PARALLEL ACCELERATION FOR TIMING ANALYSIS AND OPTIMIZATION OF ADAPTIVE INTEGRATED CIRCUITS

A Thesis

by

YIREN SHEN

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

MASTER OF SCIENCE

Chair of Committee,	Jiang Hu
Committee Members,	Sunil P. Khatri
	Duncan M. Walker
Head of Department,	Miroslav M. Begovic

May 2016

Major Subject: Computer Engineering

ABSTRACT

Adaptive circuit design is a power-efficient approach to handle variations. Compared to conventional circuits, its implementation is more complicated especially when we deal with the fine-grained adaptivity. The unconventional and sophisticated nature of adaptive design further requires timing verification to validate the design. However timing analysis becomes more complicated due to complexities arising from nanometer VLSI technologies. A well-known challenge is process variations, which need to be addressed in timing analysis at least by considering different process corners. Adaptive circuit design further needs statistical static timing analysis (SSTA), which is much more time consuming than variation-oblivious timing analysis. Besides timing analysis, gate implementation selections of the adaptive design process are also computational expensive. This research focuses on parallel acceleration techniques for timing analysis and optimization of adaptive circuit. General purpose graphic processing units (GPGPU) and multithreading techniques are used in this work. Previous works on GPU acceleration for SSTA are mostly based on Monte Carlo based SSTA. By contrast, the parallelization techniques for principle component analysis (PCA) based SSTA are explored in this work, which is intrinsically more efficient.

We develop a batch-based scheduling algorithm to partition the circuit graph into topological levels for GPU processing and investigate other techniques such as latency hiding. We propose a multithreading based acceleration method for the process of gate implementation selection and use the same batch-based scheduling result. The experiment

result shows effectiveness of our parallel acceleration for timing analysis and for optimization with the performance up to 130× and 5× speedup respectively.

ACKNOWLEDGEMENTS

I met many people in the past two years. Because of their love and help, I can become who I am and make this possible. I want to say thank you to my advisor Prof. Jiang Hu for his remarkable and patient instruction. It is my fortune to have an advisor who led me into the VLSI CAD field and taught me with his high expertise and deep insight. Without his support, the research may have no chance to see the breakthrough.

I am grateful to my committee members, Prof. Sunil P. Khatri and Hank Walker, for their helpful discusses and valuable feedback for my research and thesis.

I will never forget those days working together with those brilliant group members. I would like to thank Rohit, Hao and Jiafan for collaborations on the adaptive circuit design project; Yang, for sharing his experiences of operating system design; and all my friends for their unconditional supports.

Last, I owe it to my family – Mom, Dad, Grandma and my love Zhaofeng, for always being on my side.

NOMENCLATURE

AT	Arrival Time
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
PCA	Principle Component Analysis
PDF	Probability Density Function
RAT	Required Arrival Time
SSTA	Statistical Static Timing Analysis
STA	Static Timing Analysis

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

CHAPTER I

INTRODUCTION

Unlike the conventional circuits, adaptive circuits can apply power differently on each circuit block and tune the circuit performance. This makes the adaptive circuits a promising technique for handling variations. An adaptive circuit contains sensors to detect the performance and individually compensate the performance variation by using body biasing [1], voltage interpolation [2] and other technologies. Unfortunately the adaptive circuits are far from widely adopted in realistic products and one reason is the lack of corresponding mature timing verification tools. The unconventional and sophisticated nature of adaptive design implies relatively large risk of design errors so it requires reliable timing verification tools. One way to validate the adaptive circuit design is to enumerate all the adaptivity scenarios of adaptive circuits when verifying the timing. This solution is reliable but it is very time-consuming especially when the adativity is fine-grained.

Timing analysis is an indispensable part of mainstream digital IC designs. It can also pinpoint logic paths where timing needs to be improved or slack can be traded for power savings. In nanometer technology regime, process variations are no longer negligible and must be considered in timing analysis. Considering the sophisticated nature of adaptive design, timing analysis is also fundamental to provide assurance for design timing closure.

Static Timing Analysis (STA) is a fundamental tool of timing verification in circuit design. A traditional STA tends to overestimate the path delay which causes design

pessimism. The traditional STA method is deterministic because it analyzes the circuit under specific process condition. Therefore it cannot handle the variations across a wafer die (also called within-die variations) and will misestimate the circuit delay [3]. To model the process variations more effectively, statistical static timing analysis (SSTA) was proposed and well formulated [4-8]. Unlike the STA, SSTA represents the circuit delay and arrival time using PDF and CDF. There are several primary SSTA approaches. Monte Carlo method is based on sampling and enumeration. It carefully chooses samples that are sufficient to represent the timing feature of a circuit. Each sample is then analyzed by the classical STA to get the deterministic circuit delay. The yield of the whole circuit can be defined by the percentage of the passed samples against the total samples extracted. Another approach is typically based on probabilistic analysis [9]. Other than propagating the arrival times along the circuit graph in STA, this approach propagates the PDF of arrival time and performs statistical maximum and summation function. There are two ways of performing probabilistic analysis based SSTA, one is path-based circuit delay computation and another is block-based. Path-based algorithm is to calculate the delay distribution along the critical path and then perform statistical maximum operations to obtain the entire circuit delay distribution [10], [11]. Block-based algorithm uses sum and max functions to obtain the timing slack of each gate. This thesis will discuss more about block-based probabilistic analysis method – principal component analysis (PCA) based SSTA for adaptive circuit.

A scenarios-reduction based acceleration technique for adaptive circuit SSTA is proposed in [12]. Compared to [12], we study how to accelerate SSTA for both

conventional and adaptive circuit using GPU (Graphics Processing Unit). A GPU consists of many small processing cores and is friendly to large scale SIMD (Single Instruction Multiple Data) parallelism. It often provides large computing speedups at very modest hardware cost. This appealing advantage leads to many GPU acceleration researches for CAD algorithms [13], and SSTA is no exception. GPU-based parallel SSTA techniques are reported in [14], [15]. However, both of the SSTA techniques are based upon Monte Carlo (MC) simulation, which is relatively easy to be parallelized but intrinsically slow. Among sequential SSTA methods [9], an efficient one is the integration of conventional block-based static timing analysis with PCA (Principal Component Analysis). It is demonstrated in [16] that the PCA-based SSTA is orders of magnitude faster than Monte Carlo simulations and usually incurs less than 1% error on timing yield estimation. In [12], the PCA-based SSTA is extended to adaptive circuit designs, and also obtains dozens to thousands of times speedup over Monte Carlo simulations. Therefore, it is more beneficial to parallelize the PCA-based SSTA than to parallelize Monte Carlo-based SSTA. The GPU parallelization for PCA-based SSTA is quite different from that of MC-based. The MC-based SSTA includes many randomized runs of STA and the computing of each individual STA is relatively simple. In contrast, the PCA-based SSTA contains only two circuit traversals like the STA, but its intermediate computation steps within a traversal is quite complicated.

Gate implementation selection process is to select the candidates of the gate sizes and gate threshold voltages [17] for circuit optimization. A dynamical-programming like circuit traversal is proposed to propagate candidate solutions along the circuit graph [18].

The computation of this gate implementation selection is expensive and a corresponding GPU-based acceleration technique is developed in [19]. Our work will also consider the acceleration for the gate implementation selection for adaptive circuit design.

The main contribution of this work is development of new GPU techniques for parallelizing the PCA-based SSTA for both conventional and adaptive circuit. We also use multithreading on CPU to accelerate the gate implementation selection for adaptive circuits. A new task scheduling algorithm is designed and data/instruction techniques are also studied. Further, we also make the parallelization compatible with analyzing adaptive circuit designs. As far as we know, this is the first work using GPU to accelerate PCA-based SSTA and also the first work using GPU to accelerate the SSTA considering adaptive circuit design. We apply our GPU scheduling result for the multithreading of gate implement selection. Experiments are conducted on ISPD'13 gate sizing benchmark suite, which is prepared by Intel and includes circuit as large as 150K gates. Compared to sequential PCA-based SSTA, our approach delivers the same timing results with 22X and 134X faster speed for conventional circuits and adaptive circuits, respectively. Our approach is also compared with GPU-based Monte Carlo SSTA and obtains about 39X computing acceleration with an average of 0.77% error on timing yield estimation. And the multithreading of gate implement selection helps to get a 5X speedup compared to sequential approach.

The rest of thesis is organized as follows. Chapter II introduces the background on adaptive circuit design, timing analysis, and GPU programming. Chapter III proposes the batch scheduling and GPU memory techniques for parallel SSTA of adaptive circuits.

Chapter IV briefly discusses the application of the batch scheduling for multithreading of gate implement selection. The experimental results are shown in Chapter V and finally this thesis is concluded by Chapter VI.

CHAPTER II

BACKGROUND

II-A. Variations

The variation of circuit parameters is a common issue in modern VLSI technologies. The main reasons include the modeling analysis errors, the process variations and operation environment variations. The modeling and analysis errors come from the inaccuracy of design implementation, timing analysis and so on. The environment variations include the operation context like the temperatures and power supply. The process variations are due to manufacture procedure uncertainty, which we focus on handling.

