Acceleration of Statistical Timing Analysis Algorithm on Graphical Processing Units

A Project Report

submitted by

SUDHARSHAN. V

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY
(ELECTRICAL ENGINEERING)

AND

MASTER OF TECHNOLOGY
(MICROELECTRONICS AND VLSI DESIGN)

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2013

THESIS CERTIFICATE

This is to certify that the thesis titled **Acceleration of Statistical Timing Analysis Algorithm on Graphical Processing Units**, submitted by **Sudharshan.V**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology and Master of Technology**, is a bona fide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Nitin Chandrachoodan
Research Guide
Associate Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 28th May 2013

ACKNOWLEDGEMENTS

I would like to thank Dr. Nitin Chandrachoodan for giving me this opportunity of working in a very exciting topic that meets my interests and strengths. I would also like to thank Dr. Shankar Balachandran for giving us his guidance during our work towards the TAU Contest 2013 which was to form the base of the work that I had carried forward and extended towards GPUs.

I would also like to thank Jobin Jacob Kavalam for being the ever enthusiastic chap whose persistence and drive was a constant source of motivation and marvel for not only me but everyone in the lab doing our projects. Thanks are in order for Ramprasath who was the *guru* of SSTA and for providing different inputs and feedback on the work that I was doing. Both of them showed the value and necessity of sharing and learning from peers without which this project would be nowhere near what it is right now.

I have to thank and acknowledge the awesome year spent along with the following group - Abishek, Bharath, Bhargava, Numaan, Prasad, Siddharth and Vignesh who provided support during moments of uncertainity and provided much needed distraction outside the project activities and course work. I would also like to thank all my friends whom I met in class and outside who have made my stay at campus so memorable.

None of this would have been possible without my parents who have been a pillar of support and constant source of encouragement - a huge thanks to them.

ABSTRACT

KEYWORDS: Statistical Timing Analysis; GP-GPU; CUDA

Statistical Static Timing Analysis has been an active area of reasearch in the past decade because of its importance towards accounting various variations that affect timing analysis. The traditional STA algorithm while having grown increasingly sophisticated when accounting for so many variations is inherently pessimistic and has other disadvantages such as inability to model within die variations. Simultaneously, the field of high performance computing has changed with the advent of General Purpose Graphics Procesing Units(GP-GPU) where the emphasis is more on SIMD(Single Instruction Multiple Data) and higher computational power over caching and extensive pipelining that have powered CPUs. The block-based SSTA algorithm is used in this work because its level based approach lends itself into parallel computation of arrival time distributions in each level. It should be noted that the algorithm neither fits the embarrasingly parallel nor is a completely serial algorithm that can be converted into a scan-reduce problem that has been solved for the GPU previously.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

STA	Static Timing Analysis
SSTA	Statistical Static Timing Analysis
CPU	Central Processing Unit
GPU	Graphical Processing Unit
AT	Arrival Time
RAT	Required Arrival Time
pthread	POSIX Threads
CUDA	Compute Unified Device Architecture
PDF	Probability Density Function
CDF	Cumulative Desity Function
SIMD	Single Instruction Multiple Data
RV	Random Variable
PCA	Prinicipal Component Analysis
BFS	Breadth First Search
ALAP	As Late As Possible Scheduling Algorithm
ASAP	As Soon As Possible Scheduling Algorithm
SM	Streaming Multiprocessors

NOTATION

ϕ	Gaussian pdf
Φ	Gaussian cdf
ρ	Correlation Ceoefficient
T_A	Tightness probability
a_X	Corner point based coefficients
ΔV	Voltage Variation
ΔL	Gate Length Variation
ΔW	Width Variation
ΔH	Threshold Variation
ΔM	Metal Variation
d_e	Elmore Delay
β_e	Second order moment of impulse response
$\hat{s_o}$	Nominal Output slew of the impulse response

CHAPTER 1

Introduction

1.1 Motivation

Timing Analysis over the past 2 decades has been mainly dominated by static-timing analysis(STA) - with STA playing a very important role in the optimization stage of digital design. The advantages of STA have been

- Linear Runtime with circuit size allowing fast computation of results even for designs with the order of 10^8 gates
- Conservative estimate in terms of the delay being computed and hence providing sufficient timing constraints during design
- STA algorithms have evolved over time to tackle issues such as false paths, multi-cycle paths and lastly
- Delay characterization for cell libraries with the help of foudries which are easily available.

Nevertheless, as the Silicon industry is approaching dimensions in the 10s of nm this decade - the impact in the circuit due to process variations or variations due to change in environmental conditions have driven designers to verify their designs at different extremeties referred as corner points. The STA analysis is now performed at the different corner points to ensure that the design does not fail at these points. The obvious disadvantage of such a scheme of verifying the design results in very pessimistic timing constraints leading to the chips being over-engineered. Other reason as to why STA does not really work well with variation is that while global variations can be taken into account by running STA for different corner points, within-die variations cannot be taken into account when performing the timing analysis. The variations both global as well as intra-die have motivated the introduction of Statistical Timing Analysis.

Simultaneously in the past decade, there has been tremendous emphasis on parallel computing on Graphical Processing Units(GPU) - mostly due to the lack of increase in computation power anymore in CPUs, as was the case in the previous decades due to constant scaling and increase in clock frequency. The new era of high performance computing is being constantly referenced to the tremendous compute power that the GPU offers in trading more die area for compute units instead of memory units such as huge caches that dominate the present state of the art CPUs.

1.2 Previous Work

In the traditional STA algorithm, the combinational circuit is abstracted as a timing graph with the pins of gates forming the vertices of the graph and the gate delays as well as the interconnect delays serving as the edges of the timing graph. The subsequent algorithms of SSTA retains the basic timing graph while specifying the problem structure and it will be used in the following subsections.

1.2.1 Numerical Integration and Monte Carlo Simulation for SSTA

The earliest work in accounting for variations of different parameters and generating a distribution in order to produce a statistical account of the timing were obviously that of numerical-integration based on known equations and distributions available from foundries and Monte-Carlo simulations where the PDFs of varying RVs are sampled and based on which the delay computation is now performed using the traditional STA algorithm. The advantage of both the algorithms are that they are agnostic to the type of distributions that are propagated while they are heavily expensive in terms of the computation involved. (Jess *et al.*, 2006) and (Jaffari and Anis, 2008) each suggest efficient methods for numerical integration and Monte Carlo simulation respectively.

1.2.2 Path Based Approach

Path Based approach was among the initial algorithms of SSTA where in the timing graph probabilistic analysis was used in order to find out the critical paths as found in (Gattiker *et al.*, 2001) and (Agarwal *et al.*, 2003). In Path based algorithm the delay is found for all paths starting from the input to the output and finally a statistical MAX operations is performed on all the delay values obtained. This obviously retains the accuracy in computation because the addition of random variables happens without any loss of information unlike the MAX operation - there have been methods by which the order in which the calculation is performed minimizes the error obtained due to the MAX operation. Nevertheless, enumerating the delay for all path is not efficient because the runtime taken in order to traverse through all possible paths doesn't scale linearly with the number of gates.

1.2.3 Block Based Approach

The Block Based approach resembles the traditional STA algorithm in terms of the traversal of the gates in a topological manner. The computation of the delay values are either of the two steps - a) Adding delay value to the arrival time variable to calculate the arrival time at the gate output and b) Statistical MAX operation of all such values. (Visweswariah *et al.*, 2006) and (Chang and Sapatnekar, 2005) have established a canonical way of representing the arrival time as well as using (Clark, 1961) formulation of calculating MAX of two normal random variables and approximating it as a Normal random variable. The method followed in this work is also along same lines for which the reasons would be explained in subsequent chapters.

1.2.4 Accelerating Monte Carlo SSTA using GPU

There has been previous work done on accelerating the SSTA where Monte Carlo simulations are performed In (Gulati and Khatri, 2009), where Texture Memory of the GPU to act as a LUT for the cell library, and run the STA multiple times for gates in each level. The advantage of such an adoption of the Monte Carlo on the GPU is that the SSTA is agnostic of the distribution assumed and has no errors and gives order of magnitude improvement in runtime when compared to the CPU implementation. The disadvantage of this method is that the issues that prevailed with traditional STA such as inability to model within-die variations still carry forward with this method.

1.3 Contribution of this work

The SSTA algorithm has evolved from the initial stages a decade ago - and has grown to tackle different issues such as identification of False Paths as well as multi-cycle paths introduced by level triggered latches. While additional complexities have been added, the core idea of block-based SSTA of establishing a canonical form still continues. This work focusses on the ability to split the work of computation of arrival times as a statistical quantity into parallel computations. A `pthread` implementation of the SSTA algorithm was first realized to convince that a speed up is possible and motivated the idea of using a better equipped parallel infrastructure in running the algorithm. The CUDA architecture has its own set of advantage as well as disadvantage - and the thesis would give a brief introduction about the architecture to give a context. It should also be noted here that the CUDA architecture rewards more computation intensive algorithm where the need for load and store memory accesses are limited. The drawback is being acknowledged here, but the basic framework provided here would extend well when the same algorithm is extended to account for non-linearities in the dependence and non-gaussian random variables.

1.4 Structure of the thesis

The thesis is organized as follows:

- **Chapter 2: An overview of CUDA architecture**
  This chapter gives an overview of the CUDA architecture. The following topics are given a brief outline to give a context and understanding of the work done here and the results obtained - comparison b/w CPU and GPU, the programming

model, the memory hierarchy.

- **Chapter 3: Statistical Timing Analysis - Formulation**
  The chapter deals with the various variations that we encounter in SSTA, and the variables chosen for this work. The problem formulation of block based SSTA and some basic results for error obtained due to the assumptions made during formulation. Further the extensions that can be added to the above formulations without considerable difficulty in order to tackle different problems are also explained.

- **Chapter 4: GPU accelerated SSTA**
  The results obtained on implementing this algorithm using `pthread` and the speed up so obtained are presented here as a motivation towards parallel implementation of the SSTA algorithm. The implementation of the problem formulated in previous chapter on the GPU is then dealt in this chapter - in particular the changes that were implemented so that it can be implemented on GPU. The results obtained are also explained in terms of the architecture discussed earlier.

- **Chapter 5: Conclusion and Scope for Future Work**
  The work done is summarized and the improvements that can be built upon the existing engine are suggested.

