UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Partitioning Approach for GPU Accelerated Level-Based On-Chip Variation Static Timing Analysis

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Michael Longqiang Zhang

2010

DEDICATION

This thesis is dedicated to Angeline Feng, whose love and support has no upper bound.

To my family who have always been there for me even when I am too busy to stay in touch.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thanks to Chung-Kuan Cheng for his patience, guidance, and abundant opportunities he has made availabe to me in the past two years.

VITA

2008	B. S. in Computer Science, Worcester Polytechnic Institute
2010	M. S. in Computer Science, University of California, San Diego

ABSTRACT OF THE THESIS

A Partitioning Approach for GPU Accelerated Level-Based On-Chip Variation Static Timing Analysis

by

Michael Longqiang Zhang

Master of Science in Computer Science

University of California, San Diego, 2010

Professor Chung-Kuan Cheng, Chair

Technology and design trends have made timing analysis the bottleneck of electronic design automation (EDA) tools. Efficient and accurate timing analysis is a challenge that the EDA industry must overcome in order to move forward. Using LLC-OCV leverages Physical Location, Path Level, and Cell type information to further increase timing accuracy. This model introduces increased data complexity as a result of maintaining delays for each unique path-level. We parallelize this computation for co-processing on a CUDA enabled GPU. We introduce a novel divide-and-conquer partitioning approach for computing the per-level delay data used in the level-based aspect of LLC-OCV. Partitioning the circuit graph halves the inherently serial structure of a topological traversal of the circuit graph with a costly but more parallel merge step that combines the solutions of the two partitions. Using a massively parallel GPU-based approach allows us to absorb the cost of merging by performing it in parallel. Our experimental results on the ISCAS '85 benchmark demonstrate our parallel algorithm scales with timing graph size more efficiently than the serial algorithm. Results also show that our partitioning approach allows us to more fully utilize the massively parallel computational resources of the GPU. Our experiments on artificial test cases demonstrate that the parallel algorithms outperform the serial algorithm on large non-linear graph structures. We also find that LOCV timing analysis is a memory bound computation. We expect our algorithm to perform better on the newer Fermi architecture because of the new cached memory architecture.

Chapter 1

Introduction

Static timing analysis (STA) is a critical phase of the digital circuit design process. Technology and design trends have made timing analysis the bottleneck of electronic design automation (EDA) tools. As the feature size scales downwards of 32nm, the effect of process variation is magnified, increasing the importance of STA. The scaling of feature sizes also increases the number of gates such that designs can often reach or exceed the 100-million gate mark. Furthermore, static timing analysis is only one stage of a much larger flow of design verification, making memory consumption a concern. Efficient and accurate timing analysis is a challenge that the EDA industry must overcome in order to move forward.

As process geometries shrink, the ability to control manufacturing process variations on a single die becomes increasingly difficult. The classic Min-Max approach used in [11], propagates the minimum and maximum delays through the circuit graph. Using a the simple model used in classical STA results in an overly pessimistic analysis. The simplistic model fails to factor the inherent statistical variation. This motivated the rise of statistical static timing analysis (SSTA), which has been covered in literature extensively [2, 20, 17, 9, 7]. SSTA uses a cumulative probability distribution function (CDF) and probability density function (PDF) to represent arrival times, rather than the single arrival time used in STA [2, 5]. Although CDFs and PDFs can be represented using a piece-wise linear approximation with a few sample points, the arrival time propagation is still computationally expensive. In particular, it involves the computation of the convolution of two CDFs [5].

The concept of Location-based On-Chip Variation (LOCV) is introduced in [3], and extended to Location Level and Cell-Based On-Chip Variation (LLC-OCV) in [9]. Recently, it has been shown that intra-die variations are spatially correlated with device characteristics [18, 13, 2]. LOCV factors in physical location using a variable derating factor to derive a more accurate timing analysis[3]. LLC-OCV extends this further to include the effect of Level and Cell type to further increase timing accuracy [9]. The LLC-OCV model can be used given the necessary intra-die process silicon data and Monte Carlo simulations. The derating factor is then based upon path location, number of levels, and types of cells. In comparison with CDF and PDF approaches, this model is computationally cheaper and easier to implement. As an extension to the STA, LOCV analysis is familiar territory, making it easy to adopt and integrate into existing EDA tools.

Another approach to accelerating SSTA is to leverage parallel computation. In general, the trend has been a movement towards parallelism in order to satisfy the aggressive demand for faster EDA. Multi-threading approaches have been utilized in the context of SSTA [19]. Though concerns have been expressed that the lifespan of multi-threading in EDA is limited as a result of the limited number of CPU cores [15]. Others have sought alternative and more massively-parallel approaches using GPUs. The GPU has been used to accelerate Monte Carlo based SSTA with 200x or more speeds ups [7]. The fundamental disadvantage of CPU-based parallelism is its limited scale of parallelism. Parallelism is limited by the number of cores in a CPU, which falls in the single digit range, as well as the overhead of thread management. Massively scaled parallel computation using CPUs tends to be far too expensive. Despite multi-threading on CPUs being more robust than GPUs and more suited to task level parallelism, multi-threading on CPU fails to grow at the rate at which massively parallel threads has grown on GPUs. Therefore, we chose to pursue GPU-based parallelism.

We implement a GPU parallel algorithm for computing the per-level delay data used in the level-based aspect of LLC-OCV. We also introduce a divide and conquer approach that partitions the timing graph into two partitions, and performs the propagation of delays from both input and output. Our approach reduces the sequential complexity of the problem. We divide the problem into two sub-problems, whose solution can be combined to derive the solution to the original problem. Partitioning the circuit graph halves the inherently serial structure of a topological traversal of the circuit graph with a costly but parallel merge step that combines the solutions of the two partitions. Using a massively parallel GPU-based approach allows us to absorb the cost of merging by performing it in parallel.

By partitioning the problem, we also reduce memory consumption by reducing the amount of data propagated through the graph. Since per-level delay data must be propagated from source to sink, the number of levels that must be processed and stored at a given node is directly proportional to circuit size. A partition reduces the effective depth of propagation, reducing the amount of data propagated and stored.

We test serial, parallel, and partitioned parallel algorithms on three CPUs and two GPUs. Our results show that our parallel algorithm scales more efficiently than the serial algorithm. Our partitioning approach successfully increases the amount of parallelism that can be exploited by the GPU, and accelerates computation. Also, we conclude that the timing analysis problem is a memory bound computation.

This thesis is organized into a total of five chapters. Chapter 1 is the introduction. Chapter 2 presents background information on timing analysis and parallel computation. Chapter 3 presents the algorithms for computing the LOCV delay propagation serially, in parallel on the GPU, and in parallel on the GPU using our partitioning approach. Chapter 4 presents the experimental design, the experiments, and their results. Finally, Chapter 5 will wrap up the thesis with the conclusion.

Chapter 2

Background

2.1 Timing Analysis

The verification of a circuit design consists of three major phases: functional design verification, physical design verification, and timing verification [16]. Functional design verification ensures that the circuit performs the correct function at a high level. Physical design verification ensures that the design meets physical constraints such as wire separation and component dimension constraints. Timing verification, of which timing analysis is the primary computation, validates the path delays from the input to the output. When a signal propagates through a circuit, it propagates through various components and wires to reach any given point. Timing verification ensures that when a bit changes, the signal generated propagates through the circuit in time for the next clock cycle.

2.1.1 Static Timing Analysis

Static timing analysis (STA) is a method for verifying the timing of a circuit without performing circuit simulation. Circuit simulation is intractable for large scale designs. STA can be performed using either path-based or block-based methods. Both approaches have been implemented and discussed in literature [16, 13]. The primary criticism of path-based approaches is the exponential growth in complexity with problem size as a result of its usage of path enumeration [16]. This has led to the popularity of block-based approaches[20, 5], which use a non-enumerative approach, reducing computational complexity at the cost of increased memory. With circuit sizes already prohibitively large and growing, we choose to use a block-based approach.

Block-based methods use the idea of slack to avoid the enumeration of paths. Slack is simply the difference between the required time of arrival of the signal and the arrival time of the signal. In STA, the earliest and the latest arrival times for a signal to propagate to a given point in the circuit graph are computed. Computation of the earliest and latest arrival times can be performed by propagating the maximum and minimum delays in a topological traversal of the circuit graph. Delay measures the time from the creation of the signal to the arrival of the signal. Each edge in the graph carries a weight, which represents the time it takes for a signal to propagate across that edge, or wire. At each node in the circuit graph, only the maximum and minimum delays from all paths leading to the node are propagated. The maximum and minimum delays of the whole circuit graph can be found by finding the max and min of the output nodes.

Figure 2.1 demonstrates a simple three node graph with weights for each edge. The maximum delay for reaching each node is shown inside the node's circle. The output node retains the maximum between 0.9 and 1.0. However, classical static timing

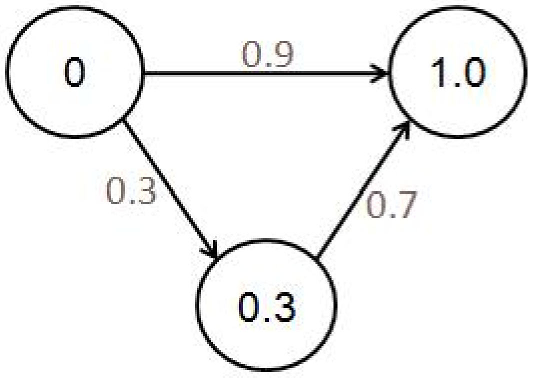

Figure 2.1: A simple three node graph demonstrating propagation of delays in static timing analysis.

analysis is inaccurate and tends to be pessimistic as a result of its overly simplistic model.

2.1.2 Intra-Die Variation

Intra-die variation is the variation in the properties of the silicon on which the circuit is manufactured. It is impossible to manufacture a perfectly uniform piece of material, and consequently there is process variability. As circuits scale increasingly smaller, the effect of variation is magnified. At larger sizes, the variation is masked because a given wire may cross several regions whose properties average out. However, as feature sizes become smaller, the slightest variation can drastically affect the actual delay for propagating between two pins. Intra-die variation often exhibits spatial correlations. Devices in the same region are likely to have similar properties. This has been shown to be true for gate location and length [3, 9]. The simple STA model presented above fails to capture intra-die variations, resulting in inaccurate timing analysis. Statistical static timing analysis attempts to remedy this, though at the cost of additional computational cost.

2.1.3 Statistical Static Timing Analysis

Statistical Static Timing Analysis (SSTA) was designed to address the shortcomings of STA. Whereas STA is deterministic and cannot account for process effects, SSTA uses a probabilistic model. SSTA models delay as a distribution rather than a single value. Like STA, it can be done using a path-based or block-based approach. In a path-based approach, SSTA requires that you specify the paths of interest, as path enumeration remains intractable. The block based approach follows the same flow as STA. The distributions of arrival and required times are propagated in a topological traversal of the circuit graph. Figure 2.2 shows how SSTA uses distributions rather than single values. The major computational challenge of SSTA is in the propagation of delay distributions [2]. Since SSTA models the correlations between arrival times and gate delays, the correlations must be taken into account when propagating the distributions. This is typically done using Monte-Carlo methods, which draw random values from the distribution, and performs STA over the entire graph repeatedly with random samples for each run. Monte Carlo SSTA has been successfully parallelized on the GPU in [7]. Analytical theoretical frameworks for performing SSTA have also been proposed [13].

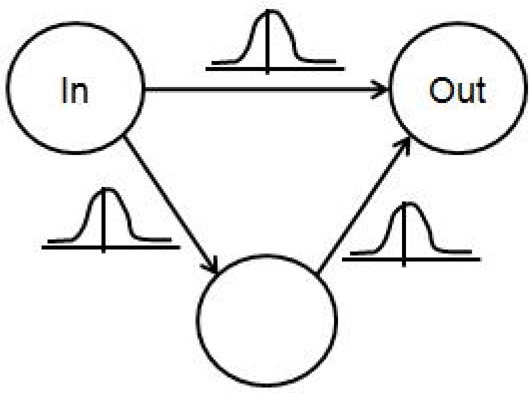

Figure 2.2: A simple three node graph demonstrating propagation of distributions in statistical static timing analysis.

Criticisms of SSTA target its complexity, and difficulty in using within optimization flows. Another challenge comes from getting the necessary manufacturing process data from fabrication companies. Ultimately, the improvement in accuracy is not worth the effort of it. Instead, deterministic STA has been modified to model process effects. Location-based On-Chip variation (LOCV) is one such method that exploits the location of a cell to derive a variable derating factor based on the distance covered by a path.

2.1.4 Location Level and Cell Based On-Chip Variation Analysis

LOCV has already been proposed as a standard, and receives wide support from fabrication companies. LLO-OCV is a methodology that extends LOCV using Gate-Level and Cell-based perspectives to further enhance the accuracy [9]. [9] shows that information about the gate-level, the number of gates along a path, can be used to account for process variation more accurately. Like Location-Based OCV, Level-Based OCV also generates a variable derating factor which is used to enhance traditional STA. While Location information is attainable given the placement of the circuit, gate-level information must be propagated through the graph during the STA flow. In order to exploit Level-Based information, the max and min delay for all paths of a given gate-level must be computed. The delay per gate-level is computed by propagation of the max and min for each gate level to each node in topological order. Figure 2.3 demonstrates the information maintained in Level-based OCV. As the number of unique path lengths

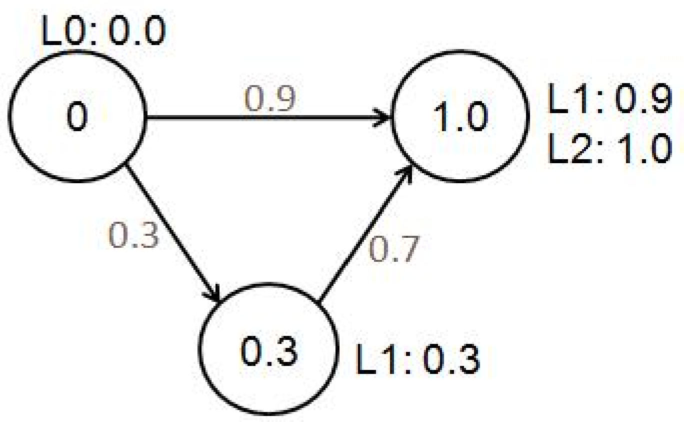

Figure 2.3: A simple three node graph demonstrating propagation of delays per path length in level-based on chip variation analysis.

increases with the depth of the graph, the number of delays that must be computed and propagated becomes expensive. Our goal is to address this problem by making this computation more efficient, allowing the LLC-OCV methodology to be used for more accurate STA. Our approach is to use parallel processing.