The physical parameters may vary from die-to-die or within-die because of the fluctuations of the manufacture processes. The variations occur in gate length, oxide thickness, channel doping and etc. [20], [21]. These physical parameters decide the electrical parameters of the device like gate capacitance or threshold voltage that will actually affect the circuit delay.

All the device parameters on the same die are affected by die-to-die variations. It represents as discrepancy of physical parameters on different dies due to the shifts in the process. A typical solution to this problem is to run STA on multiple process corner files. Another type of variations is the within-die variation which results from the intrinsic non-uniform properties of silicon on which the chip is built. Within-die variations have the property that physical parameters show similar characteristics in one location than in other

locations which are far. It is as known as the spatial correlation. SSTA we will discuss later has strategies to take this problem into consideration.

II-B. Adaptive Circuits

Adaptive circuit design can handle variation effects, especially process variations and circuit aging. An adaptive circuit has sensors to detect performance degradation and certain circuit tuning knobs for compensating performance loss due to variations. Common tuning techniques include body biasing [1] and adaptive supply voltage [2]. The tunings are usually performed after chip fabrication and therefore can be targeted to the actual variations occurred. Compared to design-time techniques, which are more or less over-design, adaptive circuit spends extra power only when it is needed. Figure 1 gives an example of the architecture of the adaptive circuit, the controller tunes the circuits based on the observation of the sensors.

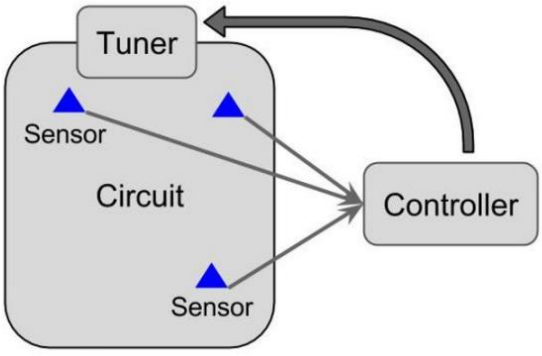

Figure 1 An Overview Architecture of Adaptive Circuits

II-C. Static Timing Analysis

STA is a fundamental tool to support VLSI circuit design. Traditional STA estimation tends to be pessimistic but will ensure the circuit design and avoid hold-time variation. And all the device parameters like the gate length, wire width are treated as deterministic in STA.

There are two main types of STA, one is path-based and another is block-based. The path based STA [11] performs the path enumeration to identify the critical paths and check the timing satisfaction. The block-based STA propagates the arrival time (AT) in topological order (required arrival time (RAT) in reversed topological order) and checks AT and RAT at each node. The gates along the critical paths should have the same negative timing slack. Figure 2 shows an example of path-based and blocked-based STA.

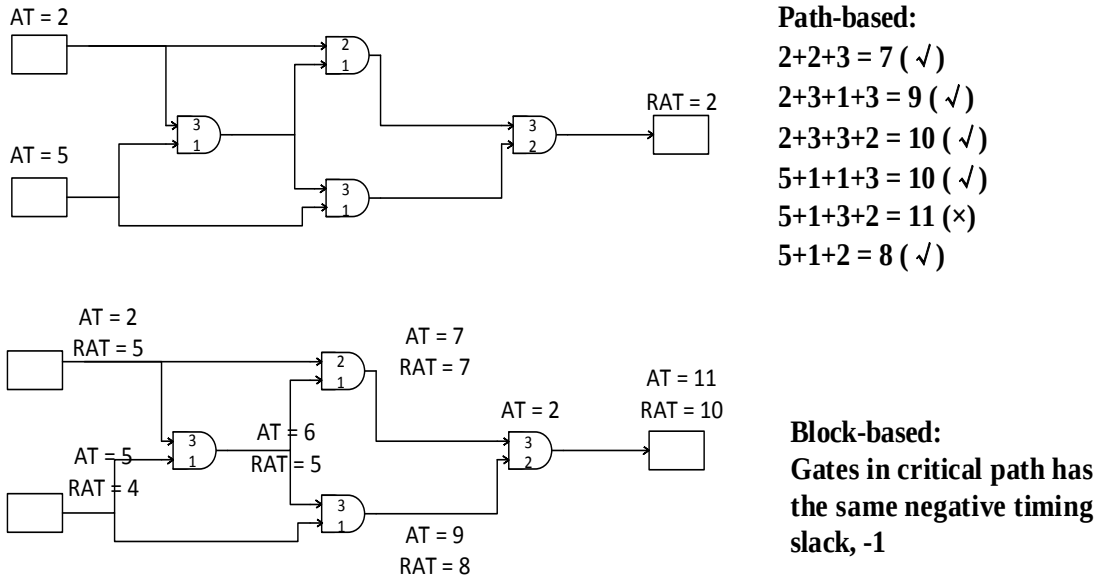

Figure 2 Arrival Time Propagation of Path-based and Block-based STA

The weakness of PCA based SSTA is that it usually needs to analyze a large number of paths, which implies a large computing cost. By contrast, the block-based STA has linear runtime and uses sum and max function to compute the delay for each gate.

STA can handle the die-to-die variation by analyzing multiple corner files. However, the limitation is that it cannot rigorously model the within-die variations especially the spatial correlation.

II-D. Statistical Static Timing Analysis

The SSTA is proposed to remedy the STA's weakness and effectively model the process variations especially the within-die variation. The within-die variations typically have two types: the spatially correlated variations and the independent variations. The physical parameters are statistically independent from other devices. It is showed in [9], the fully-correlated assumption may overestimate the circuit delay while the non-correlated assumption may underestimate the delay spread.

SSTA uses probabilistic model to treat each device parameter as random variables. The path-based and block-based STA manner can still be used here to do circuit traversal. The difference is that SSTA needs to propagate the delay PDF rather than deterministic values and the sum/max operation should be performed statistically. Due to the correlations, the delay propagation in SSTA becomes a difficult topic. Besides the spatial correlations mentioned above, topological correlations and nonlinear max operation also makes the SSTA problem more difficult to solve.

There are several approaches to deal with the abovementioned problems. SSTA can be carried out as Monte Carlo simulations which repeatedly draw a number of samples for STA runs. The Monte Carlo based SSTA is typically reliable and easy to perform. But the problem is that the runtime is significant when the sample space becomes large.

To avoid the large runtime of Monte Carlo simulations, one can propagate probability distributions of timing information in a couple of circuit traversals like in STA. In such SSTA, a main challenge is the complexity of handling spatial correlations among the random delays. An elegant approach to account for the correlation is Principal Component Analysis (PCA). A PCA-based SSTA is first introduced in [16]. The key idea is to represent each random delay variable d as

$$d = d_0 + k_1\epsilon_1 + k_2\epsilon_2 + \cdots + k_m\epsilon_m \quad (1)$$

where d_0 is the nominal delay, $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_m\}$ is a set of independent Gaussian random variables which has zero mean and unity variance, and k_1, k_2, ... are the coefficients. Besides the propagation of principal components $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_m\}$, the computation of principal components for arrival time and required arrival time is much more expensive than its counterpart in conventional STA. Yet, the PCA-based SSTA is normally much faster than running Monte Carlo simulations.

Adaptive circuit designs cause additional computation load to SSTA as many different tuning scenarios need to be covered. In [12], the PCA-based SSTA is extended for adaptive circuit designs.

II-E. Parallel Programming

II-E.1 Parallel Programming Model for Graph Algorithms

EDA software involves many advanced algorithms in diverse domains. Our task is to find the parallelizable patterns among the algorithms for adaptive circuit timing analysis. The techniques for solving STA and SSTA are mostly based on graph theory. The breath-first search (BFS) and depth-first search (DFS) traversal can be used here to propagate the gate parameters. GPU-accelerated solutions are proposed in [22] for many of the graph problems and give code skeletons for the graph traversal algorithms.

```
1 Create and initialize the working arrays in GPU
2 while true do
3       for each vertex/edge in parallel do
4               invoke Kernel 1
5       end for each
6       synchronize
7       for each vertex/edge in parallel do
8               invoke Kernel 2
9       end for each
10      synchronize
11      check termination condition; continue if necessary
12 end while
```

Algorithm 1 GPU Code Skeleton for Graph Problems [22]

The main idea of this code skeleton is to find the parallelizable nodes/edges in the current topological level (line 4 Kernel 1) and do some calculation. Then in line 8, the algorithm moves forward to the next topological level of the graph.

STA/SSTA can use this pattern to do parallelization easily. For example, in the block-based STA, we can do the BFS-like circuit traversal and calculate the PDF for each gate concurrently as line 4 of Algorithm 1.

When the computing bandwidth of parallel devices is infinite, the algorithm above can be an optimal solution because it always processes all of the parallelizable nodes concurrently. However in reality, the number of nodes can be processed simultaneously is bounded by the computing abilities of the devices (i.e., number of cores, memory bandwidth and etc.). We will discuss our scheduling method regarding this issue in later chapters.