CHAPTER 2

An Overview of CUDA architecture

2.1 Why GPUs?

Performance comparison with CPUs

CPUs have been developed over the past decades to increase their performance and they have been able to do that with different techniques such as

- Heavy **Pipelining** of the units - prone to different types of hazards while trying to mask the latency of the instructions.
- **Superscalar architecture** - where more functional units are used and data gets processed while always keeping in mind to reduce idle functional units.
- **Out-of-order execution**, **Branch Prediction**, **Eager Execution**

The compiler as well as the architecture try to leverage "Instruction Level Parallelism" in order to mask the latencies due to different operations and try to achieve more than 1 instruction being executed per cycle. All these changes with architecture along with increasing clock-speeds have been able to make sure that the hardware was delivering the promised speed up with each node. But, the way the CPUs are designed to handle the latency in accessing data structures available in the main memory - which might be in the order of 100s of clock cycles - is by using enormous caches and control

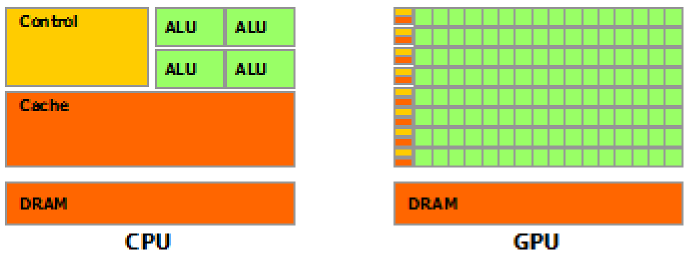


Figure 2.1: More die area allocated to compute units instead of cache (CUDA, 2012)

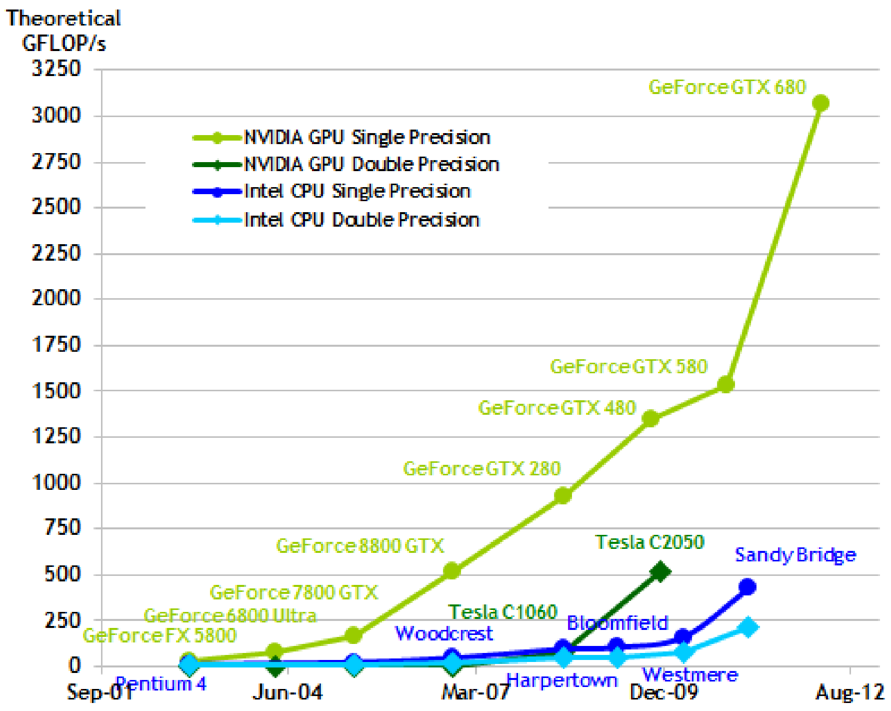


Figure 2.2: Comparison of computational power of GPU vs CPU (CUDA, 2012)

hardware which take the majority of the die space in comparison to the computational units ALU.

In comparison to the CPUs the GPUs were designed to maximize arithmetic performance. In Figure 2.1 and 2.2 it can be seen the difference in the power of GPUs in terms of computational power that it offers while in an equal way having huge memory bandwidth to actually feed the computations. The GPUs in order to reduce on complex control hardware employ two features - **SIMD Execution** and **Latency Hiding**.

2.2 Graphics Processing Units - Features

2.2.1 SIMD Execution

The GPUs are made of “Streaming Multiprocessors (SM)” which are independent execution units - with each SM having upto 48 scalar processors each capable of doing fused multiply-and-add per cycle. A **kernel** launches a grid of **thread blocks** which can cooperate between each other. The threads within a block are divided into **32 thread warp** which run simultaneously on the scalar processors.

2.2.2 Occupancy

The Fermi architecture(2.x) supports upto 1536 active threads (or 48 active warps) per SM. The ratio of the number of threads running on the SM to the maximum number of threads possible gives us the occupancy of the SM.

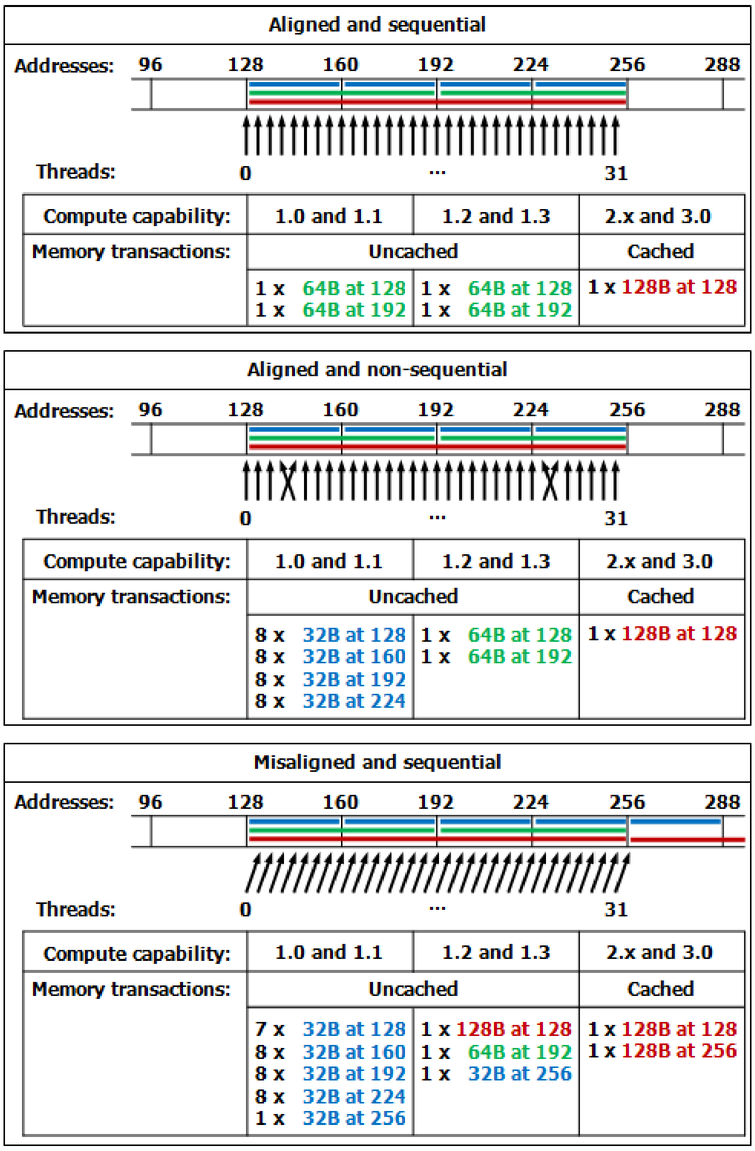
2.2.3 Latency Hiding

In CPUs, there is little benefit in running more threads than the number of cores available. Because, if there is a stall in the current thread then for the CPU to change its current working state to another thread must require that a thread scheduler select a new thread to wake, remove the contents of the old execution from the registers and load the state of the new thread into the registers. Hence, coarse-grained parallelism is used for driving multi-core.

Coarse-grained parallelism is also used in GPUs - where the newly free SM is allocated to the next block for computation. Fine-grained parallelism is the additional feature that GPUs provide in comparison to the CPU. The SM scheduler is capable of switching warps quickly when a hazard is encountered in the current warp - thus ensuring that there are no stall cycles and enables maximum throughput.

This feature of Latency Hiding makes sure that the GPUs deliver as much instruction throughput which can go upto 32 instructions per cycle when in comparison to modern CPUs which have around six execution pipelines which can deliver maximum throughput only at 6 instructions per cycle.

Synchronization between blocks is too costly and mostly not attempted. But, synchronization within a block can be done using `__syncthreads()`. While Intra-warp is automatically synchronized, inter-warp synchronization implies whenever a barrier is reached within a warp - that is removed by the scheduler and marked as inactive until all other threads reach the barrier. As the number of active threads decrease, the latency hiding ability of the SM decreases simply due to lack of active warps. As an obvious conclusion from the above discussion, it must be noted that in order to make maximum use of the latency hiding the kernel must be written so that there arithmetic operations


Aligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31
Aligned and non-sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31
Misaligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31

Aligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Misaligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Figure 2.3: Examples of Global Memory Accesses in different architectures (CUDA, 2012)

and memory operations are interleaved so that hazards are minimized and also there are sufficient number of active warps alive for each SM to ensure efficient latency hiding.

2.2.4 Memory Considerations

Threads in an active warp issue their global load and store operations simultaneously - and can wait while the scheduler brings another warp to do its computation. But when such synchronous calls are made to the memory controller - each transaction addressing a 128 byte wide global memory line, it should be made sure that the requests are *coalesced* as in Figure 2.3 so that the entire load operation can be serviced by a single transaction whereas in the worst case there can be 32 different serial accesses by the controller to the global memory.

2.2.5 Drawbacks

While the GPU offers the advantages of extremely high computational power as well as memory bandwidth, it comes at the exchange of some oft-used traditional programming routines. The drawbacks are as follows:

- No dynamic memory allocation
- Very less shared memory
- Bank conflicts for poor use of shared memory
- Branch divergence penalties
- Serialization of memory transactions for non memory-coalesced access - breaks the idea of wrapping data within structures
- Costly Synchronization across Blocks.

2.3 CUDA Programming Model

This section would give a brief outline about the programming abstraction presented to the programmer in order to use the GPU as represented in Figure 2.4

2.3.1 Kernels

CUDA C extends C in order to define functions called *kernels*. Kernel is defined using `__global__` declaration and the number of blocks used in the grid and the number of threads per block is then specified within «<`...`»>. The parameters that are passed on to specify the dimension of the kernel can be integers or can take 2-D or 3-D dimension values. The block and thread of each particular execution can be identified by

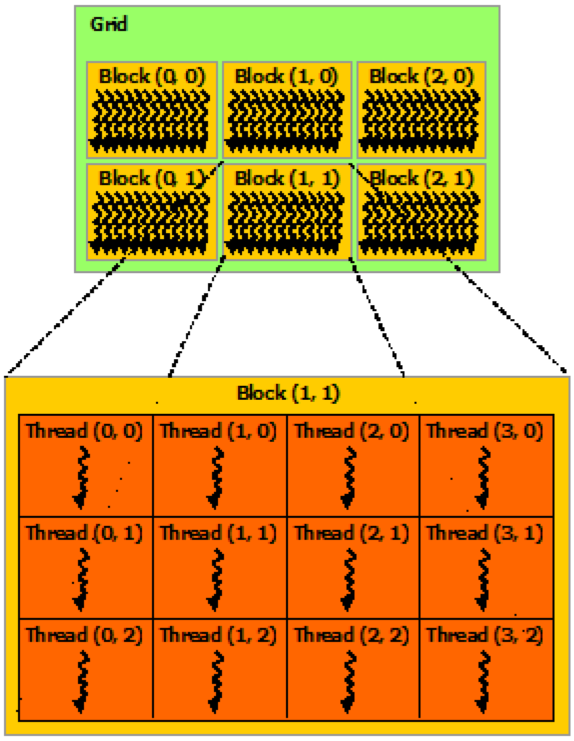


Figure 2.4: CUDA Programming Model (CUDA, 2012)

`blockIdx` and `threadIdx` variables. The maximum number of threads that can be initialized in a block is 1024 threads.

Thread blocks are same as the "Blocks" described in the previous section and have no simple way of synchronizing across them. Though, synchronization is hard across blocks - this models helps in easily scaling the code for more number of processors as shown in Figure 2.5.