2.2 Parallel Processing

2.2.1 Forms of Parallelism

An algorithm may be decomposed into parallel computations at various granularities. Fine grained parallelism assigns each thread to perform a small task, and then communicate its results to contribute to the greater computation. Coarse grained parallelism uses threads that perform a larger amount of work before communicating with other threads. Fine grained parallelism has a low computation to communication ratio. Coarse grained parallelism has a high computation to communication ratio.

A given problem may have one or more types of parallelism that can be exploited. Traditionally, focus has been placed on instruction-level parallelism, which re-orders a stream of instructions and executes them in a pipeline. However, growth with this approach was not sustainable. Task level parallelism is the execution of multiple potentially different programs in parallel on the same data. A major challenge with task level parallelism is dependencies. Communication and sharing of data is complex, costly, and difficult to debug. In contrast, data parallelism exploits a computation where the same instructions are performed on different data, in which case the computations are independent of each other. Many scientific applications exhibit strong data parallelism.

2.2.2 Parallelization Using CPUs

Parallel processing can be broadly categorized as using either CPUs or GPUs. With CPUs, coarse grained parallelism is achieved by executing multiple threads on multiple processor cores. Multiple computers are networked together into clusters which are coordinated using the MPI protocol. MPI facilitates the distribution of threads across clusters of computers to perform computation in parallel. More recently, the trend has shifted from clusters of computers towards increasing the number of cores on a single chip. Each core is self sufficient in that it maintains its own instruction pointer, register state, and provides extensive instruction level parallelism. Multi-core chips rarely exceed 10 cores per chip . The scaling of the number of cores is not expected to grow rapidly in the near future. However, the overhead of creating and managing threads in a multi-core environment is less than using the MPI protocol for a cluster of CPUs.

CPUs provide fine grained parallelism through instruction level parallelism. Historically, CPU scaling has focused on out of order execution, pipelining, and a memory hierarchy that can support this fine-grained parallelism. The CPUs memory hierarchy is essential for supporting all types of parallelism on the CPU. Up until very recently, the GPU has lacked a proper memory hierarchy.

The cost of purchasing and setting up a cluster of computers is high. The power consumption for a cluster is also extremely high. However, shifting to a multi-core CPU is unsatisfactory because it lacks the scale of parallelism achievable by a cluster. GPUs are the best choice for a cheap solution to massively data parallel computations. A five-hundred dollars state of the art Fermi architecture GPU has 480 cores with a clock rate of 1.4 GHz.

2.2.3 Parallelization Using GPUs

The usage of GPUs for general purpose computation is motivated by the massively parallel architecture it developed as a result of being optimized for the intensely data parallel application of computer graphics. The same operations are performed on millions of vertices, pixels, and other graphics primitives. As GPUs moved from their fixed-function pipelines to programmable shader pipelines, adventurous individuals explored representing their data as textures, and performing computations using custom shaders. Many scientific applications exhibit high levels of data-parallelism, and General Purpose computation on GPUs grew until Nvidia, a leader in the industry, began directly supporting it with their CUDA architecture. Along with the CUDA architecture, Nvidia provided a C API for managing memory transfer, and executing programs in parallel on the GPU. Recently, a standardized API with industry wide support has gained momentum, and Microsoft has entered the arena with its own DirectCompute as part of their DirectX 11 release. The programming model used in C for CUDA, Nvidia's C API, is similar to that of OpenCL. C for CUDA is used here for its currently superior documentation and support. GPU for co-processing is a rapidly growing field. C For Cuda

2.2.4 CUDA Programming Model

Nvidia provides a very simple set of C extensions, called C for CUDA, for programming the GPU. The goal of the programming model is to make parallel programming transparently scalable, much like how 3D graphics applications scale to take advantage of increasingly more powerful GPUs with widely varying numbers of cores. The goal is to allow code to be written once, and have performance scale with newer hardware using more cores. Fundamentally, it consists of three core abstractions: a hierarchy of thread groups, a hierarchy of memory, and barrier synchronization.

C for CUDA provides a C extension which invokes a kernel. A kernel is piece of C code intended for running on the GPU. When you call a kernel for execution on the GPU, you specify dimensions for the execution of that kernel in parallel. The GPU will automatically create a single thread to run a single instance of the kernel. C for

CUDA uses a hierarchy of threads abstraction, which allows the programmer to specify the dimensions of a grid and block. As shown in figure 2.4, a kernel call executes a one or two dimensional grid of thread blocks. A thread block is a one, two, or three dimensional set of threads. A kernel invocation with grid dimension (3, 2) and block dimension (5, 3, 1) will spawn a 3x2 grid of thread blocks for a total of 6 blocks. Each block will create a 5x3x1 block of threads, for a total of 15 threads per block. This kernel call creates in total 90 threads. Each thread has a unique ID number within its block. Each block has a unique ID number within the grid. These IDs and the dimensions of the kernel invocation are automatically available within a kernel thread, allowing a thread to determine its position in the scheme of things, and ultimately its role in the parallel computation. The proportions of the thread hierarchy used to generated threads from a kernel call is called the execution configuration. C for CUDA also provides an abstraction to the memory system, as shown in figure 2.5. There are five different types of memory: global, local, texture, constant, shared, and register. The global memory is the memory shared and accessible to all threads. However, since global memory is off-chip reading and writing to it is extremely slow. Local memory is also off-chip memory but is accessible only by a single thread. Local memory is used only when there are insufficient registers for all threads to share. Texture memory is a cached memory that is accessible by all threads, and provides special texture-based access. Texture memory is optimized for 2D spatial locality, so threads in the same warp accessing texture addresses that are close together will achieve best performance. Constant memory is a cached memory also accessible by all threads, but provides register access latencies when all threads in a warp access the same address. Shared memory is an on-chip memory shared by all threads within the same block. The benefit of shared memory is that its access time is comparable to register access times, assuming there are no bank conflicts. Shared memory is divided into banks across the chip potentially allowing all executing threads to access memory at once. Bank conflicts occur when two threads access memory within the same memory bank, which forces the memory requests to be serviced sequentially rather than in parallel. Finally, registers are accessible by only the thread it belongs to.

Finally, C for CUDA provides an abstraction for synchronization. Barrier synchronization allows all threads in a block to synchronize at the barrier. Barrier synchronization forces all threads in a block to stall until all threads in the block have reached the barrier. Then all threads will be able to proceed beyond the barrier. Barrier synchronization is handled by the hardware, and is done with virtually no overhead. However, there are no guarantees on the synchronization of threads in different blocks. A kernel call can be seen as a barrier synchronization for blocks. C for CUDA also provides a way to execute kernel calls asynchronously with memory. A kernel can be executed on a specified stream, a sequential queue of tasks to be performed by the GPU. Memory transfers can be executed simultaneously with kernel calls. A memory transfer can be executed on a separate stream, allowing some of the memory transfer time to be masked by kernel execution time in a separate stream.

2.2.5 CUDA Architecture

The CUDA architecture consists of a set of Streaming Multiprocessors. An streaming multiprocessor consists of multiple Scalar Processors cores, each of which executes a thread. Each multiprocessor also has a multi threaded instruction unit as well as on-chip shared memory. The multiprocessor creates, manages, synchronizes, and executes threads concurrently with zero scheduling overhead.

CUDA uses an architecture called single instruction multiple thread (SIMT). Each thread runs on a single processor core, and has its own instruction address and register state. Groups of threads are organized into warps. A warp is a group of threads that are executed simultaneously. Individual threads in a warp all start on the same instruction, but are otherwise free to branch and execute independently. Since each warp executes simultaneously on a multiprocessor, and a multiprocessor can only execute one instruction at a time, each branch must be executed in serial. When a multiprocessor is given multiple thread blocks to execute, it organizes them into warps, and executes whichever warps are ready for their next instruction. With large numbers of blocks, a multiprocessor can mask the memory access latency of a warp by executing another warp rather than stalling. Since several threads may be assigned to a core at any point in time, the total registers are divided amongst the threads. A limited number of registers is what creates the necessity for local memory, allowing what normally would have been stored in register memory to overflow into device memory. Having each thread own its own registers allows for the no overhead warp scheduling. SIMT differs from Single Instruction Multiple Data architectures (SIMD), in that it provides an abstraction by specifying only the behavior of branching, rather than enforcing no branching. From a programmer's perspective, SIMT can be ignored, though at the cost of substantial performance degradation.

In particular, the Nvidia G200 architecture used in the Tesla C1060 GPU has 8 cores in each multiprocessor, and a total of 30 multiprocessors. It schedules the execution of half-warps rather than full warps. We use the Tesla C1060 GPU as well as the 8600M GT, both of which follow the 8 core architecture and uses half-warps. The 8600M GT only has 4 multiprocessors for a total of 32 cores.

The Nvidia GF100 Fermi architecture is a redesigned CUDA architecture released in April 2010. Its architecture is discussed here briefly as a prelude to the later discussion on the new architecture's effect on applications such as ours. The Fermi architecture has 32 cores in a Streaming Multiprocessors, and supports double precision performance comparable to CPUs. It can simultaneously schedule two independent warps, and no longer uses half-warps. It also provides a unified address space along with 64-bit support. Most importantly, it introduces a full memory hierarchy consisting of a configurable L1 and unified L2 caches. It supports concurrent kernel execution as well as out-of-order thread block execution, and two overlapped memory transfer engines.

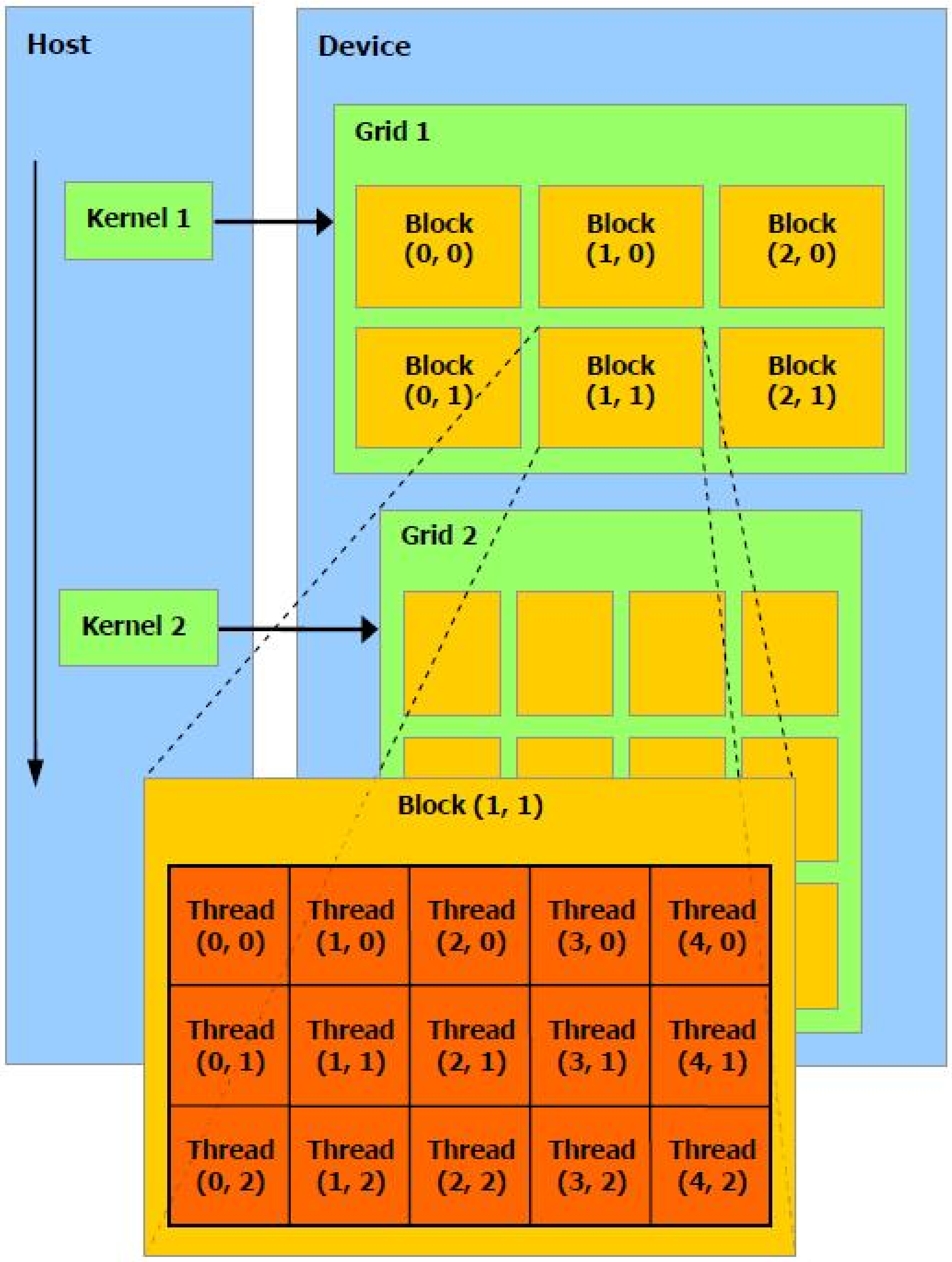

Figure 2.4: The thread hierarchy abstraction used in the CUDA programming model [12].