II-E.2 Parallel Programming Using GPU

1 Parallel Programming Using GPGPU

Graphics Processing Unit (GPU) is designed for graph applications. Compared to CPU, a GPU typically consists of many small processor cores and is good at massive parallel computing because of its SIMD (Single Instruction Multiple Data) fashion. GPU can support data intensive computing for scientific research and other applications. CUDA and OpenCL are the application program interfaces (API) for developers to use a GPU for general purpose computing as known as the GPGPU. CUDA is supported by Nvidia [23] for its own GPU products and OpenCL is supported by Khronos Group [24] for heterogeneous platforms. CUDA architecture provides more detailed control of Nvidia GPU and OpenCL provides a more portable API across multiple platforms. In this work,

we choose CUDA because we can have more controls like memory management using some provided APIs.

2 Nvidia GPU Architecture

The recently released GPU architecture of Nvidia is Kepler which is also the architecture we consider in our experiments. The overall of the multiprocess structure of Kepler is shown in Figure 3.

Figure 3 Multiprocessor Architecture of Kepler [25]

The multiprocessor contains 192 single-precision CUDA cores, 64 double-precision units and 65536 32-bit registers [25]. Compared to last version, Kepler has more computing resources and more warp controllers which can dispatch instructions to CUDA cores concurrently. Another important feature of Kepler architecture is the support of dynamic parallelism. With this capability, GPU can execute kernels itself instead of waiting for the assignments from CPU.

3 CUDA Programming Model

CUDA is GPGPU development platform provided by Nvidia. It contains both the hardware devices and software model. Figure 4 shows the programming model of CUDA. The smallest processing unit is the thread. Multiple threads make up a thread block and multiple blocks make up a grid. The threads in a block reside on the same multiprocessor so the number of threads of a block is limited. Each thread/block has a unique index inside a block/grid. This index can be used by the programmer to control the desired thread or block in programming. Figure 5 shows the relationship between the software model and hardware devices in CUDA.

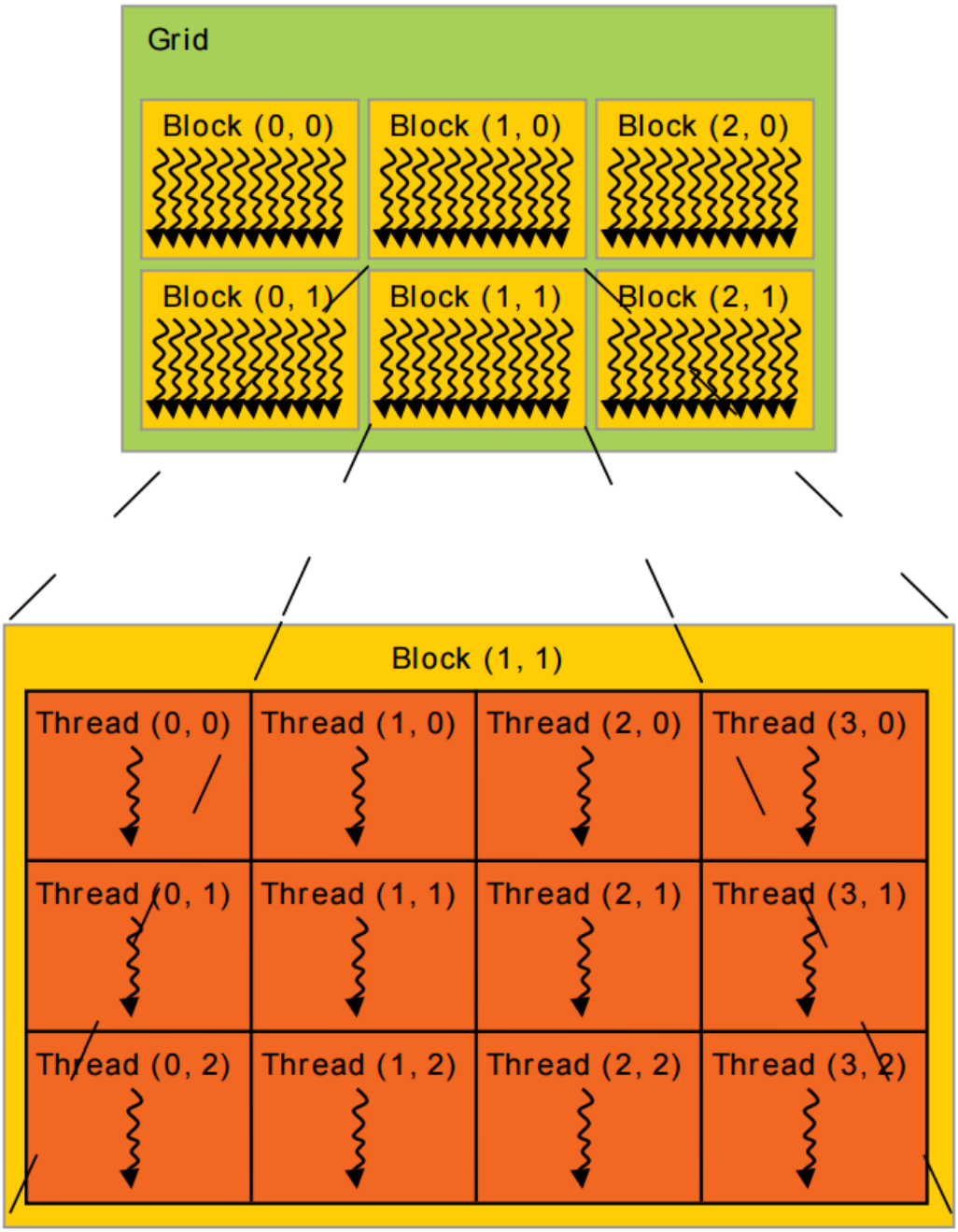

Figure 4 CUDA Programming Hierarchy [30]

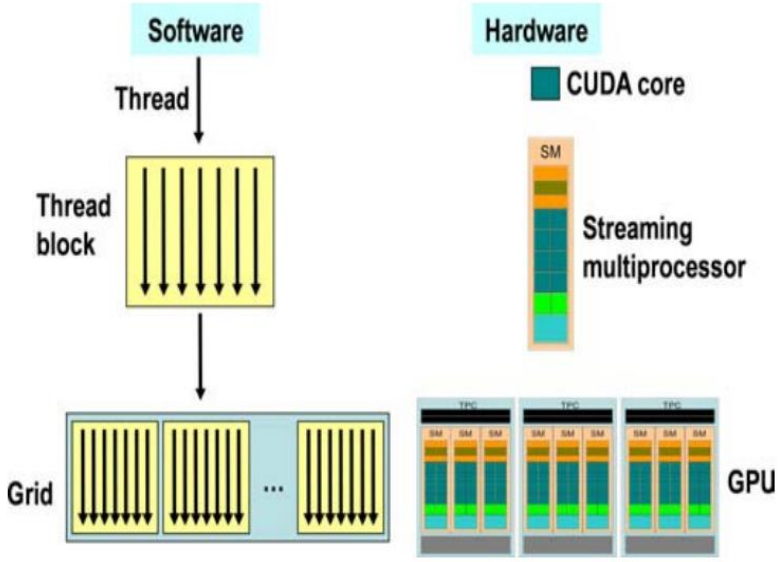

Figure 5 Software Models against Hardware Hierarchy [33]

From the software program perspective, a piece of code is called kernel. For Kepler GPU, 32 consecutive threads form a warp, which share the same kernel. All threads of a warp must be executed on the same multiprocessor. Each multiprocessor has a scheduler that dispatches warps onto its CUDA cores. Multiple warps constitute a thread block.

II-F. Related Works on Adaptive Circuit Design

A joint gate implementation selection and adaptivity assignment methodology for adaptive circuit optimization is proposed in [5]. The selection problem is transformed by Lagrangian relaxation and solved by Joint Relaxation and Restriction (JRR) [18] which is a dynamic programming like solution search. The idea of JRR is to perform topological and reversed topological circuit traversals and propagate solution candidates along a

circuit graph. The adaptivity assignment is solved by sensitivity-based optimization [26]. Liu and Hu [19] proposes a GPU-based acceleration for joint gate sizing and threshold voltage assignment for a circuit optimization problem in [18]. They observe that during the JRR traversal in [18], the different gates undergo identical computations and they can be parallelized using the GPU. In this work, we explore the parallelism of the optimization process which is described in [17]. Multithreading are used to accelerate the design process.

A reduction-based acceleration technique for SSTA of adaptive circuit is proposed in [12]. It prunes the adaptivity configurations by analyzing the performance and discarding the configurations which for sure fail in satisfying timing constraints. Circuit partitioning and block merging are also proposed to reduce the problem space. However, these reduction techniques may not be fully reliable when they are applied to a wide range of realistic adaptive circuits. In our work, we enumerate the possible adaptivity configurations and accelerate the timing analysis using the GPU. The experiment shows it can get a significant speedup and maintain the same precision at the same time.