2.3.2 Memory Hierarchy

CUDA threads may access data from multiple memory spaces. Each thread has a private local memory comprising of registers. Thread blocks have the ability to use the shared memory space available to them through L1 caches. All threads have access to the same global memory. While the registers are accessible with one clock cycle delay, the shared memory is usually around 3 clock delays away whereas the global memory is typically takes around 400 cycles to do operations. Additionally there are read-only memories provided by the constant memory and texture memory spaces. The shared memory is available only as long as the block is under execution whereas the constant, texture

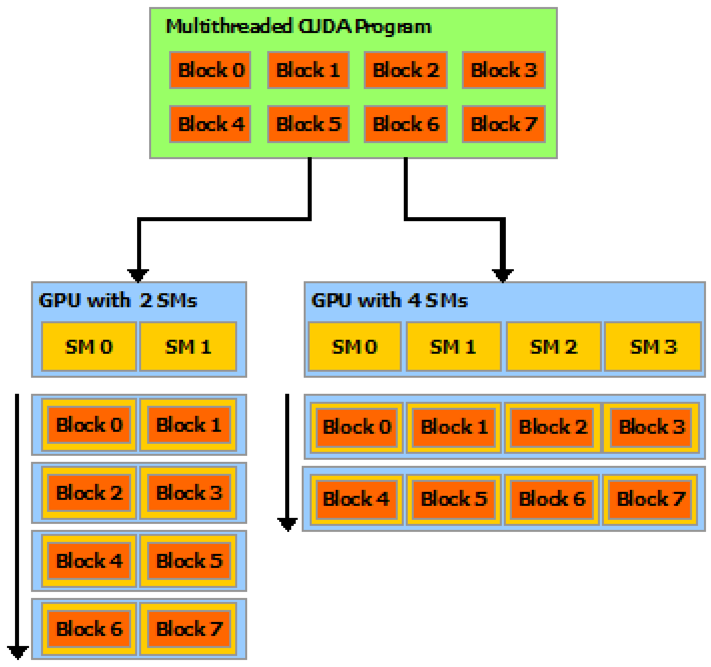


Figure 2.5: Automatic Scalability of Blocks which is architecture agnostic (CUDA, 2012)

and global memories are available throughout the lifetime of the kernel. The different memory spaces available to different programming constructs are shown in Figure 2.6.

The texture memory space has not been used much in this work and hence the details of the texture memory and its API is omitted.

Note: Optimizing CUDA code can lead to different paths which might or might not lead to better results if just the CUDA Programming Model is taken into account with just the limited knowledge about the latencies that different memory spaces offer. **The CUDA C Programming Guide** has a section on **Performance Guidelines** - subsection Multiprocessor Level which combines all the ideas from SIMD, Latency Hiding to the bounds in memory created by the limited shared memory and register usage and serves as the best possible guide when it comes to optimization instead of searching online for CUDA optimization techniques.

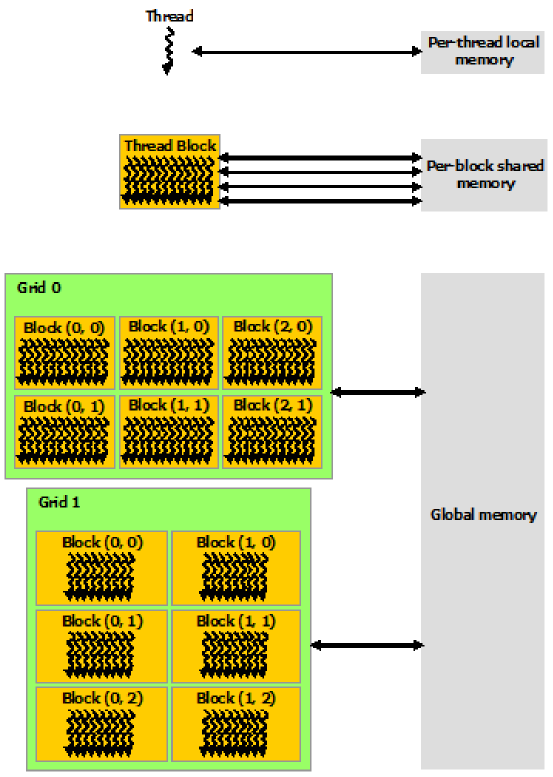


Figure 2.6: Memory Hierarchy correspondence to the programming model (CUDA, 2012)

CHAPTER 3

Statistical Static Timing Analysis

This chapter is divided into 4 sections. The first section would give details about the variations present that need to be accounted for in SSTA in more detail than that was covered in the Introduction chapter. The second section would explain the challenges that face SSTA and how they have been dealt with. The third section would explore the topic of Block-Based SSTA and venture into the problem statement. The final section would talk about the exact equations used which map the physical quantities in silicon and the timing quantities such as arrival time and slew and establish the relations used in this work. The work by (Blaauw *et al.*, 2008) gives a good survey of the work done so far in SSTA.

3.1 Variations in SSTA

In the Introduction chapter there was some explanation about STA being used for different corner points which was leading to highly pessimistic results. The variations that were mentioned there mainly stem out of the following causes.

- **Manufacturing Variations** - uncertainty that arises in parameters due to manufacturing processes becoming more complex with the latest nodes. These variations occur from die to die as well as within die.
- **Operating Variations** - the variability that comes with a range of operating conditions that the circuit is expected to work during its lifetime such as temperature, V_{dd}, and wearing out of the circuit.
- **Analysis Errors** - inaccuracy in device modelling which might affect the parameters used in timing-analysis algorithm.

Since variations in environmental conditions as well as modelling errors are given worst-case treatment because of the huge range in which it is expected to work, SSTA mainly encompasses of dealing with process-variations. The variations due to other components have been treated over and above the framework developed for the process variation case.

Process Variations

The manufacturing process has become complex with shrinking size of the transistors. There have been variations due to CMP(chemical mechanical polishing) used to planarize metal lines and insulating oxides, optical proximity effects due to usage of light whose wavelength exceeds the dimensions being realized and lens imperfections. The variations in physical parameters cause the device parameters to change which in turn affect the electrical parameters and hence the delay. A point in case is the width of the interconnects which on increasing leads to lesser resistance but higher capacitance leading to a negative correlation between them. The worst case model of the product RC would definitely lead to a highly pessimistic value given the way the electrical parameters change. The number of variations from all possible variations that arise are too high - and an analysis including all of them would be too complex. Hence, analyses have taken physical variations as basic RVs.

The variations can be classified as either systematic variations or random variations. Systematic variations are well documented and understood variations and arise mainly due to optical proximity effect. The effect of such variations can be modelled by studying the layout. Since the layout is not available at initial stages of design even these variations are given a statistical treatment. For the random variations only the statistical quantities are known during design stages. Even with the variations there might be variations from one die to another due to differences caused during alignment of the mask, as well as within-die variations caused because of the way dies are exposed in smaller blocks called reticles - leading to variations across different reticles within a same die.

3.2 Challenges in SSTA

The choice of bringing in RVs to model variations in order to predict arrival times bring in their own set of challenges. First given the context of different variations possible and how they are inter-dependent, the impact of correlation on delay calculation is discussed here.

3.2.1 Impact of correlation on Delay

This study of correlation impact is important here because it largely determines the sources of error in the analysis. In this analysis it would be seen how correlation affects the results of the two fundamental operations that are used here.

Addition of RVs

Assume there are n RVs, which are identical in their distribution. $(x_1, x_2, x_3, \ldots, x_n) \sim \mathcal{N}(\mu, \sigma^2)$. Let $s_n = x_1 + x_2 + \cdots + x_n$.

If the n RVs are independent RVs then the σ/μ ratio would be as follows

$$\left(\frac{\sigma}{\mu}\right)_{s_n} = \frac{1}{\sqrt{n}}\left(\frac{\sigma}{\mu}\right)_x$$

But, if the RVs are such that there is a correlation ρ between two random variables then

$$\left(\frac{\sigma}{\mu}\right)_{s_n} = \sqrt{\frac{1+\rho(n-1)}{n}}\left(\frac{\sigma}{\mu}\right)_x$$

In the above equation when the random variables are completely correlated then the σ/μ ratio remains the same as the original ratio. So from the above two equations it can be inferred that the variance(spread) of the sum of independent random variables is lesser than that of the completely correlated case.

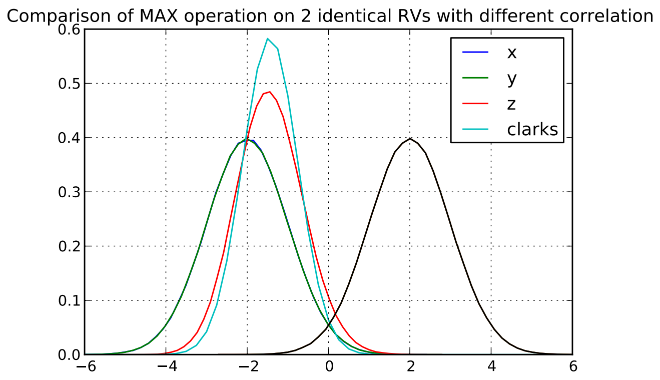


Figure 3.1: Maximum operation of RVs - nonlinearity and dependence on correlation

Statistical Maximum of RVs

The max operation is used in order to find the arrival time value at the output of any gate and is the other important fundamental operation that is performed. In the Figure 3.1 independent normal random variables and completely correlated random variables are considered. In the case of completely correlated random variables it can be seen that the random variable $s = max(x1; x2)$ is same as $x1$. While in the case of the independent random variable the pdf of the curve is shifted towards the right leading to higher estimates.

Hence from the above two paragraphs it can be seen that independent RVs while giving lesser spread in the case of addition give a higher-estimate of values with respect to the Max operation, while the completely correlated case gives higher spread while adding RVs but gives the same distribution as the RVs themselves in the Max operation in the identical distribution case.

Hence, in a given circuit if the random variables are assumed to be correlated then it over-estimates the variation while adding path delays whereas in case of the independent assumption the same happens with the Max operation.

3.2.2 Topological Correlation in SSTA

From the previous section, it must be clear that the SSTA analysis must choose RVs such that the correlation between them does not affect the estimates made. Hence, all timing quantities are made up of a known set of RVs whose distribution is known and the timing quantity/delay dependence on these variables is also known. Apart from these variables, there is an extra random variable which accounts for variations not taken care by these variables and is independent of them. Such a way of modelling the analysis reduces the problem to known variables and completely independent random variables - both can be given good analytical treatment, thus avoiding problems where there are RVs with partial correlation

The topological correlation problem stems out in the case of re-convergent fan-out. When two paths that diverged out from the same fan-out again come together as inputs then the independent random variables at the inputs are not independent anymore which leads to over estimation in the case of the Max operation.

3.2.3 Spatial Correlation

There has been a discussion about within-die variation in the section of process variations. Such within-die variations often lead to partial correlation with variables across the die. Within-die variations can also bring correlations to not only edges that are there within a gate but also edges across two gates that are located closely. With spatial correlation, both the operations - addition as well as Max do not operate easily each leading to different pessimistic results depending upon the extent of correlation.

3.2.4 Nonlinear Dependencies and Non-Gaussian distributions

In the analysis, the assumption made is that the variations affect timing quantities in a linear manner, and the variability is only as Gaussian random variables. Example of non-normal variable is gate length(Figure 3.2 which appears due to variability in manufacturing. Even if the physical parameters vary in a normal manner, the dependence on electrical parameters doesn't necessarily need to be linear which result again in using corner analysis.