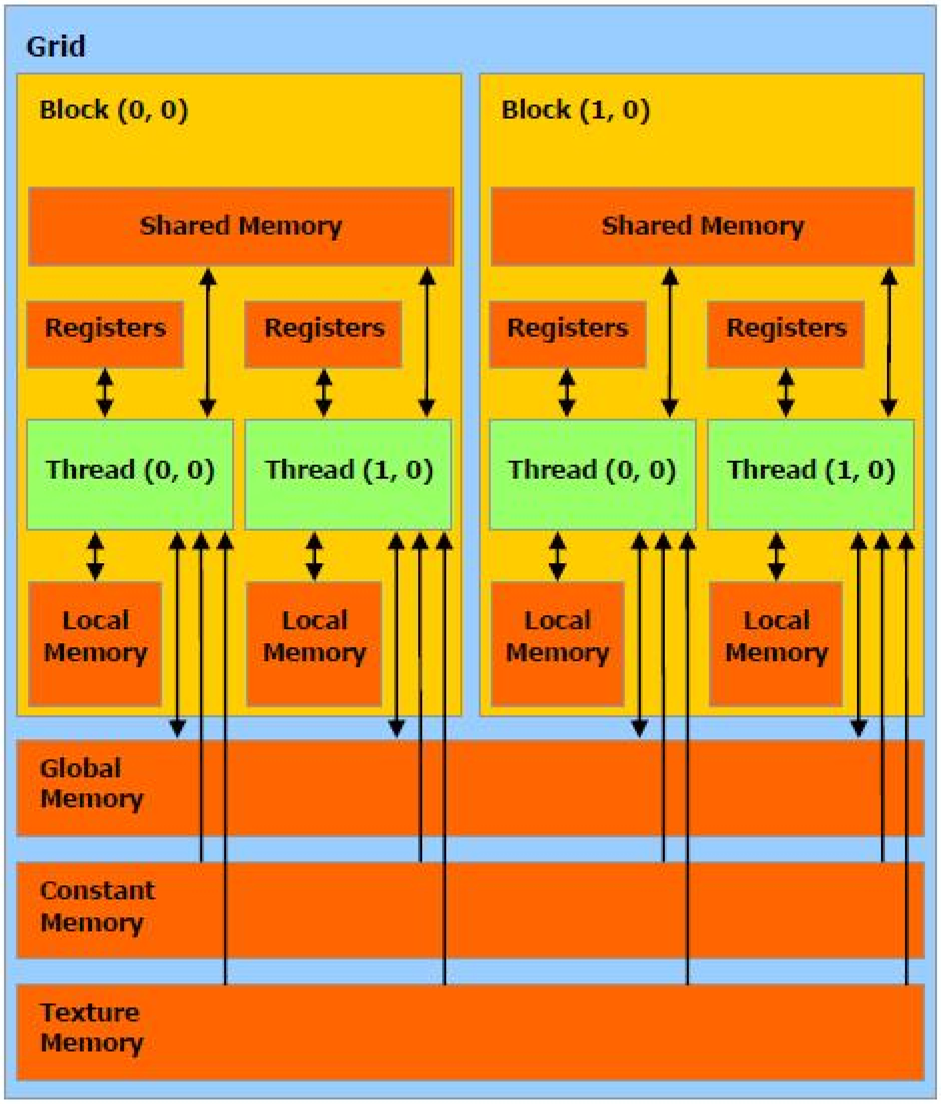

Figure 2.5: The memory system used in the CUDA architecture [12].

Chapter 3

Algorithms for Level-Based On-Chip Variation

3.1 Problem Statement

Given a directed acyclic circuit graph with weighted nodes, compute the minimum and maximum delays for each unique path length from input to output for use in LOCV STA using a scalable and low cost solution. Delay is defined to be the sum of the weights of the nodes in a path. Since computing the minimum delay is identical to computing the maximum delay except for a change from using the maximum to minimum operator, the remainder of this thesis will discuss only determining the maximum delay. Parallelization of the computation using GPU is used to achieve the scalable and low-cost objectives of the problem. The problem can also be phrased as an efficient usage of the GPU for accelerating the computation of maximum delays through a directed acyclic circuit graph with weighted edges.

Given a timing graph G, a directed graph whose vertex set $V(G)$ contains all pins. The weighted edges of the graph represent the wire connections with propagation delays corresponding to the weight of the edge. Timing analysis is performed from register to register. We cut the graph along the register components, resulting in a directed acyclic graph. The input to a register component becomes the output of the timing graph, and the output of the register component becomes the input of the timing graph. The timing analysis problem is formulated as the topological propagation of the maximum delay through the timing graph. Since the concept of a topological traversal is inherently serial, we define the notion of a topological block.

3.1.1 Topological Block Ordering

A topological traversal is defined as a traversal through a directed graph in which each node is visited after all of its parents have been visited. We refer to a topological block as the set of nodes whose order can be freely interchanged in a topological traversal. This definition implies that nodes in a topological block have no interdependencies. The concept of a topological block enables a data parallel perspective on the timing analysis algorithm. The algorithm for sorting nodes of a graph into topological blocks is essentially a breadth first search and is not presented here. In figure 3.1, a simple graph is shown seperated into topological blocks. Note that the longest path through the circuit graph is exactly one less than the number of topological blocks. In practice, since the input nodes of the graph do not need to be visited, the number of topological blocks that need to be computed is equal to the longest path. The delays for a circuit are recomputed

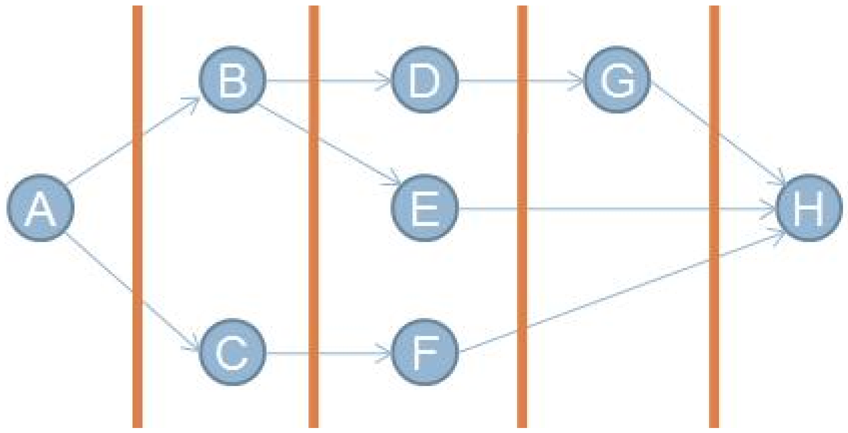

Figure 3.1: A simple graph separated into topological blocks by the vertical lines.

frequently in optimization programs. Rather than traversing the graph, which may be originally stored in a graph structure of node objects connected via pointers, we compute the topological block ordering and store it in a data structure that provides a structured access pattern. Computing the topological block ordering provides a reusable ordering for repeated calculations of the delay. Computing the topological block ordering can be easily updated for incremental changes.

3.2 The Serial Algorithm

The basic serial algorithm for computing the maximum delays for each unique path length is simply a topological traversal of the circuit graph where each visit to a node computes the maximum delay for each unique path length propagated from the node's parents.

3.2.1 Data Structures

The serial algorithm does not use any specialized data structures except for one used to store the maximum delay for each path length. The delay data structure is a two dimensional array where the first dimension has a size equal to the longest path, and the second dimension is equal to the number of nodes. If an element of the delay array has a value of -1, it represents that no delay of that path length has been computed. Figure 3.2 shows the strucutre of this array. In the serial algorithm, a smaller data structure

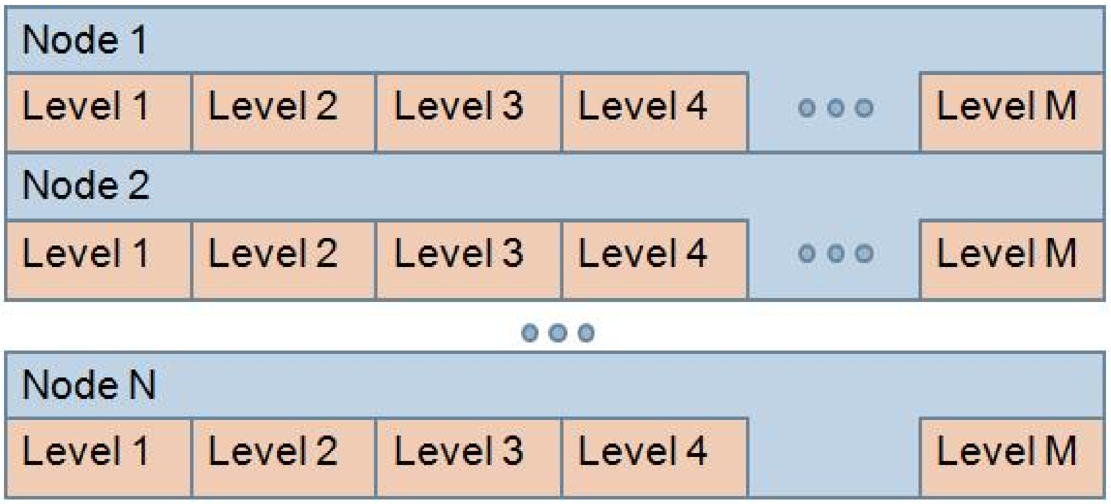

Figure 3.2: An array structure used to store the delays for each unique path length for each node in the graph.

involving lists rather than an array could be used to store delays of levels, however the predictability of the access pattern for this structure ultimately provided superior performance. The array wastes memory because memory is allocated for storing the delay of all possible path lengths, even though some path lengths do not occur.

3.2.2 Algorithm

The serial algorithm simply computes each topological block in order. Each node in the topological block is visited. Each parent of each node in the topological block propagates its delays. The propagated delay for each path length of each parent node is compared to the delay to the current max delay of the corresponding incremented path length to determine whether to overwrite the current maximum delay with the new maximum delay. The pseudo code for this algorithm is presented in figure 3.3.

```
1:  for each block $b$ do
2:      for each node $n$ in block $b$ do
3:          for each parent $p$ of node $n$ do
4:              for each level $l$ < current block $b$  do
5:                  if $delay[p][l] \neq -1$ then
6:                      $d \leftarrow delay[p][l]$ + delay of edge
7:                      if $d > delay[n][l+1]$ then
8:                          $delay[n][l+1] \leftarrow d$
9:                      end if
10:                 end if
11:             end for
12:         end for
13:     end for
14: end for
```

Figure 3.3: Pseudocode for the serial LOCV STA algorithm.

3.2.3 Approaches to Improving Performance

In order to improve upon the serial algorithm, we considered two major approaches: minimize the number of delays that need to be incremented and compared, and the second is to parallelize the algorithm and execute it on a GPU. The primary difference between classical STA and LOCV STA is the increased number of delays that must be propagated resulting from maintaining a maximum delay for each unique path length. The increase in computations also causes an increase in the data parallelism of the computation. In order to determine how to parallelize the algorithm, it is helpful to

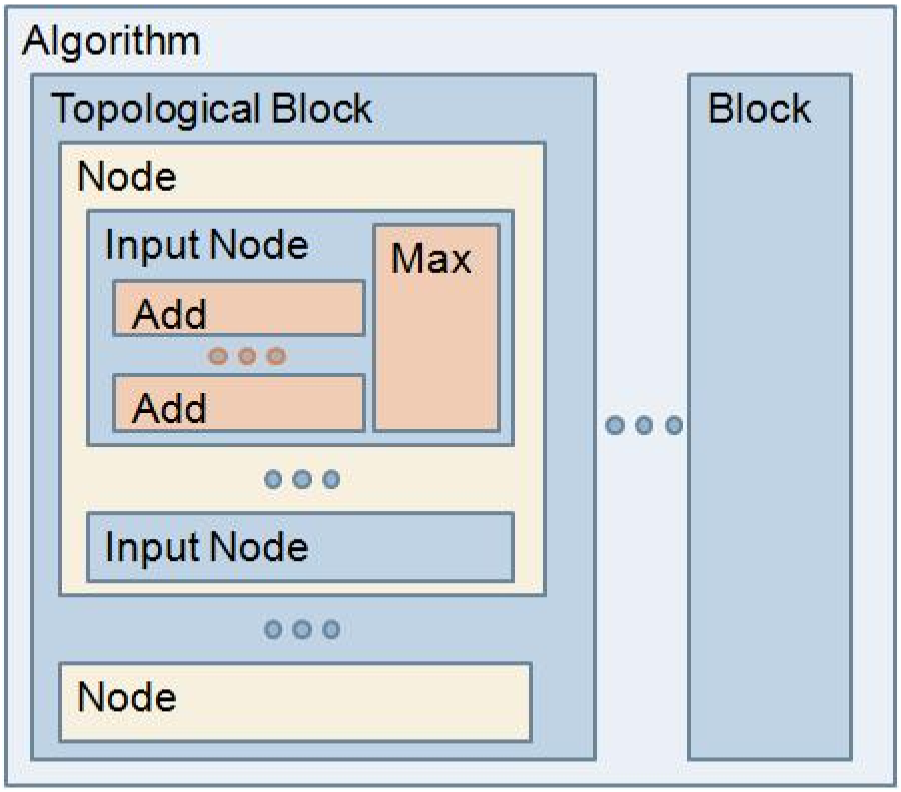

Figure 3.4: This figure shows the serially dependent components of computation as horizontally adjacent boxes.

view the algorithm in a deconstructed manner showing which computations are sequentially dependent. Figure 3.4 is a diagram showing which parts of the computation can be done in parallel, and which parts of the computation must be done serially. Horizontally adjacent boxes are serially dependent, and vertically adjacent boxes can be computed in parallel. The key units of computation that must be done sequentially are the topological blocks, and the maximization stage that compares the updated delays from each input node to determine the maximum delay that should be retained at each node. For each topological block, there are two sequentially dependent phases of computation, the addition of the delays for each level, and the maximization over the input nodes for each level.

3.3 A GPU Parallel Algorithm

The following sections describe a parallel algorithm implemented on the CUDA architecture using C for CUDA. The first step in designing an algorithm for the GPU is to deconstruct the algorithm for execution on the GPU. In order for the algorithm to run on the GPU, it needs to be structured as a set of independent sub-problems that can each be broken into potentially dependent sub-sub-problems. The first division forms a coarse-grained parallel decomposition. The second division is a fine-grained parallel decomposition. At the lowest level, we must decide what function each individual thread will perform. The deconstruction corresponds directly to the execution configuration of the GPU kernel. Given the execution configuration, the data structures must be designed for optimizing accesses to memory. The execution configuration dictates the form of the data structure because the accesses to the data structure need to be structured and efficient. Finally, the actual kernel program follows naturally from the execution configuration and data structure design.