CHAPTER III

PARALLELIZATION OF PCA-BASED SSTA FOR ADAPTIVE CIRCUIT*

III-A. Problem Statement

III-A.1 PCA-based SSTA

A statistical timing graph is defined in [16] to represent the SSTA problem in a combination circuit. The traditional STA performs a program evaluation and review technique (PERT) like traversal and calculates the arrival time using the deterministic sum and max operation in topological order. The SSTA is, however, to propagate the PDF and do statistical sum and max operations. SSTA is similar to STA but it can handle variations among timing paths. The propagation of STA is shown in the top subfigure of Figure 6 in which we use deterministic parameters. The SSTA has the same propagation manner (bottom subfigure of Figure 6), but the data becomes PDF. The vertical arrows in the top subfigure of Figure 6 represent the deterministic delay of the gates while the distribution curves in the bottom subfigure of Figure 6 represent PDFs of the gates. Therefore, the MAX operation in g3 then becomes statistical max operation.

First, we formulate the SSTA problem for process variation aware timing verification as in [16].

---

*Part of this chapter is reprinted with permission from "GPU Acceleration for PCA-Based Statistical Static Timing Analysis" by Y. Shen and J. Hu, 2015. In Proceedings of the International Conference on Computer Design, pp. 674-679, ©2015 IEEE.

Problem 1: Given a directed timing graph G = (*V*, *E*) where *V* is the set of nodes and *E* is the set of edges. Each edge has a weight d_i which is a random variable. The d_i contains the gate delay and interconnection delay information and d_i of every edge are partially correlated with each other.

The objective of the SSTA problem is to find all the arrival time distributions of the nodes in G and then calculate the timing yield for the whole circuit.

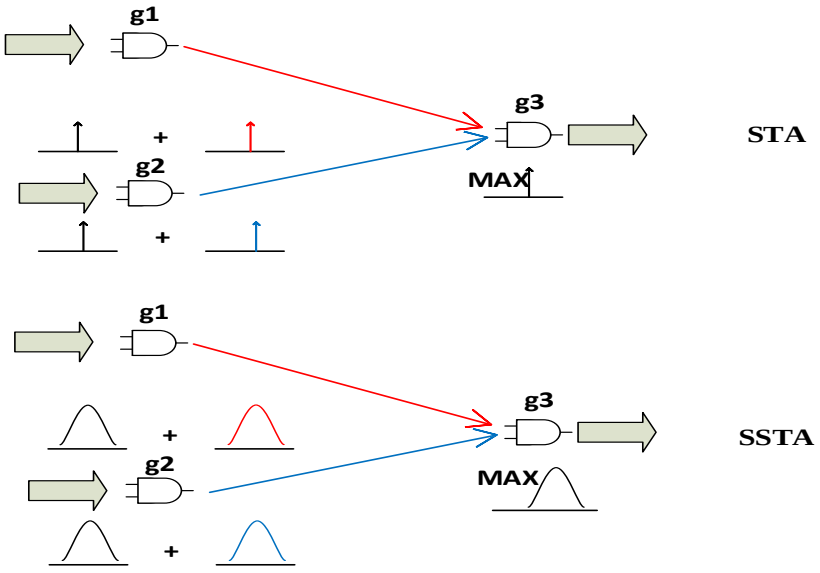

Figure 6 STA vs SSTA

III-A.2 GPU Scheduling

Though PCA SSTA is faster than Monte Carlo SSTA as we mentioned previously, the statistical max and sum functions are still time-consuming. Moreover, when we try to

examine multiple adaptivity scenarios for adaptive integrated circuits [12], the runtime may become overly large.

Therefore it is reasonable to parallelize the PCA-based SSTA for conventional and adaptive circuits. For almost every parallel computing work, a central problem is how to partition the overall computing into small tasks and schedule the tasks onto parallel processing units [27]. The objective is typically to minimize the makespan, which is the time difference between the start and end of the computing. Therefore when we try to solve *Problem 1* on GPU, we should focus on how to partition the set V under the precedence constraints implied by E. From the hardware perspective, if we can make the better use of GPU resources (processing cores, registers, shared memory and etc.), the runtime can be decreased accordingly. We generally formulate the GPU scheduling problem for PCA-based SSTA as *Problem 2*,

> *Problem 2*: Given a directed timing graph G (V, E) where V is the set of nodes and E is the set of edges. PCA-based SSTA performs block-based circuit delay propagation on G and outputs the timing yield at primary output. Partition the timing graph G into small parallelizable sets (we call it batches) such that the total makespan of parallel computing is minimized.

The following sections will focus on *Problem 2* and discuss more details. To examine multiple adaptivity scenarios, we just need to repeat the PCA SSTA to calculate each of them.

III-B. Simplification Strategy

The scheduling problem is difficult to solve due to the following reasons: on one hand dynamic scheduling conceivably results in large overhead considering the fine-grained parallelism at GPU, on the other hand each logic gate has an unknown computation time on GPU in advance.

To circumvent the difficulties, we propose a simplification formulation — batch-based static scheduling. That is, the gates are partitioned into ordered batches, which are processed in topological order. To process a batch of gates, one needs to ensure that all their predecessor gates have already been processed in earlier batches. In addition, the gates in the same batch should be independent of each other so that they can be processed concurrently. Figure 7 shows a small example for the simple approach. In this example, G1-G4 are the primary input gates and they form the first batch. The successors of G1-G4 which are ready will be grouped into the second batch. “Ready” means that all the predecessors of the gate have already been processed. Then the next batch is generated in the same manner.

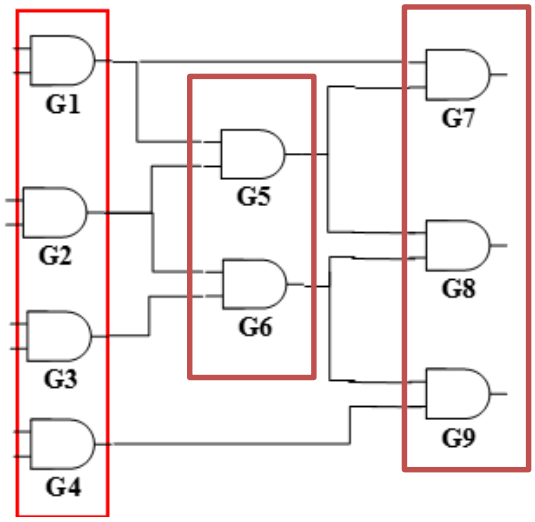

Figure 7 An Example of Simplification Strategy

III-C. Analysis of the Simplified Problem

The makespan is determined by the total number of batches and the computing time of each batch. To minimize the number of batches, we wish the size of each batch, in terms of the number of gates, is as large as possible. And we also want to compact the computing time of each batch.

The first problem, which the simplified approach does not consider, is the size of each batch, which may affect the total number of batches. The algorithm actually groups the nodes into the original circuit levels. As shown in Figure 7, the toy circuit has three levels which are marked by three rectangles. As discussed in Chapter II, PCA-based SSTA accounts for correlation by using principal components (PCs). During the timing graph traversal, the calculation of PDF of the AT for one node (i.e., the mean and standard deviation value) needs the principal components (PCs) which should be propagated as

well. We need to take the memory consumption of the PCs into consideration when designing the parallel algorithms because GPU's on-chip memory resources are limited and the off-chip memory access is expensive. Due to this restriction, the maximum number of gates in each batch is limited by the shared (on-chip) memory size in the GPU. We should also notice that the number of total batches highly depends on how we partition the circuit graph because there are precedence constraints among the gates.

Another problem is the runtime bottleneck of each batch due to unbalanced computing time for nodes in a batch. Figure 8 gives the runtime of G1-G4 that are the logic gates within the same batch. G1 – G4 start together, but G1 requires much more calculation. The speedup will be compromised because that the computing for G1 forms a bottleneck. To minimize the computing time of each batch, we hope that all tasks (or logic gates) in a batch incur about the same computation time so that no task forms a bottleneck.

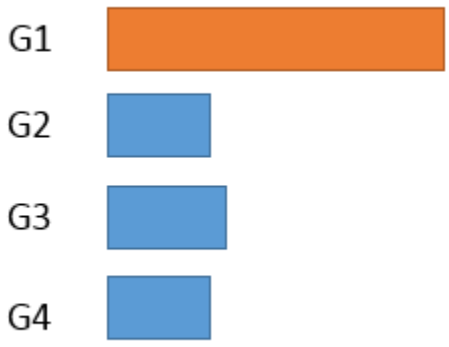

Figure 8 A Runtime Bottleneck Is Formed Due to Unbalanced Computing Time among Gates in A Batch.

It is still a challenge on how to simultaneously minimize the number of batches and equalize gate computation time in each batch. A naive approach is to keep a ready list like in the list scheduling [27] and randomly select up to K gates (batch size limitation) for processing in each iteration. Such approach does not pay attention if processing a gate can release sufficient number of gates to be ready for the next iteration [19]. The result may be quite below the GPU processing capacity. Consequently, the GPU is under-utilized and the makespan is unnecessarily long. And it also neglects the bottleneck problem in one batch – the chosen gates may have the non-uniform runtimes.

III-D. Solution for GPU Scheduling

We handle two main difficulties of this scheduling problem separately. First we model the makespan problem to an N-tasks-K-processors problem that is a well-known NP-complete problem [28]. Then we handle the makespan and bottleneck together.