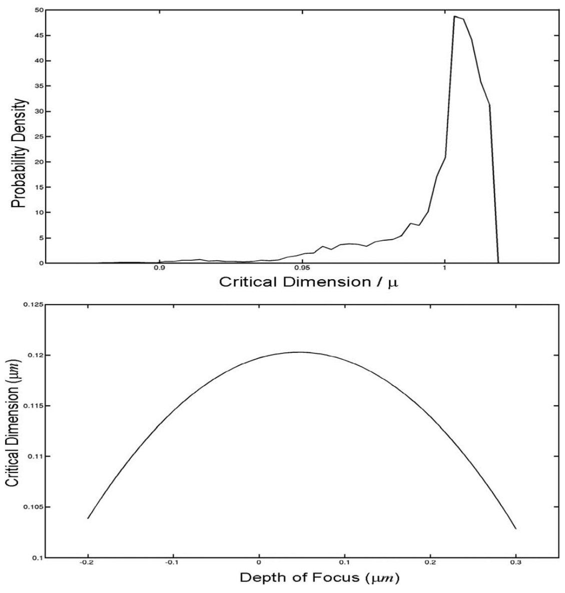


Figure 3.2: Non Gaussian distribution of gate length due to optical effects (Blaauw *et al.*, 2008)

3.2.5 Skewness of Max Operation

The Max operation plays a very essential role in the SSTA computation - but inherently the operation is a non-linear operation. Moreover, maximum of normal quantities results in positive skew of the resulting distribution which is a non-normal quantity. An approximation here is made, assuming that the output of the Max operation is also gives a Gaussian RV. The error due to the Max operation approximation is maximum when the random variables have similar means and different variances, and also when the variables are scaled versions of each other.

3.3 Block-Based SSTA

Previous work on other approaches to SSTA were covered in the Introduction chapter. Since the work done in the thesis can be extended to different analyses that are block-based, a short background on the earlier block-based techniques are presented. The method used in this work is described in the section after that.

3.3.1 Distribution Propagation Approach

In earlier approaches towards block-based SSTA, the idea was to propagate the entire distribution of the delay. The basic assumption over here is that the distributions are independent in which case both the Add as well as the Max operation computation is simple. But, in case of topological correlation due to re-convergence there will be errors - alternatively the correlations between different delay variables which have some dependence needs to be propagated along with the delay values - where the sum and max operation cease to remain trivial.

Other approach to block-based SSTA is to discretize the distribution and propagate the PMF(Probability Mass Function) as in (**?**). In case of the PMF, there is a slight difference in the way calculations need to be done. Assuming independence, in case of addition of 2 RVs the resulting PMF calculation is a convolution - while for the Max operation the calculation involves the sum of product of the PMF of each with the CMF of the rest. This method also has the issue of topological correlation - which is solved by defining cones of gates falling within the re-convergence spot and finding the PMF at the output using Bayes Theorem. The runtime complexity of the same is worst case exponential. As an extension of the above work, (Agarwal *et al.*, 2003) showed that ignoring topological correlation results in a pessimistic upper bound which can still be propagated while having a linear runtime.

3.3.2 Dependence Propagation Approach

In distribution propagation approach the whole information is contained within the distribution - this method had been used in earlier works and subsequently this line of work had been useful in tackling the issue of Topological Correlation. But, it isn't possible to handle within-die variations effectively in the distribution approach. It follows that the basic device parameters should be modelled as RVs. The spatial variation across the die has been handled in 2 ways

- **Correlation based variable order reduction** - divide the die into small grids and establish the correlation between the RVs of any 2 grids and reduce the RVs

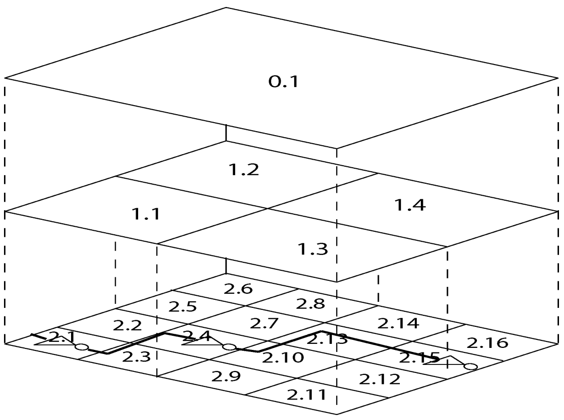


Figure 3.3: Spatial correlation using Quad-tree structure (Blaauw *et al.*, 2008)

into lesser number of independent unit variance normal RVs using methods such as PCA.

- **Quad-Tree model** - in Figure 3.3 reduce the grid recursively into quads and assign a RV for each of the quad - thus giving effectively accounting for global variations by variables in the top levels as well as local variations given by variables in the lowest level of the quad tree.

Once the correlation within die is taken care of, now it should be all be expressed in a way which can be useful to do the SSTA operations. The timing quantities are expressed in a canonical form which is as follows

$$a = a_0 + \sum_{1}^{n} a_i \Delta X_i + a_{n+1} \Delta R_a \tag{3.1}$$

where the RVs $\Delta X_i \sim \mathcal{N}(0,1) \forall i$

Now it is to be seen as to how the variables are handled with the basic operations of Addition and Max operations. When delays get added as $C = A + B$ the variables of C expressed in canonical form is quite straight-forward.

$$\mu_c = \mu_a + \mu_b \tag{3.2}$$

$$c_i = a_i + b_i \qquad \forall i = 1, 2, \ldots, n \tag{3.3}$$

$$c_{n+1} = \sqrt{a_{n+1}^2 + b_{n+1}^2} \tag{3.4}$$

It is to be noted here that the random coefficients are treated as independent RVs.

The next operation is the Max operation which is a nonlinear operation and the output of the operation cannot be expressed in the canonical form. The result of the Max operation is hence approximated to a Gaussian distribution. The following is the procedure in (Visweswariah *et al.*, 2006) which has been adopted in this work.

Consider 2 RVs A and B where A and B are represented in their canonical form. $C = max(A, B)$. The procedure to compute the coefficients of C to express in its canonical form is as follows.

1. Compute the variances of A and B and the covariance.

$$\sigma_a^2 = \sum_{i=1}^{n+1} a_i^2, \quad \sigma_b^2 = \sum_{i=1}^{n+1} b_i^2, \quad r = \sum_{i=1}^{n} a_i b_i \tag{3.5}$$

2. Compute the tightness probability T_A defined as the $Pr(A > B)$.

$$T_A = \Phi\left(\frac{a_0 - b_0}{\theta}\right) \tag{3.6}$$
$$\theta = \sqrt{\sigma_a^2 + \sigma_b^2 - 2r}$$

   where Φ is the CDF(cumulative density function) of the Gaussian distribution.

3. Compute mean and variance of $C = max(A, B)$ given by (Clark, 1961)

$$c_0 = a_0 T_A + b_0(1 - T_A) + \theta\phi\left(\frac{a_0 - b_0}{\theta}\right) \tag{3.7}$$
$$\sigma_c^2 = (a_0^2 + \sigma_a^2)T_A + (b_0^2 + \sigma_b^2)(1 - T_A) + (a_0 + b_0)\theta\phi\left(\frac{a_0 - b_0}{\theta}\right) - c_0^2 \tag{3.8}$$

4. Compute sensitivity coefficient for C to be expressed in the canonical form.

$$c_i = a_i T_A + b_i(1 - T_A) \quad \forall i = 1, 2, \ldots, n \tag{3.9}$$

5. The calculation of c_0 and σ_c^2 are exact as well as the above equation. The error in the above calculation stems from the assumption that the result from these Max

operation is Gaussian, while it is not because of the non-linearity of the operation. So, in order to match the variance of C to σ_c^2 the residue of variance from σ_c^2 from the variance offered by the remaining variables i.e. $\sum_{i=0}^{n} c_i^2$ is assigned to c_{n+1}.

$$c_{n+1} = \sqrt{\sigma_c^2 - \sum_{i=1}^{n} c_i^2} \tag{3.10}$$

It has been shown in (Sinha *et al.*, 2005) that the argument to the sqrt operator is always positive.

The above subsection reduced the problem into that of a computation one where the canonical form plays an important role. Subsequent works which have included non-normal behaviour as well as non-linear behaviour into the canonical form - where the sum operation remains the same while the Max operation still keeps the idea of finding Tightness probability and moment matching with variations on methods to do the computation of the tightness vary depending on the case.

3.4 Specifics of the Problem Statement in this work

The problem taken in this work has taken the specifics given by the (Sinha *et al.*, 2013). The variations chosen in the problem statement were the following

- environmental : **voltage (V), temperature (T)**
- process : **channel length(L), device width(W), voltage threshold(H), metal(M)**
- random variation (R)

And hence the canonical form would be expressed as

$$A = a_0 + a_V\Delta V + a_T\Delta T + a_L\Delta L + a_W\Delta W + a_H\Delta H + a_M\Delta M + a_R\Delta R$$

The coefficients are found by corner analysis as

$$a_X = \frac{A_{|\Delta X=+3\sigma} - A_{|\Delta X=-3\sigma}}{6\sigma}$$

In the case for metal variations alone for which ΔM doesn't take negative values the coefficient calculation is carried out as

$$a_M = \frac{A_{|\Delta M=+3\sigma} - A_{|\Delta M=0}}{3\sigma}$$

Interconnect Model

The RC-Tree is given for the interconnects between cell outputs and the connecting inputs. The delay values through the RC Tree is given by the Elmore Delay formula. The delay due to RC-Tree doesn't vary with only the metal parameter. Hence, the variation due to the metal is computed as given by the corner analysis equation above and using the Elmore delay. The Elmore delay is computed as

$$d_e = \sum_k R_{ke} C_k \tag{3.11}$$

In order to calculate the impact of how metal variations affect the slew propagation through the interconnects, the calculation involves finding the second order moment β of the impulse response at the port of the interconnect.

$$\beta_e = \sum_k R_{ke} C_k d_k \tag{3.12}$$

using which the *nominal output slew of the impulse response* is calculated using β which is used to calculate the output slew through the interconnect, note the non-linear relationship between the input slew and the output slew.

$$\hat{s}_o = \sqrt{2\beta_o - d_o^2} \tag{3.13}$$

$$s_o = \sqrt{s_i^2 + \hat{s}_o^2} \tag{3.14}$$

The metal dependence on slew for $\hat{s}_o$ is calculated using the above equation and the corner analysis and with the variable expressed as $\hat{S}_o = \hat{s}_o + \alpha_m^{\hat{s}_o} \Delta M$. This expression is used for calculating $\hat{s}_o$ at any given metal point. The corner point treatment is extended for finding coefficients across interconnects (refer (Sinha *et al.*, 2013) equation (22) for details).

Combinational Cells

The delay and the slew at the output of a combinational cell from the input is calculated in the following way

$$D = a(1 + k_{d,v}\Delta V + k_{d,t}\Delta T + k_{d,l}\Delta L + k_{d,w}\Delta W + k_{d,h}\Delta W + k_{d,r}\Delta R) + bC_L + cS_i \tag{3.15}$$

$$S_o = x(1 + k_{s,v}\Delta V + k_{s,t}\Delta T + k_{s,l}\Delta L + k_{s,w}\Delta W + k_{s,h}\Delta W + k_{s,r}\Delta R) + yC_L + zS_i \tag{3.16}$$

In the above equation it should be noted that the canonical form of S_i brings more variation than just the product $a * k_d, x$ and should be $a * k_d, x + c * s_x$ with the random part being the root of sum of squares $\sqrt{(a * k_{d,r})^2 + (c * s_r)^2}$. The capacitive load is the capacitance of the entire interconnect as well as the input capacitance at the inputs of the taps that the interconnect drives. Care needs to be taken to calculate the metal dependence of C_L to include just the interconnect capacitance and not the input capacitances.