Due to the synchronization within a kernel being limited to the block scoped barrier synchronization, the synchronization of each topological block must be done at the kernel level. For each topological block, the natural decomposition of the computation is into two kernels: an add kernel, and a max kernel. These kernels correspond directly to the add and max phases performed for each topological block described in the previous section. We simply state without, presenting pseudo code, that the CPU code simply iterates over each topological block and executes the add and max kernels for each block after transferring the necessary input connectivity data onto device memory. For each kernel, we describe, in order, the execution configuration, data structures, and the kernel code.

3.3.1 The Add Kernel

Execution Configuration

In order to fully utilize the fine-grained data parallel capabilities of the GPU, we decided to have each thread on the GPU perform a single addition for propagating the delay for a single level from a single input node for a single node of a topological block. We also want to ensure that each thread block will have enough threads. We assign each thread block to perform the propagation from all input nodes for a given node. Therefore the 1st dimension of the grid has a size equal to the number of nodes in the topological block.

In order for the kernel to execute efficiently, memory accesses should be coalesced. Coalesced memory accesses occur when threads of the same warp access memory within the same block of memory. In order for memory accesses to be coalesced, the width of the thread blocks needs to be a multiple of the half-warp size and the data structure needs to have a width with a multiple of 16. As mentioned previously, the memory alignment of 16 bytes is enforced by the method of allocation provided by C for CUDA. Therefore, we are only restricted by the requirement of the execution block width being a multiple of a multiple of half-warp.

Memory coalescing dictates that a warp of threads should access the delays for the same input node. We conclude that the execution configuration for a thread block should have the first dimension be the number of levels padded to a multiple of 16, and a second dimension equal to the number of input nodes. Furthermore, adhering to constraints on the number of threads in a block, the number of blocks should be scaled by increasing the dimension of the grid in the 2nd dimension.

Data Structures

Figure 3.5 shows the three dimensional array used to represent the input connectivity of the circuit graph. A three dimensional data structure is allocated in device memory as a contiguous space padded to maintain memory alignment. The first dimension has a size of the number of topological blocks. The second dimension has a size equal to largest number of nodes in a single topological block. The third dimension has a size equal to the largest number of input nodes to any node in the graph plus one. The input connectivity data is structured to facilitate predictable access patterns. Using the syntax (x, y, z) to signify the element with index x in the 1st dimension, y in the 2nd dimension, and z for the 3rd dimension, we specify that the (x, y, 0) element is the integer name of the yth node in the xth topological block. The element at (x, y, l) is the integer name of the node that is the lth input to the yth node of the xth topological block. A consequence of this representation is that it wastes space because not all topological blocks have the same number of nodes, nor do all nodes have the same number of inputs. Elements that are not used have a value of -1. The input connectivity data structure is

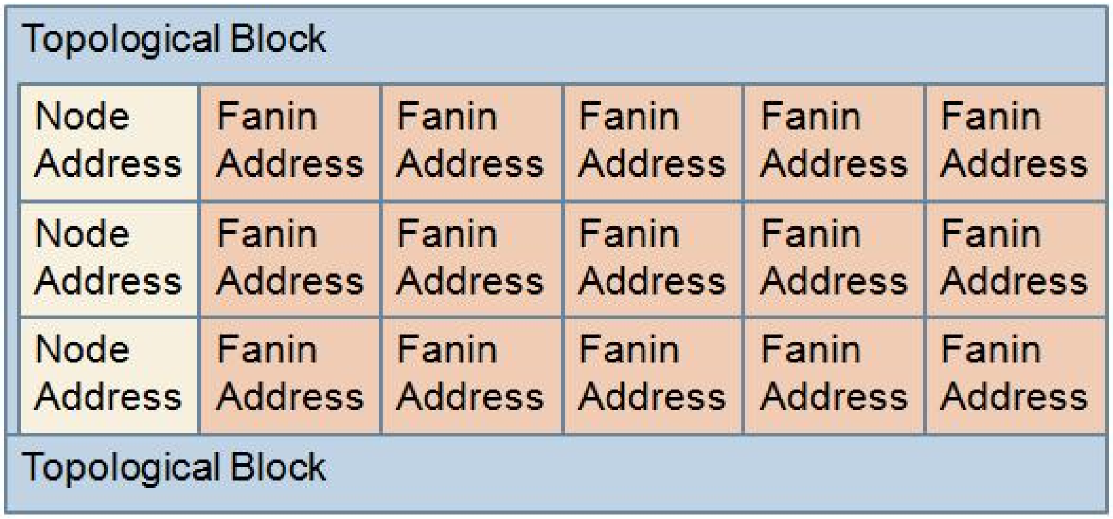

Figure 3.5: This figure shows the input connectivity data structure which stores the node of each topological block and its fanin node addresses.

placed in texture memory on the GPU device. Texture memory has the advantage of being cached. Since the input data is laid out in a structure corresponding to the execution configuration, the accesses to input data texture exhibits strong 2D spatial locality. This improves the access latency and overall performance of the algorithm. Although this data is never written to, constant memory is not a good choice because constant memory is limited to 64KB. For larger cases, input data easily exceeds 64KB.

The data representing the maximum delay for a path of each level depth is also stored in a three dimensional array, as shown in figure 3.6. The first dimension has a size equal to the number of nodes. The second dimension has a size equal to the number of topological blocks. The third dimension has a size equal to the largest number of input nodes to any node in the graph. The delay data array is designed to optimize the maximization kernel in the parallelized kernel, which is discussed below. The input nodes have a delay of 0 for level 0 by default. All other elements of the delay array are initialized to a value of -1. Actual delays for wires and components were not used because

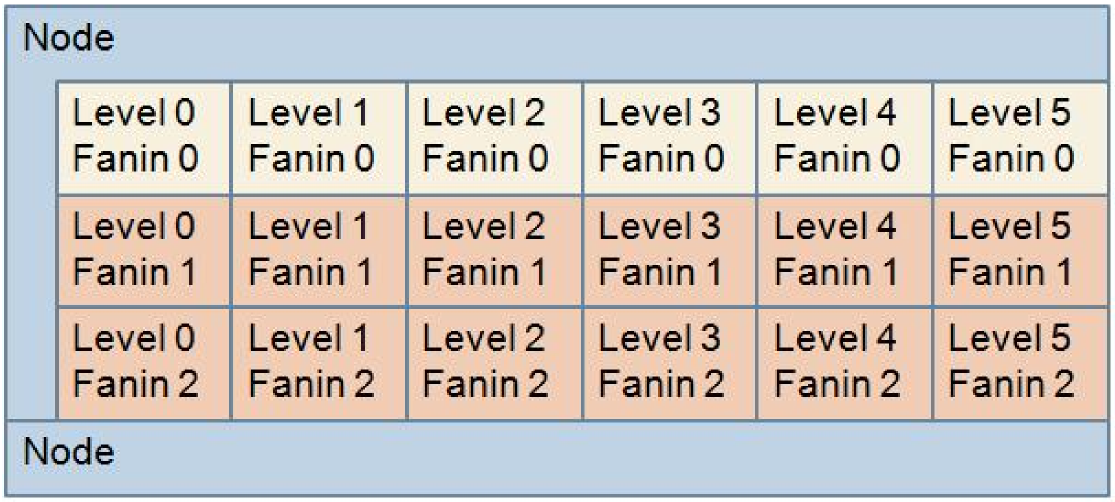

Figure 3.6: This figure shows the delay storage data structure which stores the delays propagated from each fanin for each level of each node.

for most of the test cases, no actual data existed, and accessing the delay information for the larger industrial test case required writing a parser for a complex data format. However, we anticipate that using actual delay data will not affect the performance at all, because delay data would be stored in constant memory, and due to the execution configuration of our kernels, each warp of threads will always be accessing the same delay data address.

Since this memory needs to be accessed from global memory, memory coalescing is crucial for good performance. A requirement for memory coalescing is that the width of the first dimension of each data structure needs to be a multiple of 16 bytes. C for CUDA memory allocation calls automatically pad the memory to ensure that it is aligned.

The Kernel

Using the execution configuration presented above, a thread can easily determine which node, which input node, and which path length delay it should propagate by accessing the input connectivity data structure. Each warp will have coalesced reads from the delay data structure. Each warp will always read and write to a contiguous space in memory because each thread in a warp differs only in which path length it is computing.

The pseudo code for the add kernel is presented in 3.7. A key aspect of GPU

```
1: $nodeID \leftarrow$ blockID in the 1st dimension
2: $scaleID \leftarrow$ blockID in the 2nd dimension
3: $labelID \leftarrow$ threadID in the 1st dimension + $scaleID*$ size of block in 1st dimension
4: $faninID \leftarrow$ threadID in the 2nd dimension
5: **if** $labelID <$ current topological block **then**
6:    $nodeAddress \leftarrow input[nodeID][0]$
7:    **if** $nodeAddress >= 0$ **then**
8:       $faninAddress \leftarrow input[nodeID][faninID+1]$
9:       **if** $faninAddress >= 0$ **then**
10:         $parentDelay \leftarrow delay[faninAddress][labelID][0]$
11:         **if** $parentDelay >= 0$ **then**
12:            $delay[nodeAddress][faninID][label+1] \leftarrow parentDelay + edgedelay$
13:         **end if**
14:      **end if**
15:   **end if**
16: **end if**
```

Figure 3.7: The pseudocode for the add kernel.

programming is making sure to keep the processor busy. Keeping the processor busy allows you to mask the potentially slow memory access times. To keep the GPU busy, the kernel should run with a large number of threads per block, and a large number of blocks per grid. A large number of threads allows multiple warps to be swapped between, keeping the multi-processor busy. A large number of thread blocks allows at least one, and hopefully more than one thread block to be assigned to a multi-processor. The cost of blocks that are operating on an element in the input array that does not specify a node is masked by the GPUs ability to schedule several blocks on the same multiprocessor.

The dimensions of the data structures, as well as the execution configuration, cater to the maximum dimensions of certain features such as: maximum number of input nodes (maximum fanin), or all the possible unique path lengths. However, not all nodes have that many fanins, nor do all nodes have a delay for each path length. The data structures are filled with negative values to denote the lack of a value. As shown in the pseudo-code, checks are made to see if these conditions are true. If a thread is responsible for a value that does not exist, it exits. Warps that complete will not be scheduled, which does not contribute to branching penalty. Due to our execution configuration, any filler entries in the input connectivity will cause entire warps, or thread blocks to complete after a few conditionals. Once retired, these warps or blocks do not negatively impact performance. We make the trade off between a well structured GPU kernel with structured memory accesses versus efficiency of computation. We choose to waste some computation in favor or superior memory access latency and bandwidth. As we will see later, LOCV timing analysis is memory bound.

3.3.2 The Max Kernel

Execution Configuration

We decompose the maximization problem such that each thread performs the maximization over all fanins for a given level of a specific node. This corresponds to doing the max over the columns of figure 3.6. Once again, in order to have a large number of thread blocks without having too few threads per block, we assign each thread block to perform computations for a specific node. Consequently, the grid's 1st dimension has a size equal to the number of nodes. The block's 1st dimension has a size of the number of levels, which is upper bounded by the current block number plus one. This execution configuration ensures that when writing the maximum delay back to global device memory, the write is coalesced. The writes are coalesced because, as shown in figure 3.6, we are the writing to the top row of each node.

The Kernel

The max kernel acts upon the same data structures used in the add kernel. The pseudo code for the max kernel is presented in figure 3.8. From the figure, we can see that a thread determines which node it operates upon by checking its blockID in the 1st dimension. It determines whether or not scaling was used to overcome the maximum thread limit, by checking the 2nd dimension of the blockID. The level that it is assigned to maximize is then its thread ID in the 1st dimension plus the width of a thread block if scaling was used. It declares a register variable to store the maximum delay over the fanins. Then it acquires the actual node address from the input connectivity array. For each fanin, it acquires the delay stores in the delay structure and stores it in the max delay register if it is greater than the current max delay. This process is done serially rather than with a divide and conquer approach because in the vast majority of cases, the number of fanins is 2. Furthermore, synchronization needed for a log order maximization is expensive. We keep the maximum delay in register memory to minimize accesses to global memory. Once the maximum is computed, it is written back to global memory.

Asynchronous Kernel Calls and Memory Transfer

Memory transfer times and kernel executions can be expensive in terms of performance. Each kernel execution comes with an overhead. Each memory transfer is naturally expensive as the data must be transferred across the PCI bus. However, the cost of memory transfers can be hidden by executing memory transfers asynchronously with kernel calls. The algorithm uses a separate stream for memory transfer, and the default stream for kernel calls. The input data structure is transferred onto device memory in pieces, where each piece represents the input connectivity of a single topological block. Since each pair of add and max kernels act solely on a single topological block, those kernel calls are dependent only upon the one piece of memory transfer relating to that topological block. The kernel calls wait on the appropriate memory transfer. The memory transfers are executed asynchronously, and queued one after another. This al-

```
1:  $nodeID \leftarrow$ blockID in the 1st dimension
2:  $scaleID \leftarrow$ blockID in the 2nd dimension
3:  $levelID \leftarrow$ threadID in the 1st dimension + $scaleID*$ size of block in 1st dimension
4:  $maxDelay \leftarrow$ -1.0
5:  $nodeAddress \leftarrow input[nodeID][0]$
6:  for each fanin $f$ do
7:      $delay \leftarrow delay[nodeAddress][f][levelID]$
8:      if $delay > maxDelay$ then
9:          $maxDelay \leftarrow delay$
10:     end if
11: end for
12: if $maxDelay \geq 0$ then
13:     $delay[nodeAddress][0][levelID] \leftarrow maxDelay$
14: end if
```

Figure 3.8: The pseudocode for the max kernel.

lows the memory to transfer at full speed, and the kernels are executed as soon as the necessary memory is available. Note that kernel calls are by default asynchronous as well.

3.4 A Partitioned GPU Parallel Algorithm

In order to maximally exploit the parallelism of the GPU, we seek to increase the amount of parallelism available in the computation. The basic parallel algorithm is forced to perform each topological block in sequential order. However, a key insight is that it is possible to partition the graph into two halves. Then propagate delays from the input to the boundary of the partition, but also propagate delays from the output to the boundary of the partition. Then, by combinatorially summing the levels and their delays, we recover the final solution for the maximum delay for each path length from input to output.