III-D.1. Minimize the Makespan

Figure 9 shows an example of N-tasks-K-processors problem. The three rows in Figure 9 represent three processors and the length of every column represents a time duration for a single task. An assumption is made here that every task consumes identical processing time. The goal is to minimize the number of columns under the precedence constraint. We can take every task node Ti in Figure 9 as the processing task of a logic gate when do the SSTA. Then every row of the table becomes a GPU core (We assign

each logic gate to one GPU core). We currently make the same assumption that processing time of every gate is identical and we will discuss realistic cases without this assumption.

Fortunately, this problem can be solved by the famous Coffman-Graham algorithm [29], which is a polynomial time heuristic algorithm when there are more than two processors. The Coffman-Graham algorithm generates a numerical label λ_i for each task $v_i \in V$ such that a schedule following the labels would result in the minimum makespan. The convention is that a task with larger label value is processed earlier.

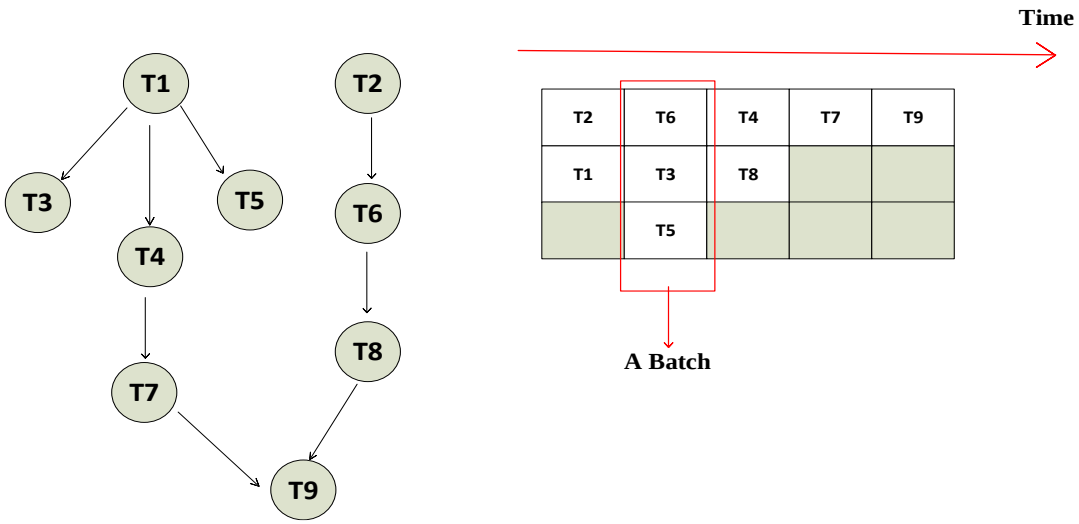

Figure 9 N-tasks-K-processors Scheduling Example

III-D.2. Balancing Computing Loads in a Batch

The labels obtained from the Coffman-Graham algorithm are based on the assumption that all tasks (or gates) cost the same computation time. In SSTA, this assumption rarely holds and the computation time of each gate $v_i \in V$ is proportional to

its fanin size ϕ_i. So we should take both the labels λ_i and fanin size ϕ_i of logic gates into consideration.

Our batch generation algorithm is a topological traversal of *G* while a ready-gate list is maintained like in the list scheduling [27]. Before we do the scheduling, we redefine the label of each gate as a scheduling priority as

$$q_i = \alpha \cdot \frac{\lambda_i}{\lambda_{max}} + (1-\alpha) \cdot \frac{\phi_i}{\phi_{max}} \qquad (2)$$

where λ_{max} and ϕ_{max} are the maximum label and fanin size, respectively, among all gates in the current ready-gate list. If we group the gates with high and similar priority *q* values together into a batch, the gates in a batch would have similar fanin sizes, which remedy the load imbalance problem, and similar label values, which to certain degree embrace the Coffman-Graham algorithm result

III-D.3. The Combined Batch Scheduling Algorithm

After we define the priority of each gate for the scheduling, we can use the list scheduling algorithm as the skeleton to generate the batches.

Input : Circuit G = (V, E), max batch size K
Output : Disjoint batches $B_1 \cup B_2 \cup ... = V$
1 initialize *ready_list*
2 j ← 1 // batch_index
3 while *ready_list* ≠ ∅ do
4 if | *ready_list* | ≤ K then
5 $B_j = \{v_i | v_i \in ready\_list\}$
6 end
7 else

8 sort *ready_list* in decreasing order of q
9 index gates in *ready_list* by the order
10 $\delta_{min} = -\infty$
11 for i = 1; i $\leq |ready\_list| - K + 1$; i++ do
12 $\delta_i = q_i - q_{i+K-1}$ //q_i corresponds to v_i
13 if $\delta_i < \delta_{min}$ *then*
14 $\delta_{min} = \delta_i$
15 $B_{min} \leftarrow \{v_i, v_{i+1}, \dots, v_{i+K-1}\}$
16 end
17 end
18 $B_j \leftarrow B_{min}$
19 end
20 j++
21 update *ready_list*
22 end

Algorithm 2 Batch Generation

The pseudo code of batch generation algorithm is shown in Algorithm 2. In line 1, the ready list is initialized by gates whose predecessors are all primary inputs. In line 4 and 5, if the number of gates in the ready-list is no greater than the maximum batch size *K*, then all gates in the ready list form a batch. Otherwise, we select *K* gates in the ready list with the minimum difference on their *q* values to form a batch (line 10-18). This selection procedure follows the ascending order of *q* values (line 8-9). In other words, gates with large q values have higher priority to form batches. After a batch is formed, the ready list is updated (line 21) by removing the gates that have been added to batches and adding new ready gates.

The Pre-labeling and Batch generation are performed once on CPU and the scheduling result is stored. We will reuse this scheduling result for multiple scenarios analysis of the adaptive circuit design which will be discussed later.

III-E. Memory Techniques

As mentioned before, the PCA-based SSTA is a data intensive algorithm for GPU to perform. The information as well as the PC values of a single gate of large circuits in our experiments (over 10K gates) can make the memory size of a single gate to 0.1K bytes. Many memory access patterns are introduced in [30] to help enhance the device memory usage of the GPU. We next show how we embrace these design patterns in our data structures and algorithms.

CUDA has a memory coalescing mechanism to reduce global memory access pains. It shows that the CUDA warp tends to coalesces the memory accesses into one or more memory transactions depending on the size of the words accessed by each thread and the distribution of the memory addresses across the threads [30]. Our timing graph after batch scheduling will store gate data of the same batch adjacent to each other. Since the data of the same batch are utilized at the same time (i.e., the statistical max and sum operations for each gate data in the same batch), we can use the coalescing mechanism to simultaneously fetch the data and therefore the efficiency of data access is improved. Another important design pattern is that the memory access should be aligned because the instructions of global memory support accessing the words of size equal to 1, 2, 4, 8, or 16 bytes [30]. This requires us to build the struct as

```
struct __align__(16){
        float x;
        float y;
        ……
}
```

Each arithmetic operation takes about 20 clock cycles while an access to global memory may take more than 400 clock cycles [31]. In this regard, a well-known approach is to increase the parallelism and thereby hide the latency. And accessing the global memory of GPU is typically slower compared to the on-chip shared memory/L1 Cache. We can exploit the on-chip fast shared memory to help us hide the latency to further improve the memory performance.

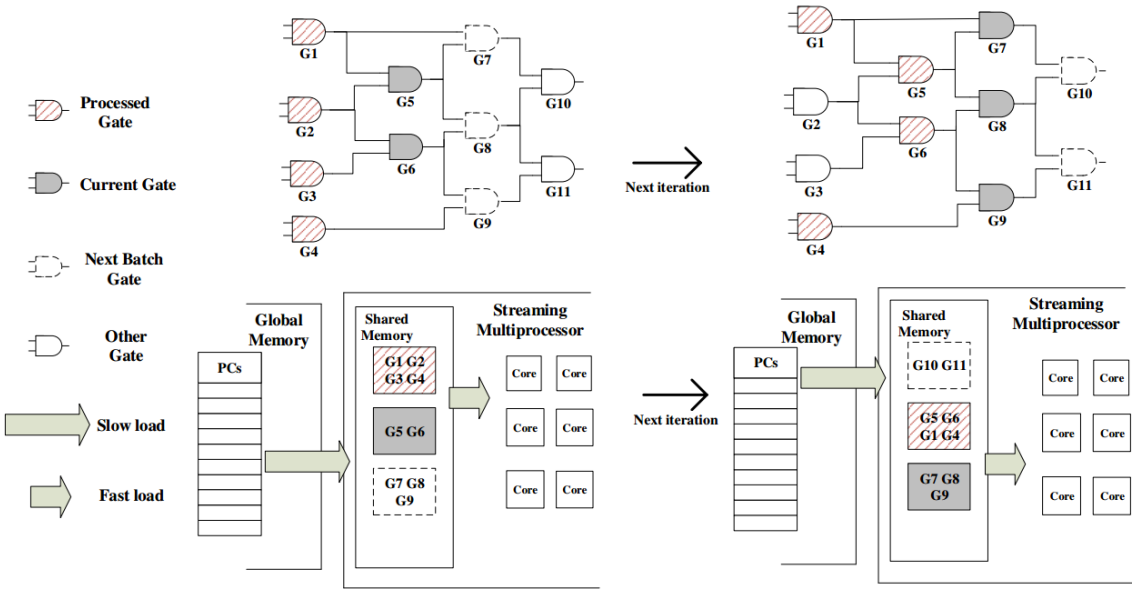

Figure 10 Example of Moving from One Batch to Another in Iterations and the Shared Memory Usage

We show how to hide memory latency for SSTA using an example in Figure 10. When processing gate G5 and G6, we need timing data of G1, G2, G3, G5 and G6. In processing the next batch (G7, G8 and G9), we still need to use the timing data of G5 and G6. We organize the shared memory in a pipelined fashion as shown in the lower half of Figure 10. When gates G5 and G6 are processed, data of G1, G2, G3, G5 and G6 reside in the shared memory. Meanwhile, data of G7, G8 and G9 are loaded from the global memory. In other words, the computing of G5 and G6 hide the latency of loading data of G7, G8 and G9. The maximum batch size K (see Chapter III-C) is decided by the amount of data we can store at the shared memory.