CHAPTER 4

GPU Accelerated SSTA

In this chapter the first section would be a presentation of the work done for the TAU Contest - a parallel implementation of the above algorithm and the results for the benchmarks provided by the TAU Contest committee. The parallel implementation is compared against the serial implementation. In the next section the work done and the results with the SSTA algorithm in GPUs are presented

4.1 Circuit Traversal

Levelization

The circuit traversal and parsing is done in a serial manner using a Breadth-First-Search(BFS) approach. The steps are as follows

1. Add all the primary inputs to a virtual source instance and add it to the BFS data structure which is a queue.
2. Pop the first instance from the queue and for each output in the instance find all the ports connected via the interconnect.
3. Increment the number of visited input counter for all connected instances - if the number is equal to the number of inputs as given in the cell library, then add this instance to the queue
4. Goto 2) if queue is not empty otherwise exit.

The above is a scheduling algorithm and the scheduling can be done in two ways - As-soon-as-possible(ASAP) and As-Late-as-possible(ALAP). The above steps proceed using the ASAP procedure. The timing graph is being treated as Directed Acyclic

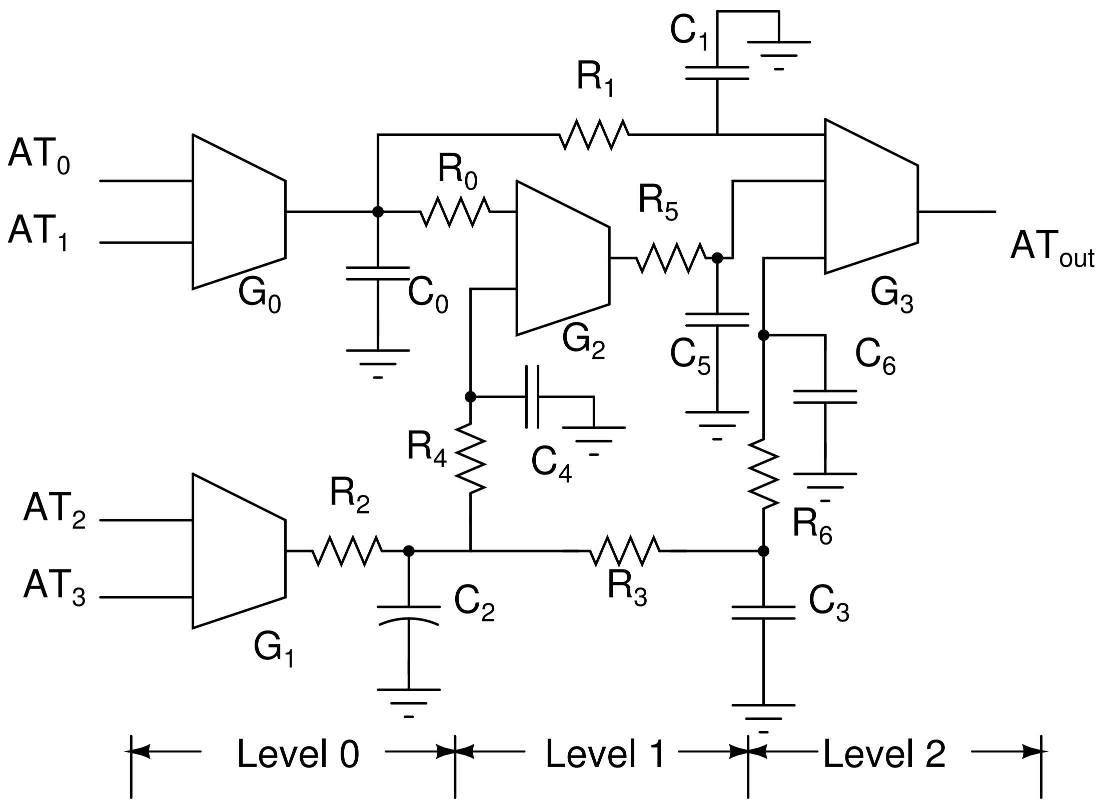


Figure 4.1: Sample circuit - Levelization

Graph(DAG) which will not be the case if the circuit contains Level-triggered latches. The scheduling gives a sense of level ordering based on the way the circuit is traversed. In ASAP scheduling all elements of a particular level consists of inputs driven by outputs from previous level but each instance should contain atleast one input being powered by the previous level output - which follows from the ASAP definition.

Also in the above enumeration, there has been no mention of clocked-circuits. The only clocked circuits considered for this work was flip-flops. In case of edge triggered flip flops the signal for being “computationally ready” is when the clock arrival time is computed. Hence for a clocked instance, the instance is pushed into the queue when the clock pin is reached and not when the number of visited inputs and actual inputs are equal as in the normal case.

Arrival Time computation

In the arrival time computation, the instances are assumed for computation in the order of levels they are in. This is straight forward given that once all outputs at level i are ready implies the calculations pending at level $i + 1$ can be started and hence giving the order of computation. This is similar to the traditional topological sorting.

For each instance I the arrival time computation does the following

Algorithm 1 Algorithm for calculating arrival times in each instance

```
for all output pins $op$ of instance $I$ do
  if $I$ contains a clock pin then
    $AT_{op} \leftarrow AT_{clk} + T_{CK \to Q}$
  else
    for all input pins $ip$ of instance $I$ do
      $AT_{op}^{ip} \leftarrow timingsense(AT_{ip}) + T_{ip \to op}$
      $AT_{op} \leftarrow MAX(AT_{op}, AT_{op}^{ip})$
    end for
  end if
  for all ports $p$ connected to $op$ through interconnect $w$ do
    $AT_p \leftarrow AT_{op} + T_{w:op \to p}$
  end for
end for
```

The above code has arrival time AT just as a symbol. Internally, the arrival time data structure consists of both time and slew at each node for combinations of rise/fall and early/late mode. The MAX operation would suitably change to MIN depending on the case. $timingsense()$ function takes in an AT structure and converts into corresponding timing sense equivalent depending on whether the input to output is defined as one among `UNATE, NEGATIVE-UNATE` and `NON-UNATE`.

Serial Code Results

The traversal algorithm is expected to work as $O(n)$ where n is the number of instances based on the BFS algorithm chosen here. In Table 4.1 and Figure 4.2 this can be seen establishing a linear relationship with the number of instances. The runtime as reported in the table were from running the code on **Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz** with cache of **8192 kB**.

Benchmark	Number of Gates	Number of Levels	Runtime(μs)
c17.net	8	3	73
s27.net	20	7	106
s344.net	131	14	445
c432.net	136	22	598
s349.net	143	14	477
s400.net	152	13	577
s386.net	158	14	554
c499.net	178	14	789
c1355.net	182	14	814
c880.net	223	22	908
c1908.net	224	20	1006
s526.net	235	11	743
s510.net	272	12	860
c2670.net	346	15	1580
s1196.net	586	20	1897
c3540.net	693	30	2884
s1494.net	785	17	2417
c5315.net	920	23	3798
c7552.net	1149	20	4568
c6288.net	1669	81	5600
wb_dma.net	2654	15	11241
systemcdes.net	2854	31	9703
tv80.net	4213	45	18051
systemcaes.net	4519	48	21071
mem_ctrl.net	7468	38	30816
ac97_ctrl.net	7865	17	27719
pci_bridge32.net	9212	37	38494
usb_funct.net	10627	36	43297
aes_core.net	21374	34	70904
des_perf.net	79028	25	296585
vga_lcd.net	88405	30	313359

Table 4.1: Runtime and Levelization results for the SSTA serial code

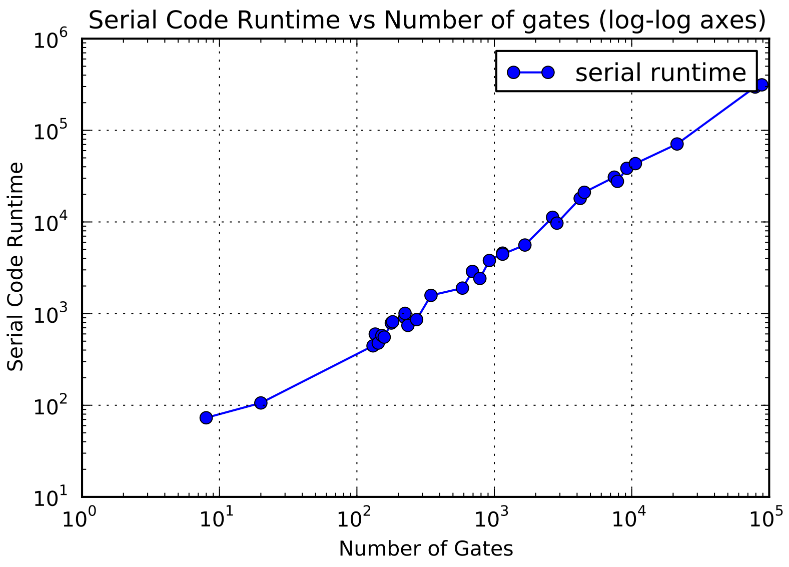


Figure 4.2: Linear runtime of traversal algorithm

4.2 `pthread` parallel implementation

In this section, the results obtained for the `pthread` implementation are discussed and compared along with the serial code results. The ASAP scheduling gives a level ordering which even though is scheduled one level after another - the order in which the instances are computed within one level really doesn't change the result. This can be used in order to speed up the runtime of the serial code - where the work load of computation within a level is divided between the threads.

Algorithm 2 pthread implementation of SSTA

```
for $i < numlevels$ do
  for $tid < maxthreads$ do
    for all instance $I \in work(tid, i)$ do
      computeinstance($I$)
    end for
  end for
  barrier synchronization for threads
end for
```

The $work(tid, i)$ is to ensure that the work in terms of computation is split across equally to all threads and in this work is a very simple way of splitting the work based on the number of instances at level i divided by $maxthreads$.

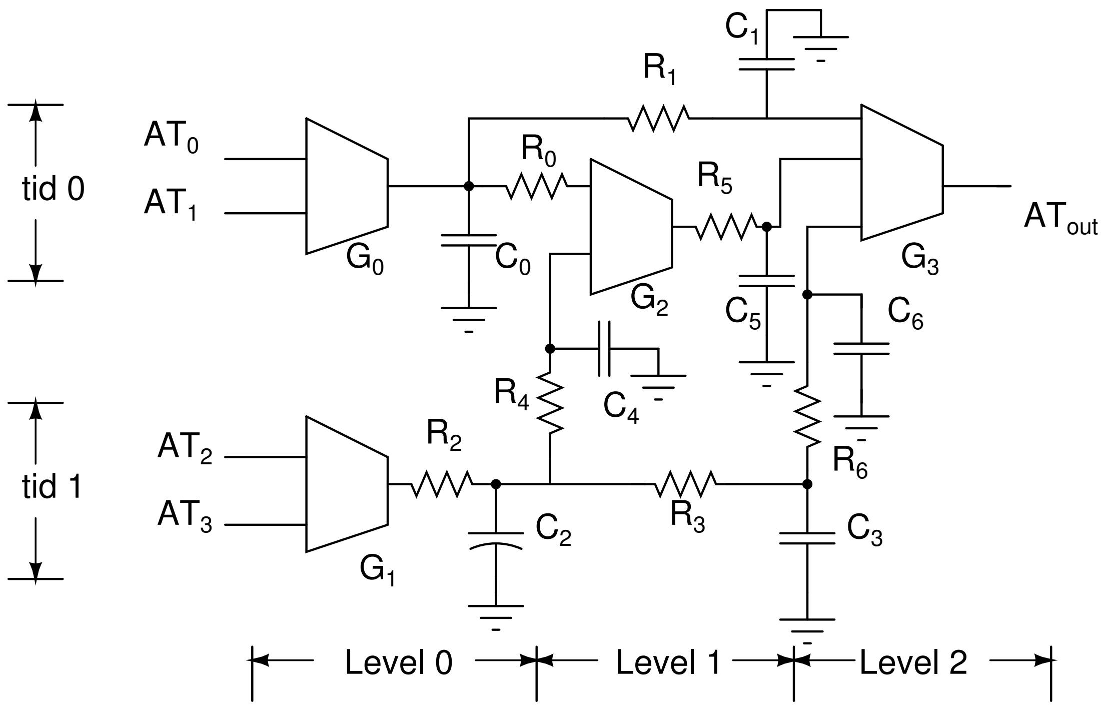


Figure 4.3: Parallel pthread implementation of SSTA

Results for pthread implementation

The algorithm was implemented on 8 Core Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz and shown in Figure 4.4 are the results for the bigger benchmarks. There was almost a 2x speed up with 2 threads but it doesn't scale linearly with the number of threads. There is around 4x speedup with 7 threads.