The process of merging is shown in Figure 3.9. The node shown in green is a boundary node. It receives delays from nodes in the left partition, and nodes in the right partition. The figure shows that this particular boundary node has three unique path lengths, ranging from 2 to 4, from the input nodes. The boundary node also has three unique path lengths between it and the output nodes. We combinatorially add every combination of one delay from the left partition to one delay of the right partition. For example, we take the delay 3.2 for the path of length 3 from the left partition, and add it to the delay of 0.6 for the path of length 3 from the right partition. We can conclude that 3.8 is the maximum delay for paths from the input to the output with a length of 3 to reach this specific boundary node, and a length of 3 to reach the output node from this boundary node. However, we must compare this with the sum of the delays for path length of 2 from the left partition and path length of 4 from the right partition. This combination also gives us a total path length of 6. Therefore, we compare these two sums: 3.2 and 3.8, and retain the maximum. Thus, we conclude that for path lengths of 6 from the input to the output of the graph passing through this boundary node, the maximum delay is 3.8. Note that we have only discussed one boundary node, and the same computation must be performed for all boundary node. Then, to find the overall maximum delay for a given path length, the delays for each boundary node must be compared. However, the final maximization over boundary nodes is analogous to the maximization over output nodes, and can be performed on the CPU for no penalty in performance. This combinatorial addition and maximization must be performed for each and every node in the boundary cut. We use a node-cut for partitioning the graph. Both left and right partition propagations will propagate delays to the nodes on the boundary cut.

By partitioning the graph in this method, we can now perform the the propagation for each topological block from each partition in parallel. Figure 3.10 shows the new structure of this computation. Partitions are performed in parallel, though an additional merge step must be performed. In a large graph, a careful selection of the partitioning can reduce the number of topological blocks in a partition by half, and replacing it with only two sequential steps in the merging operation. Partitioning the graph reduces the sequential complexity of the algorithm. A natural progression of this idea is to perform further partitions, and replace them with a merge operation. However, fur-

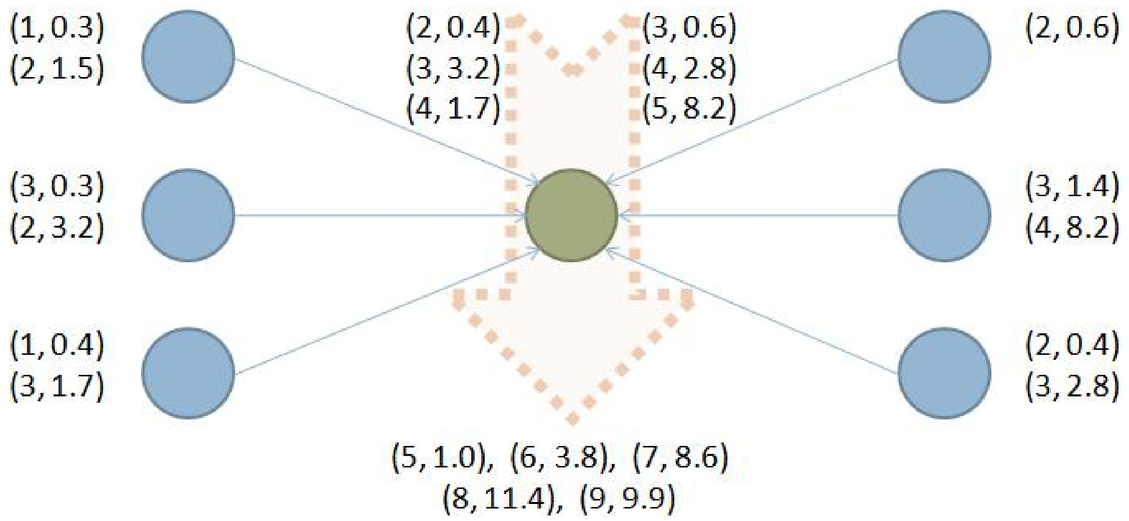

Figure 3.9: This figure shows the process of merging delays propagated from the left and right partitions at a boundary node.

ther partitions would incur significantly greater computational cost. The first partition is relatively cheap because the timing analysis problem treats all input nodes of a graph as the same. All output nodes of the graph are also the same. Input and output nodes can be merged into super input and super output nodes respectively, without loss of information. However, further partitions would need to maintain the maximum delay from each specific input node of the sub-partitions to each specific output nod of the sub-partition. This leads to a combinatorial increase in delays that must be propagated. Effectively, this performs an extra LOCV STA pass through the graph for each input node.

The question of how to partition the graph is also a challenge. The objective of this partition is to find a topologically balanced cut with as few boundary nodes as possible. A topologically balanced cut optimally reduces the number of serial topological blocks by splitting the number of serial topological blocks in half. Minimizing the number of boundary nodes reduces the cost of merging the final solution. We take a very simple approach of performing a topological block ordering, and selecting nodes to add to the left partition in topological order until the partitions are equal in size. A more complex implementation was not used due to time constraints.

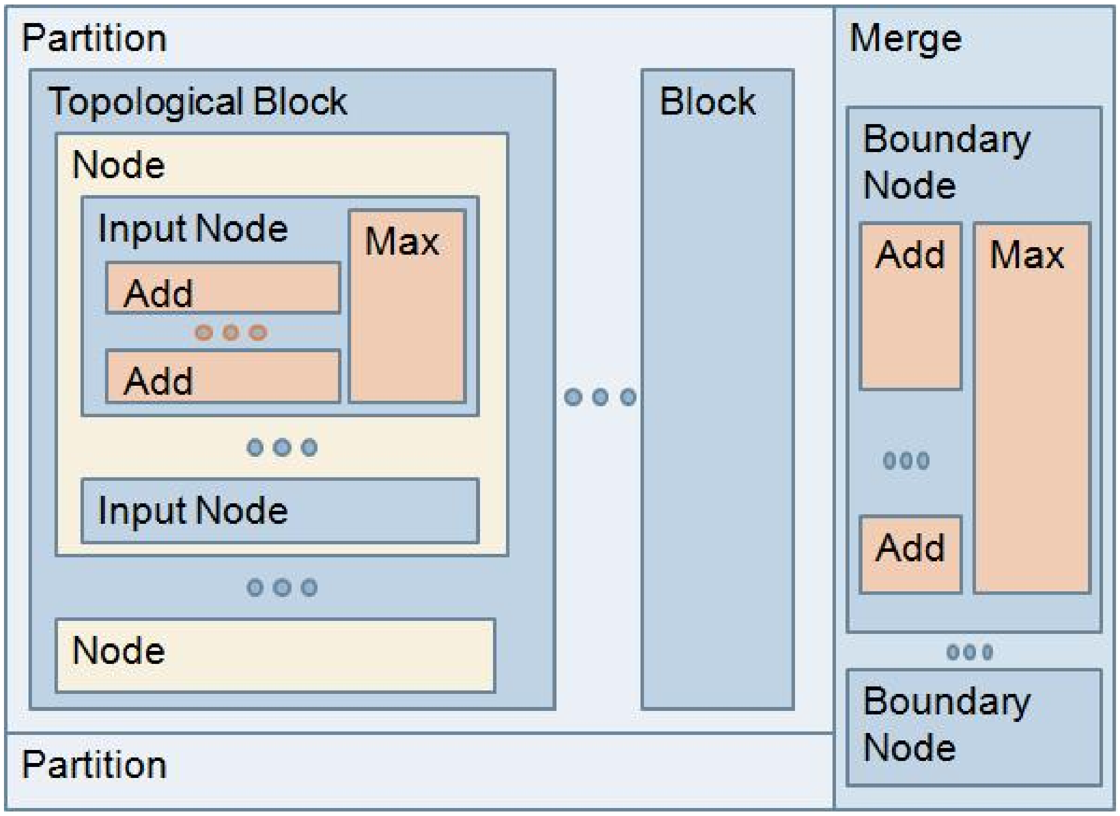

Figure 3.10: This figure shows the structure of the timing analysis computation using a partitioned graph.

3.4.1 A Modified Add Kernel

The add kernel must be modified to handle boundary node computations. The procedure remains mostly the same because a propagation from the output to the boundary can be encoded elegantly into the input connectivity data structure just be taking the output of a node as its fanin, rather than the input. However, the only area in which it must differ is in the computation of the boundary nodes. The delays for each level from both left and right propagations must be stored in a data structure. The original delay data structure has room for only one propagation. Therefore, we create another delay data structure dedicated to storing the delays propagated from the right partition to the boundary nodes. The boundary delay structure is identical to the original delay structure, except that instead of containing delays for every node in the graph, it only contains delays for each boundary node. The original delay structure was indexed by

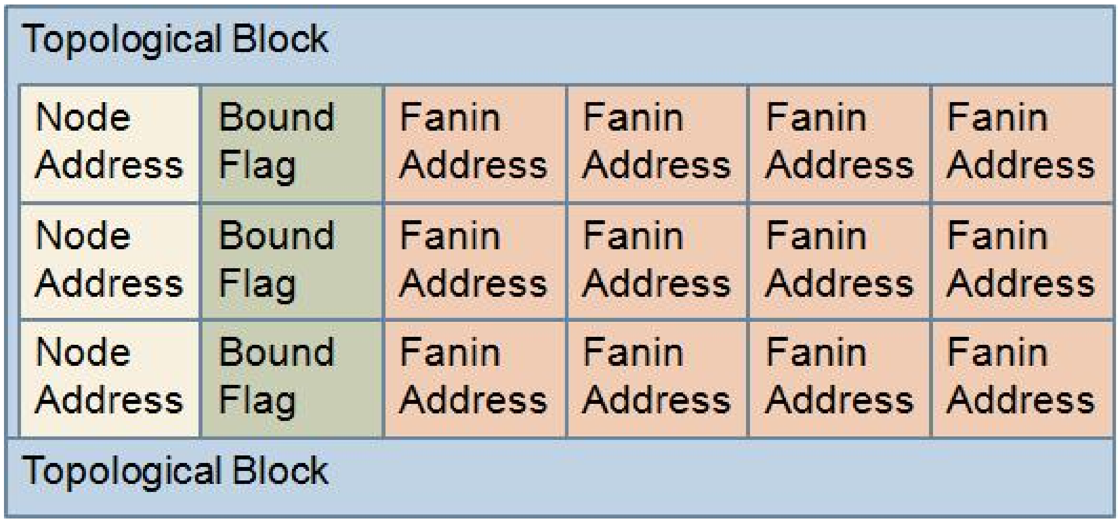

Figure 3.11: The input connectivity for the partitioned parallel algorithm.

the node address, the boundary delay structure is not. Consequently, a method is needed to inform a thread which index of the boundary delay to write to. This is achieved by embedding the index into the input connectivity data structure. Figure 3.11 shows the modified input connectivity array. An additional element is added, called a boundary flag. The boundary flag is set to 1 if its entry is a propagation from the right partition to a boundary node. If this flag is set, the thread will interpret the node address as the index into the boundary delay data structure. The pseudo code for the modified add kernel is presented in 3.12. The structure is identical to the original add kernel, with the exception of an added if statement which checks the bound flag. Depending on the bound flag, the computations are written to either the delay or bound delay structures.

3.4.2 A Modified Max Kernel

The max kernel is similarly modified for the addition of the boundary nodes. Using the same modified data structures as the modified add kernel, it determines which delay data structure to act upon using the boundary flag. The pseudo code for the modified max kernel is presented in Figure 3.13.

3.4.3 The Merge Kernel

Execution Configuration

The merge kernel needs to perform the task of combinatorially adding the delays for each combination of levels for a given boundary node. Once again, to maximize parallelism, we assign each thread to compute one such addition. We assign each thread block to each boundary node. Thus, the grid has a 1st dimension with size equal to the number of boundary nodes. The thread block is a two dimensional arrangement of threads where the 1st dimension is the number of levels from the left partition, and the second dimension is the number of levels from the right partition. Therefore, each thread in the thread block is assigned to a single combination of levels. As a whole, the thread block covers all combinations of levels.

Data Structures

A couple additional data structures are used for merging. The first is a simple one dimensional array where the index of the array corresponds to the index of the boundary delay data. Each element of the array is the node address of the boundary node with the corresponding index. This is a simple mapping from boundary node index to boundary node address. The second data structure is used to store the maximum delay for each level for all boundary nodes. Figure 3.14 shows the final delay data structure with the 1st dimension having a size equal to the number of boundary nodes. The 2nd dimension has a size equal to the largest possible level combination resulting from the merge operation.

The Kernel

The merge kernel is design differently than the add and max kernels because of its combinatorial structure. The combinatorial structure makes coalescing of global memory accesses difficult. Instead, we opt to use shared memory. Unlike the add and max kernel, data retrieved from global memory is used by more than one thread in a thread block. Such an access pattern is good for shared memory. Furthermore, the access pattern of our execution configuration reduces bank conflicts because threads in the same warp access strided sequential memory addresses. Any bank conflicts are a relatively minor set back compared to the cost of accessing global memory.

Figure 3.15 shows the pseudocode for the merge kernel. Our strategy is to first load the left and right delays for a boundary node into shared memory. We use only the first few thread IDs because these threads will be in the same warp. Other threads will reach the barrier synchronization and wait on the memory transfers to shared memory. Once all the threads reach the barrier synchronization, the threads proceed to perform the addition of the delays they are assigned to. An atomic max operation is then used to store the maximums for each level into shared memory. While the atomic max operation provided by the CUDA library only works on integers, we are able to circumvent this restriction by converting a float into an integer. Once the computation is complete, these integers can be converted back into floats. Once again, waiting on a barrier synchronization, the thread block waits on all threads to complete the addition and atomic max before writing the maximum delays to the final delay structure in global memory.