III-F. Handling Adaptive Circuit Designs

The SSTA for adaptive circuit designs are very time consuming since many different adaptivity scenarios need to be covered. Reduction-based speedup techniques are described in [16]. Compared to the work of [16], our approach can produce the same results as sequential computing without reduction, i.e., no approximation or errors are incurred. Since the SSTAs for different adaptivity scenarios are independent of each other, we simply add scenario-level parallelism upon the parallel techniques described in previous sections. Specifically, we allocate the SSTA for one adaptivity scenario to one thread block as shown in Figure 11. SSTA within the thread block is performed as the aforementioned batch-based scheduling, latency hiding, etc. The final timing yield for adaptive circuits are captured based on the timing yields of different scenarios that are analyzed and processed concurrently.

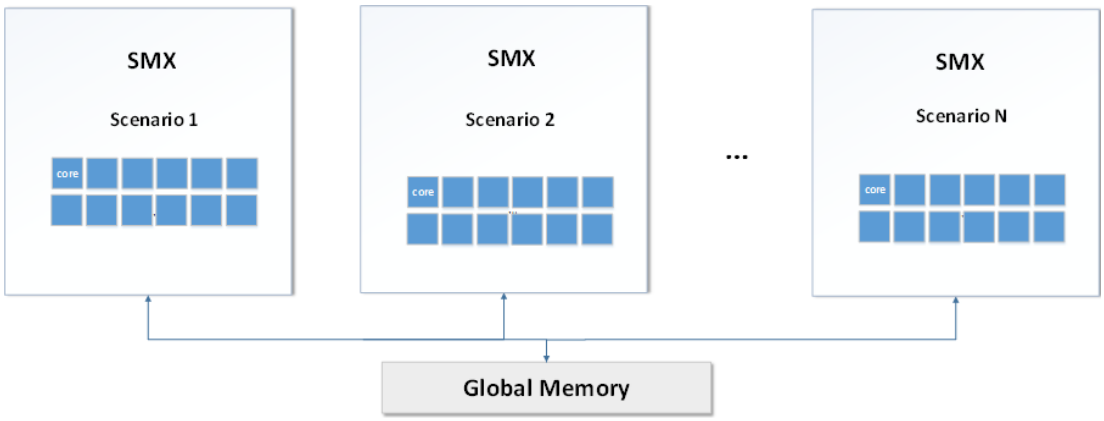

Figure 11 The Illustration of Examination of Multiple Adaptivity Scenarios

CHAPTER IV

PARALLELIZATION OF GATE IMPLEMENT SELECTION

IV-A. Gate Implementation Selection

Gate implementation selection is to simultaneously choose the gate size and threshold voltage for each gate in the circuit from a given cell library [17]. It is solved by Lagrangian relaxation in [18]. The bottleneck of the design process is the Lagrangian subproblem which is performed using dynamical-programming like circuit traversal proposed in [18]. Algorithm 3 shows this DP like traversal algorithm which is also called JRR.

```
1 begin
2 Phase I: Initial solution by relaxation
3 history consistency relaxation via topological traversal
3 history consistency restoration via reversed topological traversal
4 Phase II: Iterative refinement by restriction
5 while not converged do
6       generate candidate solutions with restrictions via topological order search
7       select solution via reverse topological order search
8 end while
9 end
```

Algorithm 3 Joint Relaxation and Restriction (JRR) Algorithm for Simultaneous Gate Sizing and Vt Assignment Algorithm [18]

Line 5-7 iteratively preform the topological and reversed topological order traversal to generate the candidate solutions. And the number of candidate solutions may

be very large depending on which cell library is used. Therefore, the runtime of the gate implementation selection in [17] is usually very long and we also need to accelerate this part for the adaptive circuit design.

IV-B. Multithreading

The batch scheduling can be taken as a general circuit parallel topological traversal templet. We also apply the batch scheduling to accelerate dynamical-programming-like circuit traversal in [17] using multithreading. The threads take the logic gates from the same batch in first-come-first-serve manner. After a batch is processed, the threads synchronize and move to the next batch. The difference from the GPU acceleration [19] is that the threads take turns to fetch the computing tasks from one batch while GPU itself dispatches tasks to the GPU threads at one time.

```
Input: Batch queue BQ
Output: Processed BQ
1 Pointer = BQ.front();
2 While (pointer < BQ.size())
3       mutex.lock()
4               Cell = pointer -> value
5               pointer = next position
6       mutex.unlock()
7       Size_Vt_Selector (Cell)
8 End while
```

Algorithm 4 Multithreading Using Mutual Exclusive

Algorithm 4 is an example for the relaxation part of the circuit traversal. That is we relax the history consistency constraint and then propagate the solution in reversed topological direction [18]. Notice that a gate will be locked to prevent being modified by other threads (Line 3-6). We also maintain a thread pool to manage the idle threads to fetch data accordingly and keep all the threads busy.

CHAPTER V

EXPERIMENTS*

V-A. Experimental Setup

We use ISPD'13 gate sizing benchmark suites [32] to test our design. The circuit graph is extracted from the benchmark. The largest circuit in ISPD'13 contains over 150K logic gates.

We need to compare the sequential and parallel algorithm to examine the effectiveness of our design. The sequential SSTA [14] and gate implement selection [17] are implemented in C++ and tested on Intel Xeon E3-1231 3.4 GHz CPU with 16GB DRAM. The parallel counterparts are implemented with CUDA C and tested on NVIDIA GeForce GTX780 GPU, which contains 2304 CUDA cores operating at 863MHz and 3GB global memory. The GeForce GTX780 has the Kepler GK110 architecture [25] and a typical Kepler GK110 contains 15 multiprocessors (SMXs) in total and each of them contains 192 CUDA cores. The GPU is configured to have 48K shared memory and 16K L1 cache for each SMX. All the devices are connected on the motherboard of a desktop, the host environment runs 64-bit Windows 6.2. Development platform is Microsoft Visual Studio 2013 with Nvidia NVCC compiler.

The delay and process variation models are the same as in [16] and the adaptive circuit design models are the same in [12]. Process variations are considered with spatial correlations, which are modeled as in [11] and handled in SSTA by PCA as in [16]. Depending on circuit sizes, the number of principal components varies from around 10 to around 100 for each case. The maximum batch size K is related to the number of principal components and typically around 500. The parameter α in Equation (2) empirically takes value of 0.8.

V-B. Experimental Result

In the experiments, we first compare the sequential SSTA with three variants of parallel SSTA on conventional circuits without adaptivity:

- **Naive parallel**: Like in the list scheduling [27], a ready list is maintained. Each time, up to K gates from the ready list are scheduled for GPU processing. This is a naive approach of using GPU for the PCA-based SSTA without our techniques described in Chapter III-B.
- **Our batch parallel**: This is GPU computing of the PCA-based SSTA using our batch-based scheduling but without the techniques in Chapter III-E.
- **Our best parallel**: This is the complete version of our approach using techniques in Chapter III.

All these parallel SSTA methods reach the same timing yield results as the sequential SSTA. The runtime comparison among them are shown in Table 1. On average, our best approach achieves 22.6× speedup, in which near a half is attributed to our

techniques. One observation is that the acceleration is usually greater for larger cases, where the runtime saving is more desired.

Circuit	#gates	Sequential	Naïve Parallel		Our Batch Parallel		Our Best Parallel	
		Runtime(ms)	Runtime(ms)	Speedup	Runtime(ms)	Speedup	Runtime(ms)	Speedup
usb_phy	0.6K	9.3	6.9	1.3×	6.9	1.3×	5.4	1.7×
pci_bridge32	25K	250.4	25.9	9.7×	17.7	14.1×	12.5	20.0×
fft	30K	324.4	84.1	3.9×	61.0	5.3×	42.6	7.6×
cordic	40K	460.3	130.8	3.5×	122.8	3.7×	87.5	5.3×
des_perf	100K	1763.8	95.1	18.5×	67.1	26.3×	61.8	28.5×
edit_dist	130K	2312.1	172.3	13.4×	106.5	21.7×	92.7	24.7×
matrix_mult	150K	3342.1	254.5	13.1×	164.2	20.4×	135.5	24.7×
			Average	12.8×		19.3×		22.6×

Table 1 Runtime Comparison for GPU-based Parallel SSTA VS. Sequential SSTA on Conventional Circuit Designs [34]

We further collect data to show how our techniques take effects. A key algorithm design objective in Chapter III is to minimize the number of batches for runtime reduction. From Table 2, the number of batches obtained from our batch parallel method is indeed significantly less than that from the naive parallel method. The effectiveness of the techniques in Chapter III-E is also evidenced by the throughput data depicted in Figure 12. The throughput is measured by GFLOPS (Giga FLoating-point Operations Per Second). The throughput improvement from techniques of Section IV is more obvious in three large circuits, each of which has at least 100K gates. For circuit *matrix_mult*, the throughput is increased from 0.62 to 1.15 GFLOPS.

