the GPU device should be approached. Then we propose few ideas on how the workload in the setup phase of our prediction model can be divided so that the runtime of parallelizable sections in the setup phase can be reduced.

A.1 CUDA Background

Before explaining GPGPU techniques, we introduce the background of CUDA. CUDA, a GPGPU language created from NVIDIA, is an extension of C language that uses CPU as host and GPU as device for data computation. CUDA programs are executed inside functions called the device functions or kernels. Device functions are callable from both the host and the device, and run exclusively inside the GPU device. Device functions take the following arguments: data content transferred from the host to the device, the size of data, and the total number of threads executed in the GPU [30].
For data used inside a device function, several types of GPU memories can be utilized to enhance computing efficiency. Constant memory and texture memory are accessed by all GPU streaming multiprocessors (SMs) and are good substitutes for global memory. Shared memory is shared by all threads within a thread block. The on-chip memories can be accessed as fast as registers as long as there are no bank conflicts among all threads within a thread *warp*, which is a collection of 32 threads. The drawback of using on-chip memories is that the size of these memories is limited. Therefore, programmers must be careful in creating the kernel functions so that CUDA programs can run without out-of-memory errors.

When specifying the total number of threads executed in the GPU, choosing the block size, i.e., the number of threads per block that runs within a SM, is important in terms of runtime performance. To maximize the utilization of computing resources in GPU and to minimize block switching time, maximizing the block size is encouraged. Up to 512 threads can be assigned in a block for the existing NVIDIA GPUs. However, maximizing the block size must consider several factors such as compute capability, the number of registers used per thread, and the amount of shared memory used per thread block. Depending on these factors, maximizing the block size may not optimize computing performance [21].

For the NVIDIA GPUs, compute capability is the hardware capacity defined by the GPU architecture. For our case study, we use a GPU with the compute capability 1.0. In each SM under the compute capability 1.0, 8192 32-bit registers exist. The size of shared memory per SM is 16KB. The maximum number of active threads allowed per SM is 768. Clearly, the number of registers used per thread times the number of active threads cannot exceed 8192. Moreover, the shared memory size per thread block times the number of blocks cannot exceed 16KB. The number of registers per thread and the shared memory usage per block are determined by the program behavior. Therefore, the program behavior affects how much computing resources in GPU are utilized. We define the measure of such utilization as the occupancy rate.

Another factor to consider in CUDA programming is the amount of computations executed in the GPUs. When data are transferred between the host and the device, the kernel invocation
time and the data transfer time are added to the total runtime. If the amount of computation is too small, the sum of the kernel invocation overhead and the data transfer time may not be offset by the runtime reduction due to GPU parallel computations. Therefore, it may be not be advantageous to use GPUs for runtime reduction.

A.2 Previous Work on VLSI CAD Algorithms

Previously, two case studies were conducted to demonstrate that runtime of VLSI CAD applications can be successfully reduced with parallel programming in conjunction with GPUs. For sequence pair based floorplanning using simulated annealing [9], an average speedup of 13.5 is achieved with no performance degradation and less than 16 percent of reconfiguration cost. For power-aware buffer insertion algorithm introduced in [35], an average speedup of 3.43 is reported [10]. NVIDIA Quadro FX5600 graphic card of 1.35 GHz is used for the hardware that achieves the speedups.

Mainly, these two case studies focus on breaking data dependencies between consecutive iterations so that data can be processed as parallel as possible. For the sequence pair based floorplanning, multiple sequence pairs are generated in advance where some data dependencies may exist during the generation of these pairs. Each pair is then assigned for each GPU thread so that only area calculation can execute in the GPU device. At the end of execution in the device function, the sets of area computed are transferred back to the CPU host. The host compares them with current area solution, then decides whether to update the current solution or not. Rather than updating area of the floorplan inside the device by designating the floorplan area as shared data for all threads, area solution is updated in the host only. Such a decision helps to avoid possible synchronization conflicts. In order to enjoy high speedup in simulation time, the size of sequence pair batch is important since the size must get closer to the maximum number of threads that can process in parallel. Such setting depends on the specification of the GPU that the program is executed.

For the buffer insertion algorithm, the original linked list data structures that stores delay
and power information is converted into arrays for a convenient indexing of selection data. The selection data is a set of two decisions: whether a buffer should be inserted to each potential location and which buffer size should be chosen. Insertion of the selection data consists of managing delay-power pairs and capacitance values associated with these pairs. The challenge in parallelization of this insertion is that all new capacitance values must be inserted before the corresponding delay-power pairs. To overcome this challenge, we partition the code into two. In the first stage executed in the CPU host, insertion of capacitances is executed, and an index list is added to track all delay and power values from the downstream location. The index list stores starting index of delay-power pairs in both the downstream location and the current location, and the extra delay based on slew rates and capacitances. In the second stage executed in the GPU, each thread responsible for each combination of slew rate and capacitance updates selection data in parallel based on the index list, as shown in Figure A.1.

Figure A.1: Using Index List to Divide Workload between CPU and GPU
A.3 Applying GPGPU

As shown in these two case studies, the biggest challenges in applying GPGPU programming in our research are: 1) recognizing parallelizable section of our code, and 2) dividing workload between the CPU host and the GPU device properly so that the speedup is maximal.

For the setup phase, the assignment of arithmetic or memory commands into command clusters can be executed in parallel. For each GPU instruction, the insertion of commands into the array that contains the corresponding command cluster does not depend on any other instructions when there are no control instructions involved. The partition of instructions and assignment of these partitions into GPU threads can be executed in the CPU host. Group IDs can be used to partition the instructions since only instructions with the same group ID can execute in parallel. For the instructions sharing the same group ID, control instructions such as branch or jump function as the breakpoint that divides the instruction partition. Since the nature of GPU executions are parallel and arithmetic computation intensive, we expect that the total number of control instructions is relatively smaller than that of arithmetic instructions. This means that it is possible to launch a massive amount of partitions where each thread is responsible to execute a single partition.

After the partition, the array that contains sub-block information can be passed as input parameters of the GPU device alongside the instruction partition list. As a result, the inclusion of power, timing, and area information for all command clusters can be executed in parallel. To produce a high speedup in simulation time, it should be noted that thousands of partitions must be created so that the occupancy rate of GPU cores can be as high as possible. This means that the total number of GPU instructions tested in our environment must be very high, expecting a day of runtime in RTL and hours in TLM. In addition, the initialization overhead for launching function calls into GPU [9] should be taken into account when speedup is estimated. A detailed measurements of the speedup remain as a future research work.