3.5 Complexity Analysis

The complexity of the serial algorithm is $O(D)$ where D is the number of delays that must be computed and propagated. The complexity of the parallel algorithm is $O(B)$ where B is the number of topological blocks, which is also the longest path through the graph. The complexity of the partitioned parallel algorithm is $O(B_1 + B_2)$ where B_1 is the block depth of the first partition, and B_2 is the block depth of the second partition. Ultimately, the partitioned algorithm remains linearly proportional to the topological depth of the original graph.

```
if $labelID <$ current topological block then
```

```
$nodeID \leftarrow$ blockID in the 1st dimension
$scaleID \leftarrow$ blockID in the 2nd dimension
$labelID \leftarrow$ threadID in the 1st dimension + $scaleID*$ size of block in 1st dimension
$faninID \leftarrow$ threadID in the 2nd dimension
if $labelID <$ current topological block then
    $nodeAddress \leftarrow input[nodeID][0]$
    if $nodeAddress >= 0$ then
        if $input[nodeID][1] == 1$ then
            $faninAddress \leftarrow input[nodeID][faninID+1]$
            if $faninAddress >= 0$ then
                $parentDelay \leftarrow bound[faninAddress][labelID][0]$
                if $parentDelay >= 0$ then
                    $bound[nodeAddress][faninID][label+1] \leftarrow parentDelay + edgedelay$
                end if
            end if
        else
            $faninAddress \leftarrow input[nodeID][faninID+1]$
            if $faninAddress >= 0$ then
                $parentDelay \leftarrow delay[faninAddress][labelID][0]$
                if $parentDelay >= 0$ then
                    $delay[nodeAddress][faninID][label+1] \leftarrow parentDelay + edgedelay$
                end if
            end if
        end if
    end if
end if
```

Figure 3.12: The pseudocode for the modified add kernel.

```
$nodeID \leftarrow$ blockID in the 1st dimension
$scaleID \leftarrow$ blockID in the 2nd dimension
$levelID \leftarrow$ threadID in the 1st dimension + $scaleID*$ size of block in 1st dimension
$maxDelay \leftarrow$ -1.0
$nodeAddress \leftarrow input[nodeID][0]$
if $input[nodeID][1] == 1$ then
    for each fanin $f$ do
        $delay \leftarrow bound[nodeAddress][f][levelID]$
        if $delay > maxDelay$ then
            $maxDelay \leftarrow delay$
        end if
    end for
    if $maxDelay \geq 0$ then
        $bound[nodeAddress][0][levelID] \leftarrow maxDelay$
    end if
else
    for each fanin $f$ do
        $delay \leftarrow delay[nodeAddress][f][levelID]$
        if $delay > maxDelay$ then
            $maxDelay \leftarrow delay$
        end if
    end for
    if $maxDelay \geq 0$ then
        $delay[nodeAddress][0][levelID] \leftarrow maxDelay$
    end if
end if
```

Figure 3.13: The pseudocode for the modified max kernel.

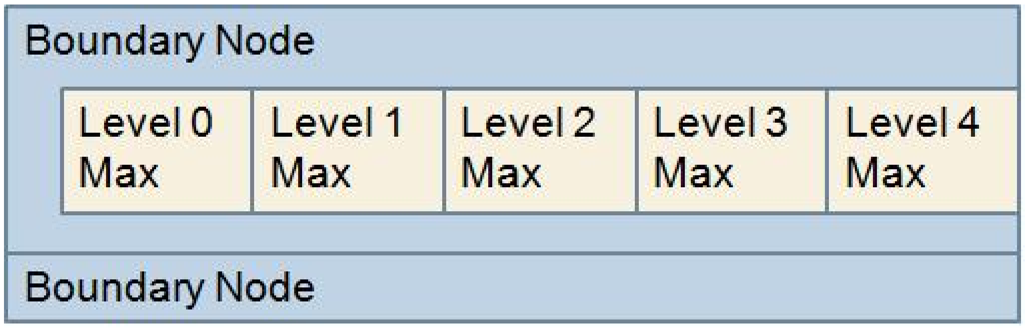

Figure 3.14: The final delay data structure which stores the maximum delay for each unique path length for each boundary node.

```
1: $nodeID \leftarrow$ blockID in the 1st dimension
2: $leftLevel \leftarrow$ threadID in the 1st dimension
3: $rightLevel \leftarrow$ threadID in the 1st dimension
4: $ID \leftarrow$ threadID in the 1st dimension + thread ID in the 2nd dimension * thread block
   size in the 1st dimension
5: $nodeAddress \leftarrow map[nodeID]$
6: if $ID < numberoflevelsfromleftpartition$ then
7:     $sharedLeftDelay[ID] \leftarrow delay[nodeAddress][0][ID]$
8: end if
9: if $ID < numberoflevelsfromrightpartition$ then
10:    $sharedRightDelay[ID] \leftarrow bound[nodeAddress][0][ID]$
11: end if
12: $syncthreads$
13: $delay \leftarrow sharedLeftDelay[leftLevel] + sharedRightDelay[rightLevel]$
14: $atomicMax(sharedFinalDelay[leftLevel + rightLevel], delay)$
15: $syncthreads$
16: if $ID < leftLevel + rightLevel$ then
17:    $final[node][ID] = sharedFinalDelay[ID]$
18: end if
```

Figure 3.15: The pseudocode for the merge kernel.

Chapter 4

Experimental Design and Results

4.1 Experimental Design

4.1.1 Test Cases

Industrial Test Cases

We used an industrial case that was provided in the LEF/DEF format, a common format used in the industry. The DEF file contains the graph connectivity data. Using a DEF parser acquired from the si2.org website, we extracted the circuit graph from the test case. In order to determine the direction of each edge, a complementary library describing each cell component and the direction of each pin was also parsed. A final preprocessing step was needed to convert this circuit graph into something we could use. We determined all of the flip flop and memory components and removed them from the circuit graph. All children of these removed nodes became the input of the graph, all parents of these nodes became the output of the graph.

We also used the ISCAS ’85 benchmark cases. The ISCAS’85 benchmark contains netlists ranging from 5 nodes to 7552 nodes. We chose to not use the smallest case, starting with the netlist with 17 nodes. The purpose of the test cases is unpublished, and are designed to be viewed as random logic circuits. The ISCAS ’85 benchmark is a popular benchmark, that has been studied extensively [8, 10].

Artificial Test Cases

In order to examine the performance characteristics of our algorithm on varying graph structures, a variety of artificial test cases were devised. There are four different artificial test cases named according to their graph structure as they would be drawn on paper: Line, Triangle, Grid and Mesh. The line and triangle test cases are fundamentally

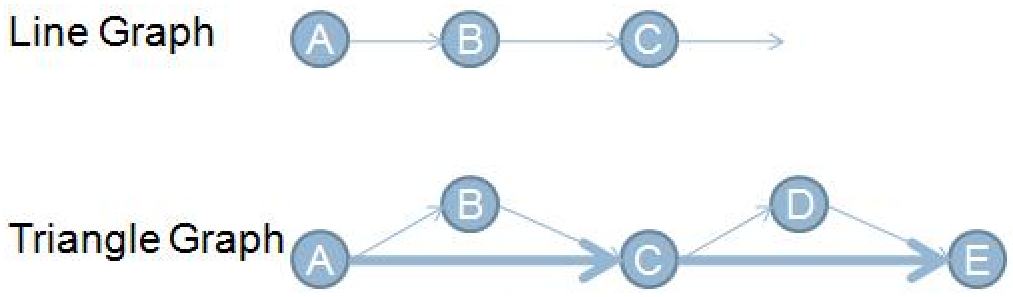

Figure 4.1: The line and triangle graph structures.

the same kind of graph, a purely sequential graph with a number of topological blocks equal to the number of nodes. The only difference is in the connectivity. The line node connects each node to the immediately prior node creating a line, as shown in figure 4.1. Assuming we number the nodes sequentially from 1 to n nodes, the triangle graph differs in that it adds an additional edge from each even edge to the immediately preceding even edge. This creates a structure of a series of linked triangles, as shown in 4.1. The grid

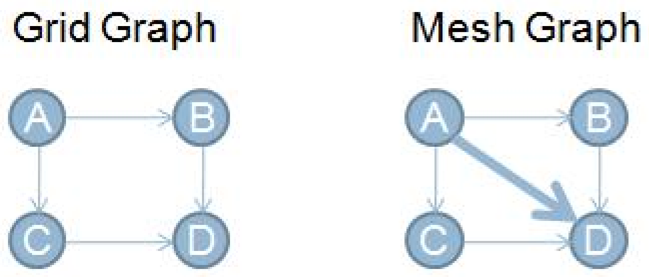

Figure 4.2: The grid and mesh graph structures.

and mesh test cases are also fundamentally similar. As shown in figure 4.2, the grid graph consists of a square of *nxn* nodes arranged in a grid. Each node has parents above and to the left, unless no node exists there, and has children below and to the right. The nodes along the top edge of the square are all input nodes. The nodes along the bottom edge are all output nodes. The mesh builds upon the grid graph structure by including an additional input and output for each node. The node diagonally above and to the left of each node is also an input. The node diagonally below and to the right of each node is an output. This extension to the grid graph structure is analogous to the triangle extension to the line graph.

4.1.2 Experimental Setup

Experiments were performed on three different machines. The first machine is a Macbook Pro running Windows Vista on a Core2Duo T7700 with 2.4GHz clock speed and 4GB of 333MHz RAM. The second machine is a Core i7 920x with 2.66 GHz clock speed and 6 GB of RAM running at 1066MHz. The third machine is a Core2Quad running at 2.4 GHz with 4GB of 800MHz RAM. The Core i7 machine is equipped with a Tesla S1070 which consists of 4 GPU devices. Each GPU device is a Tesla C1060. The Core2Duo machine is equipped with an 8600M GT. The Core2Quad is used only for timing the serial algorithm.

The Core 2 Duo T7700 2.4 GHz is used in our experiments for measuring the speed of the serial algorithm. It comes with a 4MB L2 cache, and a front side bus (FSB) speed of 800 MHz. The Core i7 920 2.66 GHz is also used in our experiments for measuring the speed of the serial algorithm. The i7-920 does not use a front side bus, rather it uses the Intel Quick Path Interconnect technology, which provides superior bandwidth and latency compared to the older FSB technology.

The Tesla C1060 has 30 multiprocessors, each with 8 cores, for a total of 240 cores. The 8600M GT has 4 multiprocessors for a total of 32 cores. The Tesla C1060 is connected to the CPU through a PCIe card on the motherboard, rather than being a PCIe card itself. The 8600M GT is directly connected to the motherboard, as it is part of a laptop machine.

Both the Core i7 and Core2Quad run CentOS, and are 64-bit. The Core2Duo machine is a 32-bit machine running Windows. Compiilation on the linux machines is done using g++ and the nvidia compiler, nvcc. Compilation on the windows machine is done using VS2008 compiler and the nvcc compiler. All experiments are compiled with

Table 4.1: Runtime in milliseconds for the serial, parallel, and partitioned parallel algorithms on the ISCAS '85 benchmark.

	Runtime (milliseconds)					
ISCAS '85	Serial			Parallel		Partitioned
Nodes	Core2Quad	Core i7	Core2Duo	8600M GT	C1060	C1060
17	0.01	0.00	0.03	0.72	1.24	1.08
432	0.06	0.04	0.79	3.18	6.25	6.01
499	0.05	0.03	0.64	1.84	3.77	1.95
880	0.12	0.11	1.47	4.26	7.02	5.75
1355	0.20	0.13	3.80	4.13	8.62	7.24
1908	0.36	0.40	12.90	10.33	12.80	10.12
2670	0.35	0.23	4.46	6.26	11.43	9.02
3540	0.58	0.39	10.90	12.88	15.22	13.17
5315	0.75	0.50	11.24	13.89	15.45	12.51
6288	3.27	2.11	122.5	32.49	50.27	46.22
7552	1.34	0.85	37.44	9.46	14.52	10.12
27751	5.97	3.94			21.76	10.80

release build and debug code removed.

4.2 Experiments and Results

4.2.1 Runtime and Speedup of Serial vs Parallel Algorithms

We tested the serial algorithm on the three different CPUs, and partitioned and basic parallel algorithm on both GPUs for all ISCAS benchmarks. For the largest industrial test case, the GPU lacked sufficient device memory to run the test case. The runtime performance of the algorithms is shown in table 4.1. Also, the 8600M GT was unable to run the merge kernel because its architecture version does not support the atomic max operation. Consequently, only the C1060 has results for the partitioned parallel algorithm.

Comparing the runtime for the serial algorithm versus the parallel algorithm, we find that the serial algorithm outperforms the parallel algorithm in every case except when comparing the Core2Duo's runtime. However, if we look at the speedup, the ratio between the serial algorithm's runtime and the parallel algorithm's runtime, we find that as the graph size grows, the speedup grows linearly. A graph of the speedup as a function of graph size is shown in figure 4.3. From these results, we can conclude that the parallel algorithm implementation carries an overhead when compared to the serial algorithm. However, as the graph size grows, the parallel algorithm scales better than the serial algorithm. Unfortunately, we were unable to acquire larger test cases.

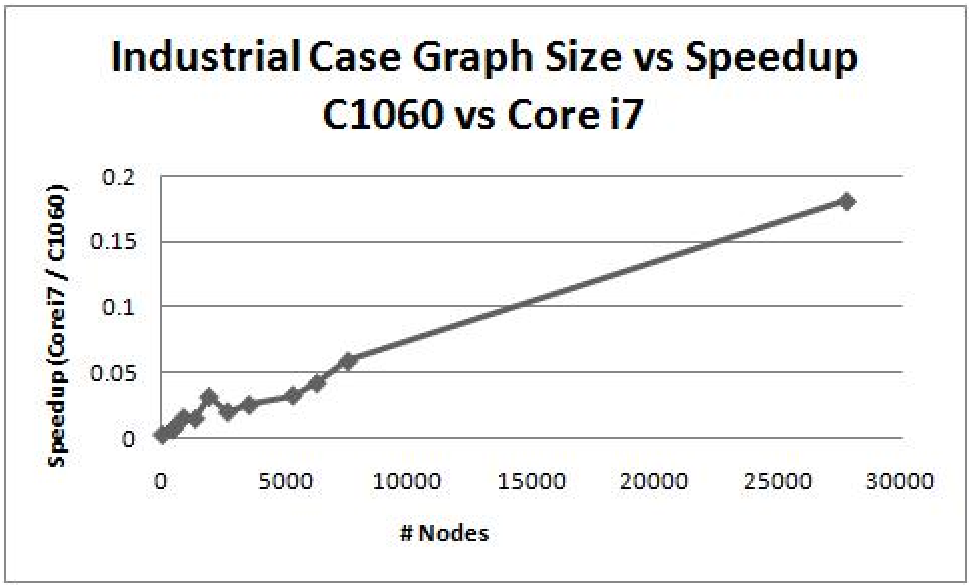

Figure 4.3: A graph of the speedup of the parallel algorithm run on the C1060 over the serial algorithm run on the Core i7 as a function of the graph size.