Circuit	Naïve Parallel	Our Batch Parallel	
	#batches	#batches	Reduction
usb_phy	12	12	0.0%
pci_bridge32	42	33	21.4%
fft	55	48	12.7%
cordic	102	99	2.9%
des_perf	72	55	23.6%
edit_dist	121	98	19.0%
matrix_mult	137	110	19.7%
		Average	18.6%

Table 2 Comparison of the Number of Batches between Naive Parallel and Our Batch Parallel [34]

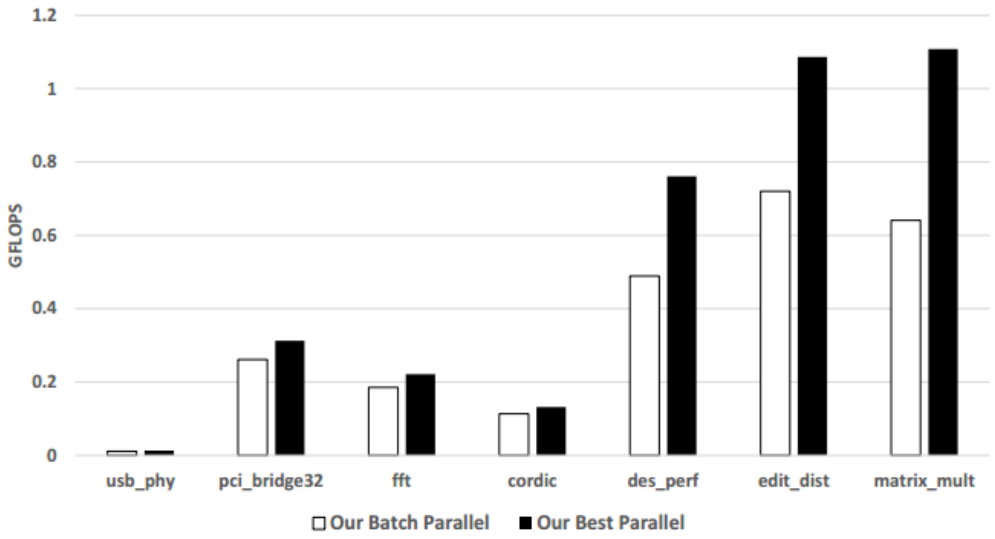

Figure 12 Throughput of Our Batch Parallel and Best Parallel [34]

We investigate the impact of parameter α in Equation (2) in experiments. Our batch parallel method is performed with α value varying between 0 and 1 on a medium case *fft* and a large case *edit_dist*. The obtained runtime-α curves are depicted in Figure 13. One can see that considering similar fanin size alone (α = 0) or the Coffman-Graham label

alone (α = 1) result in large runtime. Small runtimes are obtained when both factors are accounted for, especially when α = 0.8.

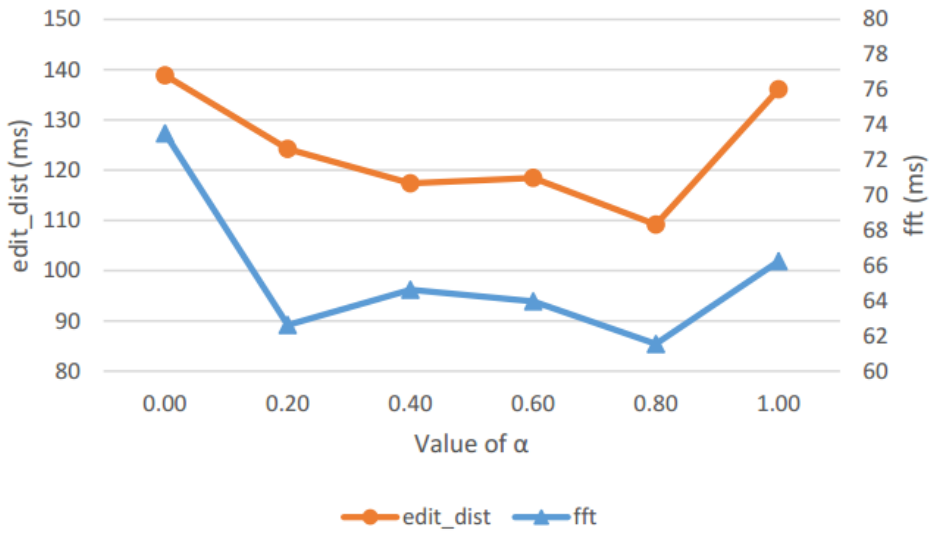

Figure 13 Runtime of Different α Values Using Our Batch Parallel Method [34]

We also compare our approach with Monte Carlo-based SSTA of both sequential CPU computing and parallel GPU computing, which is implemented in a way similar as [14]. Each Monte Carlo simulation takes 10K and 20K runs for small and large circuits, respectively. The results are displayed in Table 3. On average, our approach has −0.77% error on timing yield estimation. However, our approach is about 39× faster than the GPU parallel computing of Monte Carlo SSTA.

Circuit	Monte Carlo (MC) SSTA			Our Best Parallel PCA-SSTA				
							Speedup	
	Timing Yield	CPU(ms)	GPU(ms)	Timing Yield	%Error	GPU(ms)	vs MC-CPU	vs MC-GPU
usb_phy	0.947	228.4	117.1	0.947	0.00	5.4	42.3×	21.7×
pci_bridge32	0.992	23719.3	824.8	0.983	-0.91	12.5	1897.5×	66.0×
fft	0.968	27135.7	1171.9	0.977	0.96	42.6	637.0×	27.5×
cordic	0.945	35313.5	1819.4	0.943	-0.21	87.5	403.6×	20.8×
des_perf	0.994	168913.0	2965.1	0.975	-1.91	61.8	2733.2×	48.0×
edit_dist	0.997	209116.0	3747.5	0.983	-1.40	92.7	2255.8×	40.4×
matrix_mult	0.998	266607.0	4774.4	0.999	0.05	135.5	1967.6×	35.2×
				Average	-0.77		1985.8×	39.2×

Table 3 Runtime Comparison between Our Best Parallel PCA-SSTA and Monte Carlo SSTA on Conventional Circuit Designs [34]

Experiments are also performed on adaptive design version of the benchmark circuits. Each circuit is partitioned into 6 adaptivity blocks and each block can be independently configured to either high VDD or low VDD according to the voltage interpolation [2]. The results are summarized in Table 4. One can see that the speedup from our techniques is 134× on average, which is even greater than in conventional circuit designs.

Circuit	Sequential	Our Best Parallel	
	Runtime(ms)	Runtime(ms)	Speedup
usb_phy	156.8	38.6	4.1×
pci_bridge32	19106.0	248.7	76.8×
fft	26116.4	397.6	65.7×
cordic	35115.8	535.5	65.6×
des_perf	131553.9	937.7	140.3×
edit_dist	179853.4	1188.2	151.4×
matrix_mult	265426.6	1689.2	157.1×
		Average	134.1×

Table 4 Runtime Comparison between Our Best Parallel SSTA and Sequential SSTA on Adaptive Circuit Designs [34]

Our best parallel runtime shown in Table 4 is the total runtime including the CPU and GPU runtime. In addition, we show the ratio between the GPU SSTA runtime for adaptive circuits and the entire runtime in Figure 14. The ration ranges from 0.4 to 0.74 and the large circuits have higher ratio of GPU runtime.