From the Table 4.2 it can be seen that the benchmarks with lesser number of levels seem to benefit more from the parallelization - especially when comparing "des_perf.net" and "vga_lcd.net" both of which are circuits who size is of the same magnitude but the speed up realized by the former is better than the latter because of lesser number of levels indicating more parallelism. This is again fairly straightforward because lesser levels with same number of instances imply more number of elements / level and hence more parallelism.

Other features

Apart from the arrival time computation and slew computation - the code written for the TAU Contest 2013 also had Reverse Arrival Time(RAT) and slack(difference between

Benchmark	no. instances	no. levels	2 threads	4 thread	7 thread
wb_dma	2654	15	1.35	2.88	3.45
systemcdes	2854	31	1.79	2.73	2.97
tv80	4213	45	1.79	2.42	3.39
systemcaes	4519	48	1.78	2.38	3.33
mem_ctrl	7468	38	1.82	2.81	3.95
ac97_ctrl	7865	17	1.53	2.51	2.38
pci_bridge32	9212	37	1.82	2.49	2.99
usb_funct	10627	36	1.88	2.90	4.25
aes_core	21374	34	1.90	2.69	3.75
des_perf	79028	25	1.97	3.03	4.64
vga_lcd	88405	30	1.93	2.74	4.43

Table 4.2: Speed up of 2-4-7 thread vs serial code comparison with number of instances and number of levels

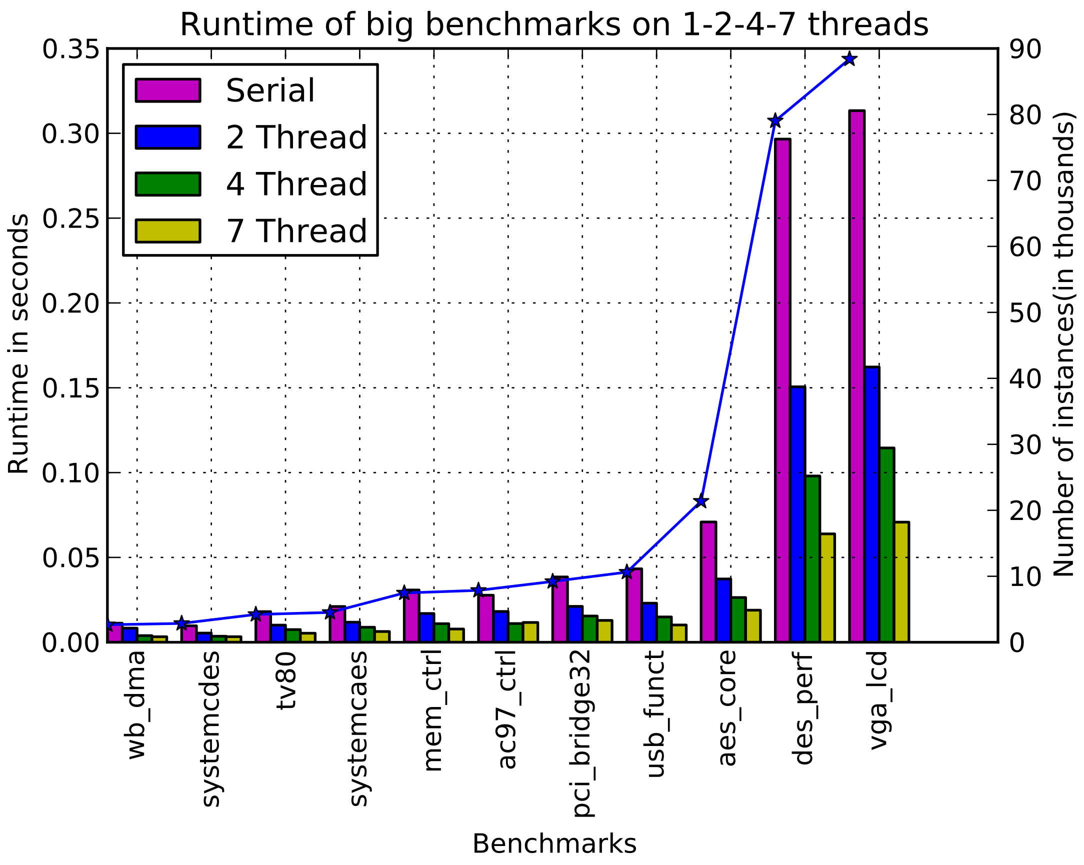


Figure 4.4: pthread runtime vs benchmarks and their size on the other axis

AT and RAT) computation. The computation followed to do the same is similar to one that has been explained in this chapter. The slack computation hasn't been included in the GPU version because of the fact that it can be extended easily hence keeping the current version simple.

CUDA Driver Version / Runtime Version	4.2 / 4.2
CUDA Capability Major/Minor version number	2.1 (Fermi Architecture)
Total amount of global memory	1024 MBytes
(8) Multiprocessors x (48) CUDA Cores/MP	384 CUDA Cores
GPU Clock rate	1660 MHz (1.66 GHz)
Memory Clock rate	2004 Mhz
Memory Bus Width	256-bit
L2 Cache Size	524288 bytes
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	32768
Warp size	32
Maximum number of threads per multiprocessor	1536
Maximum number of threads per block	1024

Table 4.3: Device Features for `NVidia GeForce GTX 560 Ti`

4.3 GPU Accelerated SSTA

In the SSTA implementation discussed in the previous section, it can be seen that the `pthread` implementation gives a 4x speed up when used with 7 threads. Nevertheless, the scope for improvement in runtime provides a motivation towards exploring the speedup possible with the GPU. The device used in obtaining results in this work was "`NVidia GeForce GTX 560 Ti`". The device constraints and features are as in Table 4.3

Structure Difference in Code

The way CUDA accepts an input from the host(CPU) is through an API call `cudaMemcpy` which copies an array of structures from the host to the device. This brings in an issue of not using pointers in the host code - because any such usage of pointer would be pointing to memory in the host which would be copied from the host to device. This constraint restricts the data structures to be filled with indices instead of pointers or change the pointers to the device pointers after doing a `cudaMalloc` and reassigning the values in the host before copying. The second way of doing this essentially means a duplicate set of the entire data must be made - because once reassigning of a pointer to the device equivalent is done, its connection is lost in the host which means the levelization algorithm will cease to work. To avoid duplication the first approach is chosen. Furthermore, the same applies to 2D pointers which were used in the earlier version and had made programming easier - any realization containing 2D pointers should now be broken into 2 arrays with one for the basic array and the second one containing the indices to the first array and its size, essentially book keeping a lot more details in order to stick to the linear array structure that is imposed on the programmer. Some examples are shown in Table 4.4. In the code comparison as shown, the structure definitions as seen in the C headings were defined by Jobin Jacob Kavalam.

4.3.1 Results - v1.0

The version 1.0 of parallel GPU code that was implemented carried forward the idea of the `pthread` implementation directly to a thread based equivalent in the GPU. A single block is launched with different number of threads and the results are reported. The kernel launch call for the GPU was as follows

```
dim3 DimGrid(1, 1, 1), DimBlock(NUM_THREADS, 1, 1);
forwardTimingAnalysis_GPU<<<DimGrid, DimBlock>>>(devicePCell, deviceIArray,
    deviceWArray, devicePArray, deviceTArray, deviceLArray, deviceLStart, numLevels);
```

while the kernel function itself is written in the following way

```
int tx = threadIdx.x;
for(int presentLevel = 0; presentLevel < levels; presentLevel++)
{
    int startid = lStart[presentLevel] + tx;
    int endid = lStart[presentLevel + 1];
    for(int instanceid = startid; instanceid < endid; instanceid+= NUM_THREADS)
        forwardTiming(lArray[instanceid], dpcell, diarray, dwarray, dparray, dtarray);
    __syncthreads();
}
```

In the above case it can be seen that this is a plain case of porting code and the resulting execution would not have any optimizations. The results can be seen as in Figure 4.5 and compared with the serial results obtained in previous sections in 4.5.

C	CUDA C

`struct instance`
`{`
`    int numIPins, numVisIn;`
`    int numOPins, numVisOut, numFanOut;`
 	
`    struct _instanceIPin **ipin;`
`    struct _instanceOPin **opin;`
`    struct _instanceIPin *cpin;`
 	
`    int celltype;`
`};`
	`struct instance`
	`{`
	`    int cellid;`
	`    int ipinid[MAX_IPINS];`
	`    int opinid[MAX_OPINS];`
	`    int clkpinid;`
	
	`    int numVisIn, numVisOut;`
	`};`

`struct wire`
`{`
`    int numTaps;`
 	
`    struct _wirePort *port;`
`    struct _wireTap **tap;`
 	
`    WIRE_TIMING_DATA iotiming[MAX_TAPS];`
`};`
	`struct wire`
	`{`
	`    unsigned int portid;`
	`    unsigned int tapzeroid;`
	`    unsigned int numtaps;`
	`    float risecap[2], fallcap[2];`
	`};`
	
	`struct tap`
	`{`
	`    unsigned int tapid;`
	`    float falldelay[2], risedelay[2];`
	`    float fallslew[2], riseslew[2];`
	`};`

`struct pin {`
 	
`    char nodename[65];`
`    int pinid;`
`    int isclock;`
`    NODE_TIMING_DATA tData;`
 	
`    struct instance *instance;`
 	
`    // Links`
`    struct pin *iop;`
`    struct tap *wt;`
 	
`};`
	`struct pin`
	`{`
	`    pintype type;`
	`    unsigned int instanceid, wireid;`
	`    unsigned int pinid;`
	`    pintiming pintime;`
	`};`