4.2.2 Serial Algorithm on Varying CPUs

When you compare the runtimes in figure 4.1 for the varying types of CPUs, we find that despite similar clock speeds, the actual performance can vary drastically. However, by examining the corresponding memory frequency, we find a relationship between the memory frequency and serial algorithm's performance. The Core i7 is coupled with a 1GHz memory, and performs the fastest. The Core2Quad has the second fastest memory, and its runtime reflects this difference. Finally, the Core2Duo using the slowest memory and an outdated front side bus architecture demonstrates the slowest runtimes. From these results, we can conclude that the timing analysis problem is a memory bound computation. The dependence upon memory bandwidth and latency helps to explain why the GPU performs slower than the CPUs using faster memory.

4.2.3 Parallel Algorithms on Varying GPUs

A surprising outcome of our tests is the difference in performance between the two GPUs. From figure 4.1, we see that the 8600M GT actually outperforms the C1060, despite having fewer cores and a lower clock rate. We suspect that his may be a result of the C1060 being connected through another PCI card rather than being directly connected to the motherboard. The 8600M GT is also only capable of running small test cases due to its limited device memory. Consequently, all comparisons between the 8600M GT and C1060 in our results reflect test cases in which the C1060 is underutilized. Thus, the C1060 cannot leverage its higher number of cores, while at the same time incurring the overhead of a slower connection between the CPU and GPU.

4.2.4 Line and Triangle Graphs

We also tested the algorithms on the line and triangle graphs. Figure 4.4 shows a graph of the runtimes of the various algorithms on the C1060 and Core i7. We find that the parallel and partitioned parallel algorithms are robust to the changing graph structure. The serial algorithm's performance becomes slower moving from the line to the triangle graph. It is not surprising for the parallel algorithms to be much slower than the serial algorithm. In both the line and the triangle test case, each topological block consists of exactly one node. This effectively removes the majority of parallelism that can be exploited. The runtimes for the parallel algorithms are dominated by the overhead of using the GPU. However, we do find that the partitioned parallel algorithm approximately halves the run-time, which confirms that we improve the parallelism that can be exploited using the partitioning approach. Similar to the ISCAS'85 benchmarks, we find that the scaling of the parallel algorithms is superior to the scaling of the serial algorithm as a function of graph size. Figure 4.5 shows the speedup of the C1060 over the Core i7 for both the line and triangle graphs. For both cases, the speedup increases as the graph size grows. We also tested the algorithms on the Core2Duo and 8600M GT machine for much smaller test case sizes. As mentioned previously, the 8600M GT has

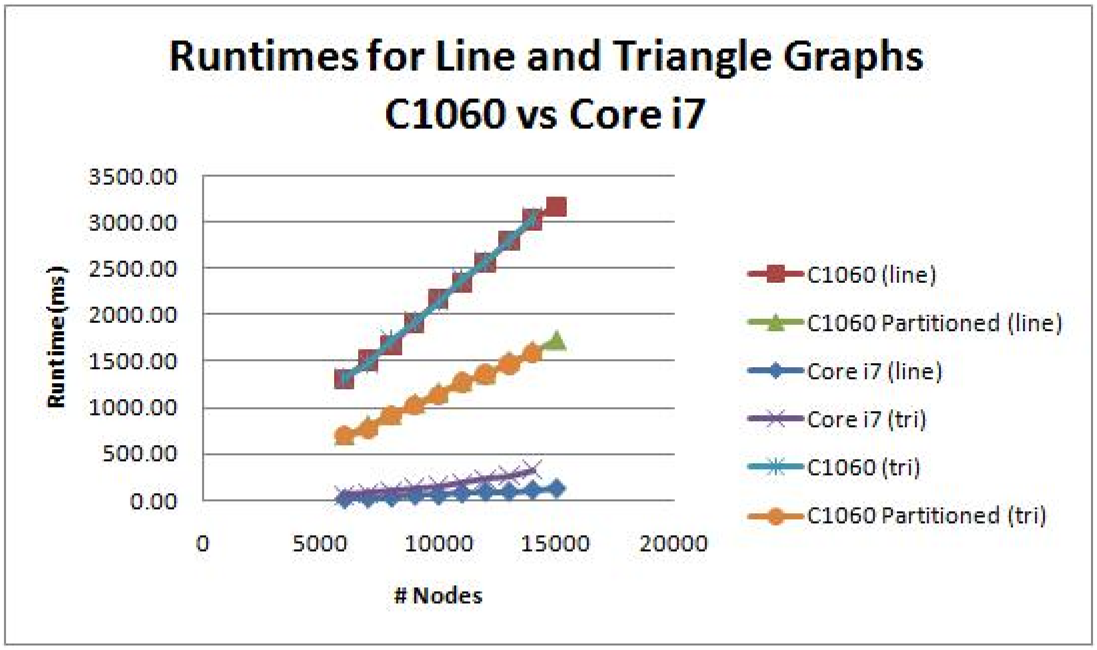

Figure 4.4: A graph of the runtimes of the serial, parallel, and parallel partitioned algorithms on the line and triangle test cases. The tests are run on the C1060 and the Core i7.

limited device memory. Figure 4.6 shows a graph of the runtimes. The C1060 machine, the parallel algorithm is robust to the changing graph structure. However, the serial algorithm's performance drops drastically when moving to the triangle graph. On this machine, the parallel algorithms begin to outperform the serial algorithm for test case sizes greater than 1600.

4.2.5 Mesh and Grid Graphs

The algorithms were tested on the mesh and grid graphs as well. Figure 4.7 shows the runtimes of the serial, parallel, and partitioned parallel algorithms on the grid and mesh test cases run on the C1060 and Core i7. The mesh and grid test cases have a significantly larger graph size compared to the line and triangle test cases. Furthermore, the topological block sizes peak at the width of the square graph structure. Continuing with the pattern from the line and triangle tests, we find that the serial algorithm performs slower on the mesh graph compared to the grid graph. The parallel

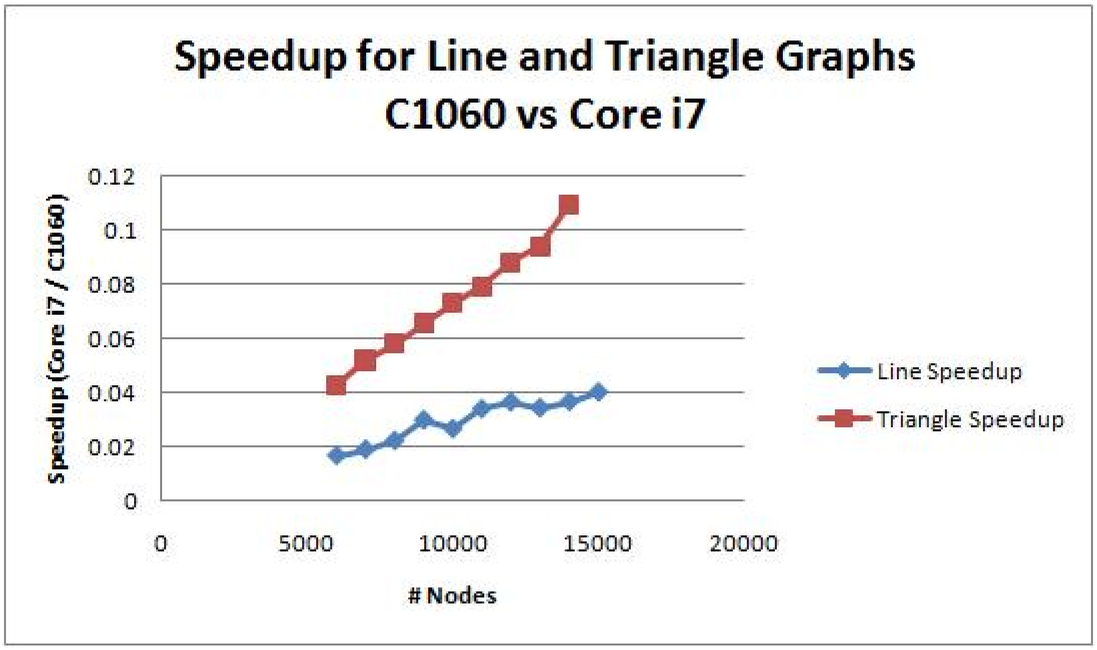

Figure 4.5: A graph of the speedup of the parallel algorithm run on the C1060 over the serial algorithm run on the Core i7 as a function of the graph size of line and triangle test cases.

algorithms remain robust to graph structure changes. Also, the partitioned parallel algorithm continues to show that it improves the amount of exploitable parallelism. We can also conclude that these test cases still do not fully utilize the parallel processing power available on the GPU, because the GPU is capable of accelerating the increased parallelism generated by the partitioning approach. In this case, we find that the parallel algorithms outperform the serial algorithm for the larger test cases exceeding 100000 nodes. Figure 4.9 shows the speedup of the parallel algorithm on the C1060 over the serial algorithm on the Core i7. The speedup grows linearly with respect to the graph size. However, towards the larger end, starts to drop off in growth. This suggests that at a sufficiently large test case size, the parallelism available on the GPU will be fully utilized, at which point larger test cases will not increase the speedup achieved. We tested the serial and parallel algorithms on the grid and mesh graphs running on the Core2Duo and 8600M GT machine. The speedup for these tests is shown in figure **??**. The graph sizes are much smaller than those used in the C1060 tests. However, for this machine, the parallel algorithm quickly overtakes the serial algorithm. It achieves a speed up of

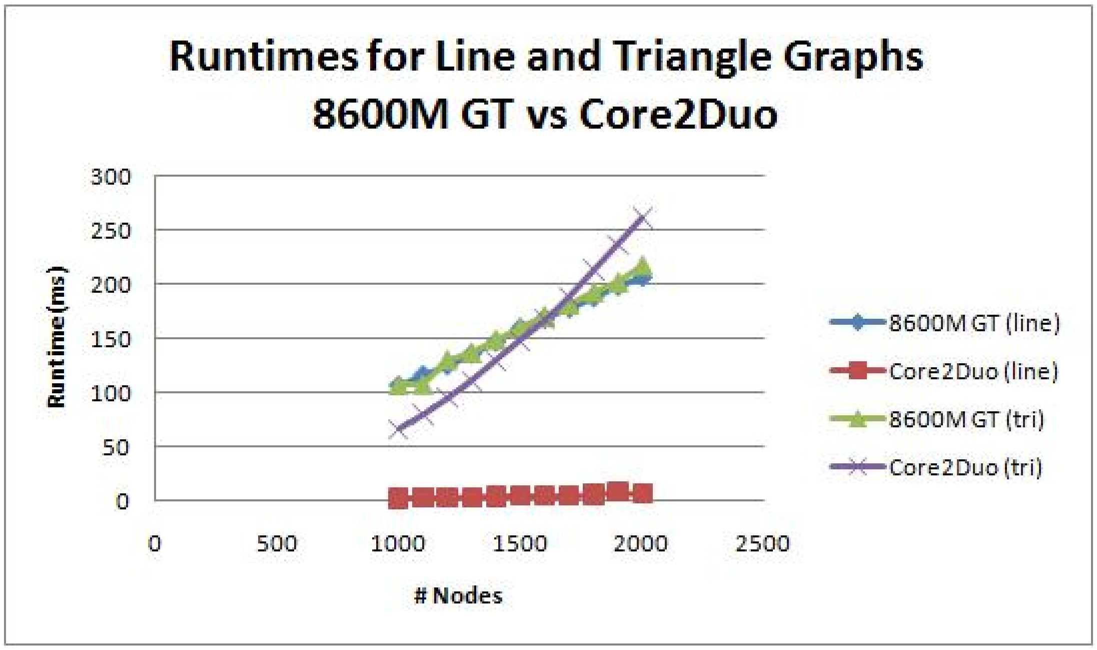

Figure 4.6: A graph of the runtimes of the serial, parallel algorithms on the line and triangle test cases. The tests are run on the 8600M GT and the Core 2 Duo.

over 7x for mesh graphs with 10000 nodes.

4.2.6 Distribution of Nodes in Topological Blocks

Finally, we also took a look at the distribution of the nodes into topological blocks. When generating the topological block, we placed each node in the earliest topological block that it could be placed within. Consequently, the distribution of nodes in a topological block took the shape shown in figure 4.10. Figure 4.10 shows the distribution of nodes into each topological block for the largest industrial test case. It is clear that the largest topological blocks, and thus the most parallelism, occurs in the first few topological blocks. The amount of parallelism drops off quickly as propagation spreads through the graph. We suspect that this negatively impacts the performance of our parallel algorithm by saturating earlier topological blocks, and under utilizing the GPU in later topological blocks.