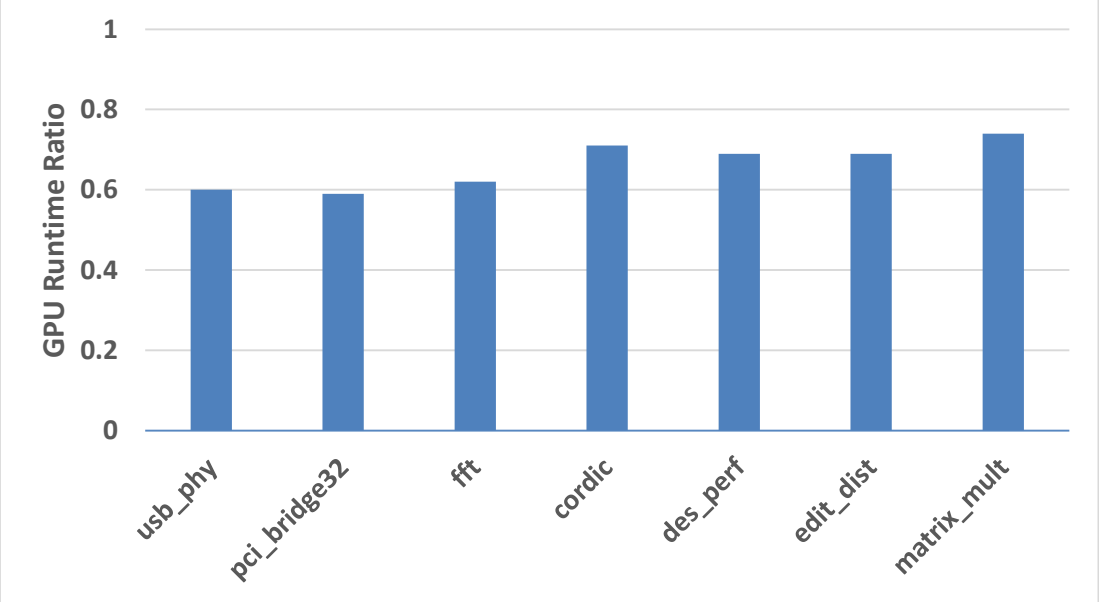

Figure 14 GPU Runtime Ration of Adaptive Circuits SSTA

We apply the multithreading techniques described in Chapter III. The adaptivity assignment is jointly performed with gate implementation selection, whose runtime is reduced by 5X on 8-core CPU processor and 8GB RAM as shown in Table 5.

Circuit	Sequential	Multithreaded on 8-core processor	
	Runtime(s)	Runtime(s)	Speedup
usb_phy	1.5	0.9	1.7×
pci_bridge32	102	21	5.0×
fft	55	11	5.1×
cordic	89	18	4.8×
des_perf	173	37	4.6×
edit_dist	252	47	5.3×
matrix_mult	327	63	5.2×
		Average	5.0×

Table 5 Speedup by Multithreading for Gate Implement Selection

CHAPTER VI

CONCLUSION

In this thesis, we introduce parallel computing techniques to alleviate runtime cost of PCA-based SSTA and accelerate the optimization process of adaptive circuit designs. To minimize the runtime of the computing and handle the bottleneck issue, a new task partitioning and scheduling approximate algorithm is developed. After the scheduling, the circuit graph is partitioned into multiple batches within which the logic gates can be processed concurrently. Memory organization is also investigated. The fast on-chip memory helps to hide the off-chip memory latency. The memory coalescing mechanism is also exploited by arranging the data structure. The scheduling strategy is also applied for the multithreading for gate implement selection. Experiments on benchmark of ISPD'13 show that the proposed approach can achieve 22X and 134X speedup for conventional and adaptive circuit designs, respectively. It is also 39X faster than GPU parallel computing of Monte Carlo based SSTA. In addition, 5X speedup is achieved for gate implement selection by using the multithreading.

REFERENCES

[1] J. W. Tschanz, J. T. Kao, S. G. Narendra, R. Nair, D. A. Antoniadis, A. P. Chandrakasan, and V. De, "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage," IEEE Journal of Solid-State Circuits, 37(11):1396–1402, November 2002.

[2] X. Liang, G.-Y. Wei, and D. Brooks, "Revival: a variation-tolerant architecture using voltage interpolation and variable latency," IEEE Micro, 29(1):127–138, January 2009.

[3] G. Nanz and L. Camilletti, "Modeling of chemical-mechanical polishing: A review," IEEE Transaction on Semiconductor Manufacturing, 8(4): 382–389, November 1995.

[4] H. Jyu, S. Malik, S. Devdas, and K. Keutzer, "Statistical timing analysis of combinational logic circuits," IEEE Transaction on Very Large Scale Integrated Systems, (1)2:126–137, June 1993.

[5] A. Agarwal, D. Blaauw, V. Zolotov, and S. Vrudhula, "Statistical timing analysis using bounds and selective enumeration," IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, 22(9):1243-1260, September 2003.

[6] L. Zhang, Y. Hu, and C. Chen, "Statistical timing analysis in sequential circuit for on-chip global interconnect pipelining," In Proceedings of the ACM/IEEE Design Automation Conference, pp. 904–907, 2004.

[7] R. Rutenbar, L. Wang, K. Cheng, and S. Kundu, "Static statistical timing analysis for latch-based pipeline designs," In Proceedings of the ACM/IEEE International Conference on Computer-Aided Design, pp. 468–472, 2004.

[8] C. Visweswariah, K. Ravindran, K. Kalafala, S. Walker, and S. Narayan, "First-order incremental block-based statistical timing analysis," In Proceedings of the ACM/IEEE Design Automation Conference, pp. 331–336, 2004.

[9] D. Blaauw, C. Kaviraj, S. Ashish, and S. Lou. "Statistical timing analysis: From basic principles to state of the art," IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, 27(4):589-607, April 2008.

[10] A. Gattiker, S. Nassif, R. Dinakar, and C. Long, "Timing yield estimation from static timing analysis," In Proceedings of the International Symposium on Quality Electronic Design, pp. 437–442, 2001.

[11] A. Agarwal, D. Blaauw, V. Zolotov, S. Sundareswaran, M. Zhou, K. Gala, and R. Panda, "Statistical delay computation considering spatial correlations," In Proceedings of Asia and South Pacific Design Automation Conference, pp. 271–276, 2003.

[12] R. Kumar, B. Li, Y. Shen, U. Schlichtmann, and J. Hu, "Timing verification for adaptive integrated circuits," In Proceedings of Design, Automation and Test in Europe Conference, pp. 1587–1590, 2015.

[13] Y. Deng and S. Mu, "The potential of GPUs for VLSI physical design automation," In Proceedings of International Conference on Solid-State and Integrated-Circuit Technology, pp. 2272–2275, 2008.

[14] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," In Proceedings of Asia and South Pacific Design Automation Conference, pp. 260–265, 2009.

[15] V. Veetil, Y.-H. Chang, D. Sylvester, and D. Blaauw, “Efficient smart Monte Carlo based SSTA on graphics processing units with improved resource utilization,” In Proceedings of the ACM/IEEE Design Automation Conference, pp. 793-798, 2010.

[16] H. Chang and S. S. Sapatnekar, “Statistical timing analysis under spatial correlations,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24(9):1467–1482, September 2005.

[17] H. He, J. Wang, and J. Hu, “Collaborative gate implementation selection and adaptivity assignment for robust combinational circuits,” In Low Power Electronics and Design, 2015 IEEE/ACM International Symposium on, pp. 122-127, 2015.

[18] Y. Liu and J. Hu, “A new algorithm for simultaneous gate sizing and threshold voltage assignment,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 29(2):223–234, February 2010.

[19] Y. Liu and J. Hu, “GPU-based parallelization for fast circuit optimization,” In Proceedings of the ACM/IEEE Design Automation Conference, pp. 943–946, 2009.

[20] L. Scheffer, “Physical CAD changes to incorporate design for lithography and manufacturability,” In Proceedings of Asia and South Pacific Design Automation Conference, pp. 768–773, 2004.

[21] G. Nanz and L. Camilletti, “Modeling of chemical-mechanical polishing: A review,” IEEE Transactions on Semiconductor Manufacturing, 8(4):382–389, November 1995.

[22] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” In Proceedings of High Performance Computing, pp. 197–208, 2007.

[23] NVIDIA’s Next Generation CUDA Compute Architecture. Whitepaper, 2009.

[24] OpenCL: The Future of Accelerated. Whitepaper, 2011.

[25] NVIDIA Kepler GK110 Architecture. Whitepaper, 2012.

[26] J. Hu, A. B. Kahng, S. Kang, M.-C. Kim, and I. L. Markov, "Sensitivity-guided metaheuristics for accurate discrete gate sizing," In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 233–239, 2012.

[27] Y.-K. Kwok and I. Zolotov, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," ACM Computing Surveys, 31(4):406–471, December 1999.

[28] A. F. Bashir, V. Susarla, and K. Vairavan, "A statistical study of the performance of a task scheduling algorithm," IEEE Transactions on Computers, C-32(8):774–777, August 1983.

[29] C. Hanen and A. Munier, "Performance of Coffman-Graham schedules in the presence of unit communication delays," Discrete Applied Mathematics, 81(1-3):93–108, January 1998.

[30] CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/, September 2015.

[31] V. Volkov, "Better performance at lower occupancy," In Proceedings of the GPU Technology Conference, vol. 10, pp. 16, 2010.

[32] M. Ozdal, C. Amin, A. Ayupov, S. Burns, G. Wilke, and C. Zhuo, "An improved benchmark suite for the ISPD-2013 discrete cell sizing contest," In Proceedings of the ACM International Symposium on Physical Design, pp. 168–170, 2013.

[33] Y. Deng, and M. Shuai. Electronic Design Automation with Graphic Processors: A Survey. Now Publishers Incorporated, Netherlands, 2013.

[34] Y. Shen and J. Hu, “GPU Acceleration for PCA-Based Statistical Static Timing Analysis,” In Proceedings of the International Conference on Computer Design, pp. 674-679, 2015.