Table 4.4: Comparison of code between CPU and CUDA

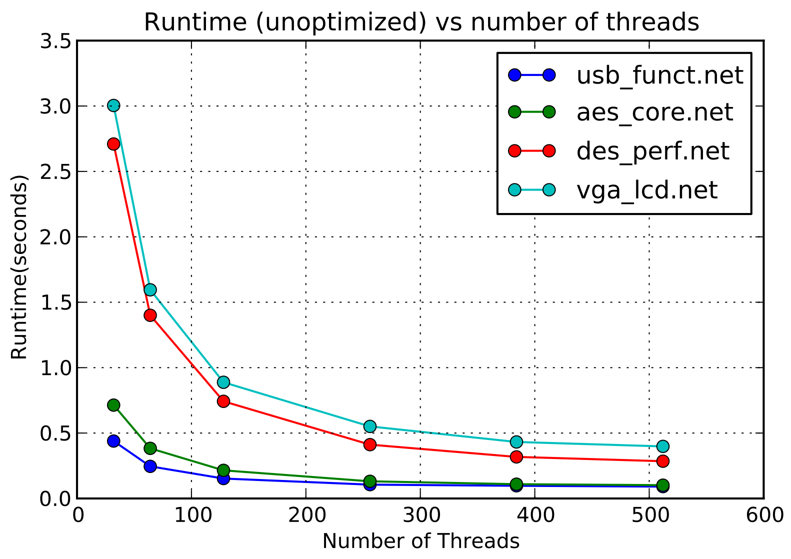


Figure 4.5: GPU runtime vs number of threads

Benchmarks	256 thread runtime(μs)	512 thread runtime(μs)	Serial Runtime(μs)
systemcdes.net	37717	37703	9703
wb_dma.net	35276	32937	11241
tv80.net	67995	66162	18051
systemcaes.net	75279	73738	21071
ac97_ctrl.net	73611	59512	27719
mem_ctrl.net	104127	91024	30816
pci_bridge32.net	114638	104056	38494
usb_funct.net	105665	90664	43297
aes_core.net	131124	101601	70904
des_perf.net	411406	284740	296585
vga_lcd.net	550720	398059	313359

Table 4.5: Comparison of v1.0 with 256-512 threads and serial runtime

It can be clearly seen that even the 512 thread runtime of this version is lesser when compared to the serial runtime of the code. Nevertheless the speed up that is seen as the number of threads are increased from 32 to 256 threads might seem to give an impression of more number of cores crunching data - while the actual speedup is due to **latency hiding** which has more flexibility in scheduling warps with increasing number of threads. Also when the kernel is compiled with `nvcc -ptxas-options=-v` the number of registers used in the kernel per thread is visible and in this case the number of registers is **63** which in turn bounds the number of threads to $32k/64 \approx 512$ threads theoretically limiting the occupancy to 33%. The features that are not there in this implementation are

- No memory coalescing present in the way data is accessed.
- Usage of the read-only memory blocks available to us.
- Very poor occupancy - at any point in time utmost only 512/1536 threads are active
- Branch divergence issues arising out of code if/else conditions
- Poor utilizations of SMs as only 1 Block is launched leaving other SMs idle

In this work a multiple combination of the above has been implemented with various success. The results for these are presented here.

4.3.2 GPU Accelerated SSTA v1.1 Enabling Memory Coalescing and Constant Memory for Read-only use

Memory coalescing is a very important aspect of implementing an algorithm in CUDA. In general the memory hierarchy is to be kept in mind and must be used so as to solve the problem in the best possible way. Constant memory is a specially cached read-only memory that is available to the entire global scope. Shared memory is likewise low latency memory but available only to a block of threads.

Memory Coalescing In order to achieve memory coalescing the different structure objects that are fetched during must be in consecutive memory locations so that when a global memory call is made from different threads the number of calls made can be minimal. In this algorithm if memory coalescing must be used, then the data structures must be in such a way that all the accesses made by different threads are from consecutive locations. This implies that the blocks should be ordered in the following way given that each thread works on its instance at a time.

- **Instances** - ordered in the topological order so that when each thread accesses its instance, they would fit the coalescing order while reading the value from memory

- **Input Pins** - ordered such that successive pins of each instance are adjacent to each other.
- **Output Pins & Wires & Taps** - all output pins of successive instances placed next to each other to enable write coalescing and wires of output pins next to each other to enable read coalescing.
- **Taps - Input Pins** - the taps of the interconnect from one output port can lead to input pins of different levels and hence the write due to this would **not** be memory coalesced.

Constant & Shared Memory The cell library gives a lot of information in terms of coefficients that map RVs/load capacitance and slew to the delay through the cell as well as the output slew. It also gives information about the names of the pins - and in case of clocked circuits the $T_{CLK \to Q}$ delays as well as the T_{setup} and T_{hold} are held in this structure. Though it is a data structure containing different parameters, they are all read-only during the computation of arrival time as well as the reverse. Hence, the choice of constant memory is obvious. But with the complete data structure holding all information the size blows to a $2kB$ and with 100 cells that is greater than the amount of constant memory available which is $64kB$. So in order to fit the constraints, the cell library data structure is broken down into just the coefficients pertaining to one input-output couple of a particular cell. This approach enables the usage of constant memory in storing the cell library information. Similarly, shared memory is used as a cache by storing all instances in a particular level being processed by the thread onto the shared memory.

Results

The results of using the above discussed optimizations for the big benchmarks are presented in Table 4.7. The maximum gain due to the rearrangement of memory coalescing is 6.58% and minimum is 2.89%. The performance benefit due to constant memory is actually negligible. The reasons for which the benefit gained is low are

Data Structure	Size (Bytes)
Instance	48
Pin	404
Wire	28
Tap	36

Table 4.6: Primary C Data Structures used and their memory sizes

- **Big Data Structures** - The sizes of data structures are shown in Table 4.6. The way memory coalescing works is that when threads in a warp(32) issue memory requests if all of them are in a chunk of 128 bytes, then in one request the entire memory request can be serviced while in the worst case it can go upto 32 separate requests. To achieve best performance the size of the data structure must be 4 bytes - `int` or a `float`.

- **Memory Requests and Caching** - The global memory accesses are cached at L2 cache and can be addressed using 32 byte requests in case of scattered access. But, this is again assuming that each thread in a warp requires only 4 bytes - when the size of each request is greater than 4 bytes, the memory requests are combined into 128 byte requests and then serviced accordingly. Eg. 8 bytes per thread is treated as 2 half warp calls of 128 bytes each and 16 bytes per thread is treated as four quarter warp calls. In the data structure that is used here, since the sizes are at minimum 32 bytes the servicing would be broken down into eight $1/8th$ warp issuing their requests.

- **Constant Memory** - Constant memory caches work in the form of banks - and if a half warp entirely requests a particular element in the constant memory, it is serviced as a half warp broadcast, otherwise the access is serialized. No such guarantee about half warps accessing the constant memory can be given here.

4.3.3 GPU Accelerated SSTA v2.0

In the above versions, the drawbacks and improvements were pointed out. Nevertheless, in version 1.1 there was not anything that had been done to improve the occupancy. The obvious point of reference with respect to occupancy is that the number of blocks

Benchmarks	256 thread w/o opt (s)	256 thread with opt (s)	% impr.	512 thread w/o opt (s)	512 thread with opt (s)	% impr.
ac97_ctrl.net	0.074	0.071	3.739	0.060	0.057	3.503
aes_core.net	0.131	0.127	3.525	0.102	0.099	2.734
des_perf.net	0.412	0.396	3.706	0.285	0.277	2.885
mem_ctrl.net	0.104	0.098	5.735	0.091	0.085	6.583
pci_bridge32.net	0.115	0.110	4.336	0.104	0.100	4.192
systemcaes.net	0.075	0.072	3.995	0.074	0.071	4.131
systemcdes.net	0.038	0.036	4.291	0.038	0.036	4.513
tv80.net	0.068	0.065	4.198	0.066	0.063	4.391
usb_funct.net	0.106	0.101	4.302	0.091	0.087	4.436
vga_lcd.net	0.551	0.529	3.963	0.398	0.384	3.707
wb_dma.net	0.035	0.034	4.542	0.033	0.031	4.731

Table 4.7: Comparison of runtime - the optimized and unoptimized

used in the grid is just 1 which implies the rest of the SMs are always on idle. This can be improved by splitting the work in each levels into blocks of calculation for each level instead of splitting the work into threads alone. Though this might seem obvious it should be kept in mind that `syncthreads()` doesn't provide a barrier synchronization across blocks and hence v1.0 was the first careful approach where such issues lead to the above usage. This way of splitting the work in each level across blocks yields results same as the one produced by the serial code.

In making the parallelization of levels across blocks instead of threads the kernel call is changed in the following way

```
for(i = 0; i< numLevels; i++)
{
    int startid = levelStart[i], endid = levelStart[i+1];
    dim3 DimGrid((endid- startid- 1)/num_threads + 1, 1, 1);
    dim3 DimBlock(num_threads, 1, 1);
    forwardTimingAnalysis_GPU<<<DimGrid, DimBlock>>>(deviceIArray,
                 deviceWArray, devicePArray, deviceTArray, startid, endid);
}
```

while the kernel function itself gets modified to just run the timing analysis gets modified into a simple call

```
int tx = threadIdx.x;
int bx = blockIdx.x;
int indx = sid + bx*NUM_THREADS + tx;

if(indx < eid)
    forwardTiming(&diarray[indx], dwarray, dparray, dtarray);

__syncthreads();
```

Result

The results for the above implementation so as to improve occupancy is presented in Table 4.8 and for the bigger benchmarks the results are compared with the all the implementations so far. The v2.0 shown here implements the changes included in v1.1.

The Table 4.8 gives the runtime in μs for all the benchmarks while running it for different number of threads. Some immediate observations are that with increasing number of threads the runtime increases not by much. This is due to the changed structure of the calls that are made to the kernel. With increasing number of threads the number of threads that remain idle increases when the levels are shallow which leads to lesser number of active warps and hence lesser latency hiding potential. The maximum activity happens when the number of threads is 32 with a lot of blocks scheduled minimizing both branch divergent threads as well as having maximum occupancy leading to better latency hiding.

Comparison In Figure 4.6 the runtime performances of the different implementations that have been discussed are represented for the 4 big benchmarks with the “aes_core.net” and “usb_funct.net” in excess of 10k gates while the other 2 benchmarks having around 80k gates. It can be seen that the v2.0 implementation is clearly better in comparison to v1.1

The comparison with the serial code as well it can be seen that while v1.1 was slower than the serial implementation, v2.0 is better than the serial implementation. When compared with the `pthread` implementations, the runtime for the 2 thread, 4 thread and 7 thread implementations are shown here. The 7 thread implementation is

Benchmark	No.of Gates	32 thread	128 thread	256 thread	512 thread
c17.net	8	394	362	363	364
s27.net	20	940	902	904	911
s344.net	131	5302	5262	5265	5262
c432.net	136	5956	5898	5898	5898
s349.net	143	5428	5391	5389	5390
s400.net	152	6047	6015	6013	6014
s386.net	158	5842	5796	5793	5796
c499.net	178	5047	5014	5016	5016
c1355.net	182	5216	5184	5184	5183
c880.net	223	8056	8013	8014	8014
c1908.net	224	8585	8541	8541	8540
s526.net	235	7712	7683	7684	7683
s510.net	272	8140	8131	8130	8130
c2670.net	346	10008	10024	10025	10026
s1196.net	586	13582	13642	13645	13643
c3540.net	693	17757	17849	17856	17854
s1494.net	785	15066	15262	15264	15258
c5315.net	920	14933	15082	15151	15149
c7552.net	1149	18658	18917	18946	18923
c6288.net	1669	39698	39660	39821	39827
wb_dma.net	2654	26799	27874	28630	29120
systemcdes.net	2854	33051	33813	34238	34237
tv80.net	4213	60923	62827	63308	63404
systemcaes.net	4519	65281	66357	67207	68094
mem_ctrl.net	7468	67218	69237	71122	73704
ac97_ctrl.net	7865	40509	40785	42038	46563
pci_bridge32.net	9212	83811	85928	87332	89074
usb_funct.net	10627	58908	59996	62185	67271
aes_core.net	21374	47660	47699	50075	58128
des_perf.net	79028	92908	97452	98716	107669
vga_lcd.net	88405	187412	201763	202661	209074

Table 4.8: Runtimes of the GPU Accelerated SSTA v2.0 for all benchmarks reported in μs

better than the v2.0 implementation in all cases. The v2.0 implementation is almost on par with 4 thread implementation in “des_perf.net” benchmark while it performs only as good as the 2 thread implementation in the “vga_lcd.net” benchmark. The particular reason might be the amount of parallelism in each level being more in “des_perf.net” than “vga_lcd.net” as pointed out earlier in 4.2.