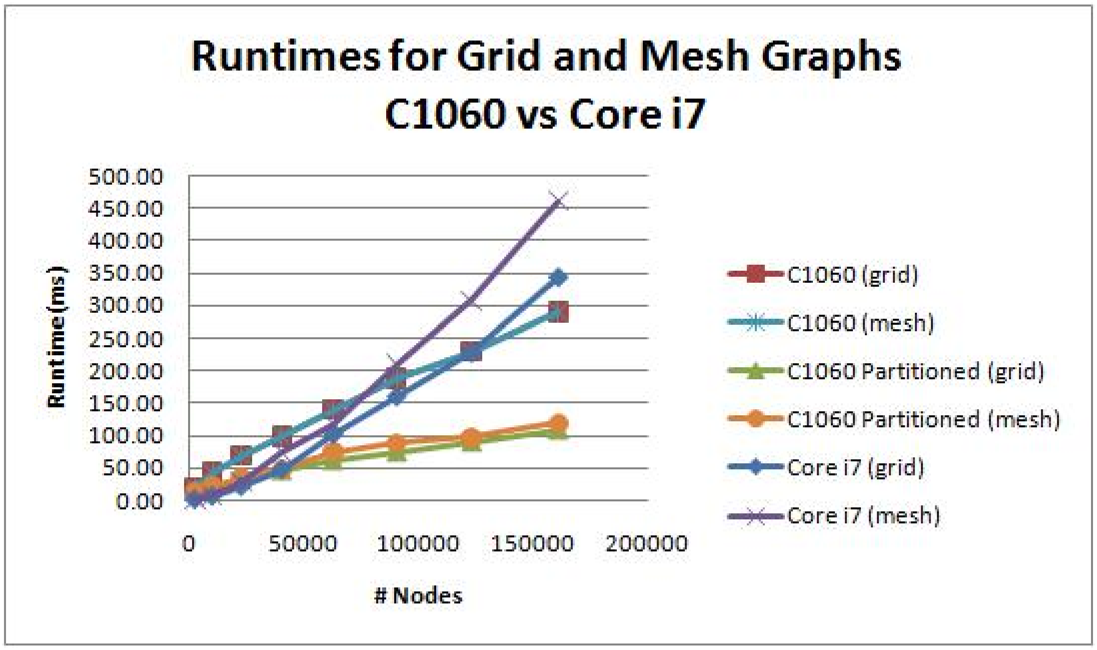

Figure 4.7: A graph of the runtimes of the serial, parallel, and parallel partitioned algorithms on the grid and mesh test cases. The tests are run on the C1060 and the Core i7.

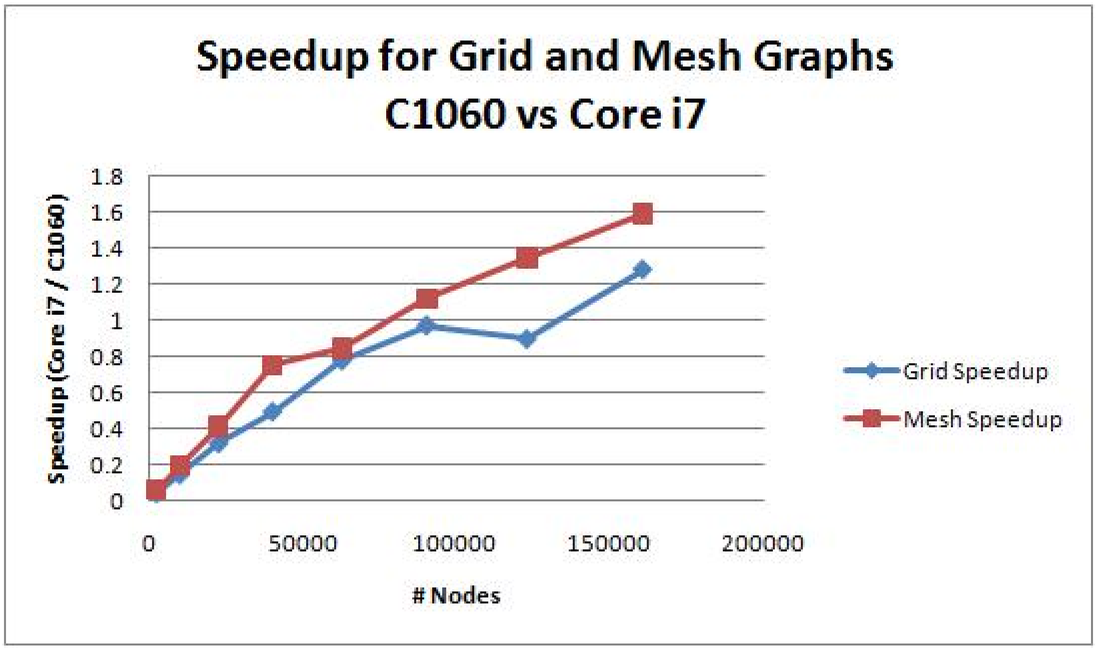

Figure 4.8: A graph of the speedup of the parallel algorithm run on the C1060 over the serial algorithm run on the Core i7 as a function of the graph size of grid and mesh test cases.

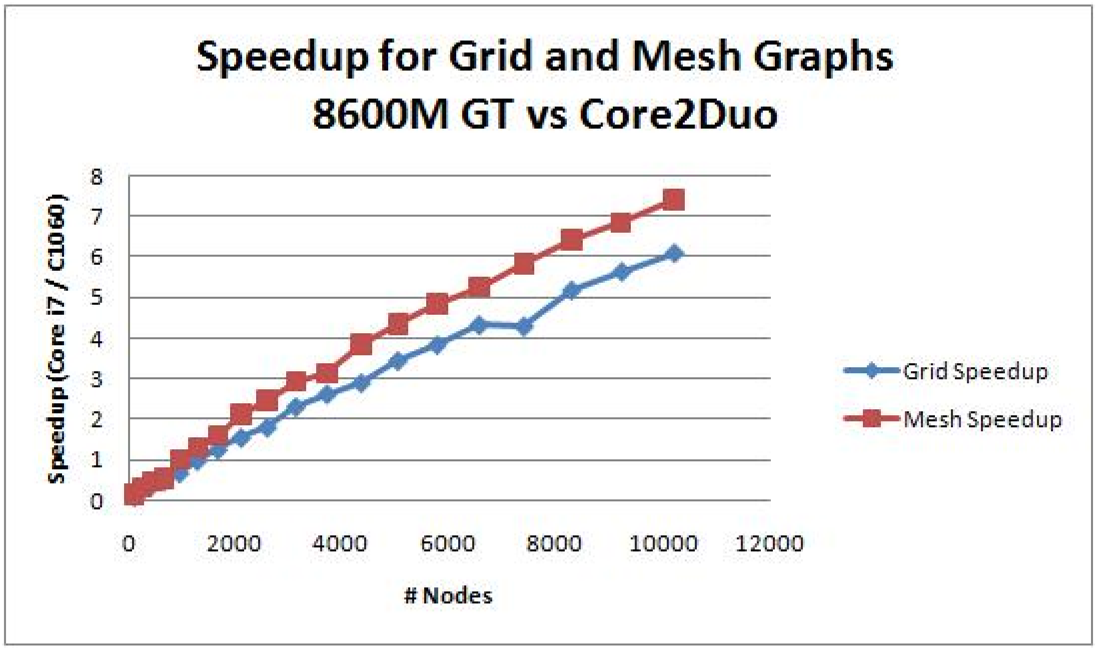

Figure 4.9: A graph of the speedup of the parallel algorithm run on the 8600M GT over the serial algorithm run on the Core 2 Duo as a function of the graph size of grid and mesh test cases.

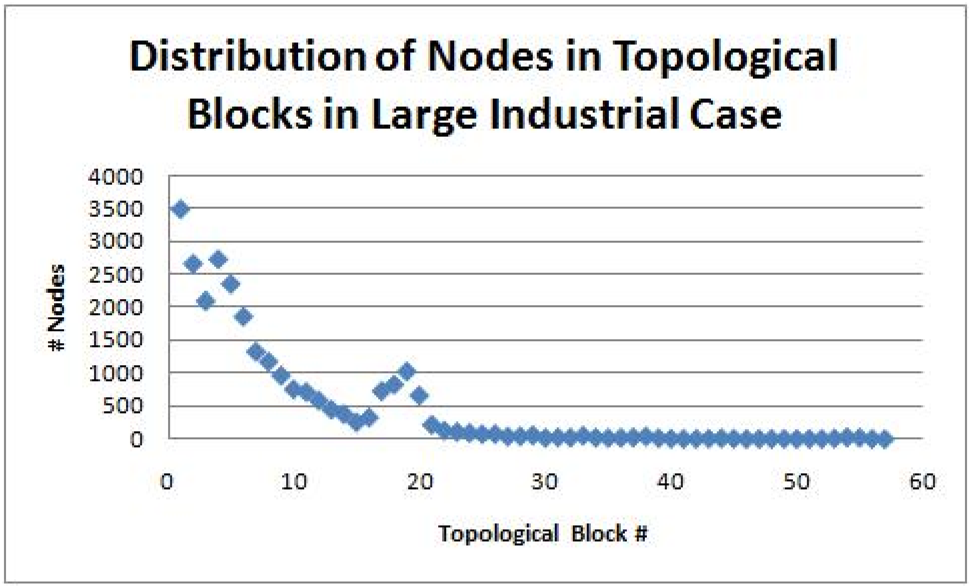

Figure 4.10: A graph of the distribution of nodes in each topological block in the largest industrial case, which has a graph size of 27751.

Chapter 5

Conclusion

In this thesis, we explored the motivations for efficient and accurate timing analysis. In order to improve upon the classic STA, we adopt the Level-Based On-Chip Variation analysis to improve the accuracy of the STA. However, adopting LOCV adds additional computational complexity. We chose to tackle this problem using GPU acceleration. We implement the basic serial algorithm on the CPU, and examined the structure of the computation to determine how best to parallelize it. We presented the design for a GPU parallel algorithm as well as a divide and conquer partitioning based GPU parallel algorithm.

Experimental results from testing the serial algorithm on multiple CPUs with varying memory speeds demonstrated that this computation is a memory bound computation. Experimental results from testing the parallel algorithms on the ISCAS '85 benchmark as well as a large industrial case demonstrate that the GPU parallel algorithm scales with graph size more efficiently than the serial algorithm. Furthermore, the partitioning of the graph improves the available parallelism of the computation, reducing the runtime in cases where the topology of the graph results in a large number of topological blocks. We also found that the using a topological blocking in which each node is assigned to the earliest possible topological block results in a right skewed distribution of nodes. Finally, we demonstrate that the graph structure is a strong factor in the computational cost of the serial algorithm. Whereas the parallel algorithms' performance is not severely impacted by the structure of the graph.

Given that the newer CUDA architecture provide a complete memory hierarchy with cached global memory, it would be interesting to see the performance of our algorithms on the latest Fermi architecture from Nvidia. We would expect performance of the parallel algorithms to improve on the new architecture. Given the importance of efficient memory access, future work should focus on further optimizing memory access. Or, another approach may be to devise an algorithm that is not memory bound.

In terms of exploiting the GPU, our test cases were insufficient to fully utilize the computational power available in the C1060 GPU. We tackled a simplified problem of only finding the maximum delay through the timing graph. The complete timing analysis problem finds the minimum as well as the maximum. Finding the minimum as well as the maximum effectively doubles the amount of data parallelism available. Also, our experiments revealed that a topological blocking results in a skewed distribution of nodes. It is possible to delay the computation of a node to later topological blocks in an effort to balance the load of the GPU. Finally, given that it seems the merging process can be efficiently computed on the GPU, it is feasible to simply perform the partitioned delay propagation in serial on a CPU with access to fast memory, and then offload the merging computation onto the GPU. We conclude that the problem of LOCV timing analysis exhibits large amounts of parallelism that can be exploited on the massively parallel GPU. In cases where graph structure inhibits this parallelization, our partitioning approach improves parallelization. Memory access efficiency is the key to further improvements in performance.

Bibliography

[1] Nvidia's next generation cuda compute architecture. Whitepaper, 2009.

[2] Aseem Agarwal, David Blaauw, and Vladimir Zolotov. Statistical timing analysis for intra-die process variations with spatial correlations. *ICCAD*, 2003.

[3] Denis Bzowy. Locv: location-based on chip variation. *SNUG Europe*, 2004.

[4] R. Chen, E. A. Foreman, P. A. Habitz, J.G. Hemmet, K. Kalafala, J.S. Piaget, P. Qi, N. Venkateswaran, C. Visweswariah, J. Xiong, and V. Zolotov. Static timing: Back to our roots. Technical report, IBM, 2007.

[5] Anirudh Devgan and Chandramouli Kashyap. Block-based static timing analysis with uncertainty. *ICCAD*, 2003.

[6] Jin fuw Lee and Donald T. Tang. An algorithm for incremental timing analysis. *Design Automation Conference*, 1995.

[7] Kanupriya Gulati and Sunil P. Khatri. Accelerating statistical static timing analysis using graphics processing units. *IEEE*, 2009.

[8] Mark C. Hansen, Hakan Yalcin, and John P. Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE*, 1999.

[9] Jerry Hong, Kevin Huang, Peter Pong, JD Pan, Jason Kang, and KC Wu. An llc-ocv methodology for statistic timing analysis. *VLSI-DAT*, 2007.

[10] International Workshop on Microprocessor Test and Verification. *Testing the Path Delay Faults of ISCAS85 Circuit c6288*, 2003.

[11] Andrew B. Kahng, Steganus Mantik, and Igor L. Markov. Min-max placement for large-scale timing optimization. *ISPD*, 2002.

[12] Nvidia. *NVIDIA CUDA Programming Guide*, 2010.

[13] Michael Orshansky and Kurt Keutzer. A general probabilistic framework for worst case timing analysis. *DAC*, 2005.

[14] Michael Orshansky, Linda Milor, Pinhong Chen, Kurt Keutzer, and Chenming Hu. Impact of spatial intrachip gate length variability on the performance of high speed digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21.

[15] Anne-Francoise Pele. Multithreading threatens to unravel beyond 45 nm. http://www.eetimes.com/news/design/showArticle.jhtml?articleID=208403976, June 2008.

[16] Sr. Robert B. Hitchcock. Timing verification and the timing analysis program. *DAC*, 1982.

[17] Amith Singhee, Sonia Singhal, and Rob A. Rutenbar. Practical, fast monte carlo statistical static timing analysis: Why and how. *IEEE*, 2008.

[18] Brian E. Stine, Duane S. Boning, and James E. Chung. Analysis and decomposition of spatial variation in integrated circuit processes and devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10.

[19] Vineeth Veetil, Dennis Sylbester, and David Blaauw. Efficient monte carlo based incremental statistical timing analysis. *DAC*, 2008.

[20] Chandu Visweswariah, Kaushik Ravindran, Kerim Kalafala, Steven G. Walker, Sambasivan Narayan, Daniel K. Beece, Jeff Piaget, Natesan Venkateswaran, and Jeffrey G. Hemmett. First-order incremental block-based statistical timing analysis. *IEEE*, 2006.

[21] Shuo Zhou, Bo Yao, Hongyu Chen, Yi Zhu, Chung-Kuan Cheng, and Mike Hutton. Efficient static timing analysis using a unified framework for false paths and multicycle paths. *IEEE*, 2006.