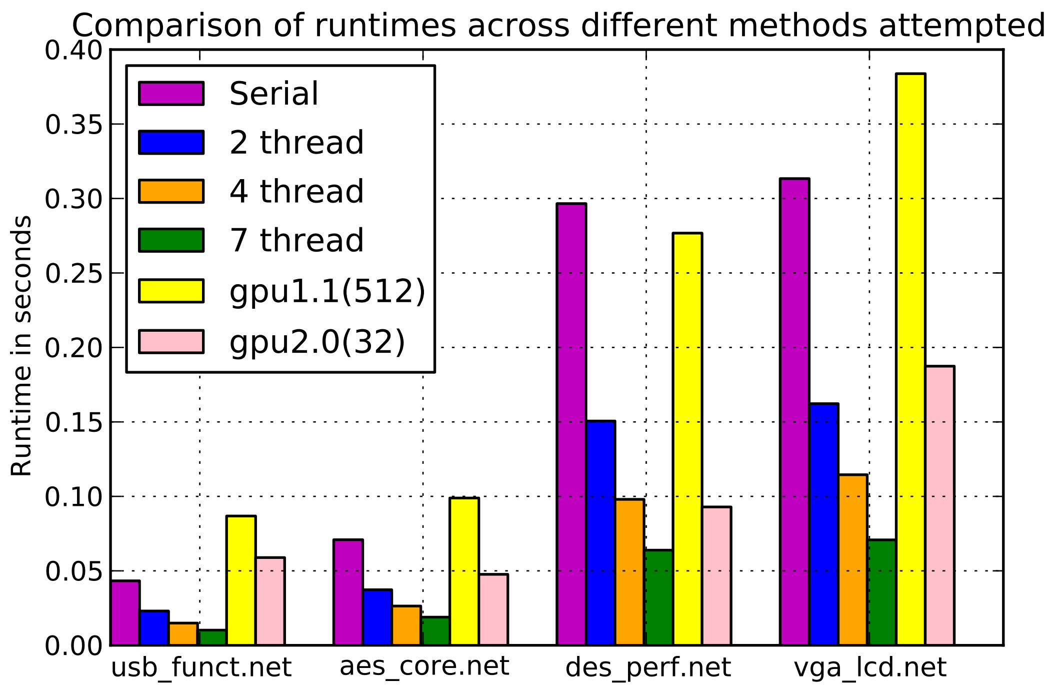


Figure 4.6: Comparison of different implementations of the algorithm

CHAPTER 5

Conclusion and scope for Future work

5.1 Contribution of this work

This work has presented a brief introduction on the need for SSTA along with the various challenges that the algorithm brings along with itself. The algorithm involved in the block based SSTA and the computations involved in addition and MAX are explained along with a qualitative idea of the errors associated with it. A brief introduction to CUDA is also presented to provide context. This work has a first order SSTA implementation on the GPU which could act as a framework for other extensions to built over this. Specifically, the areas covered in this work are

1. **pthread implementation** - A `pthread` implementation of the algorithm is implemented and the speed up associated with the pthread gives us a motivation for running the SSTA algorithm on the GPU
2. **CUDA implementation v1** - A simple translation of the idea involved in the previous case followed by optimization in memory coalescing and usage of constant memory for library look up.
3. **CUDA implementation v2** - Extension of previous implementation and to exploit parallelism across blocks instead of just a single block and multiple threads

The GPU v2.0 implementation does provide a speed up in comparison to the serial implementation, when compared to the `pthread` implementations the runtime while significantly slower than the 7 thread implementation it performs almost on par with the 2 thread implementation in a particular benchmarks and sometimes better than that depending upon the case. An indicator of parallelism $\frac{no\,of\,Gates}{no\,of\,Levels}$ though seem to point out that in cases with higher "parallelism" as indicated above, the GPU implementation does provide considerable speed up improving with respect to the pthread implementations.

Though it can be seen that for bigger benchmarks the GPU implementation performs considerably better in comparison to the pthread implementations, it is tough to predict the amount of gates before the GPU implementation performs better than the pthread versions. This is because of the very "non-linear" notion of context-switching that is in the GPU. The results for the ratio of runtime of GPU over the pthread implementation is plotted against $\frac{no\,of\,Gates}{no\,of\,Levels}$ and is shown in Figure 5.1

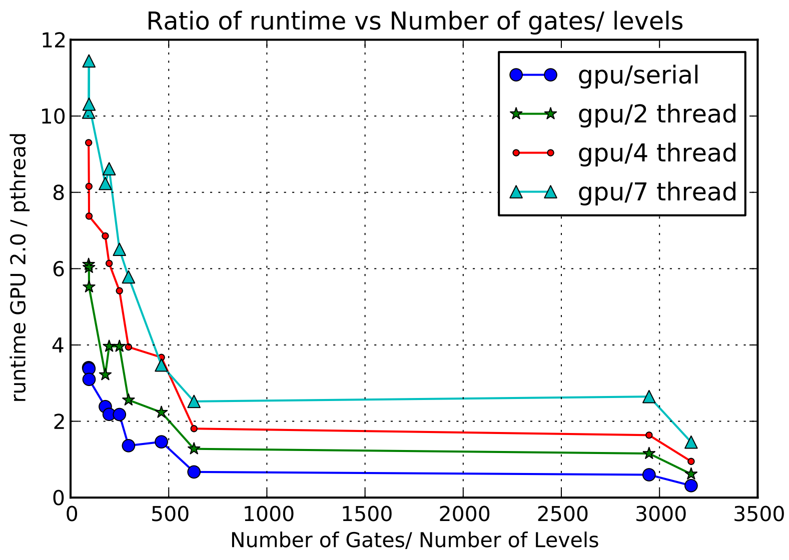


Figure 5.1: Ratio of runtime of GPU to pthread implementation vs Number of gates/level

Challenges while implementing the GPU version is that while memory coalescing takes a very important role, it automatically implies that array of structures doesn't translate necessarily as memory coalesced because of each thread accessing a member of its object which cannot be next to each other in memory. This in effect propagates the structure of arrays where a structure comprises of arrays of different objects. The implementation of array of structures of 2D array members breaks down the ease of programming and makes it a heavy task of book keeping but necessary for coalesced access. Moreover, the nature of the problem demands the data structures used here, and hence the huge data structures (egṗin [> 0.4kB]) which needs to be loaded from the memory. This again reduces the performance of the GPU implementation because of serializing of memory requests in the above case.

In comparison to (Gulati and Khatri, 2009) which takes more than 1s to complete the computations for a small benchmark such as $c7552.net$ with just 1000 gates the GPU implementation here takes in the order of ms mainly because of the algorithm used. The error in the present SSTA algorithm can be reduced by using different techniques and accounting for the known errors now.

All computations done in different platform were made sure that the output of these tools matched the serial code output. The computations done in the pthread implementation on **Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz** while the GPU implementation was on **NVidia GeForce GTX 560 Ti** with clock frequency @ 1.66GHz

5.2 Future Work

1. **Slack computation** - Slack computation is reverse traversal of the graph and computation of the required arrival time(RAT) is similar to what has already been done in this work.

2. **CUDA Implementation** - The choice of texture memory for read-only memory instead of constant-memory. This can help in setting up the Look Up Table used for finding out the delay/slew calculation across gates w.r.t load capacitance and input slew. Further it offers linear interpolation for points in between which is again a helpful and useful in this case. Also, `cudaMemcpy` takes time that is not negligible but this can be dealt by using cuda streams and/or page locked memory as the case might be.

3. **Block Based SSTA** - The GPU implementation is heavily memory bandwidth constrained. Further improvements such as the ones in (Zhan *et al.*, 2005) which involve a quadratic Canonical function and moment-matching through numerical integration as well as including non-linear dependencies and (Sinha *et al.*, 2007) which looks up the error in the MAX computation and combines path based and block based to achieve lesser error are good fits to make the problem more adaptable to the GPU framework.

REFERENCES

1. **Agarwal, A.**, **D. Blaauw**, **V. Zolotov**, **S. Sundareswaran**, **M. Zhao**, **K. Gala**, and **R. Panda**, Statistical delay computation considering spatial correlations. *In Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. ACM, 2003.

2. **Blaauw, D.**, **K. Chopra**, **A. Srivastava**, and **L. Scheffer** (2008). Statistical timing analysis: From basic principles to state of the art. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **27**(4), 589–607.

3. **Chang, H.** and **S. S. Sapatnekar** (2005). Statistical timing analysis under spatial correlations. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **24**(9), 1467–1482.

4. **Clark, C. E.** (1961). The greatest of a finite set of random variables. *Operations Research*, **9**(2), 145–162.

5. **CUDA, C.** (2012). Programming guide. *NVIDIA Corporation, July*.

6. **Gattiker, A.**, **S. Nassif**, **R. Dinakar**, and **C. Long**, Timing yield estimation from static timing analysis. *In Quality Electronic Design, 2001 International Symposium on*. IEEE, 2001.

7. **Gulati, K.** and **S. P. Khatri**, Accelerating statistical static timing analysis using graphics processing units. *In Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. IEEE Press, 2009.

8. **Jaffari, J.** and **M. Anis**, On efficient monte carlo-based statistical static timing analysis of digital circuits. *In Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2008.

9. **Jess, J. A.**, **K. Kalafala**, **S. R. Naidu**, **R. H. Otten**, and **C. Visweswariah** (2006). Statistical timing for parametric yield prediction of digital integrated circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **25**(11), 2376–2392.

10. **Sinha, D.**, **L. Guerra e Silva**, **J. Wang**, **S. Raghunathan**, **D. Netrabile**, and **A. Shebaita**, Tau 2013 variation aware timing analysis contest. *In Proceedings of the 2013 ACM international symposium on International symposium on physical design*. ACM, 2013.

11. **Sinha, D.**, **N. V. Shenoy**, and **H. Zhou**, Statistical gate sizing for timing yield optimization. *In Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*. IEEE, 2005.

12. **Sinha, D.**, **H. Zhou**, and **N. V. Shenoy** (2007). Advances in computation of the maximum of a set of gaussian random variables. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **26**(8), 1522–1533.

13. **Visweswariah, C.**, **K. Ravindran**, **K. Kalafala**, **S. G. Walker**, **S. Narayan**, **D. K. Beece**, **J. Piaget**, **N. Venkateswaran**, and **J. G. Hemmett** (2006). First-order incremental block-based statistical timing analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **25**(10), 2170–2180.

14. **Zhan, Y.**, **A. J. Strojwas**, **X. Li**, **L. T. Pileggi**, **D. Newmark**, and **M. Sharma**, Correlation-aware statistical timing analysis with non-gaussian delay distributions. *In Design Automation Conference, 2005. Proceedings. 42nd*. IEEE, 2005.